Information Processing and Retrieval Project

IMDb Data

Gonçalo Teixeira, Pedro Pinto, Pedro Azevedo
Faculdade de Engenharia, Porto

ABSTRACT

Information has never been more important and has never been more available as well. This means that the processing of acquired data and the retrieval of lost information is as crucial as never before. This report intends to document the process of creation of a good database and, with more detail, the creation of an information processing and retrieval tool to be used in that same database.

KEYWORDS

Information Processing and Retrieval, datasets, data, IMDb, movie's data, statistics, Solr, indexing, collections, documents, metrics

1 INTRODUCTION

There is a considerably large amount of data out there on the web[3], data which can be accessed by any person, from anywhere, but most of the time this data is found in a very raw format[5], which can make it hard to understand and use. Our purpose for this report is to document how we handled movie (and movie teams) data from IMDb.

We have selected IMDb movies for our project's theme because the movie industry is a rather old area and it has been accumulating data for decades now, and it's not something that will stop accumulating data for the foreseeable future.

In the following sections of this document, all the process behind the creation of the database and the development of the information search and retrieval tool is detailed, from the data refinement to the evaluation of the tool itself.

2 DATASET

Since the moment we have decided to work with movie data from IMDb, we've immediately started looking for datasets for the project. On *IMDb - Interfaces* we could find an extensive dataset, or rather datasets, containing data regarding movies, personal, ratings, reviews, and votes. We recall that on just the movie dataset there were 8 million rows worth of data, which can be challenging and exciting to work with, but we have also noticed there was not enough textual data to work with, namely, the movie synopsis for example was missing.

Our first approach to the missing textual values was to find an API for us to retrieve that information, unfortunately, there's no free API available that can process 8 million requests, the best we could find could only offer 100 requests per day. After failing with the API approach we've decided to try to scrape the data from the IMDb website directly, and despite being possible we came to the conclusion it would take forever to scrape the data we needed.

Supervised by Prof. Sara Fernandes.

Project in Computer Science (DAT620), IDE, UiS
2018.

The solution for this problem was to abandon the IMDb provided datasets and find someone who have collected the data we needed. We've found a Kaggle Dataset [7] containing exactly what we've been looking for, the only downside was it only had around 86 thousand movies, which can be a large number but it is nothing compared to 8 million movies worth of data.

In conclusion, we've selected datasets with data from IMDb but collected by someone else, with the data we needed in a smaller factor.

2.1 Dataset Content

The datasets we've chosen, while not as large as the IMDb provided ones, contains the following:

- The movies dataset includes 85,855 movies with attributes such as movie description, average rating, number of votes, genre, etc.
- The ratings dataset includes 85,855 rating details from a demographic perspective.
- The names dataset includes 297,705 cast members with personal attributes such as birth details, death details, height, spouses, children, etc.
- The title principals dataset includes 835,513 cast members' roles in movies with attributes such as IMDb title ID, IMDb name ID, order of importance in the movie, role, and characters played.

2.2 Data Quality and Source

Kaggle is a well-known community for data analysts and researchers, the post author states the data was scraped directly from the IMDb public website, and the post has over 400 votes, with this information we can infer the source is reliable.

Regarding data quality, we'd say it met our criteria, the data we've needed was there and there weren't any unpredicted values or formats found while doing a brief analysis of the data.

2.3 Additional Needs

Unfortunately, the movies dataset does not meet our information needs as it does not have a large textual field for us to perform some full-text searches against it. After some research we found several solutions to this problem: we could search for movie plots or large description on *The Movie Database* (TMDB) [13], we could try to find the same information on the deprecated *Freebase* from Google [6] or search on the School of Computer Science Corpus for movies' summaries [4] [2].
While the easiest solution would be to use the TMDB API, this interface only allows for a limited number of requests, and if that wasn't motive enough to discard this option, we've verified the requests took around 500ms to fulfill. We considered searching the Freebase for the movie summaries or plots, but Google shut down

the database in 2016, leaving 5 years worth of data behind, and on the other hand, we could only download the data dumps, which would be an extremely intensive task to understand and query for the information we wanted. Finally, we decided to go for the MCU Corpus dataset which contained around 42 thousand movie plots, and although our dataset has over 85 thousand movies, we've reached the conclusion that it was worth the additional data, even though not every movie would have the plot field.
We've included the MCU Corpus movie dataset after our original datasets refinement.

3 PIPELINE

Our Data Preparation Pipeline is built almost entirely on python scripts, the *pandas* and *matplotlib* libraries were extremely helpful for the data handling and manipulation, which lead to simple yet powerful scripts to clean and organize the data.

Our main goal was to have clean data in a structured data format such as an SQL database system.

3.1 Data Refinement

When it comes to the Data Refinement process, the first step was to exclude all columns with too many missing values, more than 1/3, with the most incomplete columns having more than 4/5 missing values. After further investigation within the dataset, we found some columns repeated in more than one table and other redundant information which were deleted. In the last step of the refinement, some other columns with irrelevant information for our project were identified and also excluded.
After the refinement process, the MCU Corpus dataset regarding movie plots was included on the refined datasets.

