UNIVERSITY OF ATHENS
DEPARTMENT OF INFORMATICS AND TELECOMMUNICATIONS

Movie Recommendation System

Student name: *Panagiotis Zazos*
sdi: 7115112300009

Course: *Big Data (M111)*
Semester: *Fall Semester 2023*

Contents

1. Abstract

The task at hand is to create a movie recommendation system, utilizing the public dataset provided by MovieLens. Similar systems, but way more complex, are used in Netflix, Hulu, Apple TV, Youtube, etc.
Several recommendation algorithms have been implementing, to adapt to user preferences. These include: user-user and item-item collaborative filtering, tag-based and content-based (using movie titles), as well as a hybrid method which combines the strengths of user-user, item-item and tag-based algorithms.
The project highlights as well the usage of four, oftenly used in recommendation systems, similarity metric algorithms to compute the similarity between users or items: Jaccard, Dice, Cosine and Pearson.

2. Data processing and management

Sparse Matrices (CSR & LIL)

- **lil_matrix:** This format is highly efficient for constructing sparse matrices incrementally (when chunking the dataset). As MovieLens (ml-latest) offers such a large dataset including movie ratings, using a dense matrix format would be incredibly impractical due to memory requirements. Therefore, the use of lil_matrix was crucial for building the matrix of ratings in chunks, getting away with memory constraints.

- **csr_matrix:** Since the matrices are very sparse, meaning that most of the entries are zeros (most users have not rated most movies), storing all those zero values explicitly in a dense matrix (pandas dataframes) consume a lot of memory.
  Once the matrix is built, converting it to csr_matrix format optimizes memory usage further, as this particularly format stores non-zero elements along with their row and column indices, dramatically reducing memory.
  As an example, the output of a csr_matrix looks like this:
  (0, 1) 5.0 - user 0, movie 1, rating 5.0

Overview of efficiency and scalability: csr_matrix is optimized for fast arithmetic operations, dot-products and slicing. The dot-products were being computed using numpy's efficient operations on csr matrices.
The combination of lil_matrix and csr_matrix for can handle the increase in data volume without an increase in computational resources.

When working on the 'ml-latest-small', differences between sparse and dense matrices aren't that important. Modern computers can hold onto dense matrices of that size. When switching into the huge 'ml-latest' dataset, though, things are completely different. Without the use of lil_matrix, the conversion to a csr_matrix becomes impossible due to memory issues. In most cases, the process of the script was terminated!

Movies mappings: The movies mapping function plays a vital role in linking movie IDs to their titles. One may observe that movie ids is the movies.csv are not continuous: a movie's id and index in the dataset won't be the same. Therefore, the need for

a proper indexing of these movies was raised. This change is crucial in understanding the nature of 'input_id' that is frequently used in the code, as in most cases, this id translates to the index of the movies mapping. Since the sparse matrices are being constructed (in shape) and populated with this indexing in mind, there will be no discrepancies when trying to print out the results of the most similar movies.

3. Similarity Metrics

Overview
In the movie recommendation system, similarity metrics functions are necessary in providing the likeness between different entities (users of items). These are also very commonly used in both collaborative and content-based filtering algorithms.
Notable distinctions: In similarity_metrics.py one can find two sets of similarity functions:

- The user-specific similarity functions, which take two parameters: the sparse matrix and the given input user. The output of such similarity functions is a score between the user and all the other users. This approach aims to get an overview on the likeness of one user with all the other distinct users in order to identify the top_k most similar users to the given user, based on the movies each user has rated.
- The items-specific similarity functions, which take three parameters: the sparse matrix, the unrated movie and the rated movies. These similarity functions were mandatory to differentiate from the user-specific ones, as the need was raised for similarity between an unrated movie a user hasn't rated with every rated movie that user has rated.

