Movies Search Engine and Recommender System

Habbash Nassim (808292) Matamoros Ricardo (807450)
Villa Giacomo (807462)

Information Retrieval Project A.A. 2019-2020
University of Milano-Bicocca
(e-mail: n.habbash@campus.unimib.it
e-mail: g.villa48@campus.unimib.it
e-mail: r.matamorosaragon@campus.unimib.it).

1. INTRODUCTION

The project required the **development** of either a **search engine or a recommender system** on a collection of textual documents following a set of formal requirements.

The following **project implemented a custom search engine and a recommender systems using three different methodologies**, personalized search recommendations, content-based recommendations and collaborative filtering recommendations.

2. DATA

The dataset used is the **MovieLens** dataset, containing 45.000 movies released before July 2017. **The data is extensive and comprised of multiple tables**, including informations partaining to the movie metadata, such as casting, crew, production and average ratings, but also individual ratings from 270,000 users. **Ratings are on a scale of 1-5** and have been obtained from the official GroupLens website.

For the current project, **only a subset of tables has been used**, that are:

- `movies_metadata.csv`, renamed `dataset.csv`, containing the movie metadata.
- `ratings_small.csv`, containing a subset of the ratings given by users to movies.

The metadata **dataset has undergone missing data evaluation and feature selection** to obtain a more suitable structure for the project; in particular, movies that didn't present a **title** or **overview** have been discarded (amounting to around *2000* documents on 45000). The final dataset (parsed_dataset.csv) has the following attributes:

- **id**: unique identificator of the movie.
- **title**: string defining the title of the movie.
- **overview**: short text to summarize the plot of the movie.
- **original_language**: ISO 639-1 code to identify original language of film.
- **spoken_languages**: list of ISO 639-1 codes for the movie's spoken language.
- **genres**: list of the movie's genres.
- **release_date**: release date in format `YYYY-MM-DD`.
- **production_companies**: list of the companies responsible for producing of the movie.
- **vote_average**: average movie vote.
- **vote_count**: movie votes count.
- **weighted_vote**: weighted vote average according to the vote count (shrinkage estimator)
- **poster_path**: URL containing an image of the movie poster.

For the purpose of the project, a **set of users** (users.csv) has been created **according to the following attributes**:

- **user_id**: unique identificator for the user.
- **username**: user's username
- **location**: ISO 639-1 code to identify the user's birth country.
- **birth_date**: user's irthdate in the format `YYYY-MM-DD`.
- **genre_preferences**: list of the user's genre preferences about movies according to MovieLens format.
- **language**: list of ISO 639-1 codes of the user's spoken language.
- **film_ratings**: list of the user's ratings in the format `(<movie_id> <rating>)`. The rating score is between 1 and 5, real values ammitted.

3. SEARCH ENGINE

3.1 Introduction

For the development of the search engine **ElasticSearch** based on Lucene has been chosen as the main tool. The **project** has been **written** in **Python 3.7** using the **elasticsearch-dsl** library that wraps ElasticSearch's classic APIs to Python. The application itself is served through a **Django** application, with an ElasticSearch connector between Django's models and ElasticSearch's documents provided by **django-elasticsearch-dsl**

3.2 Index creation

The **index was created on movies attributes to enable querying operations**. The following **field analyzers have been defined**:

- *standard_analyzer*: used for the title and overview field, and consists of a standard tokenizer based on whitespace separator, a filter for removing english stopwords, a normalize to lowercase and finally a snowball algorithm for stemming.
- *nostem_analyzer*: used for original language, spoken language, genres and production companies; it is identical to the previous one except for the absence of the stemming phase.

Other attributes haven't been subject to the analyzation process as it has been deemed unnecessary for the retrieval process itself.

3.3 Query operation

ElasticSearch offers different types of similarity algorithms, such as TF-IDF, BM25, Boolean, or custom made. For the project, **it has been decided to use a similarity based on the Okapi BM25 algorithm**:

$$score(D,Q) = \sum_{i=1}^{n} IDF(q_i) \cdot \frac{f(q_i,D)\cdot(k_1+1)}{f(q_i,D)+k_1\cdot(1-b+b*\frac{|D|}{avgdl})}$$

Where:

- $f(q_i, D)$: q_i term frequency in the document D.
- *avgdl*: average document length in the text collection from which documents are drawn.
- b and k_1: free parameters, usually chosen, in absence of an advanced optimization, as $k_1 \in [1.2, 2.0]$ and $b = 0.75$.

