M111 - Big Data - Fall 2023 – Project

Recommender System

Table of Contents

Introduction

All recommender scenarios & the bonus have been implemented. The recommender app produces a result in a matter of seconds while using the ml-latest dataset with the exception of item-item algorithm where a smaller dataset must be used to have the program finish in a logical time range (some seconds). For this reason, an additional optional flag **“-r maxRecords”** can be used to limit the dataset that recommender will work on each time. Detailed execution instructions for the binaries can be found in README.txt.

Assignment reporting

1. Data loading

After downloading and unzipping the data, I utilized the 'wc' command to count the number of rows in each CSV file, including the header row. For instance, executing "**wc -l ../ml-latest/ratings.csv**" resulted in 33,832,163 rows in the ratings CSV. Similarly, the genome-scores file contained 18,472,129 rows, the genome-tags had 1,129 rows, links and movies CSVs contained 86,538 rows, and tags consisted of 2,328,316 rows.

This recommender application utilizes **only** the files: ratings.csv, movies.csv and tags.csv.

2. Recommender system

Similarity metrics are implemented inside **“algorithms”** folder. Also included in this folder are the corresponding modules which calculate the **union** and **intersection** of 2 sets (supporting sets of ints, floats and strings) and a module to calculate **IDF** for a map of documents (in our case map of movie titles) and a module to calculate **TF** for a given document (a movie title).

Key data structures in the **"models"** directory include:

- **model.User**: Includes a map of {movieID:rating} pairs.
- **model.MovieTitle**: Contains the title string of a movie.
- **model.Movie**: Includes a map of {userID:rating} pairs.
- **model.Tag**: Consists of a map of {userID:model.userTags} pairs. Each userTag in this map is a list of tags made by the corresponding userID for a movie.

These components are organized in the main file **"recommender.go"** using a struct named **Data**. This struct holds individual maps, each with an ID as a key and the corresponding model as its value. For User objects, the ID signifies a userID, while for other maps, such as movies and titles, the key represents a movieID. This arrangement links a userID to the submitted ratings and associates a movieID with its title, received ratings and received tags respectively.

A common strategy in all collaborative filtering implementations below is the use of parallelism - more specifically, Go routines - to make the calculations faster and more efficient. The number of routines spawned can be tweaked with the **cfg.NumThreads** variable in the **config/config.go** file. The general idea is that when working with objects that are identified by a userID or a movieID, it is safe to assume these objects are distinct. Hence, before finding similar users for a user, the userIDs are split into chunks equal to the cfg.NumThreads variable and each routine works on a chunk of IDs. The same happens with movieIDs when working with movie titles, movie ratings and movie tags. In addition, each routine keeps its data in a local slice of results and thus there is no fear of race conditions or deadlocks. To avoid writing the results of similar users to the mutex-protected shared slice too often, the merge of local slices happens only when the routines are done working on the chunk. The final results are always sorted by similarity score using either a **model.SimilarUser** or **model.SimilarMovie** object depending on the algorithm used.

Finally, the **k** factor is also defined in the configuration object as Config.K = 128

Note: The following example executions of recommender take for granted that the preprocessed files have been generated in **"preprocessed-data"** directory using the preprocess binary as described in README.txt. The app doesn't work with the initial CSV dataset since the real system implementation as described in section 2.f.

2.a. User-User collaborative filtering

File: recommenders/user-user.go

Example: **go run recommender -d preprocessed-data -n 100 -s jaccard -a user -i 6752** (maxRecords - if specified - limit users, not ratings).

For this part, the original rating records are organized in **model.User** objects. To find similar users for a certain user, the IDs of each movie the user has rated are organized in a set. For Jaccard and Dice metrics these sets are used as input directly to decide on the similarity between the selected user and any other user. For Cosine and Pearson, however, these sets are processed to construct a pair of same-sized vectors which can be passed as input that represents the two users of interest. These vectors contain ratings instead of movieIDs to make the results more accurate. When creating the vectors for the selected user and any other user, each rating value refers to the same movieID indirectly. **Utils.GetMovieRatingVectors** function does that while ignoring most of the unnecessary values. This means that only ratings referring to movies the selected user has rated are kept. This is done, because if none of the 2 users have rated a movie, or even if the other user has rated some movies the selected user didn't, storing these ratings wouldn't make any difference, since during multiplication of the vectors, at least one of the 2 values would be 0 and thus have no impact to the result.

