DS210 Project Documentation

Kiefer Sherlock
May 4th, 2025

Overview:

→ **Goal:** The goal of this project is to recommend movies to a user based on their past ratings and ratings of similar users. I was inspired to attempt this project because I have many friends who are big LetterBoxd people and I wanted to help recommend them movies.

→ **Dataset:** MovieLens dataset (movies.csv and ratings.csv)

- movies.csv: ~10,000 movie records (movieID, title, genre(s))
- ratings.csv: ~100,000 rating records (userID, movieID, rating, timestamp)

GroupLens MovieLens – For this project I used a reduced-size file recommended for education and development: ml-latest-small.zip. This file only includes movies until 2018.

Data Processing:

→ **Loaded into Rust**: I used the csv crate to read both movies.csv and ratings.csv. Ratings were deserialized into a Rating struct using serde. Movie records were loaded into a MovieDb Hashmap (movieID → title) for fast lookups.

→ **Cleaning:** I didn't have to clean or transform the data, I did include movies.csv to help convert movieIDs to the names of the movies.

Code Structure

→ Modules:

- **main.rs** → Loading data, running recommendation algorithms, and outputting results.
- **movie_names.rs** → Provides the MovieDb module, which loads the movie title lookup table from movies.csv and allows retrieving a title by its movieID.
- **top_movies.rs** → Provides the top_movies function which returns the top-N movies a user has rated and sorting them by rating and (if tied) by recency.

→ **Key Functions and Types:**

- struct Rating:
    1. Purpose: Represents a single user rating record loaded from the ratings dataset. Holds all the rating data: which user (user_id) rated, which movie (movie_id) they rated, what numeric score they gave (rating), and when they gave it (timestamp, as a UNIX timestamp).

2. Inputs: Populated when we load the CSV using the serde crate, which deserializes each row into a Rating instance.
3. Outputs: As input to build per-user rating vectors (build_user_vectors), to identify which movies have been rated and when (top_movies), and to calculate aggregated score from similar users (recommend_movies).

- struct MovieDb:
    1. Purpose: Holds a lookup table (hashMap<u32, String> mapping movie_id to the readable movie title.
    2. Inputs: Built from reading movies.csv line by line using the from_path method.
    3. Outputs: Provides the get_title(movie_id) method that returns an Option<&str>, either the movie title or None. Used mainly when displaying the recommendations or the user's top-rated list.
- build_user_vectors(ratings: &[Rating]) -> RatingMap
    1. Purpose: Converts Rating records into numeric vectors (DVector<f32>) per user, which each position in the vector corresponding to a specific movie. This structure is necessary to perform mathematical operations like cosine similarity comparisons between users.
    2. Inputs: A slice of Rating objects, covering users and movies.
    3. Core Logic:
        - Collect unique movie_ids, sort, and assign each movie an index.
        - For each user initialize a vector of size = number of movies, filled with zeroes.
        - Fill in the user's actual ratings at the correct positions based on the map.
    4. Output:
       A HashMap<u32, DVector<f32>> where each key is a userID and each value is that user's full rating vectors.
- cosine_similarity(a: &DVector<f32>, b: &DVector<f32>) -> f32
    1. Purpose: Computers the cosine similarity between two rating vectors.
    2. Inputs: Two DVector<32> objects (like from two users).
    3. Core Logic:
        - Calculate the dot product of a and b.
        - Divide by the product of the magnitudes.
        - If each vector is zero (or no ratings), return 0.
    4. Outputs: A single f32 similarity score between 0 and 1. 1 is aligned and 0 means no similarity.
- build_popularity_map(ratings: &[Rating]) -> HashMap<u32, usize>
    1. Purpose: Counts how many times each movie has been rated. Metric is later used to break ties when two movies have identical predicted scores.
    2. Inputs: Slice of Rating records.

3. Core Logic: For each Rating, increment the count for its movie_id in the map.
4. Outputs: A HashMap<u32, usize> mapping each movie_id to the total number of times it was rated.

