Prjoect Report - Spotify Podcast Search Engine

DD2477 Search Engines and Information Retrieval

Shweta Misra
Yixiong Wang
Zakaria Nassreddine

Abstract

The Spotify Podcast Dataset is a collection consisting of 100,000 podcast episodes, and 50,000 hours of audio and accompanying transcripts. In this project, we designed and implemented a search engine based on Elasticsearch to perform efficient retrieval on this dataset. We use a combination method using Elasticsearch capabilities to search through podcast episodes and tf-idf similarity scoring to find relevant 2-minute clips from the episode. To improve user experience, we built the graphical user interface with React. Experiments with different queries and search fields suggested excellent performance of our search engine under various evaluation metrics. The search engine retrieves relevant clips of podcast for queries for length 1 to 4 in under 3 seconds.

School of EECS
KTH, Stockholm
8 May 2022

Contents

1 INTRODUCTION **2**

2 RELATED WORK **2**

a) Indexing Method: Elastic Search 2

b) Ranking Algorithms 2

3 METHOD **3**

a) Overview 3

b) Back-end Processing 4

c) Front-end Application 5

d) Dataset 5

4 EXPERIMENTS and RESULTS **6**

5 CONCLUSION and FUTURE SCOPE **6**

1 INTRODUCTION

Information Retrieval is finding material (usually documents) of an unstructured nature (usually text) that satisfies an information need from within large collections (usually stored on computers)[6]. An information retrieval begins when a user inputs a query, and it ends with finding and ranking all the relevant documents. The information retrieval system contains a corpus which is the collection of all the documents. Indexer creates an inverted index that contains a list of words and documents and it is responsible for managing the indexing, and a query parsing system which pre-processes user's queries and does the matching between it and the system query. Besides, a ranking system calculates the relevant score according to the user's query for each of the result that is returned by the search engine and ranks them. After these several processes result is displayed.
In this project, we implemented a search engine that could find podcasts with specific contents and play the most relevant clips that were ranked and extracted from different episodes. We built this system based on The Spotify Podcast Dataset[1], and implemented the Graphical user interface based on React[9]. A series of experiments showed that our search engine achieved good performance.

2 RELATED WORK

a) Indexing Method: Elastic Search

Elasticsearch is a distributed, RESTful search and analytics engine that centrally stores your data for lightning fast search, fine-tuned relevancy, and powerful analytics that scale with ease [3]. It is essentially an ecosystem comprising of a database, a search engine and a visualization tool, 'Kibanna' for analytical purposes. It possesses the ability to store a large number of documents into one or more indices. Each index is analogous to tables in a traditional SQL database while a document represents a record/row in the table. Major components of the Elastic stack include:

- **Index**: An 'index' in Elasticsearch is actually an 'inverted index' (or 'postings-list'). An inverted index stores a mapping from the contents of a set documents to their location those documents. In the case of Elasticsearch, an index stores the location of each word in the indexed documents. This is called a word-level index. Inverted indices are used for fast retrieval of relevant documents by matching words that are being looked-up. To make the process of storing inverted indices efficient, Elasticsearch assigns a unique ID to each document that is indexed.
- **Elastic Server**: A node in Elasticsearch is the server that stores the data and inverted indices. Since Elasticsearch is distributed it is organised as a group of nodes called a **cluster**. The indices are divided into shards and each shard can be stores on any node of the cluster. The user may create replicas of these shards that are also distributed over the various nodes. This ensures redundancy and protection against hardware failures. Hence providing interrupted service to search queries triggered by an Elasticsearch client.

b) Ranking Algorithms

In Retrieval and ranking process, queries are retrieved and ranked according to their relevancy.There are various models and approaches to weigh terms and documents in a corpus.
Information models define the way to represent the documents as well as the query. Similar generation phases are applied to both the documents in the corpus and the queries, after which the similarity between documents and queries are calculated to do the relevant information retrieval. The output documents are

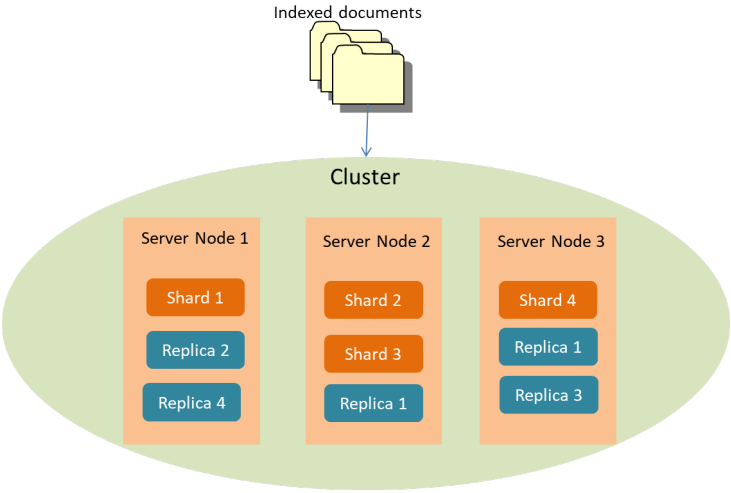

Figure 1: ElasticSearch Index Management.

