Anime List - Information Retrieval

Andreia Rodrigues
up201404691@fe.up.pt
Faculty of Engineering
of University of Porto
Porto

Francisco Queirós
up201404326@fe.up.pt
Faculty of Engineering
of University of Porto
Porto

Miriam Gonçalves
up201403441@fe.up.pt
Faculty of Engineering
of University of Porto
Porto

30th of November 2018

Abstract

Anime refers to "animation" and it represents all animation produced by Japan. It has a distinct look-and-feel compared to western animations and, over the last forty years, it has become an international phenomenon, attracting millions of fans and being translated into many languages. With the increasing number of shows being released every year, large collections of documents need to be indexed and ranked so that information retrieval tools can deliver relevant documents according to the user's information need. The aim of this paper is to focus on the selection of an information retrieval tool suitable for the type of documents present in our sample collection. It also focuses on the indexing of fields in the documents to make the delivery of relevant information easier and on the retrieval process of text-based queries.

1 Introduction

Anime is an abbreviation of the word "animation" and it is used by the Western culture to describe a Japanese-style animated film or tv show [1]. These animations are characterized by the distinct look of its characters, with huge eyes, bright colored hair and exaggerated emotional expressions and gestures [2].

Japan began producing animation in 1917, but *animes* only started to become famous after the 60s due to the creation of television, that had a crucial role in making these Japanese characteristic films an increasing trend [3] [4].

Nowadays, with the growth of *anime* popularity and the number of animated films being produced every year, *anime* fans gather in online platforms where information about *anime* and its reviews are collected and can be accessed, allowing users to interact with each other, share what animations they are interested in and keep track of what they have watched, are watching and want to watch in the future.

These online platforms gather their own data separately and there isn't a centralized place where users can have an overview of an *anime* and the rating it is given in each of them. To counter this, we previously assembled and refined the chosen information in a big sample of data stored in a single *CSV* file.

For the purpose of this paper, this refined data source file will be transformed into documents that can be indexed and queried by information retrieval tools.

We begin this paper with the characterization of the information retrieval tool chosen, along with its capacities and how it works. This is followed by the description of how the data was imported to the IR tool chosen and how the indexing and retrieval processes were done.

2 Information Retrieval Tool

For this project milestone, the application of an information retrieval tool was required. We chose *Elasticsearch*, an open source search engine based on *Apache Lucene*, after several unsuccessful tries of using *Apache Solr*.*Apache Solr* was discarded because it was not user-friendly and it didn't have as many integrated solutions as *Elasticsearch*.

Elasticsearch along with *Logstash* (used to collect, parse and enrich data) and *Kibana* (used to display data on charts, plots, maps and other visualization types), the so-called *"Elastic Stack" (ELK stack)*, eases the management, analysis and

visualization of the data previously collected [5] [6]. Being a distributed, scalable, resilient, developer friendly (supports client libraries for any language), real-time tool and used a lot in the technology market by companies makes it the best option for this project [7].

2.1 Collection and documents

The dataset containing all the data about the *animes* is in *CSV* format. Each row corresponds to a different *anime* and each column matches different data of the *anime* such as the id, title, score, synopsis, episodes.

With this structure, there is only one collection, composed by the documents corresponding to each *anime* in the dataset file (each row of the data set is transformed into a new document). In this collection, all the documents have similar characteristics as they are all characterized by the same fields. Since the data has been clustered in one single file and has a consistent structure, the need to split the several fields and create more than one index for the document was not found required.

2.2 Data processing

To index and parse all the data *Logstash* was used as it received the *CSV* data file as an input and applied the typical filtering for *CSV* files, creating a document for each *anime* and importing the full collection to *Elasticsearch* automatically.

To enable this process it was necessary to create a configuration file that described all the information necessary for *Logstash* to ingest the dataset file. It can be seen as a manual for *Logstash* to understand where is the data to be ingested, how to ingest and what should be ingested.

