Book Recommendation Engine

Shibli, Ali Kazzi, Mark Antoni Widman, Carl Johan
Udapudi, Aditya Matinzadeh, Keivan

May 11, 2021

Group 17 Topic 6

Abstract

In this project, we present a complete pipeline for books recommendation. Given a query from the user, the recommendation system finds the books that best match this query, based on the books abstracts, tags, comments and reviews, and other metrics. This process is divided into several steps. First, we collect more than 20 thousand books manually scraped from GoodReads website to form our dataset. Then, we preprocess the dataset and use Elasticsearch for both storing and processing of the data. Finally, using search and ranking algorithms, we retrieve and score the results, resulting in an engine that produces relatively good results based on the evaluation done. This implementation can therefore be used in a larger extent on bigger datasets to provide the user with book recommendations.

1 Background/Introduction

When searching for the next book to read, it can be hard to navigate the vast variate of books and authors to make a decision. Decisions might be made based on the opinions of the vast majority presented by newspapers like *The New York Times* or the recommendation of a friend. However, one might have some preferences that doesn't match the opinions of the majority which can result in the recommendation of an uninteresting book. A solution to this is to create a recommendation engine that have a collection of various books and can give a recommendation based on various variables. This has been done previously by websites such as *whatshouldireadnext* (WSIRN) and *goodreads*. WSIRN uses mass opinions to and recommends books based on collective taste. Taking users favorite lists and creating associations with the books and providing recommendations based on the associations [1]. Goodreads uses their database of around 500 million books and combines multiple algorithms to provide a recommendation of a book [2].

The purpose of this project is to design an engine that recommends books to the user based on input in the form of query searches such as authors name, title of a book or tags related to a book similar to WSIRN. However, instead of associations this report uses different algorithms based on the input to rank books and output a list of recommended books. For the base of the data, GoodReads was used as the source given there vast collection of books. When constructing the engine *Elasticsearch* was used for both the structure of the data and use of algorithms. The aim is to provide a recommendation engine that can provide good results of books based on evaluations made on the output.

2 Previous/Related Work

2.1 Elasticsearch

Elasticsearch is a search and analytics engine used for full-text search, structured search, analytics and a combination of all three [3]. The engine stands out by implementing a unique combination of full text search, analytics systems, and distributed databases.

It is built on top of an advanced full-text search engine library called Apache Lucene. Elasticsearch uses Lucene internally for its indexing and searching, while hiding library's complexities behind an simple and coherent API.

2.2 Text Processing

In order to analyze relevant information the system must fist tokenize the data, which is breaking it up in to separate words called tokens. This enables full-text analysis with queries that match individual words to information. Furthermore, the separate tokens must be normalized in order to minimize complexity and help with processing of text. In order to match variations of words, stemming is used to handle prefixes and suffixes by

reduce the word to its standard form.

2.2.1 Unicode Text Segmentation algorithm

The tokenizer used for the index is the *Standard Toknizer* provided in Elasticsearch since it is stated that it is the most suited choice of tokenizer for most languages. It uses the *Unicode Text Segmentation algorithm (UTS)* by Mark Davis and Christopher Chapman [4]. This tokenizer inputs a flow of characters and forms separate words and outputs a stream of characters removing any unnecessary characters like white space. It also records the position and order of tokens. UTS uses word boundaries to distinguish words from other text to allow for easy extraction providing grammar based tokenization.

2.2.2 The Porter stemming algorithm

The algorithm breaks down letters in to constants and vowels. It then breaks down words to only be denoted as constants (C) or vowels (V). Using the following formula $[C](VC^m)[V]$ words are then measured by the repetitious of vowels and constants (VC^m) and denoted as m. Some examples of this:

m=0 TR,EE,TREE,Y,BY.

m=1 TROUBLE,OATS,TREES,IVY.

m=2 TROUBLES,PRIVATE,OATEN,ORRERY.

