Final Assignment - Statistical Machine Learning
Professor: Hadi Farahani
Student: Sheedeh Sharif Bakhtiar (400422108)

Implementation Links:

- **Hugging Face Space**
- **GitHub Repository**

Introduction

Our final assignment (details here) was to create a Recommender system based on the Movie Dataset. A clustering-based approach was taken for this specific assignment, the details of which are described in the Implementation section later on.

Data Pre-processing

For this specific assignment, data preprocessing was rather difficult, as the data was unclean.

- All data processing was done in the `DF_Construction.ipynb` jupyter notebook. For *code* regarding data cleaning and additional explanation surrounding the code, please refer to this file.

According to the dataset page on kaggle, this dataset consisted of several different csv files:

- **movies_metadata.csv**: The main Movies Metadata file. Contains information on 45,000 movies featured in the Full MovieLens dataset. Features include posters, backdrops, budget, revenue, release dates, languages, production countries and companies.
- **keywords.csv**: Contains the movie plot keywords for our MovieLens movies. Available in the form of a stringified JSON Object.
- **credits.csv**: Consists of Cast and Crew Information for all our movies. Available in the form of a stringified JSON Object.
- **links.csv**: The file that contains the TMDB and IMDB IDs of all the movies featured in the Full MovieLens dataset.

- **links_small.csv**: Contains the TMDB and IMDB IDs of a small subset of 9,000 movies of the Full Dataset.
- **ratings_small.csv**: The subset of 100,000 ratings from 700 users on 9,000 movies.

However, not all information would be relevant to our algorithm. In fact, there are only two files: `movies_metadata.csv` and `ratings.csv` to finally construct two DataFrames that would be used in the final recommender model: `items.csv` and `users.csv`.

Items DataFrame

For the data pertaining to each movie (i.e. the "items DataFrame"), I used `movies_metadata.csv`.

My first step was to first drop columns that were irrelevant to the task at hand, such as the movie's homepage or tagline. Some columns such as `imdb_id` were only kept for identification purposes later on when I used the OMDb API for the purpose of filling in gaps in my data.

Next, I checked for any null entries. Unfortunately, there were quite a few. However, before I got started filling these cells in, I dropped all movies that did not have a status of "released" – in other words, movies that had not been released yet should not be taken into consideration when recommending movies to a specific user.

Thankfully, after filtering out unreleased movies, a number of null cells were removed from the dataframe. My next step was to make requests to the OMDb API to find the missing information for each movie.

To find each movie, I made queries with either the movie title or the movie imdb ID. The sample response from the API had the following form:

```
{'Title': 'Toy Story',
 'Year': '1995',
 'Rated': 'G',
 'Released': '25 Nov 1995',
 'Runtime': '81 min',
 'Genre': 'Animation, Adventure, Comedy',
 'Director': 'John Lasseter',
 'Writer': 'John Lasseter, Pete Docter, Andrew Stanton',
 'Actors': 'Tom Hanks, Tim Allen, Don Rickles',
 'Plot': "A cowboy doll is profoundly threatened and jealous when a new spaceman action figure supplants him as top toy in a boy's bedroom.",
 'Language': 'English',
 'Country': 'United States, Japan',
 'Awards': 'Nominated for 3 Oscars. 29 wins & 23 nominations total',
 'Poster': 'https://m.media-amazon.com/images/M/MV5BMDU2ZWJlMjktMTRhMy00ZTA5LWEzNDgtYmNmZTEwZTViZWJkXkEyXkFqcGdeQXVyNDQ2OTk4MzI@._V1_SX300.jpg',
 'Ratings': [{'Source': 'Internet Movie Database', 'Value': '8.3/10'},
  {'Source': 'Rotten Tomatoes', 'Value': '100%'},
  {'Source': 'Metacritic', 'Value': '96/100'}],
 'Metascore': '96',
 'imdbRating': '8.3',
 'imdbVotes': '1,018,595',
 'imdbID': 'tt0114709',
 'Type': 'movie',
 'DVD': '23 Mar 2010',
 'BoxOffice': '$223,225,679',
 'Production': 'N/A',
 'Website': 'N/A',
 'Response': 'True'}
```


For each row with null values, an API request was made in an attempt to fill out these null values. Thankfully, only three columns had missing data at all: `imdb_id`, `overview` and `runtime`. However, despite making API calls, not all null values could be filled, as some values were missing on OMDb's backend as well.