This configuration file is composed of three parts. In the first part, it describes the path to the *CSV* file (path) and where *Logstash* should start reading the file (*start_position*):

```
input {
    file {
        path => <csv_path>
        start_position => "
           beginning"
    }
}
```

The following part consists of the filter fields, the fields that are supposed to be parsed and how they should be converted. It consists of naming the columns that are in the dataset, the separator format of the *CSV* file, the data conversions needed for different column values to be parsed correctly and the *JSON* objects that are present in the data.

```
filter {
    csv {
    separator => ";"
    columns => ["anime_id","
       title",...]
    convert => {
    "anime_id" => "integer"
    "episodes" => "integer"
    "duration_min" => "float"
    "aired" => "date"
    "score" => "float"
    "scored_by" => "integer"
    "rank" => "float"
    "popularity" => "integer"
    "members" => "integer"
    "favorites" => "integer"
    "averageRating" => "float"
    "numVotes" => "integer"
    "rating" => "float"
    }
    }
    json
    {
        source => "aired"
        target => "aired"
    }
...
}
```

The third and last part of the configuration file deals with the output location, the name of the index created and the endpoint address where *Elasticsearch* is running.

```
output {
    stdout { codec => rubydebug
```

```
        }
    elasticsearch {
        action => "index"
        hosts => ["127.0.0.1:9200"]
        index => "anime"
        document_id => "%{anime_id}"
        workers => 1
    }
}
```

2.3 Queries

One of the objectives of this milestone was to retrieve the indexed information. The web interface and management functionalities of *Kibana* were used to help see the data more clearly, formulate the queries and validate the results provided. *Kibana* provides a graphical interface that sends *HTTP* requests to *Elasticsearch*, where the data is stored. We were able to simulate a user interface and query the information retrieval tool using its search capabilities to get the most relevant results in the collection, regarding the user's information need.

Elasticsearch makes full-text querying much easier, using a *JSON* format query where the parameters are detailed, including the text to query, the text fields where we want to search in, the number of results desired, the fields shown in the responses, among other details. The relevance of each field can be defined there as well. Several queries were done to demonstrate the capabilities of the information retrieval tool chosen for this milestone. The fields to search are adjusted regarding the information need for each query context, with the intent of retrieving the best results possible (with the highest relevance to the user). Some fields' score is boosted according to their relevance to the overall results. The types of queries we found most useful will be described in the following sections.

2.3.1 *Bool* Query

This query matches documents matching boolean combinations of other queries. It is built using one or more boolean clauses, each clause with a certain type of occurrence, defined in the query. The *bool* query takes a more-matches-is-better approach, so a document that matches the searched query in more than one field will have a better score.

Bool queries are useful when we wanted to use different techniques to match queries and score returned documents. For example, in the general matching query we created, we used two different *multi_match* queries connected by a *bool* query. In one of the *multi_match* queries we used *best_fields* to score the document equal to the score of the field that *Elasticsearch* best scored. *best_fields* analyzes each term of the query individually, that means that "Attack on Titan" is interpreted as *"Attack" OR "on" OR "Titan"*. This means that *Elasticsearch* might match documents that don't represent the "Attack on Titan" *anime* and rank them quite high. One way to circumvent this was to use *multi_match* with phrase type parameter. In this mode, *Elasticsearch* only matches documents that contain the whole phrase "Attack on Titan". Additionally, we boosted the score of matches with this query, since we believe it would be more relevant to the user to find exactly what they're searching for. *Bool* queries are also used with the filter operator because it was the simplest way to use this operator.

2.3.2 Multi-Match Query

The multi-match query is built on the match query to allow multi-field queries. The match queries accept text/numerics/dates, analyzes them, and constructs a query. It is of type boolean so the text provided is analyzed and the analysis process constructs a boolean query from the provided text. This type of query allows the appliance of the fuzziness method which allows fuzzy matching based on the type of field being queried, making the results obtained more precise and tolerating some poorly written query provided by the user or matching

with words close to the ones used by the user. If the best field parameter is used on multi-match queries, the document chosen is the one with the field that obtained a better score. In this type of queries, it's possible to give a different score weight to the fields used for the query match, having a better control over the relevancy of a document.

If the value for the type parameter is phrase instead of best field then the system will try to match the whole query's text to the fields specified. This can be useful because it might be more relevant to find the whole query instead of the individual words the query is composed of.

