Collaborators: None

Written Report for DS 210 Final Project(Spotify Dataset Recommendation System)- Alex Bischof

A. Project Overview

- Goal: What question are you answering?
    - Based on a song that the user likes, I am providing a way for the user to find other similar songs to the one they input into my project.

- Dataset: Source, size (link if too large for GitHub).
    - https://www.kaggle.com/datasets/amitanshjoshi/spotify-1million-tracks/discussion/425875

B. Data Processing

- Loading into rust
    - CSV rows are read into using csv::readerbuilder, and has headers
    - Each row gets deserialized into the Song struct using serde::Deserialize
    - Eventually, Song instances are collected into a Vec<Song>

- Cleaning of the original dataset was done in python using the function (drop.na())

C. Code Structure

1. Modules
    - loading_data: data ingestion and numeric preprocessing.
    - cli: contains knn logic, and the filters that are applied
    - main: everything that is implemented
    - Test: all the tests are stored here
2. Key Functions & Types

Function/Type	What it does	Input	Outputs	Core Logic

Song (struct)	Holds all CSV columns as typed and usable data	CSV row	Song instance	Deserialization
load_data	Read & parse CSV	path: &str	Vec<Song>	Loop rdr.deserialize(), push each row
feature_matrix	Build feature matrix	&[Song]	Array2<f32>	Map each Song tp a Array1<f32>; stack
calc_z_sc	z-score normalization by rows	&mut Array2<f32>		For each row: calculate mean, std, (x–μ)/σ
prompt_song	Selection of song	&[Song]	usize (row index)	Loop until track_id or track_name match
prompt_for_k	Ask user for number of k (1–20)	none	usize (k)	Parse and check to see if within range
prompt_for_popularity / prompt_for_genre	Extra filters	none	Option<...>	Prompting user
build_candidates	Apply filters to generate index list	&[Song], filters, query_idx	Vec<usize>	filter_map on songs
compute_distances	Brute-force Euclidean distance on query and said row	query_idx, &[usize], &Array2	Vec<(f32,usize)>	Map each candidate → (dist, idx)
recommend	prompt → filter → sort → print	&[Song], &Array2<f32>	none (prints to stdout)	Calls above functions in a sequence

3. Main Workflow

Load & normalize in main. Run the following:
let songs = load_data("song_data.csv")?;
let mut feats = feature_matrix(&songs);
calc_z_sc(&mut feats);

After, using recommend→ prompt the song, the k, the filters.
Using this, build the candidate indices, iterate over matrix and compute distances, sort and take the top k. Print result

D. Tests

- cargo test output (paste logs or provide screenshots).

```
warning: `final_project` (bin "final_project" test) generated 7 warnings (run `cargo fix --bin "final_project" --tests` to apply 7 suggestions)
    Finished `test` profile [unoptimized + debuginfo] target(s) in 12.84s
     Running unittests src/main.rs (target/debug/deps/final_project-78af206ba0b683e9)

running 2 tests
test tests::tests::build_candidates_basic ... ok
test tests::tests::compute_distances_basic ... ok

test result: ok. 2 passed; 0 failed; 0 ignored; 0 measured; 0 filtered out; finished in 0.00s

[/opt/app-root/src/final_project]
$
```

Build_candidates_basic: verifies our combined popularity and genre filtering logic correctly excludes and includes indices under each scenario

- With no filters, querying song index 0 returns candidates [1,2] (all other songs).
- With popularity=Popular, only songs with popularity ≥ 50 remain ([1,2]).
- With genre = Different, only songs whose genre aren't the query's genre remain

Compute_distances_basic: Makes sure that euclidean distance is implemented correctly

- What it checks:
    - Given a tiny matrix, compute distances of row - to row 1,2. They should be 5.0, 10.0

- Why it matters:
  Ensures the Euclidean-distance routine in compute_distances is implemented correctly, since accurate distance measures are core to KNN recommendations.

E. Results

- All program outputs (screenshots or pasted).

- Interpretation in project context (no need for "groundbreaking” results).
    - The recommendations usually cluster around similar acoustic tracks, or general “vibe” and loudness of the track. I noticed when same genre was selected, there would often be another song of the same artist that pops up. In a general sense, my project was good at recommending, but there could be some changes made (such as adding weights, or getting an even better dataset with more metrics) to boost the accuracy/similarity between the songs that are suggested

F. Usage Instructions

How my code works:

To run, cargo run –release

It expects song_data.csv in the working directory

- User submits
    - Single song (track ID or name)
    - And answers:
    - “Show same-genre or cross-genre, or both?”
    - 1-20 recommendations?
    - “Does popularity matter?”

        - If no, do nothing, if yes → would user like over/under a threshold(popularity over a certain/under a certain amount)

System returns

- Top-K similar songs
- K = however many recommendation the user said they wanted.
- It is filtered out by the same genre, different genre, or it doesn't matter("none)
- Popularity is 50 or above. If under 50, then will be labeled as underground/not as popular

Takes 15 seconds to import and build code at first run.

Around .05 seconds to run the code through 1 million + rows.