3.2 Data Analysis

To get a better understanding of the data in our hands, we developed a python script with several functions to obtain general information about the dataset. The more information we have, the better the decisions we will have to make throughout the development of our project. By taking a quick look at the plots below, some interesting conclusions can be easily drawn.

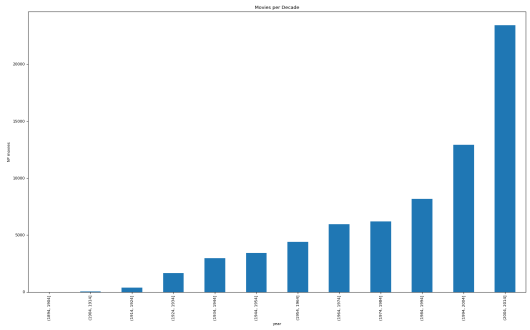

Figure 1: Movies Produced per Decade

First, the number of movies produced per decade has been increasing since the earlier decades to the present day, as shown above in Figure 1, which means that the information about the most recent years and decades is much more rich and detailed, just as it implies more movies are being produced every decade, of course. The bar corresponding to the earliest decade has been taken out of this plot on purpose due to how big it is compared to the remaining bars, as it would make the distinction of the sizes of the smaller bars very hard.

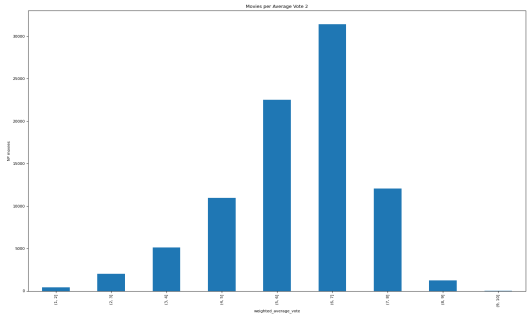

Figure 2: Movies Grouped by Rating

The second plot (Figure 2) shows that movies tend to get a score from 5 to 7, with only a small minority of movies being able to get scores above 8 or below 3.

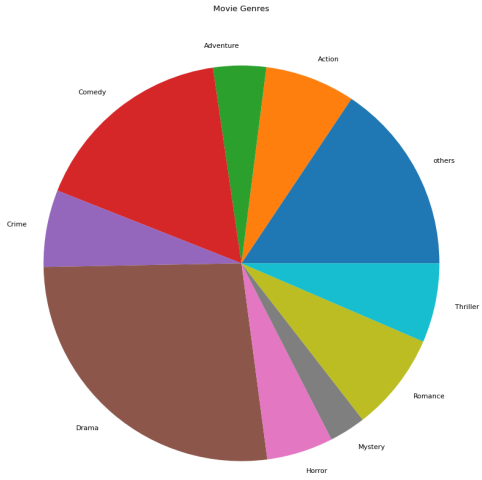

Figure 3: Movies Grouped by Genres

Last but not least, grouping the movies by genre as illustrated in the piechart of Figure 3, Drama and Comedy seem to be the most popular movie genres by a big margin, with Romance, Crime, and Thriller completing the top 5. It is worth mentioning that all

movie genres with less than 5000 movies were included in the "other" category to reduce the number of slices in the chart and thus greatly improve readability.

3.3 Database Creation

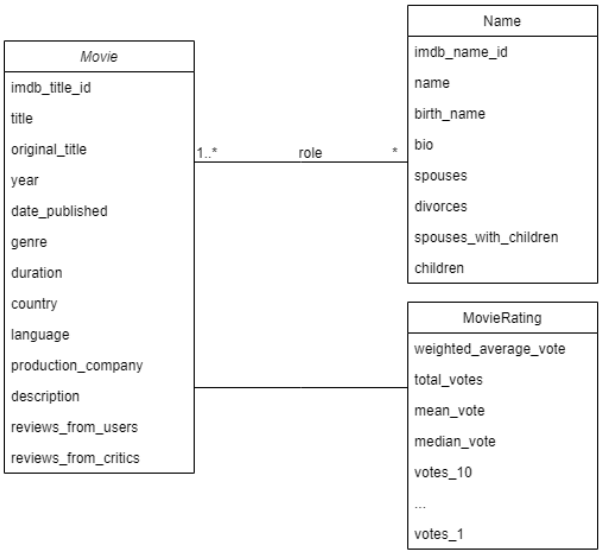

Figure 4: Database Model

The creation of the database was done with another python script that, considering all the analysis done, creates an SQL database accordingly using the refined datasets, as shown in Figure 4. The database consists of four tables. The Movie table stores the general information about every movie. The Movie Rating table stores the rating-related information for every movie. The Name table contains data about people related to the movie industry. And a MoviePersonal table, every person is connected to a movie with the help of a role association that also serves the purpose of identifying, as the name suggests, the role played by that person in the creation of the movie.