ranked according to this similarity.
One typical information model is the Term Frequency–Inverse Document Frequency (**TF-IDF**) 1, it aims to reflect how important a word is to a document in the corpus, by rewarding frequent words in one document and penalizing frequent words in the whole corpus.

$$\begin{aligned} tf\_idf_{dt} &= tf_{dt} \times idf_t / len_d \\ idf_t &= \log N / df_t \end{aligned} \tag{1}$$

Where tf_{dt} is the number of occurrences of term t in document d, N is the number of documents in the corpus, df_t is the number of documents in the corpus which contain t and len_d is the number of words in d.
If we know that some documents are relevant to the searching query, we can make use of these documents to expand and improve a query. A simplified version of Rocchio feedback algorithm can be formulated as[12]:

$$\vec{q}_{new} = \vec{q}_{old} + \alpha \vec{d}_R - \beta \vec{d}_{\bar{R}} \tag{2}$$

Where $\vec{d}_R$ is the centroid vector (the average) of all weighted documents in R (the relevant documents), and $\vec{d}_{\bar{R}}$ in $\bar{R}$ (the non-relevant documents).
In [11], Su et al. proposed to use machine learning approaches to train ranking systems.The features consists of keywords of different attributes in a HTML file. They labelled the ranking orders and trained the machine learning model to learn the potential weighted matrix to map the pages into ranking orders.

3 METHOD

a) Overview

There are three main components in the ecosystem of our implementation of the 'spotify podcasts search engine'. The first is a locally run Elasticsearch server which indexes and stores the contents of the metadata file that contains relevant information of each podcast episode. The metadata has been indexed into an index called 'spotify-metadata' which is a one-time task unless data for new episodes needs to be added. Each time our search engine is run a query is fired to the 'spotify-metadata' index which returns a list of relevant episodes using the inbuilt search() function in Elasticsearch. The second component is the search engine server that processes the user defined query, retrieves relevant episodes as mentioned in the previous sentence and finally finds appropriate 2 minute clips along with their transcripts from the

transcript data-set. This server is hosted using the globally available ngrok web service. The third main component of the eco-system is the search engine client which is built using React. It provides the user an interface to fire his or her query, view the results returned by the server and play chosen audio-clips. The figure below illustrates the interaction between the various components of the search engine and the major processing steps within each component.

