Project 3 - ElasticSearch

Marco Rossini
Politecnico di Torino
Student ID: s291482
m.rossini@studenti.polito.it

***Abstract*—This report illustrates a solution proposal for the creation of an ElasticSearch-powered search engine aimed at analyzing data from the IMDb and Rotten Tomatoes online movie databases. Multiple queries performed via the Python ElasticSearch library are solved and motivated. In addition, a Flask-based web application capable of performing full-text search is presented and discussed, together with an additional custom query of interest which can be useful in a real use case.**

I. Project overview

This section provides an overview of the project at hand and specifies the associated goals, specifications and requirements.

A JSON file containing data from the IMDb and Rotten Tomatoes online movie databases was made available, and must serve as the data source for creating an ElasticSearch-powered search engine for analysis purposes. As a first step, a Python script must be written in order to parse and ingest the relevant information contained in the original JSON data collection to an appropriately created ElasticSearch index.[1] Unfortunately, the provided data contain missing *strings*, *numbers* and *lists*, therefore the script must be written in such a way as to replace them with `""`, `0` and `[]` respectively. After performing this necessary step, the created index must be exploited to solve some provided queries of interests using the Python ElasticSearch library. All the reported solutions for the queries must include an explanation of the main operators involved in each of them. After performing these queries, a Flask-based web application that is capable of performing full-text search over the provided data collection must be created. Such application must be capable of highlighting the first matches of the query search, boosting results scores by movies ratings and presenting the top-20 resulting documents in a dedicated HTML page. The Flask-based web application created this way must be appropriately described, providing specific insights on the impact of boosting standard scores by using movie rating values. In addition, the TF-IDF standard scoring function used by ElasticSearch must be analyzed from a critical point of view, highlighting its strengths and weaknesses in a real world scenario. Finally, an additional novel query of interest must be proposed and solved, together with a description of when and how it could be useful in a real use case.

In this report, a solution proposal aimed at meeting these requirements is illustrated and motivated. In particular, multiple queries performed via the Python ElasticSearch library are reported and explained. In addition, a Flask-based web application capable of performing full-text search is presented and described, and the TF-IDF standard scoring function used by ElasticSearch is critically analyzed. Finally, an additional custom query of interest which can be useful in a real use case is solved and motivated.

[1]The complete Python code of the defined data ingestion script can be found at this link.

II. Interest queries

This section reports and explains the proposed solutions for the required interest queries. The queries were solved using the Python ElasticSearch library, which provides common ground for all ElasticSearch-related code.[2]

A. Get the list of the first 10 movies that have been published between 1939 and 1945 and best match the word "war" in the plot.

```
body = {"query": {
          "bool": {
              "must": [{
                  "range": {
                      "year": {
                          "gte": 1939,
                          "lte": 1945}}}, {
                  "match": {
                      "plot": "war"}}]}
       }, "size": 10}
```

This query was managed by defining a `bool` query, in order to retrieve documents matching boolean combinations of other queries. The boolean query was filled with a `must` clause, in order to force its content to appear in matching documents. The publication year constraint was imposed by using a `range` query, requiring it to be greater than or equal to `1939` (`gte`) and less than or equal to `1945` (`lte`). The word match constraint was imposed by using a `match` query, requiring the `plot` field to match the word `war`. Finally, retrieved results were limited as required by setting a value of `10` for the `size` parameter.[3]

B. Get the list of the first 20 movies that have an average IMDb rating higher than 6.0 and best match "Iron Man" in the full plot.

[2]The complete bodies of the solved interest queries can be found at this link.

[3]It is however worth mentioning that, since the number of results returned by default is `10`, this last step can be omitted.

```
body = {"query": {
           "bool": {
               "must": [{
                   "range": {
                       "imdb.rating": {
                           "gt": 6.0}}}, {
                   "match": {
                       "fullplot": "Iron Man"}}]}
       }, "size": 20}
```

This query was managed by defining a `bool` query filled with a `must` clause. The IMDb rating constraint was imposed by using a `range` query, requiring it to be greater than `6.0` (`gt`). The word match constraint was imposed by using a `match` query, requiring the `fullplot` field to match the word `Iron Man`. Finally, retrieved results were limited as required by setting a value of `20` for the `size` parameter.