Multi-match is used in the general matching query to make the query shorter and easier to read. We give different boosts relating to their potential to be relevant to the user's information need.

2.3.3 Common Terms Query

The common terms query is an alternative to stopwords, improving both precision and recall of results (by taking stopwords into account), without sacrificing the query's performance. It divides terms in two groups: high frequency terms and low frequency terms. It first searches for documents which match the more important terms, the ones that appear in fewer documents and have a greater impact on the document relevance. Then, it searches in the previously selected documents for the less important terms, which appear frequently and have a low impact on relevance, improving relevance calculation without paying the cost of poor performance.

This is useful in our case when matching the query in synopses as sometimes the query can contain stopwords that are common in most *anime*' synopse. Without this operator, many *anime* would show up with high scores even though they might be irrelevant to the user's information need.

2.3.4 Function-score Query

The *function_score* allows modifying the score of documents that are retrieved by a query. It computes a new score for each document retrieved by a specified query. One example of use for this modifier is to attribute higher scores to documents that represent *anime* with higher level of popularity. This will make so that higher ranking documents will more likely be relevant to the user as one person will more likely look for a popular *anime*.

2.4 Tool Evaluation

Elasticsearch was found easy to use as expected, with the help of *Logstash* for the import of data and *Kibana* for querying the information retrieval tool and visualization of results. The in-depth detailed documentation provided on the tool's website [8] was enough to understand the core functionalities and existing types of queries, although the differences between them and context where to use each was a little hard to figure out. The *IR* tool's main functionalities were successfully explored in this milestone and the results obtained were suitable to the context of this milestone.

The main difficulty resided in generating a data file, in this case a *CSV* file, that would be correctly interpreted by *Elasticsearch*. Double quotes used to either delimit fields or as content of the fields were being mixed together and *Elasticsearch* would sometimes complain that a field was being incorrectly delimited, among other issues. *SQL Server* would export the data in an incorrect format which *Elasticsearch* wouldn't correctly parse. Some extensive effort was necessary to cover all the issues introduced by this problem.

3 Conclusion

For this milestone it was required to look for appropriate information retrieval tools and determine the one that was most suitable for this milestone.

After experimenting with different information retrieval tools and evaluating the advantages of each one, we chose the one that was best suitable for the purpose of this project. *Elasticsearch*, along with *Logstash* and *Kibana*, pro-

vided the functionalities required and proved to be a good tool for information retrieval, working like a search engine with the *Kibana* interface. Several types of queries that were found to be more convenient to our dataset were tested and the overall results obtained were satisfactory.

References

[1] *Anime - Wikipedia*, https://en.wikipedia.org/wiki/Anime, 2018. [Online, accessed 13-October-2018]

[2] *What is Anime? A Brief History of Anime Genres, Culture and Evolution - The Daily Dot* https://www.dailydot.com/parsec/what-is-anime/, 2018. [Online, accessed 14-October-2018].

[3] *A Brief History of Anime - Thought Co*, https://www.thoughtco.com/brief-history-of-anime-144979, 2018. [Online, accessed 14-October-2018].

[4] *History of Anime - Wikipedia*, https://en.wikipedia.org/wiki/History_of_anime, 2018. [Online, accessed 13-October-2018].

[5] *Logstash: Collect, Parse, Transform Logs Elastic*, https://www.elastic.co/products/logstash, 2018. [Online, accessed 20-November-2018].

[6] *Kibana: Explore, Visualize, Discover Data Elastic.*, https://www.elastic.co/products/kibana, 2018. [Online, accessed 21-November-2018].

[7] *Open Source Search & Analytics, Elasticsearch* , https://www.elastic.co/, 2018. [Online, accessed 15-November-2018].

[8] *Elastic Stack and Product Documentation Elastic*, https://www.elastic.co/guide/index.html, 2018. [Online, accessed 17-November-2018].