In order to allow personalized search **we provide a different relevance dimensions based both on the user profile and movies characteristics**; said dimensions have been parametrized in the user interface to allow ease of customization. As such, **the query structure has been implemented as modular**, as to allow different configurations to be easily plugged-in according to the user's preferences. The personalized process is based on pre processing approach.

The **main query is structured as a tree** as follows:

- **Boolean query** A query that matches documents matching boolean combinations of other queries. This is the root node of the query tree, and contains one *Must queryset* and one *Should queryset*
  - **Must queryset** This is the first leaf node of the Boolean query, and contains those clauses that must appear in the documents and will contribute to the scores. In this case, the Must queryset contains a *multi-match query* on the fields Title, Overview and Production Companies of the document, each weighted differently. The query also allows for usage of *fuzziness* and *synonym tree generation*, which allow for more extensive searches, and have been parametrized in the user interface.
  - **Should queryset** This is the second leaf node, and contains those clauses (queries) that should appear in the matching document. As such, it is not mandatory for the documents to match those clauses, but matching them will contribute to the scores. The **should queryset has been used to implement different relevance dimensions** in the retrieval process. The relevance dimensions mapped to the should queryset are *User interest* and *User language*, in the form of two *multi-match queries* on the fields Genres, Overview and Spoken_lan and Original_lan respectively, each field weighted differently.

The above **structure implements the main query core and the relevance dimensions pertaining to the user** (Genre Preference and Language).

To experiment and test different functions of *ElasticSearch*, the **relevance dimensions related to the movie metadata** have been implemented following a different paradigm:

- **Function score Query** Function score queries allow to modify directly the *score of a query* according to a user customized function. The function score query is effectively a wrapper of the boolean query, which specifies different computations of the score on document fields according to the specifications inserted into the field *field value factor*. With this paradigm, the relevance dimensions of *Movie popularity* and *Movie weighted vote* have been implemented each, based on a square root boosting of the field values vote_count and weighted_vote respectively.

A **phrasal query has also been added** for the user to search for exact text matches, and simply substitutes the multi-match query leaf node for a match-phrase query leaf node.

Following is a summary of the parameters used:

- **Personalized**: allows personalized search based on the user profile using the should_queryset.
- **Fuzzy**: allows inexact fuzzy matching, using the fuzziness parameter. Fuzziness is interpreted as a Levenshtein Edit Distance, in particular the parameter is specified as `AUTO` (generates an edit distance based on the length of the term). The number of beginning characters left unchanged when creating expansions is setted to 2 (*prefix_length parameter*).
- **Synonyms**: multi-terms synonym expansion with the *synonym_graph* token filter. When this filter is used, the parser creates a phrase query for each multi-terms synonyms. For example, the following synonym: `‘‘ny, new york’’` would produce: `(ny OR (‘‘new york‘‘))`.
- **Popolarity Relevance**: allows personalized search based on the movie metadata using the function_score queries.
- **Weighted vote Relevance**: same as of **Popolarity Relevance**, allows personalized search based on the movie metadata using the function_score queries.
- **Phrasal search**: not a customization per-se, but an addittional query-type, by using double quotes in the query text (enclosing a word) it is possible to search for an exact match of the input string only in *title* field.

4. RECOMMENDER SYSTEM

4.1 Recommendations with Personalized Search

It's possible to have recommendations by just using the search engine previously mentioned.

Using the different relevance dimensions and a *blank query*, the system retrieves items according just to the user profile and movie metadata, forming a pseudo-recommendation system.

This recommendation system **places in-between classic approaches such as Content Based recommendation**, as it makes use of a user profile representation (in this case, *genre preference* and *language*), but also makes use of rating aggregates (vote count and weighted vote), that can represent, in a way, user-item interaction. Although aggregated, putting Collaborative Filtering approaches on the other extreme.