When removing the suffix of a word the following rule is used: (condition) S1 -> S2.Where S1 and S2 denotes the suffix of a word. When the stem, meaning the word before S1, meets the condition attached to it then S1 is replaced by S2. Where the condition is based on the value of m attached to the suffix.

(m > 1) EMENT -> null S1 = EMENT S2 = null, where m = 0 represents the null values.

Then when faced with REPLACEMENT the condition will then turn it to REPLACE since REPLACE is part of m = 2 [5].

2.2.3 Removal of stop words

To minimize the impact of common words in the data, stop token filtering was used. It utilizes Lucene's Stop Filter [6] and removes certain words from the data that are present in a list of stop words. This packages contains many different lists of words from varying languages.

2.3 Scoring

2.3.1 TF-IDF

The idea of the TF-IDF scoring is to first represent each document as a vector. Then, in the vector space, the documents with similar vectors have similar content [7].

For a document, each vector element is calculated as a combination of term frequency, $TF(w, d)$, and inverse document frequency $IDF(w)$ where w is a word. The term frequency of a word w in a document d is the number of times w occurs in d. The document frequency $DF(w)$ of a word w is the number of documents w appears in at least once. The inverse document frequency can then be calculated from the document frequency according to

$$IDF(w) = \log \frac{|D|}{DF(w)}$$

where $|D|$ is the total number of documents.

The more documents a word appears in the lower its IDF value will be.

The combination of $TF(w_i, d)$ and $IDF(w_i)$ can then be expressed as

$$d^{(i)} = TF(w_i, d) \cdot IDF(w_i)$$

where $d^{(i)}$ represents the value of a feature w_i for document d.

According to this heuristic, the highest value would then be for every word in document d to be word w_i, while not occurring in any other document except in d.

2.3.2 Vector space model

The vector space model is a way to model information retrieval objects as elements of a vector space [8]. Terms, documents, queries etc. can all be represented as vectors in the vector space. Since linear operations can be applied to vectors in a vector space, this opens up the possibility of using linear operations on information retrieval objects, e.g. summing two documents.

A common use of the vector space model to compute the similarity of two documents represented as vectors, where such a measure could be the inner product between vectors [9], or the angle between two vectors on the base of the cosine, called the cosine similarity [10].

2.3.3 The Practical Scoring Function

When searching with Elasticsearch, Lucene handles multi-term queries with a formula called *The practical Scoring Function*. It combines the Boolean model, TF-IDF, and the vector space model to collect and score documents [11].

The query

```
GET /my\_index/doc/\_search
{
    "query": {
    "match": {
      "text": "quick fox"
    }
  }
}
```

Listing 1: Example from [11]

gets rewritten to look like this:

```
GET /my_index/doc/_search
{
  "query": {
    "bool": {
      "should": [
        {"term": { "text": "quick" }},
        {"term": { "text": "fox"   }}
      ]
    }
  }
}
```

Listing 2: Example from [11]

In this example only documents containing the term quick, or the term fox, or both.

When a document matches a query, the score is calculated and the score of each matching term is combined. This is done by the practical scoring function:

```
score(q,d)  =
            queryNorm(q)
             coord(q,d)
                 (
               tf(t in d)
                idf(t)^2
                t.getBoost()
                norm(t,d)
            ) (t in q)
```

Listing 3: Example from [11]

where

- $queryNorm(q)$ is the *query normalization factor* which tries to normalize a query making the results from one query comparable with the results of another. It is calculated as inverse of the squared sum of the IDF of each term of the query.
- $coord(q, d)$ is the *coordination factor* which rewards documents containing a higher percentage of the query terms. Instead of only just adding the scores of the query terms that appear in the document, the score is then also multiplied by the number of matching terms in the document divided by the total number of terms in the

query. If a document contains all query terms, then the score will be multiplied by 1.