To make the computations faster and use less resources, each routine keeps only the top **k** most similar users before merging the results. In the end, the final slice contains only **cfg.NumThreads * k** similar users. Those are again sorted and trimmed to length **k**. This mechanism ensures that *the **k** most similar users to the user of interest are present in the final recommendations* even if they all happen to come from the results of a single routine. Most users are ignored, but since they are not very similar to the selected one, the impact on the recommendations is not big. With this slice of similar users, the recommendation part takes place, where every movie gets a forecasted rating based on weighted average of the opinions of these **k** similar users for this specific movie. Of course - having only 128 users to decide on every movie that exists - leads to many movies not being recommended, because none of the 128 users have rated them. Increasing k will improve accuracy and the number of recommendations at the cost of extra execution time.

2.b. Item-Item collaborative filtering

File: recommenders/item-item.go

Example: **go run recommender -d preprocessed-data -n 100 -s dice -a item -i 856 -r 5000** (maxRecords - if specified - limits movies, for example, the first 5k movies include 20,668,411 rating records).

The ratings are now organized in a map of **model.Movie** objects which relate a movie with all the ratings it has received per user. This method Is more accurate than the previous, but also a lot more compute-intensive due to the fact that it will be applied on decades of movies for a specific user on average. However, since movies are significantly less than users, there is no real benefit in trimming the local slices of similar movies per routine as in user-user approach, thus only the final slice is sorted by similarity and trimmed to length k.

Heading on to the actual calculation of movie-similarities, for the selected movie and each other movie, a set of userIDs is created, declaring the users that have rated them. Again, for Jaccard and Dice metrics these sets can be used directly as input. For Cosine and Pearson, however, **util.GetUserRatingVectors** function creates two same-sized vectors of ratings in a similar way to the user-user scenario. Each pair of selectedMovieVector[i] rating and otherMovieVector[i] rating refer to the **same userID** indirectly for any index i. Once again, only users that rated the selected movie are taken into consideration, aiming to drastically reduce the size of vectors that would be necessary to store a rating from all existing users for a specific movie. Hence, the absence of the rest userIDs implies that a 0 would be stored in the both vectors as rating from this user for each movie and cause no effect in the final multiplication of the vectors.

Proceeding to the recommendation part, some assumptions are made to further reduce the number of calculations needed to suggest movies to a specific user. Finding similar movies for each movie the user has rated is inefficient, so only movies that user really enjoyed (**rating >=4**) are used. This limits the number of possible recommendations, but ignores only movies that would be related to a movie the user didn't like. Hence, a map of **{ratedMovieID:similarMovies[]}** pairs is used to store the *k-most similar movies to each ratedMovieID that the user liked*. Another map called **recommendableMovies** keeps track of every movie *present in at least one of these similarMovies slices*. Each individual **similarMovies** slice has no duplicates, but there is a possibility that a movie is quite similar to more than one movie the user has watched. In any case, this map is used because in the worst-case scenario it includes **(number of movies the user has rated) * k** movies. This is significantly smaller than the total movies, especially when considering movies that are similar to more than one rated movies are only stored once as stated above. Eventually, this map of recommendable movies is used to calculate the weighted average of the user's ratings to all the rated movies that are related to any recommendable movie.

Overall, this approach is strict, discarding a lot of movieIDs in each step, but making sure the most closely related movies to the ones the user has rated are not dismissed. It helps boost the execution time at the cost of not being able to recommend as many movies as the other algorithms.

2.c. Tag-based collaborative filtering

File: recommenders/tag-based.go

Example: **go run recommender -d preprocessed-data -n 100 -s cosine -a tag -i 1** (maxRecords - if specified - limits tags).

