What does my code do?

I am developing a recommendation algorithm using two distinct methods to provide meaningful and diverse movie suggestions. I chose to do this project because I'm always fascinated by how apps like Netflix or YouTube can predict my preference videos so accurately. This curiosity inspired me to explore how such algorithms work and apply similar principles to a movie recommendation system. I'm using two approaches to give users recommendations.

1. Content-based similarity: Find movies that are directly similar to the input movie based on attributes like genre, director, country, release year, etc.
2. The second one is Graph-based importance (PageRank): Identify influential movies in a network of similar movies, providing recommendations that consider broader relationships.

Content-based similarity provides familiar and relevant recommendations based on the input movie's specific attributes. On the other hand, PageRank is beneficial for viewers seeking diverse suggestions or influential movies beyond direct similarities. By combining these methods, the algorithm ensures both relevance and variety, catering to a wide range of preferences and viewing experiences.

Code Summary/Overview

My CSV file is a list of Netflix movies or TV shows each instance includes features such as the director, released year, etc……. I'm writing a similarity and a PageRank code. The similarity code looks for the top 5 movies similar to the input movie. The PageRank looks for the top 5 movies/TV shows that have the most edges, meaning those with the most connections with others. For the similarity part: we compare a hard-coded input movie ("Dick Johnson Is Dead") to all other movies/TV shows in the dataset. The features used for similarity are Type (Movie/TV Show), Director, Country, Release Year, Rating, and Genres (using Jaccard Similarity). For the PageRank Calculation, we build a similarity graph where the nodes are movies/TV shows. The edges represent high similarity and edge weights are similarity scores. We then rank movies globally by their connectivity in the graph.

How to run and Output:

(Since the dataset is large we need to use cargo run–release to run the code or it will take an extremely long time to run it)

```
Comparing 'Dick Johnson Is Dead' to all other movies...

Top 5 similar movies:
Giving Voice
 (Similarity Score: 0.8333)
The Social Dilemma
 (Similarity Score: 0.8333)
Jeremy Scott: The People's Designer
 (Similarity Score: 0.8333)
Lo and Behold: Reveries of the Connected World
 (Similarity Score: 0.8333)
Final Account
 (Similarity Score: 0.7500)
```

- The high similarity scores (0.8333 for the top four movies) indicate that these movies share strong similarities with the input movie (Dick Johnson Is Dead).
- These recommendations are highly personalized and aligned with the characteristics of the input movie, such as genre, themes, or creators.
- However, the recommendations may be too similar, potentially limiting diversity in the viewing experience. For instance, viewers might find these suggestions predictable if they are looking for new or varied perspectives. Here comes our second approach:

```
Running PageRank...

Top 5 movies by PageRank:
Katherine Ryan: Glitter Room:
(PageRank Score: 0.000500003574416)
Amy Schumer Growing:
(PageRank Score: 0.000500003574416)
Battle Drone:
(PageRank Score: 0.000500003458001)
Michael Bolton's Big, Sexy Valentine's Day Special:
(PageRank Score: 0.000500003399793)
Jim Jefferies : BARE:
(PageRank Score: 0.000500003341585)
```

- These recommendations introduce diversity by surfacing movies that may not directly align with the input movie’s attributes but are significant within the graph.
- This PageRank approach is useful for viewers who want to explore broader connections or discover new movies
- However, one problem with my code is that it produces low PageRank scores suggesting weak connections between every movie. This is probably because the density of the graph is too large since it is a large dataset. Thus, the algorithm struggles to highlight influential movies effectively.

Future evaluation:

Some approaches that help solve the low connectivity of a graph are to construct smaller graphs for the input movie and its direct neighbors instead of building a global graph for all movies.

This approach will make the PageRank calculation more specific and computationally efficient. Another method is that we could remove nodes or edges representing weakly connected movies to simplify the graph. For instance, exclude movies with similarity scores below a certain threshold or those with very few meaningful connections.

Explaining Code:

I created 3 sub-modules (read_file, analyze_file, build_graphs) and a main.rs file for this project.