3.4 JSON Format

So that our data could be easily exported to an external tool to help us with the information retrieval tasks, we had to create a script able to convert the data on the database to JavaScript Object Notation, or rather, JSON format. The approach was simple: for each movie, we gathered the related personal (actors, directors, etc.) and joined them in a parent/child way, so that the personal would be nested on the movie information. Since the amount of data is quite large, because there was some redundancy on the data (usually a person does not take part in just one movie), we've decided to split the outcome into several JSON files with around 20 thousand records each, resulting on five chunks worth of clean data.

3.5 Pipeline Conclusion

The following pipeline scheme, Figure 5, summarizes the whole dataset cleaning, analysis, and database creation process.

This scheme summarizes how the initial 4 files of raw data were turned into a clean and functional SQL database and JSON files. First, the data was refined, then analyzed and finally, the database and JSON files were created. In future sections, we will discuss how these tasks helped create an efficient information retrieval engine capable of performing the needed tasks.

4 SEARCH TASKS

Given the nature of our dataset, we want to be able to take advantage of most of our attributes when searching for a movie or a person in the movie industry. Just like the IMDb website, we can make multiple search tasks for movies such as:

- Search for a movie by its IMDb ID (ttXXXXXXX)
- Search for a genre by name (drama, action, animation, etc.)
- Search for a movie by its title or original title
- Search for a movie by published year (with ranges)
- Search for a movie by keywords present on the plot or description
- Use sorting by average voting
- Use sorting by published year

One can even combine these and other queries to make an advanced search, and therefore refine the results to better suit the needs.

It is also possible to make search tasks for people (on the industry):

- Search for a name
- Search for a biography
- Search for an IMDb name ID

Again, it is also possible to combine different attributes to better refine the query and retrieve the best results.

It is important not only to allow for single category searches (person or movie) but also to combine them, for instance, if someone wants to know in what **drama movies from 1980 to 1990** a given actor took part in, it can be done with our dataset. This could be helpful if someone wants to watch every movie Tom Cruise took part in, for example.

This illustrates there are several options for an efficient search task, which we will cover in future sections.

5 COLLECTION AND DOCUMENTS

We've decided to build our information retrieval engine on Solr, we are running Solr 8.11, on a Docker container, without any volume configuration to facilitate sharing the work among us.

Regarding Solr configurations, we came to the conclusion that only one core was needed since we could have at least two cores: one for movie information (and ratings) and another for people (movie industry people). However, by using nesting objects on a JSON file, we could run every search task using only one core, and for that fact, only one collection.

As for the documents themselves, Solr provides means to split the records by parent and children, and each document, either child or parent will be stored as an individual document. Although

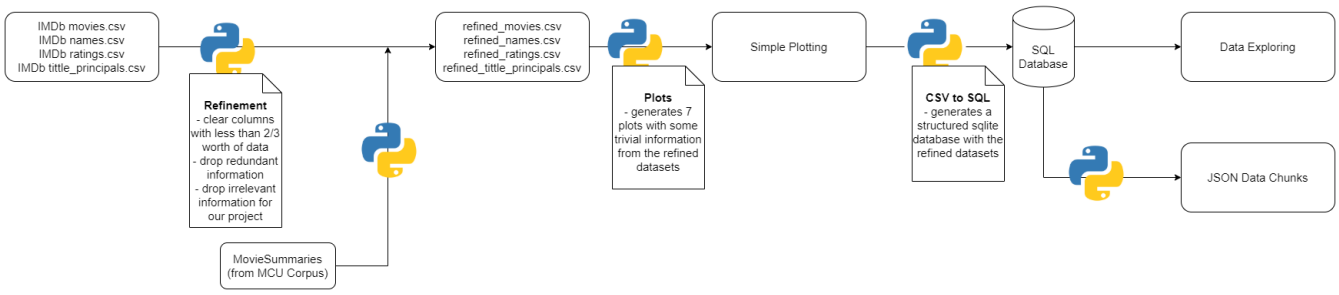

Figure 5: Pipeline Scheme

this is the default behavior for Solr, we can format the results on the queries to present the document as a join between parent and children.

The standard document, as a whole joining parent and children, consists of the multiple fields on the movies and associated reviews, and a list of documents stored under `personal`.

5.1 Different Approaches

The biggest issue we found with the approach mentioned above was that our JSON files were quite large, around 1.5GB, so in order to use the Solr API, we needed to split the whole collection into several chunks of data. We tried not using nested elements, and have a structure similar to a relational database, with three types of documents: `movie`, `person`, and `relation`. This structure would save up storage by a big amount, around 1GB, but the biggest trade-off is that for queries related to more than one type of document we would've had to make multiple queries and join them, therefore we traded storage for efficiency, which as a matter of fact is not a problem for Solr, since its job is not to store data in a structured way but rather to make efficient search tasks.

6 INDEXING DOCUMENTS

The first step in order to start indexing and running search tasks on our data is to identify the fields we are going to run searches against, and how can we improve the user experience and overall performance of the system.