In this case, once again routines work on chunks of unique movieIDs like the item-item approach. For Jaccard and Dice algorithms, the **unique** tag strings of each movie are organized into slices of strings and used as input sets for these similarity metrics. For the other two metrics, **util.GetTagOccurrenceVectors** function prepares a vector for the selected movieID and any other movieID based on their tag sets. Both vectors contain tag occurrences, which - for a given index in the vectors - refer to the same tag string. Tags that are not related to the selected movie are dismissed as they would make no difference in the vector multiplication.

Note: When loading the tag strings from the CSV (in **"utils/loadCSVData.go"**), **helpers.ExtractTokensFromStr** is used to convert the tags in a way that are case insensitive and free from characters like commas, parentheses etc. This aims to make the comparison of tags smarter by limiting the number of truly unique tags.

2.d. Title-based collaborative filtering

File: recommenders/title-based.go

Example: **go run recommender -d preprocessed-data -n 100 -s pearson -a title -i 45722** (maxRecords - if specified - limits titles).

When utilizing titles to recommend movies to a specific movieID, first the IDF map of all movie titles existing is calculated. This map contains an IDF value for each token (word) in every movie title. Then, for the selected movie the TF map is created. It is a map that associates a TF value to any token (word) in the title. A similar TF map is created for each other movie title that will be compared to the selected one. This is achieved by combining the TF maps of the two movies of interest and the IDF map using **util.GetTfIdfVectors** function. This function generates 2 TF.IDF vectors of the same size. The vectors contain only TF.IDF values that refer to *a token in IDF map which is present in TF map of the selected movie*. If the other movie has words that don't exist in the selected movie title, these will be ignored. This significantly reduces the size of the vectors, keeping only the words that matter for each pair of titles. Of course, any index in the two vectors contains a TF.IDF value that refers to the exact same word in both. The above methodology is used only in the cases of Cosine and Pearson similarity metrics.

Since the implementation of Union and Intersection can work with sets of strings Jaccard and Dice take as input the set of tokens of the two movie titles and calculate their similarity based on that.

Note: For the creation of both TF and IDF maps, **helpers.ExtractTokensFromStr** is applied to the titles prior to calculating any TF or IDF score for the words in them. This is done to remove irrelevant characters and make titles case-insensitive, which leads to a more accurate representation of unique words in all titles.

2.e. Hybrid collaborative filtering

File: recommenders/hybrid.go

Example: **go run recommender -d preprocessed-data -n 100 -s cosine -a hybrid -i 8903** (maxRecords - if specified - limits movies, not titles or tags, because item-item is the most demanding procedure).

In this case, the input is expected to be a **movieID**. What happens is that for this specific movieID, all three mechanisms of item-item, title-based and tag-based correlation filtering are applied. The final ranking of movies is decided by a *sort-merge-join* approach. More specifically, the slices of similar movies from each kind are not trimmed, but rather kept to the maximum length of movies each algorithm was able to recommend. First, tag-based correlation is applied. The whole list of similar movies that come out from this algorithm are used to create a subset of movie titles which only contain movies that are recommendable by tag-based method. The same happens with item-item approach, where instead of using the whole dataset of movies, only a subset that contains all movieIDs recommended by title-based method is considered. This strategy works because *only movies that are recommendable by all 3 methods* *are meant to be in the final recommendations*. By discarding the movies that are recommendable by the previous method in each step, the dataset that the next algorithm works on is significantly smaller, saving a lot of computation time, while ensuring that only movies that would be dismissed either way in the final calculations are missing. The three similar-movie slices are finally sorted by ID and a combined similarity score is stored for movies that exist in all three slices. The weights used for the combined similarity are: 20% from item-item similarity score, 40% from title-based and 40% from tag-based.

