Popular Animation Movies Search System

João Pires
Master in Informatics and Computing Engineering,
FEUP, University of Porto
up201806079@edu.fe.up.pt

Mafalda Costa
Master in Informatics and Computing Engineering,
FEUP, University of Porto
up202109478@edu.fe.up.pt

Tomás Gonçalves
Master in Informatics and Computing Engineering,
FEUP, University of Porto
up201806763@edu.fe.up.pt

Abstract

Search system for popular animation movies obtained from IMDB and OMDB. Each movie includes primary information (title, language, etc.) and secondary information (reviews, box office, etc.). Solr was used as the information retrieval tool. The Extended DisMax (eDismax) Query Parser was used to extend the basic functionalities of Solr. Final search system allows to search for Title, Rate, Director(s), Writer(s), Actor(s), Plot, Language, Country, Awards, Production and Reviews, in any language, with priorities and filters applied to fields to produce better results. GUI available to submit the query and see the search results in a friendly user interface. Evaluation applied to a set of configurations, using some metrics.

Keywords

movies, search system, datasets, indexes, full-text search, tmdb, omdb, python, solr, edismax, information retrieval, retrieval evaluation, flask, wikidata, gui

1 Introduction

This document represents the report for the search system developed under the curricular unit of Information Processing and Retrieval, of the 1st year of the Master in Informatics and Computing Engineering at FEUP.

The subject of the search system is the set of the most popular animation movies from all around the world.

The report is structured the same way that the search system was built, i.e. it starts with the data processing pipeline, moving into information retrieval and then achieving the goal, the search system itself. Through the following sections, each one of these steps will be described with further detail, including examples and explanations to expose and justify decisions that were taken during the development.

2 Data

The first task we found ourselves searching for was the main subject of our search system. There are a couple of search systems in this area, including the most famous one, IMDB [7], but none of them are able to allow the user to search for more than the basic details of the movie. We want to extend that possibility and allow to search not only for essential information, but also for the plot and other characteristics.

That said, we've decided to develop a search system over a collection of the most popular animation movies from all around the world, using multiple sources to get the information necessary in order to have a dataset with complementary information from more than one source and including not only primary fields like the movie title or language, but also other secondary and relevant fields.

We've fetched the data we needed and the proceeded with the analysis of the data quality and its characteristics, which will be presented in detail in the next sections.

2.1 Data Sources

The first data source, which represents simultaneously the starting point of our movie collection, was obtained from TMDB [9], a community driven database for movies that is online since 2008. It has more than 705,000 Movies, more than 121,000 TV Shows and more than 2,270,000 people contributing for the success of the database [10]. For more information regarding the contribution process, please visit the TMDB Contribution Rules page [11].

From TMDB, we downloaded the most popular animation movies from all around the world. Popular movies are all those who have the most amount of searches on the platform. To get the data, we've used the TMDB API [12], which is free to use as long as the product using the data is not used for commercial purposes. Since we're using the data for academic purposes, we've used the API Developers version.

Inside a *Python* script, we've called the following URL

https://api.themoviedb.org/3/discover/movie?api_key=API_Key&with_genres=16&page=<page_no>

which retrieves in JSON format the first page of the most popular animation movies. The URL contains our API Key and the ID of the Animation genre (16).
Then, parsing the total number of pages and using the *page* attribute, we've iterated through all the pages to get all the popular animation movies and their information.
Inside a newly created folder called 'movies', for each movie, we created a folder whose name is the edited movie title. By edited movie title we mean that the new one results from removing all non alpha-numeric characters from the original one, which can still be accessed in the Title field of the movie details. Inside that unique folder for each movie, we've created a .txt file named 'details.txt', containing the information in JSON format.
The second data source was OMDB API [8], The Open Movie Database, which allowed us to obtain extra information about the movies downloaded from TMDB. All the content and images on this API are contributed and maintained by OMDB users, with the platform being funded through donations and Patron.
Again, using a *Python* script, we've iterated the movies list and searched for each one at OMDB, using the following URL