C. Search for the movies that best match the text query "matrix" in the title. Boost the results by multiplying the standard score with the IMDb rating score, then with the Rotten Tomatoes score. Does the order of the results change?

```
body = {"query": {
          "function_score": {
            "query": {
              "match": {"title": "matrix"}},
            "script_score": {
              "script": {
                "source":
                  "doc.containsKey('imdb.rating')
                  && doc['imdb.rating'].value > 0.0
                  ? doc['imdb.rating'].value : 1"
       }}}}}
```

This query was managed by defining a `function_score` query, which allows to modify the score of the retrieved documents. The function score query was filled with a `match` query, requiring the `title` field to match the word `matrix`. In order to boost the results with the IMDb ratings as required, the `script_score` function was exploited, allowing to modify the score of the retrieved documents with a custom script. The `source` of the `script` for the function was defined in such a way as to check in advance that the rating value exists (`doc.containsKey('imdb.rating')`, to avoid runtime errors) and that it is greater than `0.0` (`doc['imdb.rating'].value > 0.0`, to avoid scores from being incorrectly nullified).[4] By default, the standard document scores are multiplied by the value returned by the custom script (`score_mode: multiply`) as required, hence no specification for the `score_mode` was provided. Since the project specifications require to repeat the same operation also with the Rotten Tomatoes ratings instead of IMDb ones, a second analogous query was written, simply replacing `'imdb.rating'` with `'tomatoes.viewer.rating'`.

If both queries are executed, it can be noticed that the order of the results change, as reported in the following text box.

```
IMDb rating boost:                        Rotten Tomatoes rating boost:
  #1) Score: 89.27725                       #1) Score: 36.94231
      Title: The Matrix                         Title: The Matrix
      IMDb rating: 8.7                          Rotten Tomatoes rating: 3.6
  #2) Score: 65.58309                       #2) Score: 31.019032
      Title: The Matrix Revisited               Title: Armitage: Dual Matrix
      IMDb rating: 7.4                          Rotten Tomatoes rating: 3.5
  #3) Score: 63.810577                      #3) Score: 30.132774
      Title: The Matrix Reloaded                Title: The Matrix Reloaded
      IMDb rating: 7.2                          Rotten Tomatoes rating: 3.4
  #4) Score: 59.379288                      #4) Score: 30.132774
      Title: The Matrix Revolutions             Title: The Matrix Revolutions
      IMDb rating: 6.7                          Rotten Tomatoes rating: 3.4
  #5) Score: 57.606773                      #5) Score: 30.132774
      Title: Armitage: Dual Matrix              Title: The Matrix Revisited
      IMDb rating: 6.5                          Rotten Tomatoes rating: 3.4
  #6) Score: 50.318214                      #6) Score: 27.675018
      Title: Return to Source:                  Title: Return to Source:
             Philosophy & 'The Matrix'                 Philosophy & 'The Matrix'
      IMDb rating: 8.0                          Rotten Tomatoes rating: 4.4
```

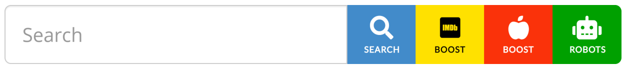
In particular, the movies *"The Matrix Revisited"* and *"Armitage: Dual Matrix"* turn out to be respectively ranked #2 and #5 if boosted by IMDb rating, whereas they turn out to be respectively ranked #5 and #2 if boosted by Rotten Tomatoes rating. This is due to the fact that the two movies are assigned different ratings on the two online movie databases: on IMDb *"The Matrix Revisited"* is rated higher than *"Armitage: Dual Matrix"* (`7.4` vs `6.5`), whereas on Rotten Tomatoes the opposite occurs (`3.4` vs `3.5`). Since the final search scores for the queries are computed by multiplying their standard scores by their respective movie ratings, *"The Matrix Revisited"* will be favored with respect to *"Armitage: Dual Matrix"* when boosting by IMDb rating, whereas the opposite will occur when boosting by Rotten Tomatoes rating.

III. Movies Search Engine

This section presents and describes the web application capable of performing full-text search, created using the Flask micro web framework and the Python ElasticSearch library.[5]

The application consists of a total of four main source files:

- `search.html`: this file defines the main web page of the search engine, of which it can be considered the homepage. It is the first page that is shown to the user who is preparing to perform a search, and contains the main search bar, the buttons that can be clicked to execute a query and an information box with instructions on how to perform a search. Each button is linked to a different application logic (defined in `app.py`) which allows the execution of a specific type of query.