- $tf(tind)$ is the term frequency of term t in document d.
- $idf(t)$ is the inverse document frequency of term t.
- $getBoost()$ returns any boost values applied to the query or the term itself. Boost values are used to give one clause more importance than others.
- $norm(t, d)$ is the *field length norm*. It is the inverse squared sum of all terms in the field. Thus, a high weight indicates a short field. A high weight implies that the field is more about the term, compared to a lower weight which would imply that the field is not only about that term but also other terms.

  This is combined with index-time field-level boosting (if specified).

2.4 GoodReads

Goodreads is one of the most extensive databases for books on the internet. It contains over 500 million books [12] information from abstract, author, tags (or genres) of the book, comments and reviews, and much more. For the purposes of our project, we develop a search (or recommendations) engine for books starting the GoodReads books. In the section for Data Retrieval 3.1.1, we elaborate in depth of the structure followed to scrape more than 20 thousand books info from GoodReads as our dataset for the project.

3 Method

3.1 Data

3.1.1 Data Retrieval

GoodReads is divided into subpages, were each page contains references to other pages. For example, the main page of GoodReads contains over 30 tags that we use as our references tags for crawling the website. Next, we collect the top 100 lists of books that come from each tag (e.g., "Best Book Boyfriends", "All Time Favorite Romance Novels", and others for "Romance" tag). From each of these lists, we scrape the top 100 books respectively after collecting their urls. We use python with Selenium to perform this task running on google Colab.
Due to a limit on the number of requests per second, we divide the dataset into batches which we collect in parallel, to reach a total number of around 21052 books. In brief, these are the steps followed:

1. Collect the top 30 book tags from the website

2. Collect the urls for the big lists of books from the tags
3. Iterate over each of these urls, and collect the urls for the books inside each of these lists
4. Divide these urls into k batches each containing $21052/k$ books
5. Iterate through these url batches and scrape the book info for from each url

3.1.2 Data Structure

Data in Elasticsearch is stored in one or more indices, where an index is similiar to what is known as a relational database. The main entities are documents, which from the client point of view are JSON objects. Each document also has a type [13].

The documents are stored in an index called "books" by reading from a .json file. The JSON file contains the information scraped from GoodReads converted into JSON objects. Each JSON object describes a book and has nine fields;

1. book_id - *The id of the book*
2. name - *The name of the book*
3. abstract - *The description of the book*
4. author - *The author of the book*
5. average_rating - *The average rating of the book, meaning the sum of the ratings divided by the number of ratings*
6. number_of_ratings - *The number of ratings*
7. tags - *The genre(s) the book belongs to*
8. comments - *the top three comments left by readers of the book*
9. url - *the url linking to the GoodRead page of the book*

3.2 The Engine

The search engine proposed is heavy centered around Elasticsearch and its API for searching the indices. The engine mainly consist of three parts: the scraper, the reader and the searcher. The job of the scraper is to scrape the goodreads website for information about books. This part is only ran once, but can of course be modified to extract other information or to update the data. The data is outputted as a .csv file containing information about around 21 000 books.

The reader then converts the .csv file to a .json in order to facilitate the indexing of the data using the Elasticsearch API. It then creates the index and specifies that the index

should use all of the text analysis that is presented in section 2.2. Worth noting here is that we apply stemming and removal of stop words for the English language although there are books that are not written in English in the data. However, they make up a very small part of the data and the benefit from stemming and the removal of stop words greatly increase the performance of the search engine.

Lastly the searcher, which will be further explained in the following section (section **??**).

3.2.1 The Searcher

The searcher works twofold. Firstly it the program begins by reading the list of books that the user has read in the past, if provided in the arguments. Each time the index is searched, the query gets tokenized (section 2.2.1), the punctuation is removed, all tokens are lower cased, English stop words are removed and stemming is performed on the tokens according to the English language using the Porter Stemming Algorithm (section 2.2.2). It is important that the same text analysis is performed both when indexing the data and when the index is queried.