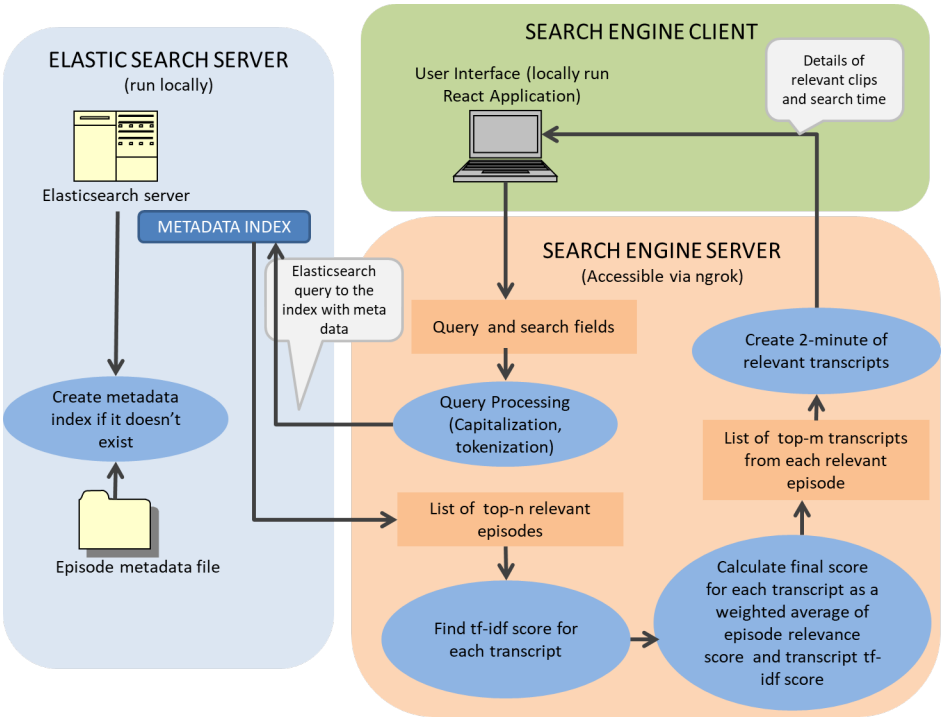

Figure 2: Spotify Podcasts Search Engine Eco-system

b) Back-end Processing

Indexing and search through Metadata using elastic search

When the user fires a query to the search engine, the first level of search that is conducted is on the metadata indexed on the Elasticsearch server. This search is done using a multi_match query that allows to search in multiple fields of an index. The fields to be searched are provided as input by the user when firing the query. The Elasticsearch server returns all the documents (in this case each line in the metadata file) that were a 'hit'. It also returns a relevance score for each document. Elasticsearch uses Lucene's Practical Scoring Function. This is a similarity model based on Term Frequency (tf) and Inverse Document Frequency (idf) that also uses the Vector Space Model (vsm) for multi-term queries [2]. We store a list of all relevant episodes that Elasticsearch returns along with other important information like show name and relevance score. This list if further processed to find relevant transcripts in the next step.

Retrieval of n-minute audio clips

The retrieval and ranking of n-minute audio clips happens a posteriori, after the search and ranking of relevant episodes is complete, after which we get the ranking and the score of each episode. We truncate the list to a top N subset where we only keep the N best ranked episodes as the intermediate corpus to start the post-processing that builds our n-minute clips. Note that this step of the search happens in real time after we get the intermediate results (episodes) upon querying the inverted index built on

the initial corpus, and its performance therefore decays with the size of the subset that we decide as a hyperparameter. We have decided this was the better compromise as opposed to indexing each episode in real time. With all the information that we get from the Elasticsearch results, we extract transcripts within each episode, calculate the TF-IDF score, rank them and make the n-minute clips accordingly.
To calculate the TF-IDF score, we retrieve all the episodes and create the word space gradually with each transcript in the episode. Then we can calculate the TF-IDF score for each word in the transcript according to Equ 1, which is then followed by calculating the cosine similarities between the query and each transcript, and rank the transcripts accordingly. One thing to mention is that to make sure the ranking of episodes counts more than that of the transcripts, we use a weighted score that considers the episode's ranking score as well as the similarity score for each transcript.

$$s_{i,j} = \alpha e_i + (1 - \alpha) sim_j \tag{3}$$

Where e_i is the score of episode i that contains transcript j, and sim_j is the similarity score between the query and j.
When making the clips from the transcripts, we start with the most relevant transcript, and repeatedly add the next one until the duration of the clip reaches n minutes. To avoid overlapping between clips that are extracted within the same episode, we assign unique id's to each transcript and add a new one to the clip unless the transcript is not visited.

c) Front-end Application

To facilitate the testing and improve the user experience, we built an application to act as a client on top of our search engine backend. The app was developed using React, an open-source JavaScript library for building user interfaces. Further documentation can be found on [9]. The backend was exposed as a Flask [4] REST [10] API endpoint. The React client communicates with the server through HTTP GET requests using the Fetch [5]. For cross-device testing, we used ngrok [7] tunnels to expose our local server ports to the internet which makes them accessible from outside of the developer's machine. This way, our React app can directly query our Flask endpoint through the public URL provided by ngrok. Since React runs a Node.js [8] server to build and serve the frontend app, the latter can also be accessed using an ngrok tunnel. These public redirections will be used for the live demo and can be requested from us on demand as the free service only provides tunnels with a lifespan of two hours.

d) Dataset

The Spotify Podcast Dataset[1] consists of 100,000 podcast episodes, and 50,000 hours of audio and accompanying transcripts. The episodes vary in length and include some very short trailers to advertise other content. For each episode, there is following metadata: show URI, episode URI, show name, episode name, show description, episode description, publisher, language, RSS link, and duration. In addition, there is the full RSS header for each show, from which additional metadata can be extracted.
To extract the link to the mp3 file that is attached to each episode, we parsed the xml file of each episode and obtained the url within the 'enclosure' attribute. We found that some of the xml files did not contain any url while some contained more than one, we did data cleaning in advance and used the first 'enclosure' attribute to get the link.