4.2 Collaborative Filtering

Collaborative Filtering is based on the assumption that people that have agreed in the past will agree also in the future and will appreciate similar items as those they appreciated in the past.
The approach used for the implementation of CF is defined as **N**eural **C**ollaborative **F**iltering (**NCF**). This technique is widely used in modern recommendation systems such as Youtube, Netflix and Google Play Store. The objective is that of realizing a **U**ser **R**ating Matrix (**URM**) factorization using a *feed-forward Neural Network*, in order to infer the valutaions for a set of items not yet evaluated explicitly by a set of users. As such, **NCF is used to realize a deep matrix factorization** (DMF), an approach that tries to get the strength of the matrix factorization, as in the incorporation of information not directly given by the user, but derived by the analysis of the user behaviour (implicit feedback) on the set of available items, and that information allows for the estimation of missing ratings.

Problem definition Given a sparse matrix URM of dimensions $n * m$, where n corresponds to the set of users that have given at least one rating to the set m, that represents the set of evaluated items, and given a *feed-forward neural network* F, it's necessary to generate a **non-sparse matrix** URM_1 of dimension $n * m$.

Deep Matrix Factorization This approach uses a **latent representation of the user profile** q_u **and item profile** q_i, similarly to the normal *Matrix Factorization* approach, which uses the same representantations together with a **linear operation to compute the missing ratings**. The **issue with MF** is that **it requires a high dimensionality to allow better expressivity and to manage complex relations**. With DMF the intuition is to use a NN to learn a non-linear function of the user-items interactions, increasing the expressivity of the final model. Formally:

$$\hat{y}_{\mathrm{u,i}} = F(u, i|\Theta)$$

Where y is the predicted ratings of the user-item interaction, Θ is the set of parameters of the model, F is the non-linear interaction function that maps Θ to y.

Network structure

- The **first level receives as input a sparse vector for the user and one for the item**. Said representation can be seen as a one-hot encoding generated through the binarization given from this level.
- **The second level is called Embedding**, and is *Fully Connected.* Starting from the sparse representation of the user and item profiles, **it generates a dense representation of the of dimension K**, giving in output a latent representation of q_u and q_i.
- The **third level corresponds to another Fully Connected layer**, with a *ReLU activation function*, introducing non-linearity.
- The **fourth level corresponds to the concatenation of the output of the previous layer**, generating a unique vector in output
- The **fifth and sixth levels are identical to the third**, differing only in dimension.
- The **last layer is an output layer** of dimensionality 1, with a Sigmoid activation.

Following are the *hyperparameters* of the NN:

- **Loss function:** Mean Squared Error
- **Optimizer:** Adam
- **Learning rate:** 0.01
- **Beta 1:** 0.9
- **Beta 2:** 0.99

The loss function is in itself defined as:

$$Lf = \sum_{(u,i)\in D} W_{u,i}(y_{u,i} - \hat{y}_{u,i})^2$$

Where D is the set of **user-item pairs**, $y_{u,i}$ is the **original rating**, $\hat{y}_{u,i}$ is the inferred rating and $W_{u,i}$ are the weights of the training instance.

Data preprocessing The **main dataset used at this step are ratings_small_extended.csv**. Users and Items are mapped together before performing a normalization of the target column in 10 values between [0.5, 5]. At last, a split between train and test of 0.8-0.2 is performed. **The only issue found in the dataset is a balancing issue between classes**.

Training **For training 10-fold cross-validation has been used**, with an additional 0.66-0.34 split for every fold. The **maximum number of epoch for each fold is 50**, but with *Early Stopping* for the control of MSE on the validation, corrisponding to 0.1 of each training set. The test set is used during the preprocessing to evaluate the performance of the trained model.

Results The mean value of the *MSE* is **0.01%**, with a *std deviation* of **+/- 0.001%**. Successively the test phase, the final model obtains the same *MSE* value.

4.3 Content Based

In this type of system, **items are mainly defined by their associated characteristics**. The **user profile is**

represented by the user's own past ratings (movies seen). As such, **this type of system recommends to users items similar to those appreciated in the past**, on a item similarity basis. The **objective of the system** implemented is to **use a representation of the user profile**, in this case, bag-of-word comprised of the overviews of seen movies by the user, **and compare it to the bag-of-word representation of every item available**, in order to **obtain a ranking** in terms of similarity between the user profile and the item catalogue.