We are going to approach this question focusing first on the fields we may run search tasks but are not text, or rather not text-intensive, and second on the full-text search configurations for the system.

6.1 Relevant Fields

Although not every field needs to be indexed, Solr enforces that the nested documents' fields are, therefore our schema includes indexes that were not in fact needed for the system.

The `imdb_title_id` and `imdb_name_id` were two important fields to the index since we can execute queries for the IDs and we want the results quickly without having to search on the entire database. We've indexed them as Solr's existing type `string`.

We've also indexed a person's role in a movie as an existing type `text_general`, allowing the query to have misspelled words or mismatching cases.

The other fields were not relevant enough to us to justify indexing them, for example, we usually don't search all the movies per year, but when we run a search for a movie genre we may want to sort the results by year, therefore indexing the year would not have been efficient.

6.2 Full-text Searches

Regarding full-text searches, starting from the simpler cases, we've decided to index the `title` and `original_title` fields, since these two are the main target for a movie search. We added a new type to the schema, with a `Lower Case Tokenizer` so we can split the terms into multiple words, discarding whitespaces and non-words, this tokenizer alone is very helpful since one can run a search for a Spider-Man movie just by searching for "spider man". Regarding the filters, the `Common Grams` and the `Beider-Morse` filters were added to the new type.

The Common Grams Filter creates word shingles by combining common tokens such as stop words with regular tokens. This is useful for creating phrase queries containing common words, such as "the cat." Solr normally ignores stop words in queried phrases, so searching for "the cat" would return all matches for the word "cat" [12]. Again, using the Spider-Man example, if a query is made for "the spider man", a title with "Spider-Man" should also return on the results.

The Beider-Morse Filter implements the Beider-Morse Phonetic Matching (BMPM) algorithm, which allows identification of similar names, even if they are spelled differently or in different languages [11].

We are also indexing every person's biography, with a new type which is already present in Solr default configuration, `text_en`, the field type consists of two different analyzers, on runs on index time and the other on query time, both use the *Standard Tokenizer*, and for the filters, both analyzers execute the *Stop Filter*, *Lower Case Filter*, *English Possessive Filter*, *Keyword Marker Filter* and *Porter Stem Filter*. In addition, the query analyzer also executes the *Synonym Graph Filter*.

- **Standard Tokenizer**
  This tokenizer splits the text field into tokens, treating white-space and punctuation as delimiters. Delimiter characters are discarded but there are some exceptions, such as dots not followed by a whitespace, but @ characters are part of the token-splitting punctuation, therefore this tokenizer does

not preserve email addresses [9], which is irrelevant to us since we are not dealing with email addresses of any sort.

- **Stop Filter**
  This filter discards, or stops analysis of, tokens that are on the given stop words list. A standard stop words list is included in the Solr conf directory, named stopwords.txt, which is appropriate for typical English language text [8]. It is essential to remove the same stop words from both indexes and queries, which is why the field type has both index and query analyzers [10].
- **Lower Case Filter**
  This filter simply converts any uppercase letter in a token to the equivalent lower case token.
- **English Possessive Filter**
  This filter removes singular possessives (trailing 's) from words. Note that plural possessives, e.g., the s' in "divers' snorkels", are not removed by this filter [8].
- **Porter Stem Filter**
  This filter applies the Porter Stemming Algorithm for English. The results are similar to using the Snowball Porter Stemmer with the language="English" argument. But this stemmer is coded directly in Java and is not based on Snowball. It does not accept a list of protected words and is only appropriate for English language text. However, it has been benchmarked as four times faster than the English Snowball stemmer, so can provide a performance enhancement [8].
- **Synonym Graph Filter**
  This filter maps single- or multi-token synonyms, producing a fully correct graph output. This filter is a replacement for the Synonym Filter, which produces incorrect graphs for multi-token synonyms [8].

Finally, we decided to add a custom type for *plot* and *description* fields, since most of the search tasks take advantage of both of these fields it was important for us to improve the performance regarding text queries for plots and movie descriptions. The type created is essentially similar to the Solr provided *text_en*, since we wanted to explore how well we could improve the system's capabilities.

7 INFORMATION RETRIEVAL

As described in Section 4, there are some interesting search tasks we can explore using the Solr query tool, given our dataset domain, there are multiple possible and relevant queries, but many possible queries were left to explore since they didn't meet our information requirements or weren't going to have much impact on a regular person's search tasks.

7.1 Query 1

Drama movies before 2000 Tobey Maguire took part in:

Option	Value
q	{!parent which="-_nest_path_:* *:*"} name:"Tobey Maguire"
fq	year:[* TO 1999] genre:drama
sort	weighted_average_vote desc

Table 1: Parameters for Query 1

by changing some parameters such as *"fl"* we can obtain the movies and the people involved.