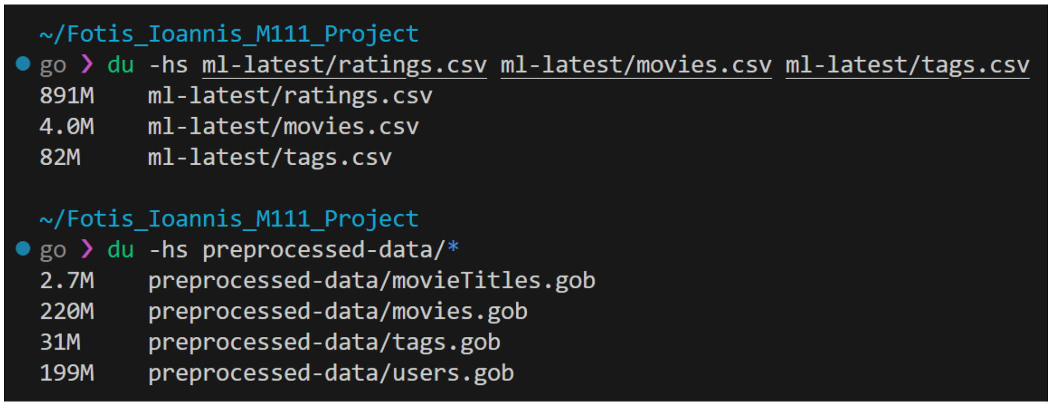
2.f. Real system

Up to this point, the data structs were populated using the **util.LoadCSVData** function in the homonym file. The only optimization was that each time the CLI program was executed, only the necessary structs for the requested algorithm were loaded to save time.

With the real system implementation, all the CSV data is loaded by executing the preprocess binary following the instructions in README.txt. So, user ratings, movie titles, movie ratings and movie tags are read and stored in the appropriate models as described in section 2. After that, the Go package **"encoding/gob"** is used to encode the data maps of objects into GOB format. This format is Go specific, is not human-readable and serializes values into bytes which are meant to be deserialized by a GOB decoder. For the needs of recommender app, each map of user ratings, movie titles, movie ratings and movie tags are encoded and stored in individual files named **"users.gob"**, **"movieTitles.gob"**, **"movies.gob"** and **"tags.gob"** respectively. The processed files exist under *the directory that was specified for flag* ***-p*** *when executing the* ***preprocess binary***, so the parameter **-d** of recommender should now target this directory.

The new data loading functions are created in **"utils/loadProcessedData.go"** to decode these files.

The use of GOB encoding/decoding saves disk space by storing the data in a more efficient format. It has also a big impact in the speed that the recommender binary now loads the required data, since the encoded objects match the objects that recommender uses (common models), are already validated and need no further processing. Namely, the reading of all four of the CSV needed took up to 35 seconds in the original recommender, while it takes less than 5 seconds in the real system. In terms of input file size, assuming that the working directory is the **root directory** of the project, the CSVs are in **"./ml-latest"** directory and we set **"preprocessed-data"** as the preprocessed data directory, below is a comparison of the original files and the processed files:



```
  ~/Fotis_Ioannis_M111_Project
● go ❯ du -hs ml-latest/ratings.csv ml-latest/movies.csv ml-latest/tags.csv
  891M    ml-latest/ratings.csv
  4.0M    ml-latest/movies.csv
  82M     ml-latest/tags.csv

  ~/Fotis_Ioannis_M111_Project
● go ❯ du -hs preprocessed-data/*
  2.7M    preprocessed-data/movieTitles.gob
  220M    preprocessed-data/movies.gob
  31M     preprocessed-data/tags.gob
  199M    preprocessed-data/users.gob
```

