Ryan Flynn
DS210: Final Project

Links:
Million Song Dataset: http://millionsongdataset.com/

CSV File on Google Drive:
https://drive.google.com/file/d/17jiNBxXqDCu0Djj2AL7k9Q-s2cyz9CZq/view?usp=sharing

Github Repository: https://github.com/rjf126/DS210-Final-Project

Goal:

The goal of this project was to create a music recommendation system based on user data. The code is designed to work on popular songs with many listeners as well as songs with very few listeners.

Pre-processing:

The user data for this project comes from a dataset called "The Million Song Dataset" (MSD). As the title suggests, this dataset contains around a million songs, with various other elements. Because the dataset is so large, the data was split into multiple different files of different formats. I used two files from the dataset. One was titled "train_triplets.txt", and contained a user ID, a song ID, and the number of times the user listened to the song, all separated by tabs. It contained around a million songs. The user ID and song ID were encrypted, but consistent across all the files on the MSD. The next file I used was called "msd_summary_file.h5", which also had around a million songs. This file contained a lot more information about songs, but most importantly, it contained song IDs with the actual song titles. However, the file was in a format ".h5" that we have never worked with before. Luckily, they provided code in python to transform the ".h5" file into a csv (link provided below). Once I converted the ".h5" into a csv file, I merged data with the triplets file by their song IDs using

pandas (they both contained song IDs). This resulted in a csv file with columns containing the user ID, song ID, song title, listen count, track ID, artist ID, and artist ID. However, for this project, the most important columns are the user ID and song title. I uploaded this new csv file, “merged_data.csv”, to Rust in the SRC file of my project file.

Link to python code:
https://github.com/RohanBhirangi/Million-Song-Dataset-Analysis/blob/master/makecsv.py

Concept:

The idea behind this recommendation system was to take a song, scan all the users that listen to it, search for all the songs each of these users listened to, and find the most common song between them. However, this would only work with popular songs that had many listeners. To account for this, for songs with five or less listeners, I would take the three most popular songs among those users, find all the users who have listened to those three most popular songs, and find the most common song between them. This process gives more user data to work with to find the most popular song, and still maintains a strong connection to the inputted song.

Code:

Module csv_reader.rs:

MSD struct: I used this struct to represent the columns in the csv file. It includes 8 different columns in the order they appear on the csv. The first field titled “unknown”, just contained the row number. “user_id” is a unique identifier for each user. “song_id” is a unique identifier for each song. The “listen_count” column contains how many times a user played a song. “Track_id” is another unique identifier for the song and “artist_id” is a unique identifier for the artist, with “artist_name” being the artist's actual name. Finally, the “title” column just contains the title of the song. All of these columns are represented as Strings.

fn read_msd(): This function converts the data in each column of the csv into a vector of the MSD structs. The code reads the csv file path, deserializes the file (meaning converting the csv into a Rust data structure), and adds the elements to a vector corresponding to a field in the

struct. Because the dataset is so large, the code keeps track of how many lines it reads, and stops at a certain number of lines read (I used 200,000). It also makes use of the result enum in case anything in this process goes wrong. The code then returns the vectors under the name “data”. I also split this function and the struct into a different module called “csv_reader.rs”, because the role of this code is to read and store the data, which is very different from the purpose of the other functions.

Module: main.rs

fn songs_to_users: This function takes in the title of a song and the data from the csv, and finds all the users who have listened to the song. It works by creating a hashset to store the user IDs. It then iterates through all of the data, checking where the inputted song title matches song titles in the data. If it finds a match, it will take the corresponding user ID, and add it to the user ID hashset. The benefit of using a hashset is that it ensures all the user ids are unique.

fn user_to_songs: This function takes in the hashset of users and the data from the dataset, and outputs a hashmap of the format (string, hashset). The string, which is the key, will be the user ID, and the hashsets, which are the values, will contain a list of songs associated with the user ID. The function starts by initializing the hashmap. It then iterates through all of the data, and checks if the user ID already exists in the hashmap. If it does, then it checks if the user ID already has a set of songs. If it has a hashset, it will just add the song to the hashset. If not, it clones the user ID and adds it as a key in the hashmap. It then creates a new hashset, and adds a song title to it. After this process ends, the code returns the hashmap as “user_songs_hm”.
Explain in deeper detail