The index is searched with the titles of the books in the file and if they appear in the index, the author, genres and abstract of that book is obtained. The index is then queried with all of these different attributes and scores the results. The index is then searched for with the author, the abstract and the genres of that book in order to find similar books. These are then stored in the searcher. When the author is searched for, the engine performs a *match_phase* query in order do only provide exact matches on the authors name. This is due to the fact that two books are generally not similar because their authors have a similar name. However, it is fairly likely that a user is going to enjoy another title of their favourite author. This comes with the caveat that other authors can share the name of the searched author. Though, the likelihood of this is fairly small and the results of this error should be mitigated by the ranking algorithm (section 3.2.2).

The second part of the searcher takes into account the actual query provided by the user. In order to retrieve the most amount of information from the query and also to improve the flexibility in writing the query, the same query is used to search the fields "name", "author", "tags" and "abstract". This means that the user can combine searching for an authors name, within a certain genre and with some keywords from the abstract in a single query without having to specify which words corresponds to what field or any other particular order. Each of these searches returns a set of results that all contribute the final result to different extents, as described in section 3.2.2.

Lastly the top results are reiterated through the searcher in order to boost similar books.

3.2.2 Ranking/Weighing

In order to evaluate whether the received results from the searches in Elasticsearch are relevant or not, the API provides a scoring of each of the retrieved results. When the size of the retrieved results of a query is limited, the API will sort the result and send back the

top chunk. This internal scoring is calculated using a version of TF-IDF (section 2.3.1) and Vector Space Model (section 2.3.2) for multi-word queries [14]. The end result is a positive score that represent how similar that documents specified field is to the query. The score is not the same for different types of queries so the internal score is normalized (called "es_score" below). If the searcher retrieves a book that is already in the result list, the scores get multiplied in order to favour books that partly matches the query on many fields or partly matches fields from the query and also is similar to a book that the user has enjoyed.

Our implemented scoring is calculated by:

$$score = es\_score \cdot weight \cdot average\_user\_ratings\_of\_book \cdot number\_of\_ratings^{1/8}$$

The reasoning behind the formula is to promote popular books in order to retrieve books that also the generic user should probably like. The "average_user_ratings_of_books" is bounded between 0 and 5 where 5 is the best score. The "number_of_ratings" is unbounded, so in order to decrease the contribution of very high number_of_ratings while very few number_of_ratings should be punished, the metric is taken to the power of 1/8. The weight was added in order to weigh fields differently to experiment with the results. In the end a weight of 1 was applied for all fields.

3.3 Evaluation Method

The searcher (as mentioned in section 3.2.1) has two arguments, namely the *list of books* that a user likes and a *query* which can both be exploited and experimented with. We have chosen the measures **precision** and **recall** to evaluate our search engines for the following test cases we have chosen. The experiment was conducted by a user who evaluated the resulting books in a binary fashion - 1 for a relevant book recommendation and 0 for a non-relevant recommendation.

Precision is the fraction of returned results that are relevant to the information needed and *Recall* is the fraction of relevant documents in the collection which are returned by the search engine. For recall, we have assumed relevant documents up to a scale factor to be 50 books. For both of the measures the engine returned 20 books.

4 Experiments/Results

1. Test Case: A Game of Thrones in *my_favourite_books.txt* and No Query added:

$$Precision = 80\%$$

$$Recall = 32\%$$

2. Test Case: No book(s) in *my_favourite_books.txt* and Query "mystery and crime" added:

$$Precision = 90\%$$

$$Recall = 36\%$$

3. Test Case: The Complete Sherlock Holmes in *my_favourite_books.txt* and Query "mystery and crime" added:

$$Precision = 95\%$$

$$Recall = 38\%$$

4. Test Case: The fault in our stars, Beautiful disaster, Twilight in *my_favourite_books.txt* and Query "plot twist" added:

$$Precision = 85\%$$

$$Recall = 34\%$$

5 Conclusions/Discussion

The first part of the searcher, where the engine finds similar books is quite straight forward. This could have been done by comparing the retrieved results to the goodreads list of similar books. The query part is of a much more subjective nature and would require many users to experiment with the engine and qualitatively evaluate the retrieved book recommendations. These evaluation steps are subjects for future research.