7.2 Query 2

Comedy movies from the 20's with at least 8.0 as mean vote, sorted by year ascending and mean vote descending:

Option	Value
q	title:* genre:comedy year:[1920 TO 1930] mean_vote:[8.0 TO *]
q.op	AND
sort	year asc, mean_vote desc

Table 2: Parameters for Query 2

7.3 Query 3

Top 10 worst rated horror movies since 2000:

Option	Value
q	genre:horror year:[2000 TO *]
q.op	AND
sort	weighted_average_vote asc
rows	10

Table 3: Parameters for Query 3

7.4 Query 4

Action movies from the 20's 1hour or shorter where at least one actor has New York mentioned on their biography, sorted by weighted average vote descending:

Option	Value
q	{!parent which="-_nest_path_:* *:*"} bio:"new york" role:actor
q.op	AND
sort	weighted_average_vote desc
fq	year:[1920 TO 1929] genre:action duration:[* TO 60]

Table 4: Parameters for Query 4

7.5 Query 5

Best movie by genre with at least 10000 votes:

Option	Value
q	mean_vote:* total_votes:[10000 TO *]
q.op	AND
sort	mean_vote desc
rows	20
group	true
group.field	genre

Table 5: Parameters for Query 5

Of course, for this particular query we only want the top-rated movie, then we select only the first result for every group.

7.6 Results

Our goal is to have a functional system capable of running the basic search tasks IMDb provides while being able to execute queries across different sets of data, such as people and movies altogether. We've proven our system can handle the basic queries, and one can go further and explore different parameters to refine the search.

8 EVALUATION

Now that we have a working system, capable of performing several search tasks, it is time to evaluate and better adjust the system in order to obtain the best results.
To evaluate the system we are going to test three different scenarios: **schemaless**, **schema without boosting and adjustments**, and **schema with boosting and some minor adjustments** to the query. For the evaluation metrics we are going to use **Precision** and **Recall**. Precision measures retrieval specificity defined as the proportion of retrieved items that are judged by the user as being relevant; this measure penalizes system retrieval of irrelevant items (false positives) but does not penalize failures by the system to retrieve items that the user considers to be relevant (false negatives). Recall measures retrieval coverage defined as the proportion of the set of relevant items that is retrieved by the system, and therefore penalizes false negatives but not false positives [1].

8.1 Experience 1 - Matrix Trilogy

We want to find the Matrix Trilogy, just by searching all movie titles for "matrix" and it may or not contain "neo" and "matrix" on the plot or the description. We believe this is a rather precise query if the system is capable enough. The movies we are looking for are: The Matrix, Matrix Reloaded and Matrix Revolutions.

For the last scenario the following parameters were used:

Option	Value
q	(original_title:matrix title:matrix) AND (description:neo plot:neo)
qf	original_title description^2 plot^3
defType	edismax

Table 6: Parameters used to enhance the query

After running the tests we came to the following results on Table 7.

Scenario	Metric	Value
Schemaless	Average Precision	0.0
	Precision at 5 (P@5)	0.0
Schema w/out boost	Average Precision	0.88
	Precision at 5 (P@5)	0.6
Enhanced	Average Precision	1
	Precision at 3 (P@3)	1

Table 7: Results for Experience 1

The following tables and P-R curves reveal the obtained results:

Movie Title	Relevance
...	N
...	N
...	N
...	N
...	N

Table 8: Movies retrieved without Schema

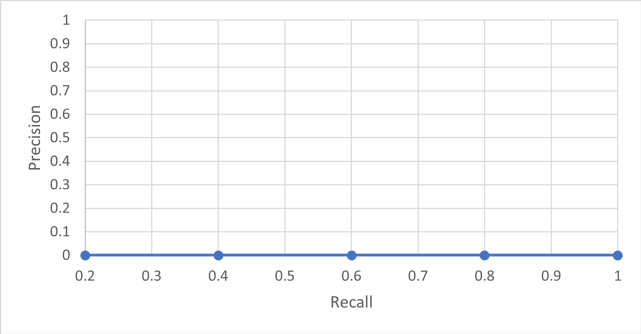

Figure 6: Experience 1 without Schema

Movie Title	Relevance
The Matrix Reloaded	R
The Matrix Revolutions	R
The Matrix	R
Buhera mátrix	N
Maarek hob	N

Table 9: Movies retrieved with Schema without Boosting

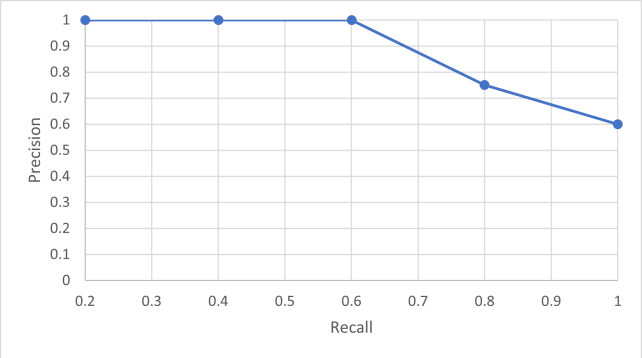

Figure 7: Experience 1 with Schema without boosting

Movie Title	Relevance
The Matrix Reloaded	R
The Matrix Revolutions	R
The Matrix	R