- `app.py`: this file defines the core logic of the whole web application, guaranteeing the actual existence of the application itself. By running this file, the application starts by connecting to an ElasticSearch cluster and rendering the default search web page (`search.html`). When a query is run by clicking one of the buttons in the search page, a search request function is triggered inside `app.py`, which in turn executes a search method

[4]It is however worth mentioning that, since the data ingestion script already replaces missing numeric fields with the value 0, the first check of existence for the rating value can be omitted.

[5]The folder containing the complete source code of the Flask-based web application can be found at this link.

filled with a specific body based on the specific button that was clicked by the user. More specifically, the user can perform four types of queries:

  - *Standard search:* results are sorted by relevance based on how well the keywords entered by the user match the title and the full plot of the documents in the index.
  - *IMBd-boosted search:* results are sorted by relevance like in a standard search, but their scores are boosted by a multiplication with IMDb movie ratings.
  - *Rotten Tomatoes-boosted search:* results are sorted by relevance like in a standard search, but their scores are boosted by a multiplication with Rotten Tomatoes movie ratings.
  - *"Robots" query search:* a pre-defined custom interest query is executed (regardless of the input keywords entered by the user).

  After a query is executed, the application renders the results web page (`results.html`).
- `results.html`: this file defines the results web page of the search engine. When a query is run by clicking one of the buttons in the search page, the applications renders this page which reports the score, title, full plot and ratings for each of the top-20 documents found in the index, sorted by descending relevance score. For each result, the first match of the full-text search for the full plot field is highlighted in bold. In this page, the search bar and query buttons are still present, hence the user can still perform further searches.
- `style.css`: this file defines the style, design and layout of the web pages of the application, and was suitably adapted to provide a nice look to the elements defined in them.

The created application is therefore able to let the user perform standard searches as well as rating-boosted searches. Exploiting the IMDb and Rotten Tomatoes movie ratings to boost standard scores can have a relevant impact on the effectiveness of the performed search. For example, a movie in the index may have a poorly verbose plot and full plot (e.g. because they were poorly filled in), and this could penalize it in the ranking of the retrieved results due to a probable poor match of the document with the input keywords. However, such a movie may actually be considered as highly acclaimed by the viewers, and therefore it may be undesirable for it to be bypassed by other possibly less acclaimed results. By boosting standard scores using the movie ratings, situations like these can be compensated, as the multiplication by the weighting-factor given by the movie ratings introduces a re-balancing of possible relevance gaps caused by poor keyword matches. Of course, this behavior is based on the assumption that the movie ratings constitutes a factor of relevance for the user who is performing a search. If, on the contrary, the user was not interested at all in the movie ratings expressed by other viewers, but only interested in just the relevance given by the level of correspondence with the input keywords, then it would not be advisable to boost the results using movie ratings, sticking to standard searches instead.

IV. Scoring function analysis

This section analyzes, from a critical point of view, the TF-IDF standard scoring function used by ElasticSearch, highlighting its strengths and weaknesses in a real world scenario.

TF-IDF is a numerical statistic that is intended to reflect how important a word is to a document, and is calculated as the product of two measures:

$$TF \times IDF$$

Where TF is the *Term Frequency* and IDF is the *Inverse Document Frequency*. There are various ways for determining the exact values of both statistics but, in general, the purpose of TF is to quantify the level of occurrence of a term in a document, whereas the purpose of IDF is to quantify the level of specificity of a term in a document. TF is aimed at increasing the importance of a term that is repeated multiple times in a document, and can be calculated as the raw count of the term in the document. For example, if a user is trying to find out documents about *"The Matrix"*, a great occurrence of the keyword *"Matrix"* in a document will result in a high TF value for that term, increasing its weight in determining the document's final relevance score. IDF, instead, is aimed at reducing the importance of a term that is repeated multiple times among multiple documents, and can be calculated as the logarithmically scaled inverse fraction of the documents that contain that term. For example, considering the same input keywords as before, the keyword *"The"* will be probably included many times in many documents, since it is a very common word. For this reason, its presence will not be of much help in determining if the documents to be analyzed are about *"The Matrix"* or not, because the term will be present in almost every document (the term *"Matrix"* will be instead much more interesting). A great occurrence of such keyword among many documents will hence result in a low IDF value for that term, decreasing its weight in determining the document's final relevance score.

Like all measures, TF-IDF cannot however be considered as an infallible tool capable of perfectly quantifying the actual relevance of a set of documents for a provided input query. Overall, TF-IDF strengths and weaknesses can be summarized as follows:

Strengths

- *Simple:* it can be computed in an extremely simple and easy to understand way.
- *Powerful:* despite being basic, it constitutes a powerful metric capable of extracting the most descriptive terms in a document.
- *Provides similarity:* it can be used as a metric to compute the similarity between two documents.