- recommend_movies(...) -> Vec<u32>
    1. Purpose: For a given target user, recommend top-N movies they have not yet rated, based on the ratings of their most similar users (neighbors).
    2. Inputs: user_id → the ID of the target user. uvecs → the per-user rating vectors. ratings → the full list of ratings. pop → the popularity map for tie-breaking. Top_n → how many recommendations to return.
    3. Core Logic:
        - Calculate similarities: Compare the target user's vector to all other users' vectors using cosine_similarity.
        - Select top-k neighbors: Sort by similarity score, take the top 5 (hardcoded).
        - Aggregate weighted scores: For each unrated movie, sum up the neighbors' ratings weighted by their similarity to the target user.
        - Compute predictions: For each candidate movie, divide total weighted score by total weight (normalize).
        - Sort final recommendations: By predicted score (high to low), then popularity (more ratings first), then movie ID (lowest first).
    4. Outputs: A Vec<u32> containing the movie_ids of the top-N recommended movies.
- top_movies(user_id: u32, ratings: &[Rating], top_n: usize) -> Vec<(u32, f32)>
    1. Purpose: Returns the user's top-N rated movies, breaking ties by favoring more recent ratings.
    2. Input: user_id → the target user, ratings → all available ratings, top_n → number of top movies to return.
    3. Core Logic: Filter, select only ratings matching the user. Sort rating (high to low) then by timestamp (newer first) if ratings tie. Trim by taking only the top-N entries.
    4. Outputs: A list of (movie_id, rating) pairs representing the user's top-rated movies.

→ Main Workflow:

1. Load ratings data and build per-user vectors.
2. Load movie metadata (movie titles).
3. For the selected user:
    - Calculate the cosine similarity to all other users.

- Aggregate ratings from the top-K most similar users.
- Predict the top-N movies (that users have not already rated).

4. Retrieve and display the user's own top-rated movies for comparison.
5. Output all results with both movie IDs and human-readable titles.

Tests

→ cargo test output:

```
running 2 tests
test top_movies::tests::test_top_movies_user2_real_data ... ok
test movie_names::tests::test_get_title ... ok

test result: ok. 2 passed; 0 failed; 0 ignored; 0 measured; 0 filtered out; finished in 0.03s
```

→ Why each test matters:

1. **movie_names::tests::test_get_title:**
   - Confirms that MovieDb correctly retrieves known movie titles by ID.
   - This makes sure that the recommendation outputs are meaningful correct movie titles instead of "unknown"/blank. Tests on a variety of movie ID and name pairs.
2. **top_movies::tests::test_top_movies_user2_real_data:**
   - Verifies that the top_movies function correctly sorts a user's ratings by score, and then by recency when the scores are tied.
   - This is important to give users a reliable view of their top favorites which can be used as a sanity check or benchmark against system recommendations.

Results

→ Output: These are the recommendations for "user 1."

```
Top 5 recommendations for user 1:
- 858: Godfather, The (1972)
- 4306: Shrek (2001)
- 6539: Pirates of the Caribbean: The Curse of the Black Pearl (2003)
- 541: Blade Runner (1982)
- 750: Dr. Strangelove or: How I Learned to Stop Worrying and Love the Bomb (1964)

User 1's top-rated movies:
- 553: Tombstone (1993) (Rating: 5.0)
- 157: Canadian Bacon (1995) (Rating: 5.0)
- 1298: Pink Floyd: The Wall (1982) (Rating: 5.0)
- 3053: Messenger: The Story of Joan of Arc, The (1999) (Rating: 5.0)
- 3448: Good Morning, Vietnam (1987) (Rating: 5.0)
```

→ Interpretation in context: The recommended movies reflect titles highly rated by similar users that the target user hasn't seen yet. These are called "system-inferred preferences"!

I also asked one of my friends for some movie recommendations while creating this project. In a past csv file I included her ratings as “user 0.” Here is a snippet of the output:

```
Top recommendations for user 0:
  7361: Eternal Sunshine of the Spotless Mind (2004)
  1197: Princess Bride, The (1987)
  1193: One Flew Over the Cuckoo's Nest (1975)
  4995: Beautiful Mind, A (2001)
  5445: Minority Report (2002)
Top movies seen by user 0:
  69757: (500) Days of Summer (2009) (Rating: 5.0)
  2959: Fight Club (1999) (Rating: 5.0)
  8376: Napoleon Dynamite (2004) (Rating: 5.0)
  66097: Coraline (2009) (Rating: 5.0)
```

This result was super cool because one of the recommendations (The Princess Bride), is one of her favorite movies (she gave me a variety of movie ratings, ~10, and left that one out).

Usage Instructions

→ Building/Running

1. Install Rust
2. Clone the project repo.
3. Ensure the ratings.csv (user rating data) and movies.csv (movie metadata) are present in the project directory.
4. Build the project:
   cargo build –release
5. Run the program:
   cargo run –release

→ Command-line arguments or user interactions:

- My program looks for hardcoded values for user_id (the user to recommend for) and top_n (the number of recommendations).
  To change these, edit main.rs:
  let user_id = 1;
  let top_n = 5;

The program takes under 1 second to run on a standard laptop.