Table 10: Movies retrieved with Schema with Boosting

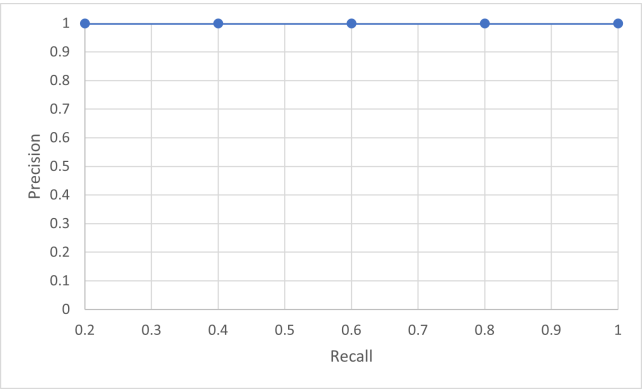

Figure 8: Experience 1 with Schema with boosting

We can conclude the boosting techniques were helpful since the curve for the enhanced system is a steady line at 1 for precision.

8.2 Experience 2 - Harry Potter Franchise

In this experience we want to find Harry Potter franchise, searching "wizard", "magic", "harry" on the plot or the description

We believe this is a rather precise query if the system is capable enough. The movies we are looking for are all Harry Potter's related movies

For the last scenario the following parameters were used:

Option	Value
q	description:(wizard magic harry) plot:(wizard magic harry)
qf	description^2
defType	edismax

Table 11: Parameters used to enhance the query

After running the tests we came to the following results on Table 7.

Scenario	Metric	Value
Schemaless	Average Precision	0.0
	Precision at 10 (P@10)	0.0
Schema w/out boost	Average Precision	0.5875
	Precision at 10 (P@10)	0.4
Enhanced	Average Precision	0.567803
	Precision at 10 (P@10)	0.4

Table 12: Results for Experience 2

And the correspondent P-R curves for each scenario:

Movie Title	Relevance
Harry Potter and the Goblet of Fire	R
The Magic Sword	N
Reaching for the Moon	N
Harry Potter and the Half-Blood Prince	R
Harry Potter and the Order of the Phoenix	R
Harry Potter and the Prisoner of Azkaban	R
Dragonslayer	N
The Care Bears Adventure in Wonderland	N
Troll	N
The Wiz	N

Table 13: Movies retrieved without Schema

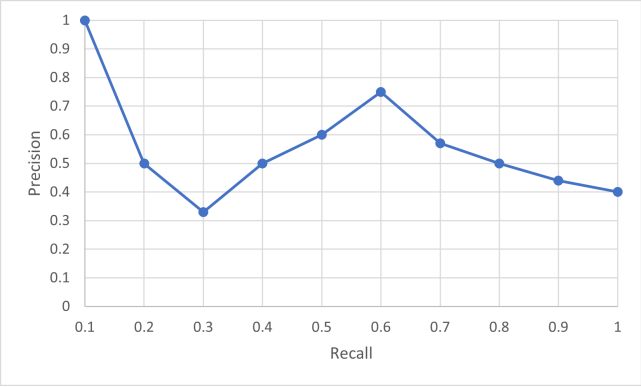

Figure 9: Experience 2 without Schema

Movie Title	Relevance
Harry Potter and the Goblet of Fire	R
The Magic Sword	N
Reaching for the Moon	N
Harry Potter and the Half-Blood Prince	R
Harry Potter and the Order of the Phoenix	R
Harry Potter and the Prisoner of Azkaban	R
Dragonslayer	N
The Care Bears Adventure in Wonderland	N
Troll	N
The Wiz	N

Table 14: Movies retrieved with Schema without Boosting

Figure 10: Experience 2 with Schema without boosting

Movie Title	Relevance
Harry Potter and the Goblet of Fire	R
Harry Potter and the Prisoner of Azkaban	R
Reaching for the Moon	N
Dragonslayer	N
Harry Potter and the Half-Blood Prince	R
Troll	N
Harry Potter and the Order of the Phoenix	R
The Wiz	N
The Wizard of Oz	N
Wizards	N

Table 15: Movies retrieved with Schema with Boosting

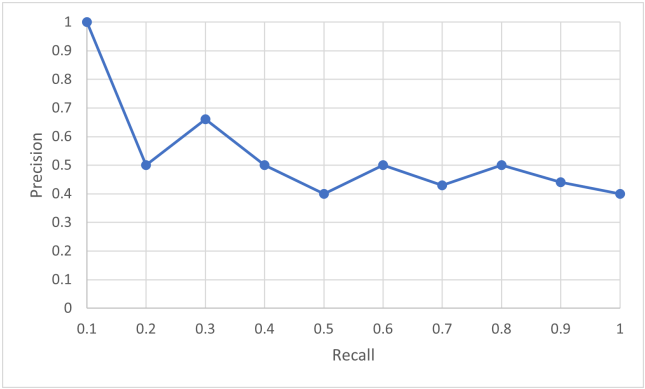

Figure 11: Experience 2 with Schema with boosting

In this experience in particular, the advantages of the schema and the boosting are less visible. On the other hand, we can clearly see what happens to the precision as the recall increases, starting at around 0.5 and going down until 0.25. This values, although not perfect, seem to be good enough for our information retrieval process.