Another subject for future research is whether a collaborative filtering approach could increase the performance of the engine further. In that case a combination of the content filtering approach presented in this report could incorporate a new dimension to the search results. The data on other users is available through the goodreads website, however it would have increased the scope of this report too extensively.

There is a stable and consistent trend that can be observed in the precision and recall values. This means that the search engine is performing fairly well in different scenarios and hence a reliable one. Though, if we were to put this engine through an optimization regarding both the scoring algorithm when weighing of the different fields (section 3.2.2), the engine could definitely provide the user with some helpful book recommendations - which was the intent of this report.

References

[1] *what should i read next*$_2$021. 2021. URL: https://www.whatshouldireadnext.com/faq.

[2] *announcing goodreads personalized recommendations*$_2$011. 2011. URL: https://www.goodreads.com/blog/show/303-announcing-goodreads-personalized-recommendations.

[3] Clinton Gormley and Zachary Tong. *Elasticsearch: the definitive guide: a distributed real-time search and analytics engine*. " O'Reilly Media, Inc.", 2015. URL: https://books.google.se/books?hl=sv&lr=&id=d19aBgAAQBAJ&oi=fnd&pg=PR3&dq=elasticsearch&ots=NAeTuQGubG&sig=pfIhRGNwfAsBdYFOK8EHdP6nIhw&redir_esc=y#v=onepage&q=elasticsearch&f=false.

[4] 2020. URL: https://unicode.org/reports/tr29/.

[5] M.F. Porter. "An algorithm for suffix stripping". In: *Program* 14.3 (1980), pp. 130–137. DOI: 10.1108/eb046814.

[6] Jan. 2021. URL: https://lucene.apache.org/core/8_8_0/core/org/apache/lucene/analysis/StopFilter.html.

[7] Thorsten Joachims. *A Probabilistic Analysis of the Rocchio Algorithm with TFIDF for Text Categorization*. Tech. rep. Carnegie-mellon univ pittsburgh pa dept of computer science, 1996.

[8] Vijay V Raghavan and SK Michael Wong. "A critical analysis of vector space model for information retrieval". In: *Journal of the American Society for information Science* 37.5 (1986), pp. 279–287.

[9] Gerard Salton, Anita Wong, and Chung-Shu Yang. "A vector space model for automatic indexing". In: *Communications of the ACM* 18.11 (1975), pp. 613–620. URL: https://dl.acm.org/doi/abs/10.1145/361219.361220.

[10] Tuomo Korenius, Jorma Laurikkala, and Martti Juhola. "On principal component analysis, cosine and Euclidean measures in information retrieval". In: *Information Sciences* 177.22 (2007), pp. 4893–4905. ISSN: 0020-0255. DOI: https://doi.org/10.1016/j.ins.2007.05.027. URL: https://www.sciencedirect.com/science/article/pii/S0020025507002630.

[11] *Lucene's Practical Scoring Function*. URL: https://www.elastic.co/guide/en/elasticsearch/guide/current/practical-scoring-function.html.

[12] *Goodreads*. May 2021. URL: https://en.wikipedia.org/wiki/Goodreads.

[13] Rafał Kuć and Marek Rogoziński. *Mastering ElasticSearch*. Packt Publishing Ltd, 2013. URL: https://books.google.se/books?hl=sv&lr=&id=NRO8AQAAQBAJ&oi=fnd&pg=PT19&dq=elasticsearch&ots=98mrEn-Otx&ig=Jhzc7h3qbjqR9Runz5_jCkfE11w&redir_esc=y#v=onepage&q=elasticsearch&f=false.

[14] Lisa Smith. *How scoring works in Elasticsearch*. 2016. URL: https://www.compose.com/articles/how-scoring-works-in-elasticsearch/.