4 EXPERIMENTS and RESULTS

We tested our search engine against queries of varying length and over multiple combinations of columns in the metadata of podcasts. We calculated the normalized discounted cumulative gain (NDCG) for top 10 results of the queries 'Air Pollution', 'Christmas Food', 'World Leaders in Politics' and 'Natural Language Processing'. The obtained results are represented in the tables below. Another important metric to measure the performance of the search engine is the 'time' to complete the search and obtain the results. The time required to fetch the results is majorly dependent on the number of episodes considered to search for relevant transcripts. Our search engine currently looks for relevant transcripts in the top 10 episodes returned by the Elasticsearch query. However, we tested on top 10, top 50, top 100 and top 1000 episodes, see figure (3). The average time required for the above mentioned 4 queries with top 10 documents was 1.09 secs and with top 500 documents was 82.06 secs. The search time is also dependent on the transcript length of the relevant episodes and the processing resources available like CPU usage and traffic on the ngrok server.

Table 1: Evaluation metrics for the Query "Air Pollution" over all the fields in podcasts metadata.

Clip Rank	Relevance Score	CG	DCG	Ideal System Relevance Score	DCG Ideal System	NDCG
1	2	2	2	2	2	1
2	2	4	3.262	2	3.262	1
3	0	4	3.262	1	3.762	0.867
4	0	4	3.262	1	4.193	0.778
5	0	4	3.262	1	4.193	0.778
6	0	4	3.262	1	4.193	0.778
7	1	5	3.595	0	4.193	0.858
8	1	6	3.911	0	4.193	0.933
9	0	6	3.911	0	4.193	0.933
10	0	6	3.911	0	4.193	0.933

Table 2: Evaluation metrics for the Query "Christmas Food" over the fields of 'episode_name' and 'episode_description' in podcasts metadata.

Clip Rank	Relevance Score	CG	DCG	Ideal System Relevance Score	DCG Ideal System	NDCG
1	2	2	2	2	2	1
2	2	4	3.262	2	3.262	1
3	1	5	3.762	2	4.262	0.883
4	1	6	4.193	1	5.123	0.818
5	1	7	4.579	1	5.897	0.777
6	1	8	4.936	1	6.253	0.789
7	1	9	5.269	1	6.586	0.800
8	2	11	5.900	1	6.902	0.855
9	2	13	6.502	1	7.203	0.903
10	2	15	7.080	0	7.492	0.945

5 CONCLUSION and FUTURE SCOPE

We have designed a podcasts search engine that retrieves clips relevant to a user-defined query from multiple podcast episodes. Ranking of retrieved clips is based on a weighted average of the similarity score obtained by Elasticsearch for the respective episode and the tf-idf based similarity score for the first transcript that constitutes the retrieved clip.

Table 3: Evaluation metrics for the Query "World Leaders in Politics" over all the fields in podcasts metadata.

Clip Rank	Relevance Score	CG	DCG	Ideal System Relevance Score	DCG Ideal System	NDCG
1	2	2	2	2	2	1
2	2	4	3.262	2	3.262	1
3	0	4	3.262	1	3.762	0.867
4	1	5	3.693	1	4.193	0.881
5	0	5	3.693	1	4.579	0.806
6	0	5	3.693	0	4.579	0.806
7	1	6	4.026	0	4.579	0.879
8	0	6	4.026	0	4.579	0.879
9	0	6	4.026	0	4.579	0.879
10	1	7	4.315	0	4.579	0.942

Table 4: Evaluation metrics for the Query "Natural Language Processing" over the fields of 'episode_name' and 'episode_description' in podcasts metadata.

Clip Rank	Relevance Score	CG	DCG	Ideal System Relevance Score	DCG Ideal System	NDCG
1	1	1	1	2	2	0.500
2	2	3	2.262	2	3.262	0.693
3	0	3	2.262	2	4.262	0.531
4	0	3	2.262	1	4.693	0.482
5	0	3	2.262	0	4.693	0.482
6	0	3	2.262	0	4.693	0.482
7	2	5	2.929	0	4.693	0.624
8	2	7	3.559	0	4.693	0.759
9	0	7	3.559	0	4.693	0.759
10	0	7	3.559	0	4.693	0.759

Table 5: Query search time (in seconds) for varying number of relevant episodes (n)

Query	n=10	n=50	n=100	n=200	n=300	n=500
'Air Pollution'	0.40	1.30	3.12	33.21	48.28	63.13
'Christmas Food'	1.55	6.14	20.92	68.70	59.15	62.38
'world leaders in politics'	0.45	2.33	6.78	43.09	96.23	128.30
'Natural Language Processing'	1.96	8.61	22.32	49.23	103.3	82.06

Concluding from the NDCG values for the queries in the section above, the search engine provides very relevant results for about the top 2 when the number of relevant episodes searched is restricted to 10. The search with this parameter setting takes 1-2 secs to retrieve results. We can also see from the graph in the previous section that search time increases dramatically as the number of retrieved episode increases. If along with the metadata we also index the transcripts data using Elasticsearch we might see an improvement in performance. We could further improve the user experience by allowing the user to tweak parameters like the duration of retrieval clips and weight assigned to episode relevance versus transcript relevance at query time. As future enhancements relevance feedback could also be included in the searching algorithm.