None of these columns save for runtime had an effect on the final recommendation model, so null `imdb_id` cells were ignored, null `overview` cells were filled in with a "dummy" string informing the user that this movie's overview was not available, and `runtime` empty cells were replaced with the median runtime across all movies. And thus, all empty cells were filled in.

An additional check to make sure all non-categorical columns had the correct data type was made before progressing onto the feature engineering phase.

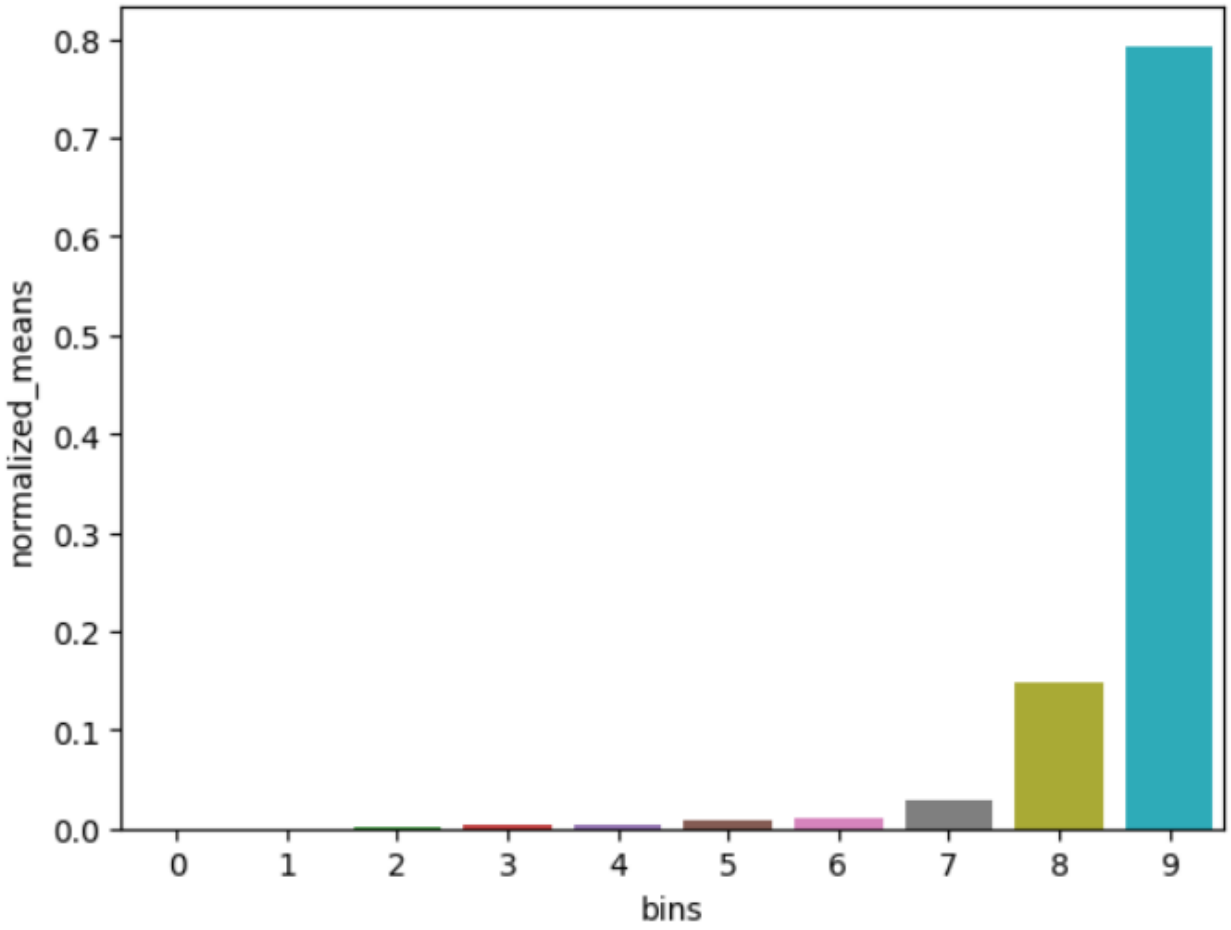
My next step was encoding the `genre` column – more specifically, using one-hot encoding. Each movie could have more than one genre, and in the new dataframe, each potential genre had a column filled with zeros assigned to it. If a movie belonged to a certain genre, that movie's specific genre column would be a one instead of a zero.