Sparse matrices: These similarity algorithms have been adjusted to work with sparse matrices. Two helper function were created to help compute the cosine similarity between rows and the **subtract_row_means** which was designed to calculate and subtract the mean without converting the sparse matrix into a dense format (this would have been computationally and memory intensive)

Some Insights on similarity metrics

- User-user Collaborative Filtering: Pearson Similarity
  Pearson similarity is proficient in capturing the linear relationship between user's ratings. It adjusts for each user's rating scale by mean-centering, which is crucial in user-user collaborative filtering, where different users may have different rating scores for different movies. But, Pearson correlation, as it is also called, has its limitations as well: its effectiveness diminishes in scenarios with sparse data or when users have a limited number of commonly rated items. Sparsity has been tweaked with the usage of csr matrices, but the second part of its limitations still exist.
- Item-item Collaborative Filtering: Cosine Similarity
  Cosine similarity is effective in comparing the orientation but not the magnitude

of vectors, making it suitable for item-item collaborative filtering where the focus is on the pattern of ratings over their values. Its more about commonality rather than scoring of ratings. As Pearson, so does cosine similarity deal with high-dimensional data, where each dimension can correspond to a different user.

- Title-Based Filtering: Cosine Similarity
  In content-based filtering, especially with text data, as the movies titles, cosine similarity excels in measuring the similarity of TF-IDF vectors. It effectively captures content similarities, regardless of document size differences (length in the case of movies titles). Cosine similarity computes the angle between two vectors (similarity in title content), therefore the selection of this metric is almost undisputed.

- Tag-Based Filtering: Jaccard or Dice Similarity
  Both Jaccard and Dice similarities are suitable for binary attributes like tags. They are good at measuring the similarity based on the presence or absence of tags, which in the case of the given dataset is binary. These metrics, by default, don't consider the frequency of tags, but knowing this, proper adjustments have been implemented to consider the frequencies as well.

4. Algorithms and Helpers

4.1. User-user collaborative filtering

The initial phase of the recommendation algorithm, following the construction of the sparse matrix from the ratings.csv file, involves calculating the similarity between movies rated by a specific user (input_id) and those rated by all other users.

Following this, the system identifies the top 'k' users who share the highest similarity scores with the target user.
Similarity matrices are represented as a list of tuples, where each tuples consists of a user id (key) and the corresponding similarity score (value).
A crucial detail in this operation, as previously mentioned concerning indexing discrepancies, is the adjustment of the input_id by subtracting 1. This adjustment accounts for Python's zero-based indexing, ensuring alignment with the matrix's indexing.

In the final stage, the algorithm leverages the identified similar users to predict ratings for movies that the target user has not yet seen. This process returns the top N movies, prioritized by their highest predictive scores.

4.2. Item-item collaborative filtering

Following the successful segmentation of the sparse matrix into chunks for user data, the ratings matrix adopts a user-item format. In this stricture, rows correspond to users, columns to movies and entries represent the ratings assigned by users to movies.

For the item-item collaborative filtering process, a pivotal transformation is applied to the matrix (transpose). This transposition is a critical step, as it shifts the matrix's focus from user similarities to movie similarities. Rows now represent movies, and columns represent users.

Upon transposing the matrix, the algorithm compares each unrated movie by the target user (input_id) with all the movies that the user has rated. In this specific collaborative filtering method, similarity metrics that take three parameters into account are being utilized: the sparse matrix, the unrated movie, and the rated movies.

The outcome of this process is a similarity matrix, generated from the applied similarity metrics. This matrix includes a list of tuples, each signifying the similarity of an unrated movie with all the movies rated by the user. The similarity scores are being computed using only the top 'k' most similar movies.

The final step involves aggregating the scores for all unrated movies, sorting them, and ultimately returning the top N highest-rated recommendations.

4.3. Tag-based recommendation algorithm

The tag based recommendation algorithm leverages the tags associated with movies to generate recommendation. Not every user has tagged a movie and not every movie has been tagged by a user. This is an important observation!
This algorithm works under two assumptions:

- input_id is a movie that is located in the dataset.
- input_id is a movie that has been tagged, otherwise the algorithm cannot recommend to a movie that hasn't been categorized with any actual tag.