Weaknesses

- *Does not capture semantic:* being based on the bag-of-words model, it disregards grammar and word order and is not able to capture the semantic meaning of sentences. Like an histogram describes an image using pixels intensity values, ignoring all the information given by how pixels are located in the image itself (e.g. a set of pixel could represent a specific object in the image endowed with a further meaning), TF-IDF ignores word positioning, therefore specific combinations of words that could carry a particular semantic meaning will have no weight in determining the relevance of a document.
- *Can be slow:* by computing document similarity in the word count space, it may be slow to compute for very large vocabularies.

In a real world scenario, TF-IDF strengths and weaknesses could have a great or little impact on the relevance of the retrieved results, depending on the specific characteristics and aims of the performed search. If a user is interested in searching for keywords which do not convey any semantic meaning (e.g. a shopping list like "bread, meat and milk"), TF-IDF could provide an effective and satisfactory sorting of resulting documents, as this type of search perfectly fits the bag-of-word paradigm. On the contrary, if a user is interested in searching for a sentence characterized by a sharp semantic meaning (e.g. a question like "who is the eldest son of the Queen of the United Kingdom?") then TF-IDF could provide an unsatisfactory sorting of resulting documents, with the risk of giving, in some cases, excessive importance to some specific terms that, despite being included in the provided query, may be distant from the actual semantic meaning of the answer that the user was looking for.

V. Additional search request

This section reports and explains an additional novel query of interest, solved using the Python ElasticSearch library. An explanation of its utility in a real use case is also presented and discussed.[6]

***Interest query**: get the list of the first 20 movies that best match the word "robots" in both the plot and fullplot and that have been published after year 2000. Boost the results by multiplying the standard score with the average between IMDb and Rotten Tomatoes ratings.*

```
body = {"query": {"function_score": {"query": {
          "bool": {
            "must": [{
              "multi_match": {
                "query": "robots",
                "fields": ["plot", "fullplot"]}},{
              "range": {
                "year": {"gte": 2000}}}]}},
        "script_score": {"script": {"source":
          "doc.containsKey('imdb.rating') &&
           doc['imdb.rating'].value > 0.0 &&
           doc.containsKey('tomatoes.viewer.rating')
           && doc['tomatoes.viewer.rating'].value >
           0.0 ? (doc['imdb.rating'].value +
           doc['tomatoes.viewer.rating'].value) / 2
           : 1"}}}
      }, "size": 20}
```

[6]The complete text of the solved interest query can be found at this link.

This query was managed by defining a `function_score` query (in order to modify the score of the retrieved documents) filled with a `bool` query (in order to retrieve documents matching boolean combinations of other queries). The boolean query was in turn filled with a `must` clause (in order to force its content to appear in matching documents). The multi-field word match constraint was imposed by using a `multi_match` query, requiring both the `plot` and `fullplot` fields to match the word `robots`. The publication year constraint was imposed by using a `range` query, requiring it to be greater than or equal to `2000`. In order to boost the results with the average between IMDb and Rotten Tomatoes ratings as required, the `script_score` function was exploited, allowing to modify the score of the retrieved documents with a custom script. The `source` of the `script` for the function was defined in such a way as to check in advance that the rating values exist (`doc.containsKey('rating')`, to avoid runtime errors) and that they are greater than `0.0` (`doc['rating'].value > 0.0`, to avoid scores from being incorrectly nullified). If all conditions are met, the standard document scores are then multiplied by the average between IMDb and Rotten Tomatoes ratings (`doc['imdb.rating'].value + doc['tomatoes.viewer.rating'].value) / 2`). Finally, retrieved results were limited as required by setting a value of `20` for the `size` parameter.

The described interest query was proposed since it could be helpful in a real use case. A user who is becoming passionate about robotics might be interested in searching for a good movie to watch which narrates about the world of robots. However, since technology evolves quickly, some hypotheses about the future of robots narrated in the movies of the past may no longer be realistic, or may have been proven wrong. For this reason, the user would like to filter the results by selecting only movies which belong to the most recent history (after year 2000). By leveraging on the results retrieved by the performed interest query, he or she would then be helped in performing his or her choice, having the ability to choose from a careful selection of the best recent movies (according to the IMDb and Rotten Tomatoes averaged ratings) which narrate about the world of robots (according to their plot and fullplot).