- My motivation for using binary encoding instead of label encoding where each genre was assigned a specific number (e.g. “Fantasy” = 1, “Adventure” = 2, and so on) was to prevent skewing results when calculating similarity, as there is no specific order to genres, and normalizing such a column might lead to information loss.

Next, I added an extra column to capture how well a movie did revenue-wise relative to the movie’s production budget. For this, I added an extra column named `rb_ratio` which was calculated from dividing the `revenue` column by the `budget` column. Should this ratio be larger than 1, the movie had made a profit, and was most likely of relatively higher quality. If this ratio was less than 1, for whatever reason, the movie had lost money.

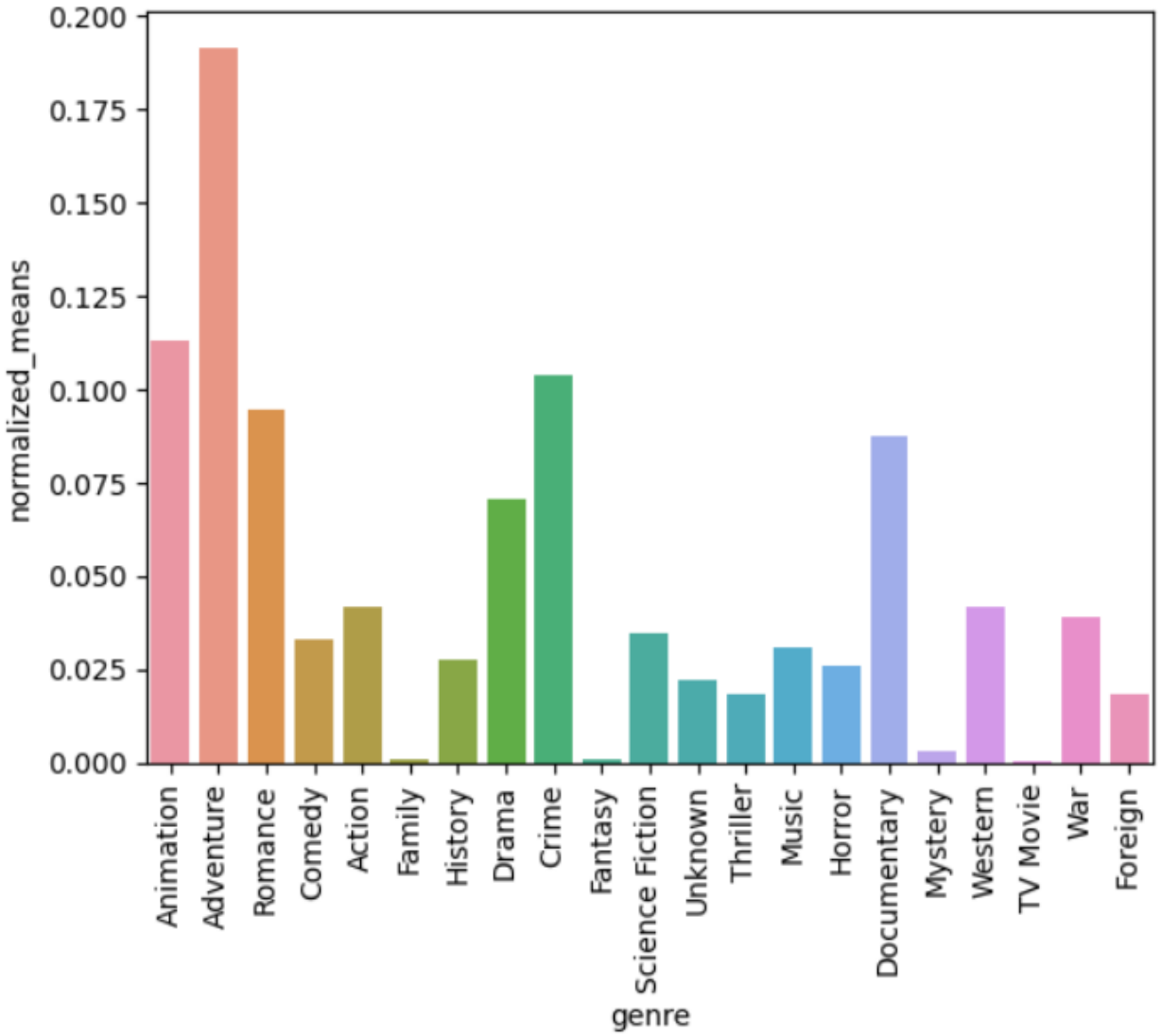
However, there was an issue here pertaining to the values of `revenue` and `budget`, as many movies had the values of either or both of these columns set to 0 (instead of being recorded as a `NaN` value). For this, I devised a quick solution for augmenting `revenue` and `budget` data.