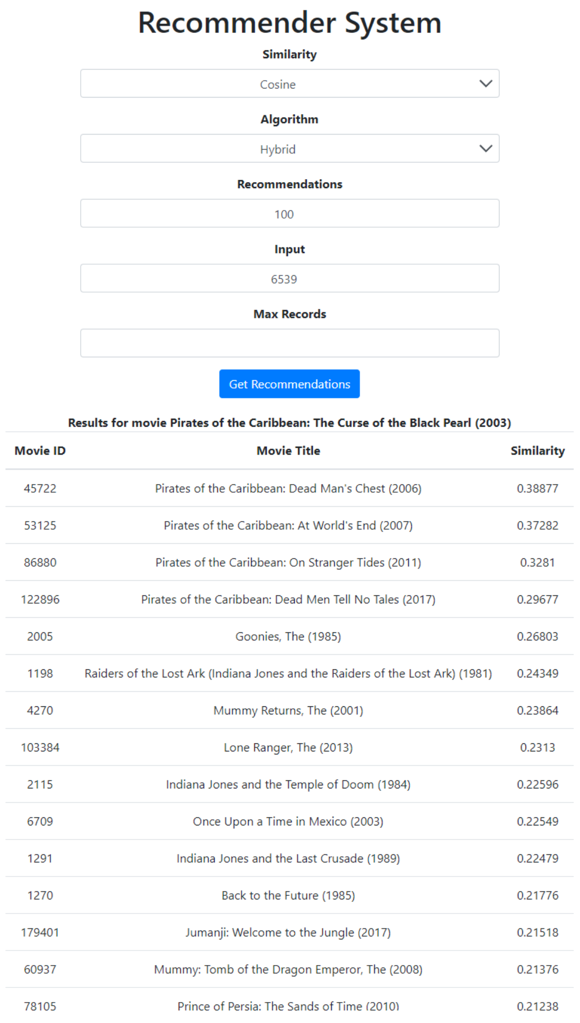
Note: The preprocess binary needs to be executed **at least once** before recommender. After the GOB files are created the recommender can be executed without any other restriction or need to repeat the preprocessing *unless the ml-latest dataset is updated*.

2.g. Web interface

The back-end logic of the web-server co-exists with the CLI version of the app which was used until now. To execute recommender in web-server mode, it is necessary to provide only the parameters: **"-d preprocessed-data -u"**. The flag **"-d"** works as described in section 2.f. The flag **"-u"** leads the configuration mechanism to ignore all other flags and set up recommender to run in web-server mode.

The core function of the API is to serve a **"/recommend"** endpoint which accepts http GET requests. In this case all the structs are populated when starting the server in order to be able to handle any user request at once. The core logic of the server is in **handleRecommendationRequest** function. What's important to note is that if the request specifies that a limited dataset should be used (maxRecords parameter is other than -1), then the appropriate struct is adapted in size according to the algorithm specified. After the request is satisfied, the dataset is restored back to the original size. This happens to be able to serve all types of requests in the optimal time.

The front-end logic is in **"ui"** folder. It consists of a simple HTML form with dropdown menus for the similarity metric and the algorithm, as well as fields for the input and recommendation limits. The is also the optional parameter maxRecords to reduce the dataset size if desired. The UI is available on http://localhost:8080/ui/.


Recommender System

Similarity: Cosine

Algorithm: Hybrid

Recommendations: 100

Input: 6539

Max Records:

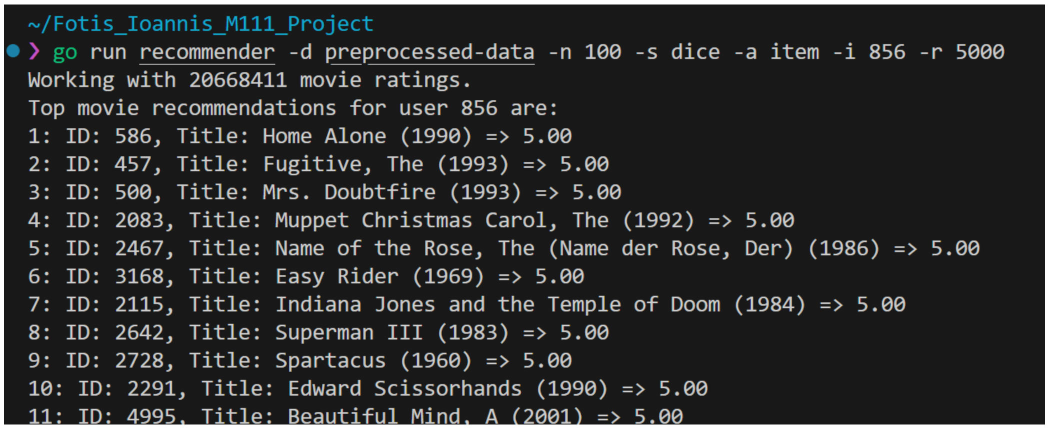
Results for movie Pirates of the Caribbean: The Curse of the Black Pearl (2003)