http://www.omdbapi.com/?apikey=API_Key&plot=full&t=<movie_title>&y=<year>

substituting the <movie_title> and <year> tags for the movie title and for the year of the release date, respectively. Note that, as we've done with the TMDB API, we've used again the API Key, this time for the OMDB API, and the *plot* attribute set to 'full'. Then, for all the movies of TMDB with a match on OMDB, we've created a new .txt file named 'omdbContent.txt' containing the JSON response of the API call.
Back into the TMDB source, we collected the available reviews for each movie by iterating through the list we've just fetched before. Again, through the TMDB API, we used the following URL

https://api.themoviedb.org/3/movie/<movie_id>/reviews?api_key=API_Key&page=<page_no>

substituting the <movie_id> and <page_no> for the id of the movie of the current iteration and for the page number, respectively. We then stored the comments in a new file, named 'reviews.txt', where each line represents one comment. This time, we've not stored the information in JSON format.
The last step was to generate a single JSON file containing all the dataset. For that, we've created a *Python* script that iterates through all movies and generates by the end a JSON file with the information of all movies, each one identified by an ID that is the same as the previous folder name used to store the information of each movie.

2.2 Data Processing Pipeline

The following diagram represent the Data Processing Pipeline of our project.

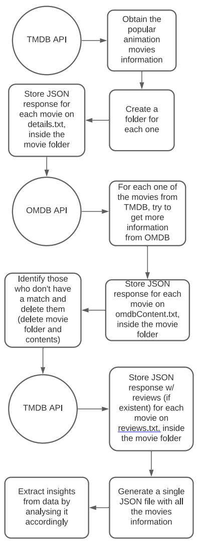

TMDB API
Obtain the popular animation movies information
Create a folder for each one
Store JSON response for each movie on details.txt, inside the movie folder
OMDB API
For each one of the movies from TMDB, try to get more information from OMDB
Store JSON response for each movie on omdbContent.txt, inside the movie folder
Identify those who don't have a match and delete them (delete movie folder and contents)
TMDB API
Store JSON response w/ reviews (if existent) for each movie on reviews.txt, inside the movie folder
Generate a single JSON file with all the movies information
Extract insights from data by analysing it accordingly

Figure 1: Data Processing Pipeline.

It starts by getting the data from TMDB and storing it in appropriate files and folders. Then, the data is collected from OMDB while being matched with the previous one and all the movies without a correspondent in the second API are deleted. The last API call is again to the TMDB one, fetching all the existing reviews for each remaining movie. The last step before the analysis and the insights extraction from the data collected and stored consists in the generation of a single JSON file that contains the whole dataset information.
Our MAKEFILE follows the same structure. It starts by calling the *Python* script that collects the data from TMDB and the one that gets the remaining info from OMDB and then calls the script that erases the movies without a match. Then collects the available reviews for each of the movies stored before and generates a single JSON file for the dataset. The last step represents the analysis of the data.

2.3 Data Analysis

The first data source, TMDB, allowed us to collect the information of 9,863 animation movies. Then, we collected the information from OMDB and erased the folders (movies) without a match in the second API and/or without release date. The reason behind this decision was the fact that the gap between the amount of information of some movies would be considerable. That said, the deletion of those movies resulted in a new total of 6,300 animation movies (3,563 movies were deleted).
From TMDB, we got the following main attributes, among others, for each movie:

- Adult (set to true if is an adult movie)
- Original language
- Original title
- Overview
- Popularity score
- Release date
- Title
- Poster path
- Vote average
- Vote count

From OMDB, we got then the following extra attributes for each movie:

- Duration
- Director(s) name(s)
- Writer(s) name(s)
- Actor(s) name(s)
- Plot
- Country
- Awards
- IMDB rating
- IMDB number of votes
- Metascore
- Ratings from other platforms (varying depending on the movie)
- DVD release date
- Website
- Production

- Box Office (the amount in American Dollars made in the ticket boxes)

From TMDB, we got also the information regarding the reviews for each movie, when they exist. In that case, the most important attribute is the *content* of each comment, since the user who made it is not so relevant.
Regarding the medium size of the *overview* attribute from TMDB, it is 47,52 words on average. For the *plot* attribute from OMDB, it is 61,02 words on average.
The analysis of this attributes included also the most used words, which we display in the following graphic, containing the top 30. We've removed all the words that don't add value to this analysis, selecting them from a stopping list, which includes, among others, the words 'you', 'the', 'a', etc.