Module: read_file.rs

This module handles reading and cleaning the data from the input CSV file

```
struct Movie
```

This struct includes key attributes that represent a movie

- We are only considering title, movie_type, director, country, release_year, rating and listed_in

```
impl Movie
```

We are implementing Movie methods to help us read the file

```
pub fn read_and_clean(file_path: &str) -> Vec<Movie>
```

This function reads the CSV file, processes each record, and creates a Vec<Movie>

```
let mut movies = Vec::new();
```

- We first initialize an empty vector to store all the Movie instances

```
let mut rdr = ReaderBuilder::new()
```

- The ReaderBuilder helps to configure the CSV reader for flexible input handling
- flexible (true): allows the reader to handle rows with varying numbers of fields because we have different types of data (features) for a movie instance
- trim(csv::Trim::All): removes extra whitespace from each field
- from_path(file_path): opens the specific file
- .expect("Could not open file"): panics if the file cannot be opened, will print the error message if the file cannot open

```
for result in rdr.records()
```

- Iternates overall movie instances in the CSV file
- Since there are some movies with empty fields so in *result* we need to handle both cases
    - Ok(record): A valid record
    - Err(err): An error encountered while reading a record

    ```
    match result
    ```

        - match both possible outcomes of *result,* Ok(record) and Err(err)
        - For Ok(record): retrieves the field at the specific column index, .unwrap_or("") ensures to handle cases when we have empty data. Each field is then assigned to the corresponding Movie struct.

        - After retrieving the data, we add the constructed Movie instance to the *movies* vector
    - For Err(err): We print the message "Skipping invalid record:" to skeps the invalid record and continue processing the rest
- Lastly, we return the *movies vector*

Module: analyze_file.rs

This module analyzes individual features for the movie instances and computes similarity scores from one movie to another. These functions return a similarity score for a specific feature based on how closely the two movies match.

```
pub fn analyze_type(type1: &str, type2: &str) -> f32
```

This function compares the types of two movies, for example, whether it is a "Movie" or a "TV Show"

- If both types are empty then return 0.0
- If both are the same, return 1.0
- If different, return 0.0

```
pub fn analyze_director(director1: &str, director2: &str) -> f32 {
```

This function does the same thing but compares the directors

```
pub fn analyze_country(country1: &str, country2: &str) -> f32 {
```

This function is also similar to the previous one but compares the countries of both movies

- If either country is empty return 0.0
- If both countries are exactly the same, return 1.0
- If the countries are partially overlap, return 0.5
- If there's no overlap, return 0.0

  Since a movie can have multiple associated countries, we need to use HashSet to store the data from both movies

  ```
  let intersection: usize = countries1.intersection(&countries2).count();
  ```

  This intersection method compares two HashSets and identifies elements that are present in both sets

    - We use count to get the number of countries overlappings and if we have more than one overlapping country, we return 0.5. Else, return 0.0.

```
pub fn analyze_release_year(year1: u32, year2: u32) -> f32 {
```

This function calculates the similarity scores based on the movie's release year

```
        let diff = (year1 as i32 - year2 as i32).abs();
```

- We use 5 years as a time frame and if the difference in release year between both movies is less than 5 years then return 1.0
- If the difference is larger than 10 years, return 0.5
- If larger than 10 years, return 0.0

```
        pub fn analyze_rating(rating1: &str, rating2: &str) -> f32 {
```

  This function compares the ratings of two movies, for example ("PG", "R")

- If either rating is empty, return 0.0

- Ratings are the same, with a return 1.0 and a different return 0.0.

```
pub fn analyze_listed_in(genres1: &str, genres2: &str) -> f32 {
```

This function analyzes two movies' genres using Jaccard Similarity (I discovered and learned about this way of comparing similarity from ChatGPT).