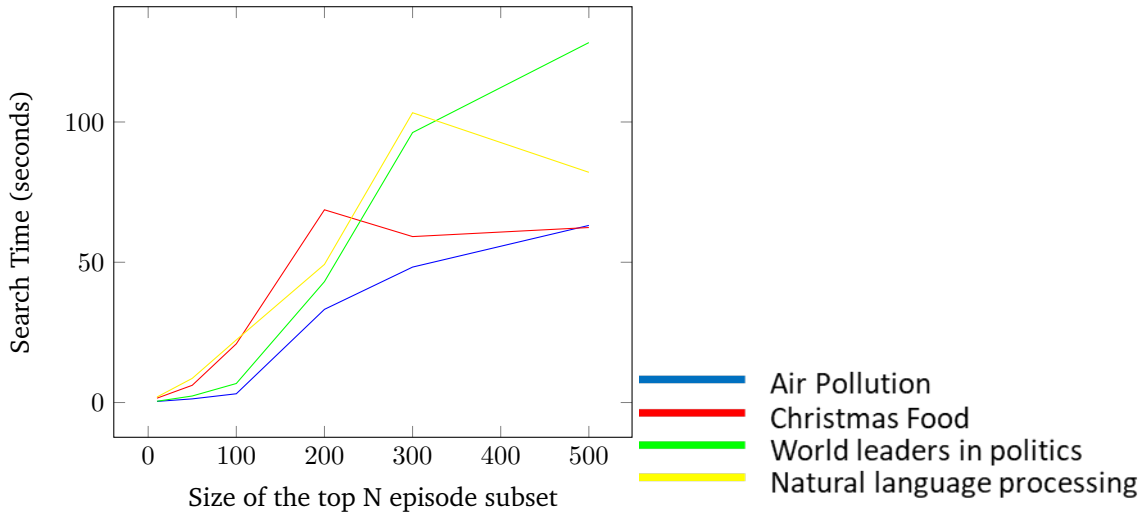

Figure 3: Search time w.r.t. retrieved episode corpus size

References

[1] Ann Clifton et al. "The Spotify Podcast Dataset". In: *arXiv preprint arXiv:2004.04270* (2020).

[2] Compose. *How scoring works in Elasticsearch*. `https://www.compose.com/articles/how-scoring-works-in-elasticsearch/`. [Online; accessed 7-May-2022]. 2016.

[3] Elastic. *Elasticsearch: Getting Started*. `https://www.elastic.co/webinars/getting-started-elasticsearch?ultron=getting-started-sitelink&blade=adwords-s&Device=c&thor=what%20is%20elastic%20search&gclid=CjwKCAjw682TBhATEiwA9crl35Q7lzAomZCuUz2yBOzUXFxbn-QWDp_8JJg32qRQR42jdj3Cfr12qxoCwvsQAvD_BwE`. [Online; accessed 5-May-2022]. 2022.

[4] *Fask: Web Development, One Drop at a Time*. `http://flask.pocoo.org`.

[5] *Fetch: Web Development, One Drop at a Time*. `https://developer.mozilla.org/en-US/docs/Web/API/Fetch_API`.

[6] Apra Mishra and Santosh Vishwakarma. "Analysis of tf-idf model and its variant for document retrieval". In: *2015 international conference on computational intelligence and communication networks (cicn)*. IEEE. 2015, pp. 772–776.

[7] *ngrok: The programmable network edge that adds connectivity, security, and observability to your apps with no code changes*. `https://ngrok.com/`.

[8] *node.js: A JavaScript runtime built on Chrome's V8 JavaScript engine*. `https://nodejs.org/en/`.

[9] *React: A JavaScript library for building user interfaces*. `https://reactjs.org/docs/getting-started.html`.

[10] *REST: REpresentational State Transfer*. `https://restfulapi.net`.

[11] Ao-Jan Su et al. "How to improve your search engine ranking: Myths and reality". In: *ACM Transactions on the Web (TWEB)* 8.2 (2014), pp. 1–25.

[12] Chengxiang Zhai. "Notes on the Lemur TFIDF model". In: *Unpublished report* (2001).