In both cases, the system terminates with a proper message.
The sparse matrix is being built from a matrix which has dimensions corresponding to the number of movies and number of unique tags. Populating this matrix includes iterating over tags.csv and for each tag associated with a move, the count is being incremented.
Essentially, this matrix represents a frequency matrix, where [i][j] represent the count of tag j associated with movie i.

lil_matrix usage: Without the usage of lil_matrix, this matrix would have been dense and therefore impossible to be converted into a csr_matrix. It would cause heavy memory issues, to the point of terminating the process.

From this point on, this matrix is being converted into a csr_matrix, a sparse matrix where each entry shows a non-zero value along with it coordinates. Example output:
(0, 439) 2.0, the movie with index 0 has been tagged with the tag corresponding to column 439 twice (by different users).
After that, the calculations are being made using the similarity metrics with two parameters (sparse matrix, movie index), which means the similarity of target movies tags with all the other tags associated with every other movie.

Finally, utilizing the helper function **similar_movies_mapping**, the extraction of N most similar movies is being made successfully.

Note about similar_movies_mapping: This helper function was built to help map the movies with actual useful indices. As stated in the beginning of this report, an observation has been highlighted which concerned the continuation of movie ids. Mapping these non-continuous ids is mandatory to avoid index discrepancies while manipulating the matrices.

4.4. Content-based (titles) recommendation algorithm

The importance of movies mapping has been stated in the previous section. Furthermore, the removal of the dates in titles before vectorizing them seemed important, as the text processing focuses solely on the title's textual content rather than the release year.

Content-based recommendation is consisted of one major component and that is the use of TF-IDF vectorization. This algorithm converts the textual data into a numerical form which will be used for similarity calculations.
TF-IDF: Term Frequency-Inverse Document Frequency is a technique used to convert text documents into numerical vectors which will be used for similarity calculations.

- **TF:** Term Frequency, which measures how often a word appears in a document, giving insight about the relevance within it.
- **IDF:** Inverse Document Frequency, which indicates how unique or rare a word is across the collection of documents.

Stopwords: Before computing the Term Frequency and the Inverse Document Frequency, the removal of stopwords takes place, to eliminate redundant computations between irrelevant words. This filtering is implemented using nltk's stopwords for the Engligh language.

Normalization: Generally, in many implementations of TF-IDF the vectors are by default L2-normalized, meaning each vector is divided by its Euclidean norm, resulting in vectors of unit length. This ensures that all TF-IDF scores won't exceed the value of 1 and are going to be between 0 and 1, where a score of 1 would indicate a high form of uniqueness in the dataset. By incorporating the filtering of stopwords a strange behavior appeared where a L2-norm resulted to 0, meaning that all elements in a vector were zero. That could happen if a movie title contained only stopwords which were removed or words not present in any other title. To prevent this, a really small constant was used to the L2-norms, to ensure that the denominator is never zero.

Improvements: It would not be necessarily bad to lemmatize the text as well, but i chose not to, as the dataset containing only the titles of the movies wouldn't be as complicated as a tweet for example, which would be more difficult to compare.

The result of the TF-IDF computation is a csr_matrix (sparse) which contains information about words in movies in the format:

(movie index, word containing in the movie) tfidf value
(0, 1640) 0.8405710076604556
(0, 5016) 0.5417013762957281
(1, 7084) 1.0

Lastly, the similarity function are being applied to the TF-IDF sparse matrix along with the index of the target movie (movie index), resulting in a matrix that designates how similar each movie title is to the input_id movie.
Finally, through the usage of the helper function **similar_movies_mapping**, the selection of the top 'N' similar movies, based on their titles, is being successfully made by mapping the movies indices to actual movie ids.

4.5. Hybrid recommendation algorithm

This hybrid implementation integrates three recommendation algorithms: user-user, item-item and tag-based.

Initially, given input_id, which is **strictly a user_id**, the top N movie recommendations for user-user and item-item collaborative filtering are being computed separately. Afterwards, a choice has been made to merge these recommendations into a list and apply predefined weights to balance their influence. Movies that have been gathered by item-item collaborative filtering are more important, than of those of users. This decision has been made by the developer (myself) under the personal preference of a recommendation system which values the recommendations based on movies similarities higher than of the recommendation of movies between similar users. Merging these two lists and applying the weights would result in a **sorted** 2*N sized matrix.