Problem definition Given a set of users each represented through a *bag-of-words profile*, where the weight of every term corresponds to the value $W_i = TF_i * IDF_i$, and given a set of items represented through a bag-of-word, **it's necessary to output the similarity value between all the pairs user-item**. In this case, cosine similarity has been used:

$$sim(U, I) = \frac{bow_u \cdot bow_i}{||bow_u|| \cdot ||bow_i||} = \frac{\sum_{k=1}^{n} W_{ku} \cdot W_{ki}}{\sqrt{\sum_{k=1}^{n}(W_{ku})^2} \cdot \sqrt{\sum_{k=1}^{n}(W_{ki})^2}}$$

Data preprocessing The **dataset used for this process are parsed_dataset.csv and users.csv**. Following are the steps taken:

- The **first step** consists in the **insertion of the users created manually into the ratings dataset**, ratings_small.csv. Every rating of every user profile is as such inserted into the ratings dataset, obtaining ratings_small_extended.csv.
- From the set of items descriptions contained in parsed_dataset.csv, **some preprocessing is done on the text**, consisting in lowercasing, tokenizing, stopword removal, stemming, punctuation removal. **The final parsed overview is saved as a feature**, and a join between parsed_dataset.csv and small_ratings_extended.csv is done to generate dataset_rs.csv.
- The **last step consists in the generation of the bag-of-word representation of the user profile based on the user's ratings**. This is done by considering only movies positively rated (above 3/5). As such the BOW representation obtained is then saved as a feature in the users_final.csv set.

Similarity step computation To **compute the cosine similarity**, first the *TF-IDF* is applied to get the weight of every term inside the overview. **Higher such a weight and higher the importance of the associated term**. Succesively, given the BOW representation of the user profile and of every item, a **mapping of the user representation in the item representation space is applied to obtain an n-dimensional space** in which every element is seen as a vector. To compute the cosine similarity the cosine between the user profile vector and the vector of each item is computed:

$$W_{x,y} = TF_{x,y} \cdot log(\frac{N}{DF_x})$$

$TF_{x,y}$ represents the frequence of x (the word) in y (the item), DF_x represents the number of items that present x in the overview, and N is the total number of items.

4.4 Recommendation process

For the recommendation step itself, the **two models have been used in parallel**, outputting two different recommendation lists. Given a user profile, **the first list generated through CF is ordered in function of the inferred ranking**, while **the second according to cosine similarity**. It is also possible to apply a filter on the movies categories and languages of the movies in output according to the user profile.

CF Given the userId, from **the items list those already seen by the user are deleted**. Successively, a list of pairs *userId-ItemId* is generated, and given in input to the model, that generates a list of inferred ratings. Given the list of inferred ratings, the list is sorted in ascending order and the first n recommendations are given back.

CB **Initially a filtering on the user's language is applied with the objective of reducing the computational complexity of the task**. Given the userId and its BOW representation, the similarity between User-BOW e Items-BOW is computed, generating a list that is sorted in ascending order and the first n recommendations are given back.

Motivations **Using both models in parallel allows the assessment of the quality of the recommendation in both system**, allowing some evaluations on their behaviours based on their specifications.

5. WEB INTERFACE AND FRAMEWORK

As initial mentioned **Django was used**; Django is a high-level Python ORM Framework used **to host and structure the search engine and the recommandation system application**. It maps the user and movie structure to Django models, allowing for fast development and deployment of the front-end and back-end of the application.

Docker has been also used to allow for fast development between team members **and allowing easy dependance management**.

6. CONCLUSIONS

It is **possible to enhance the search system by adding other relevance dimension such as user's birthdate**; possibly justified by famous movies remake trends. For example, a user born in the 2000s searching for “Batman” might be more interested in Nolan's newer trilogy rather than Tim Burton's 1989 one.
Better use of ElasticSearch mappings and custom similarities might also help in tailoring more specific query results for the user.
The recommendation systems could also be merged into one working jointly, rather than two systems working parallely. As the project showed, classic feature representations such as BOW incur into computational complexity and memory issues when computing large matrices. As such, a possible improvement could be changing the items representations to a denser one, such as embeddings.

Hybridizing the user recommender systems might also help into generating better suggestions for the users.
More formal evaluations might also be performed for each systems.