- Firstly, if one or both are empty then return 0.0
- Otherwise: we use Hashsets to split the genre strings into individual genres and then calculate the intersection and union of the sets

```
let intersection: usize = genres1.intersection(&genres2).count();
let union: usize = genres1.union(&genres2).count();
```

  - Jaccard Similarity = intersection/union, it is a measure to compare the similarity and diversity of two sets. It ranges from 0, completely disjoint sets, to 1, identical sets.
  - Our return value is the Jaccard Similarity

Module: build_graphs.rs

This module implements functions for analyzing the relationship between movies, building a graph based on their similarities, and applying the PageRank algorithm to rank movies.

```
pub fn calculate_similarity(movie1: &Movie, movie2: &Movie) -> f32 {
```

This function computes the overall similarity score between two movies by aggregating feature-level similarity scores which we calculated in the analyze_file.rs

- We create a *score* variable to hold the total scores
- We then call each feature's score calculation functions to calculate the similarity scores
- Lastly, we sum up the scores to compute the overall similarity and return it

```
pub fn build_graph(movies: &Vec<Movie>, threshold: f32) -> HashMap<String,
Vec<(String, f32)>> {
```

This function constructs a similarity graph where the nodes represent movies and edges represent the similarity between pairs of movies

- The outer loop iterates overall movies, and for each movie, we create a list of edges to other movies

```
if similarity >= threshold {
                    edges.push((movies[j].title.clone(), similarity));
                }
```

-
- This inner loop compares the current movie with every other movie using calculate_similarity().
  - We add an edge if the similarity score is above the threshold.
  - The threshold is a minimum similarity score required for two movies to form an edge in the graph.
  - If the similarity score between movies[i] and movies[j] is greater than or equal to the threshold, then an edge is created.
- We then normalize the weights to ensure the sum of edge weights for each node is 1.

        - We first iterator over the edges vector and use .map() to extract the weight from each tuple and sum up the weights to calculate the total weight of all edges
        - The edges.into_iter() method converts the edges vector into an iterator that consumes the vector and yields owned values. The map() closure normalizes the weight by dividing them by total_weight. Making sure the weights for a node is 1 is important for PageRank algorithms where the relative importance of connections matters. This normalization standardizes the similarity graph, making it easier to interpret and compare nodes. Lastly, the collect() method collects the results into a new vector.

```
graph.insert(movies[i].title.clone(), normalized_edges);
```

- The graph is represented as a HashMap, where the key is the title of the movie and the value is a vector of tuples where each tuple contains a connected movie and its normalized weight.
- Return graph

```
pub fn compute_pagerank(graph: &HashMap<String, Vec<(String, f32)>>, damping: f32,
iterations: usize) -> HashMap<String, f32> {
```

This function implements the PageRank algorithm to compute a ranking score for nodes(movies) in a graph based on their connections. We considered both direct and indirect connections. Direct connections compare how similar the movie is to others and indirect connections compare how similar the connected movies are to others.

- Movies that are well-connected to high-ranking movies will have higher ranks.

```
    let num_nodes = graph.len() as f32;
```

- This computes the total number of nodes
- The initial rank = 1/(num_nodes) to all nodes
- The .key() method retrieves an iterator over the keys (movie titles) of the HashMap representing the graph. The .map() closure applies a transformation to each key and then .collect() converts the iterator produced into a HashMap.
    - Rank is now a HashMap where the key is each movie’s title and the value is its initial PageRank Score

```
    for _ in 0..iterations {
```

- This iterative update repeats the update process for the specified number of iterations
- Inner loop: for each node, calculate its new rank
    - rank_sum += ranks[neighbor] * weight: each neighbor contributes a portion of its rank, weighted by the edge strength which is the similarity score.

    ```
    new_ranks.insert(node.clone(), (1.0 - damping) / num_nodes + damping *
    rank_sum)
    ```

    - This is the damping Factor that balances random jumps and ranks contributions from neighbors (learned this from ChatGPT)
    - (1.0 - damping) / num_nodes:
        - This is the base rank.