If given user hasn't rated any movie at all, the algorithm returns the combined top N recommendations based on user-user and item-item collaborative filtering.
But if the user has tagged movies, the algorithm determines the relevance of each recommended movie based on these top tags, refining the combined recommendations and choosing only top N most relevant movies, based on tags.

5. Real system

Preprocessing Stage: This preprocessing stage is a one-time computational process that prepares the recommendation data in advance.

IMPORTANT: The preprocess.py script has been executed using ml-latest-small dataset, primarily due to concerns regarding execution time. Furthermore, to manage computational complexity, a subset limit of 25 has been established. This means that the preprocessing will output files containing information about only the first 25 users and movies.

The output from the preprocessing script is organized into specific directories for efficient storage. Each directory, named after a specific algorithm, houses four csv files, corresponding to each of the similarity metric algorithms employed. There are

two distinct types of outputs, differentiated by the nature of the input (either a user or a movie id):

- **user-user, item-item:** The format is [userId, recommendation_list]
- **tag, title:** The format is[movieId, title, recommendation_list]

During the preprocessing stage, the recommendation list is capped with a default maximum of 250 entries. This limit is based on the pragmatic assumption that rarely does a request for recommendations exceed this number, as an excessively long list of suggestions may diminish in reliability beyond a certain point. Consequently, it is crucial to ensure that the input for N remains strictly below 250.

Recommendation Retrieval: Post-preprocessing, recommender.py serves as the tool for swift and efficient retrieval of recommendations.
-d directory: Prior to the implementation of the real-system, the 'directory' input was utilized to specify the dataset source for reading. By inputting the term **'real_system'**, the recommender is configured to switch to reading from the directories generated during the preprocessing stage. Therefore, users have the flexibility to either run the recommender in a conventional manner, specifying the desired algorithm and metric on either ml-latest or ml-latest-small, or to query the preprocessed output files.

6. Results and Analysis

Revisiting the title-based implementation strategy Upon careful reflection, an alternative approach to the title-based recommendation algorithm was considered. Initially, the design focused exclusively on leveraging movie titles for content-based filtering. However, the potential benefits of incorporating genres, as provided in the 'movies.csv' dataset, alongside titles could not be ignored. This integration could potentially enhance the recommendation accuracy by adding another dimension of similarity based on genre.

Dataset selection and processing efficiency The choice of dataset significantly impacted the system's performance. In depth analysis revealed substantial differences in processing times between the larger ml-latest dataset and its smaller counterpart, ml-latest-small. Specifically executing algorithms like the user-user and item-item algorithms (item-item is way more computationally heavy than user-user), which are the most computationally heavy, on a single user incurred considerably processing time, ranging up to 2 plus hours for the larger dataset and approximately 30-50 minutes, varying depending on the similarity metric, for the smaller one.
Given these findings, the decision to utilize the ml-latest-small dataset was both pragmatic and necessary to maintain operational efficiency. It it important to note that this choice inherently influenced the scope of the recommendations. As such, the output files and results are exclusively derived from the ml-latest-small dataset.

Output file configuration and consistency As mentioned above, the system's configuration was tailored to read from the ml-latest-small/movies.csv file for retrieving and displaying movie titles. This decision was critical to ensure consistency in the recommendations provided. This implies that all movie ids referenced in the output files

correspond to the ml-latest-small dataset and not it's larger counterpart. The larger dataset does not include every movie present in the smaller dataset.

Progress indicators Results generated by any algorithm are displayed directly in the terminal. Progress messages printed in the terminal have been particularly instrumental when processing extensive datasets, providing insights to the execution milestones and overall progress, mimicking 'loading bars'. The system includes optional console outputs such as memory usage statistics and processing stages. For instance the print_memory_usage() function outputs current memory utilization, offering a view into the system's resource consumption during execution.