- `revenue` data was estimated from the `popularity` column. As `popularity` contained continuous values, Pandas’s qcut method was used to bin the value of popularity based on *quantiles* and for movies which had a revenue of less than 100 (something considered to be quite impossible), had their revenues reset to the average revenue for their popularity bin.
  The decision was made based on the fact that there is a clear correlation between popularity and revenue;



Where the x-axis is each quantile-based popularity bin, and the y-axis is the relative average revenue compared to other bins.

- Similarly, `budget` was estimated based on genre, as movies with different genres most likely require different budgets. This too, was illustrated using a simple barplot:



With adventure movies having the highest average budget by a large margin. And thus, movie budgets with a value less than 100 were set to the median value of their genre's budget.

The resulting DataFrame from the operations described above were saved in `items.csv`.

Users DataFrame

For this DataFrame, the *massive* `ratings.csv` file was used. This file had the following columns in the following format;

	userId	movieId	rating	timestamp
0	1	110	1.0	1425941529
1	1	147	4.5	1425942435
2	1	858	5.0	1425941523
3	1	1221	5.0	1425941546
4	1	1246	5.0	1425941556

With one row per each review made by a single user at a specific time.

I instead constructed a new DataFrame, with each row for each user, and a column for every movie. Each cell included a user's rating of that movie if they had reviewed that specific movie, otherwise that cell was empty or filled with 0.

While in theory I could've taken all the users into account, the DataFrame would have gotten far too large, so only the first 500 users were taken into account.

The resulting DataFrame from the operations described above were saved in `users.csv`.

- One thing to note here is **scaling**, especially in the items DataFrame. Later on I'll be calculating the similarity between each movie and a vector that represents a user's preferences, if different features are on different scales, scaling is vital to not unintentionally give extra importance to any feature in particular.

Implementation

Model

The core code implementation of the model was implemented in `recommender.py`.

This file consisted of three 'main' classes:

1. `ItemData`: This class was in charge of loading the processed DataFrames pertaining to *item* data (i.e. `items.csv` or `items_*.csv`). This class was also in charge of scaling columns (the details of which mentioned in the previous section), and also in charge of basic operations on the items data, such as finding a specific movie title based on its row's index in the DataFrame, or obtaining a random movie id.
2. `UserData`: Similar to `ItemData`, this class is in charge of loading the processed DataFrames pertaining to *user* data (i.e. `users.csv` or `users_*.csv`) and any operations that might need to be done on user data, such as fetching a specific user.
3. `Recommender`: The main recommender class, pertaining to the recommendation model(s). The entire model was wrapped up into a single class, with instances of the `ItemData` and `UserData` class being attributes of the *main* `Recommender` class.