Movie ID	Movie Title	Similarity
45722	Pirates of the Caribbean: Dead Man's Chest (2006)	0.38877
53125	Pirates of the Caribbean: At World's End (2007)	0.37282
86880	Pirates of the Caribbean: On Stranger Tides (2011)	0.3281
122896	Pirates of the Caribbean: Dead Men Tell No Tales (2017)	0.29677
2005	Goonies, The (1985)	0.26803
1198	Raiders of the Lost Ark (Indiana Jones and the Raiders of the Lost Ark) (1981)	0.24349
4270	Mummy Returns, The (2001)	0.23864
103384	Lone Ranger, The (2013)	0.2313
2115	Indiana Jones and the Temple of Doom (1984)	0.22596
6709	Once Upon a Time in Mexico (2003)	0.22549
1291	Indiana Jones and the Last Crusade (1989)	0.22479
1270	Back to the Future (1985)	0.21776
179401	Jumanji: Welcome to the Jungle (2017)	0.21518
60937	Mummy: Tomb of the Dragon Emperor, The (2008)	0.21376
78105	Prince of Persia: The Sands of Time (2010)	0.21238

Sample request: Max Records parameter is not specified. The default behavior is to use the whole dataset.

3. Assumptions

- When the algorithm is “user” or “item” the results include a forecasted rating for each movie and sort the recommended movies according to this rating. In all other algorithms, the results contain a similarity score for a movieID, not a rating. Recommendations are sorted by this similarity score.
- When running the recommender with a maxRecords specified, this number refers to the number of user / movie / title / tag objects to be taken into consideration. For this purpose, every recommender function logs this information at the beginning of its execution in order to clarify the amount of the original records that are used.
  For example, when executing the item-item scenario, when maxRecords is specified to be 5000, the **first** 5k movies in “preprocessed-data/movies.gob” file will be used, containing all their ratings independently of their number. This is demonstrated in the following figure:



```
~/Fotis_Ioannis_M111_Project
❯ go run recommender -d preprocessed-data -n 100 -s dice -a item -i 856 -r 5000
Working with 20668411 movie ratings.
Top movie recommendations for user 856 are:
1: ID: 586, Title: Home Alone (1990) => 5.00
2: ID: 457, Title: Fugitive, The (1993) => 5.00
3: ID: 500, Title: Mrs. Doubtfire (1993) => 5.00
4: ID: 2083, Title: Muppet Christmas Carol, The (1992) => 5.00
5: ID: 2467, Title: Name of the Rose, The (Name der Rose, Der) (1986) => 5.00
6: ID: 3168, Title: Easy Rider (1969) => 5.00
7: ID: 2115, Title: Indiana Jones and the Temple of Doom (1984) => 5.00
8: ID: 2642, Title: Superman III (1983) => 5.00
9: ID: 2728, Title: Spartacus (1960) => 5.00
10: ID: 2291, Title: Edward Scissorhands (1990) => 5.00
11: ID: 4995, Title: Beautiful Mind, A (2001) => 5.00
```

Execution of item-item scenario using the first 5,000 movies of the dataset.

4. Conclusions

- The cosine similarity metric seemed to me to be the most accurate, especially when finding similar movies based on title and tags (aka when dealing with text) which is easy to validate.
- Increasing the k-factor gives more accurate results in all scenarios but increases execution time.
- User-user correlation didn’t have high variance when checking random user_ids.
- Item-item is heavily affected by the number of movies the user has rated and also how popular a movie is based on how many users have rated it.
- Tag-based method is also affected by how popular a movie is, this time based on how many tags refer to it but has overall a very stable performance when checking random movies.
- Title-based solution depends on the total length of the vectors of tokens created for the two movies which are compared. If the specified movieID has a long title, chances are it will take longer to get the final rankings. In addition, the rarity of the words plays a role too. If a movie has rare words in it, it will be compared with a significantly lower number of movies due to the way the comparable movie_ids are selected. I happened to get title-based results in under half a second for a movie and over 12 seconds in another one.
- The highest impact of the hybrid recommender is by the item-item most of the times and then by title-comparison because it has a big variance in execution time per movie_id.

Thank you for your time