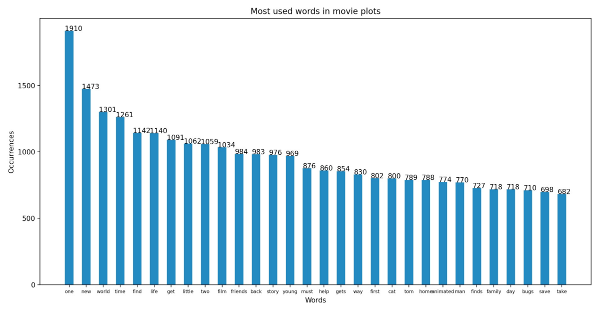


Figure 2: Most Used Words in Movie Plots.

We've also analysed the distribution of the movies regarding their original languages and the decade when they were released. The following two graphics show the distribution of the movies according to those attributes.

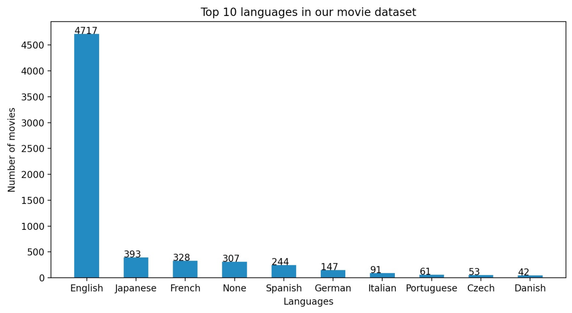


Figure 3: Top 10 Languages in our Movie Dataset.

Note that this graphic only contains the top 10 and that one of the languages is 'None', which represents language-less movies. There are also 267 movies without the original language attribute, having 'N/A' as value.

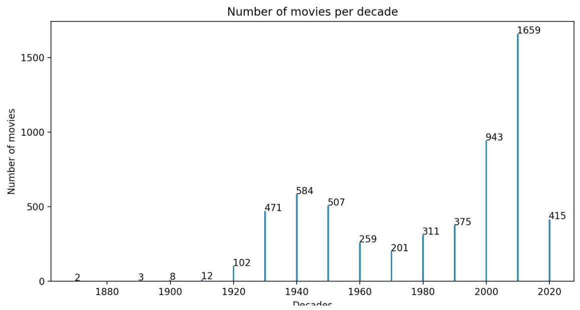


Figure 4: Number of movies per decade.

Regarding the reviews, the total number is 609 and there are 348 movies with reviews, from the 6,300 mentioned above. That said, the average number of reviews per movie is 0,097. There are 5950 movies without any review and 2 movies whose API call throws an error (in-existent movie).

2.4 Conceptual Model

The following UML represents the conceptual model for our database.

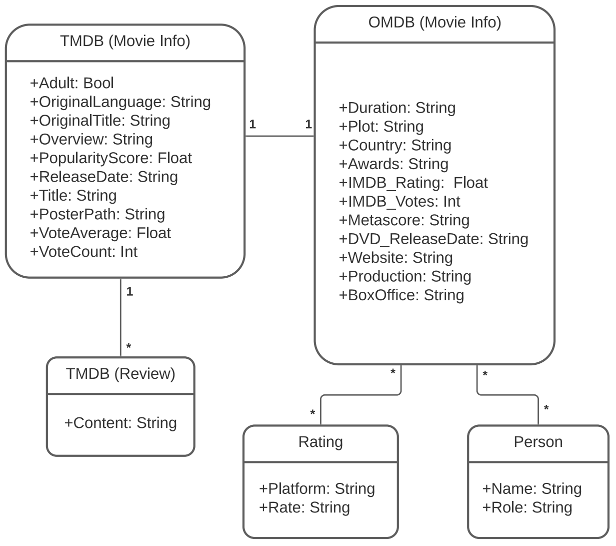


Figure 5: Conceptual Model.