- These two objects are initialized in the `Recommender` object's constructor, along with DataFrames built with the sole purpose of keeping track of a single user's opinions on different movies, such as:
    - `item_picks`: a Pandas DataFrame consisting of attributes of movies
    - `seen_movies`; Movies that the user has already given an opinion on
    - `preferences`: A simple 1-row Pandas DataFrame where for each movie, there is a column, and the value in that column is the rating the user had given to that movie, if they had given any at all. Otherwise, the value of that column is 0.

The algorithm used was *nearest neighbors*. For this assignment, I had used `scikit-learn`'s implementation (documentation).
The metric used is simple euclidean distance. There was no need to implement this manually; when initializing scikit-learn's `NearestNeighbor` object, I only needed to set the parameter `p` to 2.

Two `NearestNeighbor` objects were created: one for user-based similarity, and one for item-based similarity.

In general, at every step, the user is shown 3 different recommendations:

1. *User-based similarity recommendation*
   For this, based on the user's opinions so far, the model will look through `users.df` to find the user that is most similar to the current user. Then, the movie that has not been chosen *yet* with the highest score in the similar user's row will be chosen for this specific recommendation category

2. *Item-based similarity recommendation*
   For this, based on the user's selected movies so far, the model will calculate an *average* of every quantitative attribute in the `items.csv` file (such as genre one-hot encodings, the revenue-to-budget ratio, and so on) and then use that average to find the most similar movies in the items DataFrame. The movie with the least distance that has not been chosen yet will be chosen for this specific recommendation category.

3. *A joint user and item-based similarity*
   For this, to overcome any weaknesses either of the two previous recommenders might have had, a fusion method is implemented.
   For this, *n* similar users are found such as with to user-based similarity recommendation, and *m* similar movies are found. Each of these *n* users will "vote" on these movies using the ratings they have given these movies if they have seen them. Finally, the movie with the highest *average* vote is selected to be recommended next.

After the user logs their opinion/score on a specific movie, the `Recommender` object will update its internal DataFrames, and then obtain 3 new recommendations based on each recommendation method described above.

The Cold-Start Issue

A well known issue in the field of recommender systems, the cold start issue pertains to difficulties calculating relevant user information when little to no information is to be had about a specific user.
For this I used a random strategy; as I did not want to get caught in a “popularity” local maxima where only popular movies were recommended to a user. In other words, for my first few predictions (or at least, until enough data is obtained), the predictions are completely random.

