ΤΜΗΜΑ ΠΛΗΡΟΦΟΡΙΚΗΣ & ΤΗΛΕΠΙΚΟΙΝΩΝΙΩΝ

ΕΛΛΗΝΙΚΗ ΔΗΜΟΚΡΑΤΙΑ
Εθνικόν και Καποδιστριακόν
Πανεπιστήμιον Αθηνών
ΙΔΡΥΘΕΝ ΤΟ 1837

Big Data Management assignment report

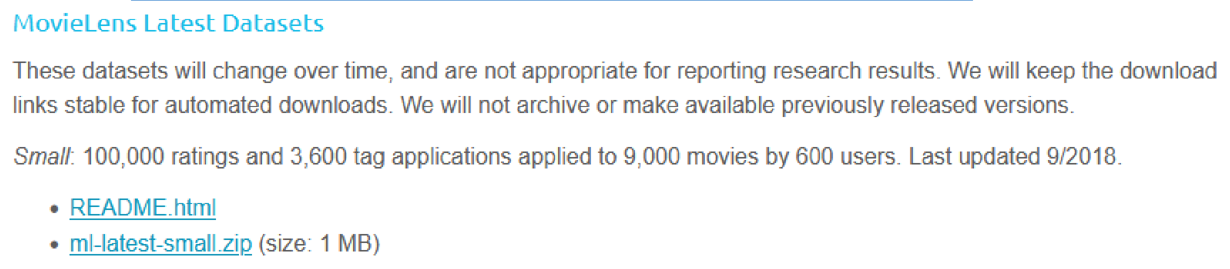
Fall semester 2023-2024

Dimitrios Tsiompikas

AM: 7115112300036

Part 1: Data loading

First of all, I have used the full dataset initially to test my algorithms. I also used Python as the programming language and Pandas to process the data. I ran the user-to-user algorithm for one user with the full dataset and it took 50 minutes to finish the calculations and provide me with the recommended movies.
For that reason, I used the ml-latest-small dataset instead of the one provided in the assignment pdf, because it is essentially the same dataset but the data is already structured in csv files.
Link -> https://grouplens.org/datasets/movielens/


MovieLens Latest Datasets

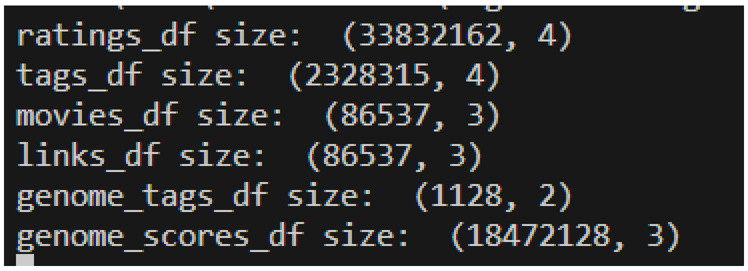
These datasets will change over time, and are not appropriate for reporting research results. We will keep the download links stable for automated downloads. We will not archive or make available previously released versions.

Small: 100,000 ratings and 3,600 tag applications applied to 9,000 movies by 600 users. Last updated 9/2018.

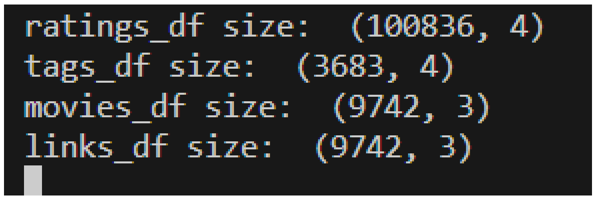
- README.html
- ml-latest-small.zip (size: 1 MB)

This allows my code to be ran both for the full and the small datasets. Below, you will find the sizes of each dataset.

- Full dataset size:


```
ratings_df size:  (33832162, 4)
tags_df size:  (2328315, 4)
movies_df size:  (86537, 3)
links_df size:  (86537, 3)
genome_tags_df size:  (1128, 2)
genome_scores_df size:  (18472128, 3)
```

- Small dataset size:



```
ratings_df size:  (100836, 4)
tags_df size:  (3683, 4)
movies_df size:  (9742, 3)
links_df size:  (9742, 3)
```

Each df size corresponds to the respective csv file. (First part of parenthesis is row count and second is column count). Genome scores and genome tags do not exist in the small dataset. There are 33 million ratings in the full dataset and 100k ratings in the small dataset. The small one also consists of 3683 tags, 9724 movies and 610 users.

Part 2: Recommendation system

All similarity metrics have been implemented by hand and I used them for all my algorithms. The system (for the initial part which runs with the recommender_initial.py) works as follows: the user chooses directory of datasets, algorithm, similarity metric and number of recommended movies , the algorithm returns a dictionary , which then gets sorted in descending order on the value (score) and then I get the top n items , I get their keys (movie IDs) and print out the respective movies in the console.