8.3 Experience 3 - Toy Story Franchise

In this experience we want to find movies related to Toy Story, searching "toy" and "woody" on the plot or the description

For the last scenario the following parameters were used:

Option	Value
q	description:(toy woody) plot:(toy woody)
qf	description^3 original title^2 genre
defType	edismax

After running the tests we came to the following results on Table 7.

Scenario	Metric	Value
Schemaless	Average Precision	0.833333
	Precision at 5 (P@5)	0.4
Schema w/out boost	Average Precision	0.833333
	Precision at 5 (P@5)	0.6
Enhanced	Average Precision	0.96
	Precision at 5 (P@5)	0.8

Table 16: Results for Experience 3

And the correspondent P-R curves for each scenario:

Movie Title	Relevance
...	N
...	N
...	N
...	N

Table 17: Movies retrieved without Schema

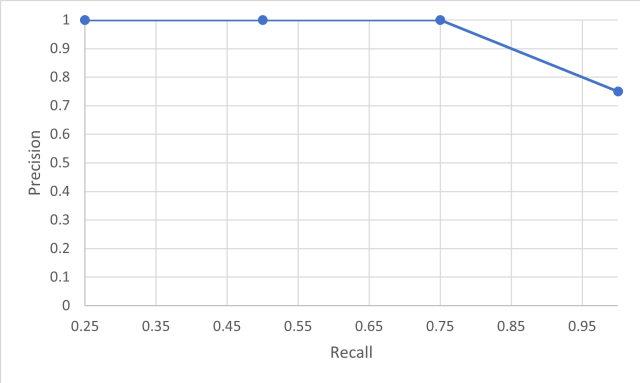

Figure 12: Experience 3 without Schema

Movie Title	Relevance
Toy Story 2	R
Toy Story 3	R
Toy Story	R
Condorman	N

Table 18: Movies retrieved with Schema without Boosting

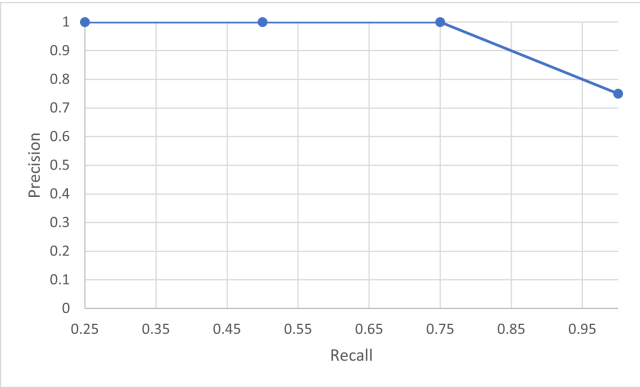

Figure 13: Experience 3 with Schema without boosting

Movie Title	Relevance
Toy Story 2	R
Toy Story 3	R
Toy Story	R
Toy Story 4	R

Table 19: Movies retrieved with Schema with Boosting

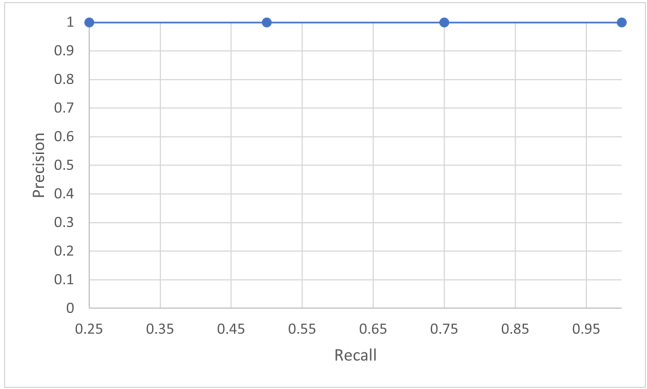

Figure 14: Experience 3 with Schema with boosting

In this example we can also conclude the boosting techniques were helpful because the precision remained at 1 for longer in the respective chart.

Analysing the values from the three experiences, a few conclusions can be drawn. First, we are happy with these results since in every single experience the intended results were retrieved. Although we know these experiences are not too demanding for our information retrieval tool, getting the intended results is always satisfactory as that is the goal of the tool. The plots built in the three experiences also let us easily conclude that the precision tends to decrease as the recall increases. this happens because, as the recall increases, the true positives increase, but so do the false positives, which leads to a inevitable decrease in precision. Is also important to notice that, in all three experiences, the precision could never be maximum at 5 (for experience 1 and 3) or at 10 (for experience 2) because the number of true positives is always less than the number of results retrieved .