The attribute names are self-explanatory, as well as each table's names, but a further analysis is necessary:

- Each TMDB movie can have at most 1 match with an OMDB movie.
- Each TMDB movie can have zero or more reviews from the same platform.
- Each OMDB movie can have zero or more ratings and people associated with. The possible roles for the people are 'Director', 'Writer' and 'Actor'.

3 Information Retrieval

Having the data fetched, analysed and in a single JSON file, the next step was to select the information retrieval tool and start working on the schema, indexes and queries. For that, we've selected the Solr information retrieval tool [1], which was covered in a PRI tutorial.

3.1 Information Needs

In this section, we include five examples of information needs and the corresponding queries. Note: **IN.** - Information Need ; **Q.** - Query

3.1.1 Information Need - Example 1

IN. "Which is the name of the movie where the toys gain life?"
Q. "toys that have life"

3.1.2 Information Need - Example 2

IN. "Who was the Hollywood actor that was part of a Christmas movie where a kid goes on a train to see Santa Claus?"
Q. "kid on trip to see santa claus"

3.1.3 Information Need - Example 3

IN. "What is the imdb score of the movie where an ogre who's voice was made by Mike Myers meets his wife's parents?"
Q. "ogre voices by Mike Myers meets wifes parents"

3.1.4 Information Need - Example 4

IN. "In which year was released the first Ice Age movie with Sid as a character?"
Q. "animals ice sid"

3.1.5 Information Need - Example 5

IN. "Was the animation movie where animals escape from the Central Park zoo awarded?"
Q. "central park zoo animals"

3.2 Indexing

Starting with the idexable components and the schema built, we've indexed all the information we had for each movie. The following list includes the fields that we have for each movie. Some were not available for all, like the Awards and the Reviews, due to not having any content on those fields for the specific movie.

Title, Rated, Release, Runtime, Genre, Director, Writer, Actors, Plot, Language, Country, Awards, Poster, Metascore, imdbRating, imdbVotes, DVD, BoxOffice, Production, Website, Overview, Popularity, Poster_path, Video, Vote_average, Vote_count

Then, depending on the field, we've defined a set of tokenizers and filters for each of the field types we've created on the schema to overcome some problems like the accentuation and the lower/upper case.
The field types we created were the following:

Text	• text class (*solr.TextField*) • standard tokenizer (*solr.StandardTokenizerFactory*) • filter to convert the input to lower case (*solr.LowerCaseFilterFactory*) • filter to convert to ASCII characters only (*solr.ASCIIFoldingFilterFactory*) • used for: id, Title, Rated, Runtime, Genre, Director, Writer, Actors, Plot, Language, Country, Awards, Poster, Production, Overview, Video and Reviews
Number	• float point class (*solr.FloatPointField*) • standard tokenizer (*solr.StandardTokenizerFactory*) • whitespace tokenizer (*solr.WhitespaceTokenizerFactory*) • no filters • used for: Metascore, imdbRating, imdbVotes, BoxOffice, Popularity, Vote_Average and Vote_Count
Date	• date range class (*solr.DateRangeField*) • standard tokenizer (*solr.StandardTokenizerFactory*) • no filters • used for: Release and DVD
Link	• text class (*solr.TextField*) • standard tokenizer (*solr.StandardTokenizerFactory*) • whitespace tokenizer (*solr.WhitespaceTokenizerFactory*) • no filters • used for: Website and Poster_path

Table 1: Field types with corresponding classes, tokenizers, filters and fields used in.

To facilitate the indexing process, we've created both a script and a Dockerfile that are responsible for uploading the collection to the information retrieval tool and for defining the schema. In total, 6227 documents were uploaded.

3.3 Queries

Regarding the queries, we've decided to use not only the standard query parameters that Solr has, but also the the Extended DisMax (eDismax) Query Parser [3] and its parameters.