User to User

For the user to user algorithm I am using the following steps:
I find all the movies of the wanted user (let’s call him user x) , I find all the movies of the users who have watched the same

movies as user x (let’s call them users y) , then I got the similarity scores of user x with users y. For the jaccard and dice scores, I put the movies that user x watched in a vector and for every other user the movies that they watched (including the same movies with x) to find the similarity scores since these two are binary algorithms. For cosine and pearson , I got the user ratings (of the movies that both users watched) as input for the similarity scores (for cosine I also normalized the ratings between 0 and 1 , pearson handles that on it’s own). I sorted the similarity scores in descending order , I got the 128 first of them (128 most similar users). Lastly , from those 128 users I calculated the recommendation score for the movies that user x hasn’t watched and returned them in a dictionary with movie ID as the key and the score as the value.

Item to Item

For the item to item the logic is similar but the values are a bit reversed. To be exact , I get the movies user x has watched , then I again get the users that have watched the same movies with x , then get their movies that user x hasn’t watched. Now, here is where the logic part comes in. Normally, item to item recommendation requires all similarities to be found (every movie with every other movie as a pair). I thought of getting similarity scores for all movies that x has watched with all other movies that x hasn’t watched from users y to save time and because I believe that these would also show the highest similarity scores from other movies.I also tried out mean-centering but it didn’t produce different results for me so I removed it. For Jaccard and Dice, instead of getting the movies as input for the vectors, I get the users that have rated the movies that I'm comparing. For cosine and pearson I get the ratings of the movies (I've filled NaN values with zero , meaning users who haven't watched the movie , have the value zero now) as input for the vectors. Lastly, as mentioned before, I sort the similarity scores descendingly , getting the 128 most similar movies and then I calculate the recommendation score with the formula given, returning the dictionary with movieId as key and recommendation score as value.

Tag based

Tag based algorithm was implemented as shown in the assignment pdf. Since the movie tags are only 3683 , for every movie that I'm searching I take its tags and compare them to every other movie's tags. As mentioned in the pdf, I take every movie's tag counts for cosine and pearson and the tags themselves for jaccard and dice in order to obtain the similarity scores. Finally, I return them in the classic form of a dictionary with movieId as the key and the similarity score as the value.

Content based (TF-IDF)

Similarly as above, since the movies are only 9724 I was able to find a relatively fast way to calculate the TF-IDF vectors for the movies that I was searching for (it took 1 minute and 20 seconds which is a lot for the real system part but it's still fast). For this part I took movie titles and genres as features (I also

used genres because some movies like Othello with movieID 26 , have a VERY specific title which would only correspond to a movie with the same title. The genre helps identify other movies like this one , dramas for this example, and gets their similarity scores properly). I start with some text preprocessing (I remove all parentheses with years and non-English language titles and remove all punctuations and special characters) , then I found the count of each token in the movie titles and genres. Afterwards, I find the TF-IDF vector for the wanted movie and for every other movie I find their TF-IDF vectors and compare them with the wanted movie to get their similarity scores.
For Jaccard and Dice , I take the tokens (title tokens and genre tokens) that have a count greater than 1 in the TF-IDF vectors and put them in the metrics , while in cosine and Pearson metrics I take the values of the TF-IDF vector and feed them to the similarity metrics. Lastly, I return the similarity scores in dictionary form with movie ID as the key and the similarity score as the value.

Hybrid

For the hybrid algorithm I took user to user, tag based and content based (TF-IDF) algorithms , I sort their scores and then apply weights to them based on importance. I multiplied the biggest weight to tag based algorithm because I believe it yields the best results for content based filtering, the second biggest to user to user (in order to get a good cooperation with collaborative filtering as well) and the smallest weight to TF-IDF (item to item wasn’t used for time saving). Then I return them in dictionary form with movie ID as the key and the score as the value.

Real System

For the real system I created two new files the recommender_preprocessed.py and the preprocess.py. The preprocess file computes all the scores for all metrics and all algorithms, places them in respective folders in the form of txt files in the text_files folder. I have created a folder for each algorithm namely: user_to_user, item_to_item, tag_based, content_based and hybrid. Formats are the following:

user_<userId>_<metric> (for collaborative filtering algorithms: user-to-user, item-to-item)

movie_<movieId>_<metric> (for tag based and TF-IDF)

score_<userId>_<movieId>_<metric> (for hybrid)

Then the recommender_preprocessed reads the movie IDs and scores from these text files, puts them in their respective dictionaries and then follows the same process as the initial system, sorting them in descending order and getting the first n recommended movies to show to the user (where n = number of recommendations). Finally, for this part, User-to-user and tag-based were the fastest algorithms , the first one finished in 2 hours and the tag-based finished in 3 hours for all users and tagged movies respectively , while for TF-IDF , item to item and hybrid I had to get a subsample as it would take days for them to finish with all movies/ users (especially hybrid). If I used the full dataset on this it would have taken weeks to finish and in order to solve this issue I would probably have to use pre-made algorithms and similarity metrics from known python frameworks like scikit-learn , TensorFlow etc, I might have had better performance and therefore much faster results which would only take a few hours to complete the text files for every recommendation algorithm.

Web Interface

For this part, I created a GUI using TKinter which is a pre-built Python package (like math for example) which provides such functionality. This is executed from the recommender_gui.py file. The GUI has two dropdown menus , one for similarity metric choice and one for algorithm choice , an input field which takes a user ID , a movie ID or a user AND a movie ID (depending on algorithm) and then returns the top 100 recommended movies by fetching them from the text files in a new window each time the “Search” button is pressed.