        - Damping factor is the probability of continuing to follow links in the graph vs randomly jumping to another node. For example: With damping (
        - *d*=0.85. You follow recommendations (based on movie similarity) 85% of the time.
        - (1.0 - damping): This is the probability of jumping to a random node
        - / num_nodes: this ensures the base rank is evenly distributed across all nodes in the graph
    - damping * rank_sum: This is the contribution from neighbors. Rank_sum is the sum of the weighted ranks of the node's neighbors. Damping scales this contribution.
    - The full formula of combining the base rank and neighbor contributions is New Rank = Base Rank + Weighted Contributions
- Return the final ranks

The main.rs file:

We use mod to import the previous 3 modules, read_file, analyze_file and build_graph

```
let file_path = "netflix_titles 2.csv";
```

We directly put the file name in the file_path variable.

- We use is_empty() method to do error handling to handle cases when the file is empty or invalid CSV file

We hardcoded a movie title, input_title = "Dick Johnson Is Dead" for testing.

```
    if let Some(selected_movie) = movies.iter().find(|m| m.title == input_title) {
```

- This line searches for a movie in the list with the input_title
- We then iterate over all Movie objects in the movies vector
- .find() method looks for the movie
- Some(selected_movie): if a match is found, it assigns the matched Movie object to slected_movie and executes the rest of the code
    - If not found, the block is skipped

```
    let mut similarities = Vec::new();
```

- We then calculate the similarity score between the selected_movie and the current movie using the calculate_similarity function
- similarities.push() method adds the result to the similarities vector

```
    similarities.sort_by(|a, b|b.1.partial_cmp(&a.1).unwrap());
```

- The similarities vector is sorted in descending order of similarity scores
- Lastly, we printed out the top 5 similar movies

```
    let sample_size = 2000;
```

- For the PageRank: we use random sampling (with a sample size of 2000) and 100 iterations to compute PageRank.

```
    let ranks = compute_pagerank(&graph, damping_factor, iterations);
```

- We use the standard damping_factor which equals 0.85 and then calculate the ranks for every movie.

- In the last line, we create an error handling code that will print "Movie not found in the dataset." when the input movie is not found

ChatGPTfor Jaccard Similarity and PageRank's Damping Factor:

what method can i use for measiring the similarity of genre

To measure the similarity of genres between movies, several methods can be used depending on the representation of genres (e.g., as sets, lists, or weighted vectors). Here are some effective methods:

1. Jaccard Similarity

Jaccard similarity is a commonly used method for comparing sets. It measures the proportion of shared elements (intersection) to the total number of unique elements (union).

Formula:

$$\text{Jaccard Similarity} = \frac{|A \cap B|}{|A \cup B|}$$

Implementation:

If genres are stored as comma-separated values:

rust Copy code

```rust
use std::collections::HashSet;

pub fn analyze_genres(genres1: &str, genres2: &str) -> f32 {
    let set1: HashSet<&str> = genres1.split(',').map(|g| g.trim()).collect();
    let set2: HashSet<&str> = genres2.split(',').map(|g| g.trim()).collect();

    let intersection: f32 = set1.intersection(&set2).count() as f32;
    let union: f32 = set1.union(&set2).count() as f32;

    if union == 0.0 {
        0.0
    } else {
        intersection / union
    }
}
```

for page rank how can i do the random jumps

In PageRank, **random jumps** are incorporated into the computation to simulate the possibility of a user randomly navigating to any node in the graph, not just through links. This behavior is controlled by the **damping factor**, typically set to `0.85`.

Here's how you can implement **random jumps** in the PageRank algorithm:

1. Random Jumps in PageRank

The updated PageRank equation with random jumps is:

$$PR(v_i) = \frac{1-d}{N} + d \sum_{v_j \in M(v_i)} \frac{PR(v_j)}{L(v_j)}$$

Where:

- $PR(v_i)$: PageRank of node i.
- d: Damping factor (e.g., 0.85).
- N: Total number of nodes in the graph.
- $M(v_i)$: Nodes pointing to v_i.
- $L(v_j)$: Number of outgoing edges from v_j.

The term $\frac{1-d}{N}$ accounts for random jumps, where the user randomly navigates to any page with equal probability.

Netflix Movies Dataset URL:

https://www.kaggle.com/datasets/rahulvyasm/netflix-movies-and-tv-shows/data