4 Appendix

4.1 Queries

In the report for the first milestone, we presented some questions that we would like to try and answer using the information retrieval tool in the second milestone. We present here queries that would attempt to answer those questions.

```
GET anime/_search
{
  "size" : 10,
  "_source": ["title","title_english","title_japanese","title_synonyms
     ","synopse","rating"],
  "query":
  {
    "function_score":
    {
      "query":
      {
        "bool":
          {
            "should":
            [
              {
                "multi_match":
                {
                  "type": "phrase",
                  "query": "Attack on Titan",
                  "fields": ["title^10", "title_english^10", "
                     title_japanese^10", "title_synonyms^9", "synopse^
                     5", "genre"],
                  "boost":3
                }
              },
              {
                "multi_match":
                {
                  "type": "best_fields",
                  "query": "Attack on Titan",
                  "fields": ["title^10", "title_english^10", "
                     title_japanese^10", "title_synonyms^9", "synopse^
                     5", "genre"]
                }
              }
            ]
          }
      },
      "field_value_factor":
      {
```

```
37          "field": "scored_by",
38          "modifier": "log2p",
39          "missing" : 1
40        }
41      }
42
43    }
44  }
```

Query 1 - This query was created with the idea of being used as the main search mechanism in a hypothetical application. The query the user would input is "Attack on Titan" for this example, which is the name of an *anime*. As discussed earlier there is a bigger emphasis in matching the whole phrase rather than only parts of the query, evidenced by the "boost" parameter within the "multi_match" query type with the "phrase" type parameter. The query is also embedded within a "function_score" block in order to use the "field_value_factor" operator which takes the documents from the query and transforms the score value using the value of the "scored_by" field, which represents how many user gave a score to each*anime* related to the documents retrieved. The value of the field gives an idea of the popularity of the *anime*. We believe the modifying the score of the document using this method will increase the rank of more relevant documents.

"What is the *anime* with the id 123?"

```
1 GET /anime/doc/123
```

Query 2 - This one just retrieves a particular document in the collection.

"What are the *anime* with an attribute with a particular value? (e.g. Rating as PG-13)"

```
1  GET /anime/_search
2  {
3    "query":
4    {
5      "match":
6      {
7        "age_rating": "PG-13"
8      }
9    }
10 }
```

Query 3 - This query is to show a simple example of retrieving *anime* by the value of a particular field, in this case the age rating.

"What are the licensors for this *anime*?"

```
1  GET /anime/_search
2  {
3    "_source": ["licensor"],
4    "query":
5    {
6      "term":
7      {
```

```
      "anime_id": 99
    }
  }
}
```

Query 4 - This query will only result the licensors of the *anime*, by giving the value "["licensor"]" to the "_source" field of the query.

"How many comedy *anime* has a certain studio made?"

```
GET /anime/doc/_count
{
  "query":
  {
    "bool":
    {
      "filter":
      [
        {
          "match":
          {
            "genre": "Comedy"
          }
        },
        {
          "match":
          {
            "studio": "Sunrise"
          }
        }
      ]
    }
  }
}
```

Query 5 - In this query the bool query is used in order to then invoke the filter operator to just retrieve the documents that has "Comedy" as a genre and "Sunrise" as a studio.

"What is the most popular *anime* of a certain studio?"

```
GET /anime/_search
{
  "query":
  {
    "bool":
    {
      "filter":
      [
        {
          "match":
```

```
            {
              "studio": "Sunrise"
            }
          }
        ]
      }
    },
    "sort":
    [
      {
        "scored_by":
        {
          "order": "desc"
        }
      }
    ],
    "size": 1
}
```

Query 6 - In this query, only documents that have "Sunrise" as a studio are retrieved. But then they are ordered by the value of their "scored_by" field in descending order. Then only one document is returned by the query. This effectively returns the *anime* from Sunrise that has the highest number in the "scored_by" field.

"What is the average score of the anime of a particular licensor?"

```
GET /anime/_search?size=0
{
    "aggs" : {
        "anime" : {
            "filter" : { "match": { "licensor": "Nozomi Entertainment"
                } },
            "aggs" : {
                "avg_score" : { "avg" : { "field" : "score" } }
            }
        }
    }
}
```

Query 7 - This query aggregates *anime* resulting from using a filter that only retrieves *anime* that have "Nozomi Entertainment" as a licensor, then creates a statistic about this aggregation which is the average value of the "score" field of each document belonging to that filter.