9 WEB APPLICATION

Since Solr's user graphical interface is a bit hard for the regular user to perform efficient search tasks, we've decided to implement our custom user interface, on a simple lightweight web application.

9.1 Technology Stack

Being a popular framework, we've decided to use React, alongside ExpressJS to build our web application. We used ExpressJS to make API calls to Solr and an external API (see next section), so we don't have to implement that type of logic on the front-end.

9.2 Additional API - TMDb

For the web application, it made sense to provide users some extra information in order to enhance user experience, we decided to reach out for another source of data, TMDb, this database would allow us to retrieve data such as poster pictures, profile pictures, and some additional information about a movie or a person. Since this data would only be necessary when a movie or a person is found, we didn't include it on the Solr core, and besides that, this external API is very limited in terms of requests, and we would have to make around 200K requests in a viable time frame.

9.3 Features

The web application features two important main sections: Movies and People. The movies section allows the user to search for a movie with some filters such as year, genre, and language, it also allows sorting for duration, votes, year, title, and search rank, the main input field tries to match the title or original title, as well as a description and/or plot for a movie. After a number of movies are found the user can click on a movie to see more info about it, as well as the people involved (cast, directors, etc.).

On the people section, the user can only search for a person on a single input, this input tries to match with the person's name or biography. After a number of people has been found, the user can click to see more info about a person, such as extensive biography and related movies.

We didn't find the necessity to include both people and movies on the same query, not because they are not related but because a regular user would want to search for a specific topic and not both.

10 REVISIONS INTRODUCED

Some parts of this document were revised in the last stages of the project. Among some minor adjustments, most of them just language-related, the biggest change was executed in the Evaluation chapter. More specifically, all plots were remade from scratch to improve their readability. Also, for every system in every experience, a table was created showing the actual movies retrieved by that system with that specific query and whether they are relevant or not to make it possible to see the results in their purest form.

11 CONCLUSIONS

In conclusion, this report summarizes how a set of raw data can be combined into a functional database and finally a search system capable of functional queries related to the problem domain.

The first part of this process was rather simple, the only difficulties were choosing the proper dataset and adapting it to our needs, extending it if necessary with additional external information.

Joining the refined data into a relational database was very helpful to the next step because we needed to merge the information on a JSON file to import it to Solr, there were, of course, some implications, being the main one the time complexity for this task which made our work very slow at first, but after the JSON files were good enough to be imported the process was fairly simple.

The indexing process was done just by exploring the Solr documentation, which in our opinion was not so detailed and clear as we might have hoped. Regarding evaluation, we could take some notes about how a schema improves (or not) the search tasks, and how refining and boosting the query parameters could improve the overall performance of the search system.

When it comes to user interface, the web application was implemented to improve the usability of the tool for the common user. This application also retrieves some more information directly from a different source.

Looking back to the whole process, the final result is very satisfactory. Although further improvements were possible for example related with the search tool like improving the accuracy of the obtained results, Solr is very functional and useful to interact with such a big amount of data, making it easily accessible and readable.

REFERENCES

[1] Sergio A Alvarez. 2002. An exact analytical relation among recall, precision, and classification accuracy in information retrieval. *Boston College, Boston, Technical Report BCCS-02-01* (2002), 1–22.

[2] David Bamman, Brendan O'Connor, and Noah A Smith. 2013. Learning latent personas of film characters. In *Proceedings of the 51st Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*. 352–361.

[3] bernardmarr (Forbes). [n. d.]. https://www.forbes.com/sites/bernardmarr/2018/05/21/how-much-data-do-we-create-every-day-the-mind-blowing-stats-everyone-should-read/

[4] CMU Movie Summary Corpus. [n. d.]. http://www.cs.cmu.edu/~ark/personas/

[5] Tim Bock (DISPLAYR). [n. d.]. https://www.displayr.com/what-is-raw-data/

[6] Google. [n. d.]. Freebase API. https://developers.google.com/freebase

[7] Stefano Leone (KAGGLE). [n. d.]. https://www.kaggle.com/stefanoleone992/imdb-extensive-dataset

[8] Apache Solr Reference. [n. d.]. Filter Descriptions. https://solr.apache.org/guide/8_11/filter-descriptions.html

[9] Apache Solr Reference. [n. d.]. Tokenizers. https://solr.apache.org/guide/8_11/tokenizers.html

[10] Karan Jeet Singh. 2019. Sitecore Solr Stop Words Made Easy. https://www.searchstax.com/blog/sitecore-stop-words-solr/

[11] Solr. [n. d.]. Beider-Morse Filter. https://solr.apache.org/guide/8_11/filter-descriptions.html#beider-morse-filter

[12] Solr. [n. d.]. Common Grams Filter. https://solr.apache.org/guide/8_11/filter-descriptions.html#common-grams-filter

[13] The Movie Database (TMDB). [n. d.]. http://www.themoviedb.org/?language=pt-PT