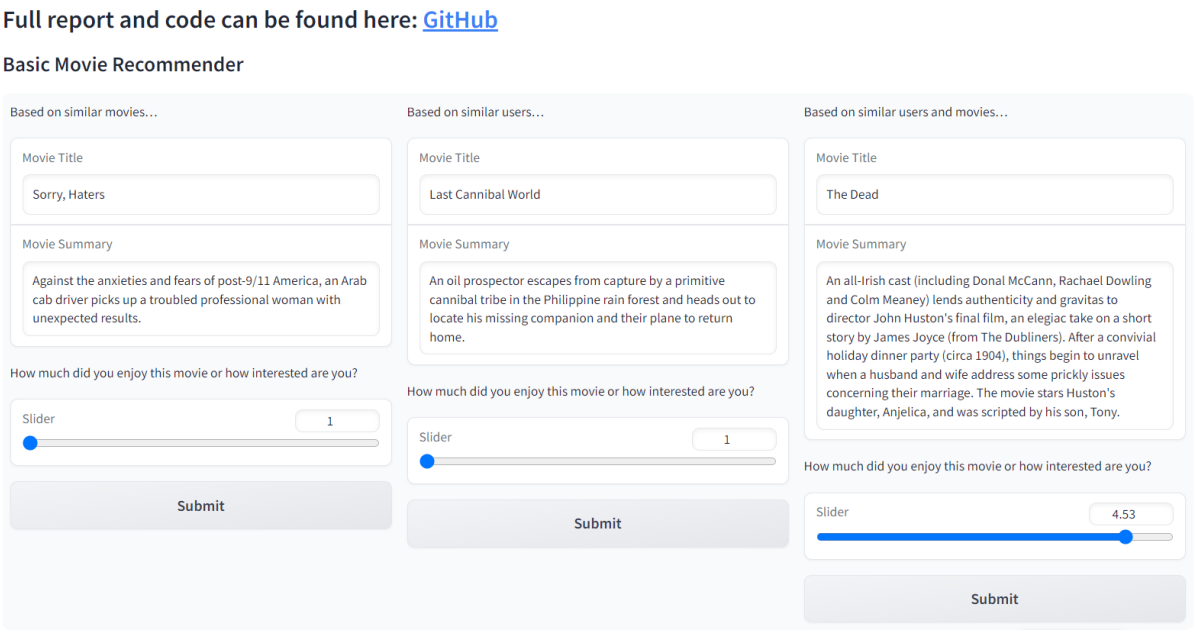
- User-based predictions are completely random until the main user records their opinion for at least one movie
- Item-based predictions are completely random until the main user records their opinion for at least one movie they *like*. (I.e. rating above 2.5)

UI

The UI was built using Gradio, as it was a recommended framework in the assignment details. Gradio is a Python package made for building demos and simple UIs incredibly quickly for the purpose of showcasing a machine learning model.

The layout is as follows:

- 3 columns for 3 suggestions given by each algorithm.
- Each column consists of a movie title, a movie description, and a slider where the user can input their opinion on a specific movie from a scale of 1 to 5.
- At the end of each column was a submit button, where upon clicking, the user would submit their opinion *on that single movie* to the recommender model.
- The recommender model would then update the values of the movie title and description according to its new predictions.



Gradio frontend

Next was connecting the layout to the recommender model implemented in `recommender.py`. As aforementioned, the entire model was wrapped in a `Recommender` class making it easy to use. Connecting this object to the layout was incredibly simple:

- I initialized my `Recommender` object at the beginning of the file, and filled in each column with the cold-start initial predictions.
- I then attached each submit button to a `submit_opinion` function, which takes two parameters: which submit button was pressed (i.e. which prediction was the one the user had an opinion on) and the user's opinion on that specific movie in the form of a rating.
- This information is then sent to the recommender object, and it's internal DataFrames are update; namely the rating for the user DataFrame for finding similarities between this user and pre-existing users, and if the rating was more than 2.5 (i.e. the user had *liked* that movie, or at least held a somewhat positive opinion), that movie's attributes was then added to be taken into account in future item-based predictions.
    - The details of this were explained in the previous sub-section.
- Finally, the recommender object returns the movie titles and overviews for the 3 predicted models, and uses those strings to fill in the movie title and overview textboxes.

Deployment

Hugging Face makes deployment incredibly easy with Gradio. All that was needed was to create a space, and upload the local files onto the Git origin for that specific space, and the application was automatically deployed. However, in order for everything to run smoothly, a couple adjustments needed to have been made:

- I needed to rename the main application file where the UI was implemented from `main.py` to `app.py`.
- I needed to provide a `requirements.txt` file to resolve dependencies that weren't pre-installed (namely `scikit-learn`)
- HuggingFace did not have support for uploading large files (more specifically, larger than 10 MB), so I needed to split the `users.csv` and `items.csv` files into several smaller csv files, and then reconstruct the original dataframe in the deployed code.
    - The details of this can be found in `DF_Construction.ipynb`.

Future Works

Obviously, the model proposed and implemented in this assignment is incredibly rudimentary, and can be greatly improved upon. Some suggested improvements in the future could be:

- Categorizing different movie features (i.e. genres, popularity, …) and assigning different weights that could be taken into account when calculating similarity
- Using deep learning models instead of clustering models.