fn most_popular_song: The goal of this function is to take all the songs found in the previous function, and find what song appears the most. The code starts by initializing a hashmap where the keys are song titles (Strings) and the values are the popularity (usize). The code runs through all of the values of the “user_songs_hm” from the previous function (the hashsets), and then through each song title in those hashsets. It is important that the code excludes the inputted song, because it would always be the most popular as all these users listened to it. So if a song is not

the inputted song, and it already exists in the hashmap, +1 is added to its count (its popularity). If the song does not exist in the hashmap, it is cloned and added with a count set to 1. Now there exists a hashmap with all the songs and their popularity. To find the most popular song, the code has to find the song with the highest count. It does this by iterating through each song and their count, checking if the current count is greater than the highest count, and if so, setting the current count to the highest count. This works for finding the most popular song, but it runs into a problem when there are two or more songs with the largest count (equal popularity). Because a hashset does not always contain data in the same order, the hashmaps will not contain the songs in the order. Because of how the code sorts the popularities, if two songs have the highest popularity, and their position in the hashmaps changes, then the most popular song will also change each time you run the code. I ran into this problem with the song "Imagine". In deciding which of the songs to recommend, I thought consistently recommending the same song was important for how the code works. To fix this, I added another parameter for the songs to be sorted by. In cases where songs have the highest but equal popularity, the song with a higher **lexicographical value will be picked.** This is just done by taking two strings and comparing them with < or >. This keeps the most popular song consistent each time you run the code. When you put in an input song, it will output the recommended song along with how many users listened to it.

fn find_more_songs(): In the previous function, the more popular a song is, the more accurate a recommendation it gives. This is because a more popular song gives you more user data, which means the code is more likely to find a pattern in the most popular songs. However, the music industry is so enormous that despite having millions of songs, many songs in the dataset have very few listeners. To account for this, I wrote a function to recommend a song based on an input song with less than 5 listeners. The function uses all the previous functions to make this work. It takes in the input song and the data, and prints out a recommended song within the function. The function works by first finding the users who have listened to the input song by calling "fn songs_to_users". If it finds less than five users, the rest of the function will run. Next, the code calls "fn users_to_songs" using the user's found from " fn songs_to_users". The code now has a hashmap "user_songs_hm" that contains the users who listened to the input songs, as well as all the other songs they listened to. The next goal of the function is to find the 3 most popular songs

in that hashmap. It does this by calling “fn most_popular” on “user_songs_hm”, finding the most popular song in the hashmap, adding it to a vector storing the top 3 songs, and then removing the song from the hashmap. The code uses a for loop to repeat this process 3 times. The purpose of this function is to take the user data one layer deeper in order to find more users, so the code now calls “fn songs_to_users” again to find all the users who have listened to these top 3 songs. It stores all these users in a hashset. The code then takes this hashset, and calls “fn users_to_songs” again, giving us all the songs these users have listened to. Finally, we take the resulting hashmap, and run it through “fn most_popular” to find the most popular song among these top 3 songs. The answer is printed out along with how many users listened to it.

How to Run the Code: Before running the code, make sure that the csv file is in the src folder. The file is too large to upload on its own (1.5 gb), so I will provide a separate link to download it from google drive. To run the code, enter a song title in the main function as “input_song”. For input songs, I chose “Imagine”, which has many users who listened to it. This will show you the output for a popular song (more than 5 listeners), and will just run the function “fn most_popular”. To show the output for an unpopular song, I set the input song to “The Foundation”, which has around 3 listeners. This will run “fn find_more_songs”. The dataset is from 2012, so you must enter a song before then. Finally, to run the code fast, use “cargo run --release”.

Tests: For my tests, I created artificial rows for the dataset using the MSD struct. I then used these rows to make sure the actual and expected output of each function was the same. This confirmed the functions were working properly. For example, for “fn songs_to_users” I input a song that I knew only had two users, and made sure that that answer matched the output of “fn songs_to_users”. My code passed all of the tests.

ChatGPT Conversation: I needed a way to compare two names of songs (strings) in order to break ties in popularity. Just by using a greater than or less than sign, you are able to compare two strings lexicographically. Lexicographically essential means comparing two words by which one would appear in a dictionary first (alphabetical order). The benefit of breaking ties this way

is that it is easy to implement, consistent, and two songs would have to have the same name for there to be another tie, which is impossible with the code I wrote.