3.3.1 Solr and eDismax Parameters

The following list includes the parameters we've used from both the standard version of Solr and from the eDismax and a brief explanation for each one. For that, we have consulted both the documentation for the eDismax and DisMax Query Parser [2].

indent = true	This is probably the simplest of the parameters, but it's important to make the JSON response human-readable. If set to true, as it's our case, the response format includes indentation.
q.op = OR	By using the operand OR, we make sure our query is flexible enough, i.e. there's no need for all the words that compose the user query to match a certain field, since a movie would match if at least one of those words is found under one of the movie fields. If we've used the AND operand, we wouldn't guarantee that the user search string would return a result since it would be less probable that its entire content would match the data of a specific movie.
stopwords = true	By setting this parameter to true, we make sure that the *StopFilterFactory* configured in the query analyzer should be respected when parsing the query.
qs = 3	This parameter, Query Phrase Slop, specifies the amount of slop permitted on phrase queries explicitly included in the user's query string with the qf parameter, described next. Slop is the number of positions one token needs to be moved in relation to another token in order to match a phrase specified in a query. For example, consider the query "toy story" and slop being 1. The slop being 1 means that there must be no more than one word between the ones used in the query. So, if we had a field with the value "toy has life story", this field wouldn't match since there are two words between "toy" and "story". Of course that, for this example, the 'Title' field would match, but it could be different in other situations. The value 3 used represents the average number of words between the first and the last on the queries we used for evaluation purposes.
ps = 3	This parameter, Phrase Slop, has the same definition as the Query Phrase Slop, except it applies to queries specified with the pf parameter. That said, the value used was also 3 for the same reasons as described in the Query Phrase Slop explanation.
tie = 0.1	The tie parameter controls how much the final score of the query will be influenced by the scores of the lower scoring fields compared to the highest scoring field. The value used was the one recommended in the documentation. A value of 0.0 (default) makes the query a pure disjunction max query, i.e. only the maximum scoring sub-query contributes to the final score. A value of 1.0 makes the query a pure disjunction sum query, where it doesn't matter what the maximum scoring sub query is, because the final score will be the sum of the sub-query scores.
q = user_input	This parameter is the query itself, with the user input.
qf = Title^8 Rated^3 Director^5 Writer^5 Actors^5 Plot^10 Language^2 Country^1 Awards^5 Production^5 Overview^10 Reviews^9	This parameter, Query Fields, includes the list of all the fields in which the search system should look for matches considering the user input and, for each field, there's a number that represents the importance. The higher the number, the higher the importance of that field.
pf = Title^3 Plot^5 Overview^5 Reviews^4	This parameter, Phrase Fields, can be used to "boost" the score of documents in cases where all of the terms in the q parameter appear in close proximity. That way, we've selected the top 4 fields, Title, Plot, Overview and Reviews, and gave them an importance number.
bq = Awards:oscar^7 Awards:nom*^6 Reviews:good^7 Reviews:excellent^7 Reviews:best^7 Reviews:recommend*^7	This parameter bq stands for Boost Query, and specifies an additional query clause that will be added to the user's main query as optional clauses that will influence the score. This way, we've decided that all the movies that were awarded or nominated for the Oscars and those whose reviews include words such as good, excellent, best, recommend would suffer a "boost".

rows = 2147483647	The last parameter used was rows, to define the maximum number of results that would be retrieved for an user input string. Since we didn't want to limit the number of results retrieved, we selected the value 2147483647, which represents the maximum int value and thus allows to see all the results and not just a part of those.

Table 2: Solr, Dismax and eDismax parameters, its values and explanations.

3.4 Evaluation

In order to calculate the effectiveness of our search system, we proceeded with the evaluation and comparison of different system configurations of our search system. To do that, we calculated the precision and recall based on the results obtained for each information need defined in the section "Information Needs".
For each information need, the results were tested with 5 different system configurations. All these systems have an Schema defined.

- **System 1:** System without Query Phrase Slop, Phrase Slop, Tie, Phrase Fields and Boost parameters.
- **System 2:** System without Phrase Fields and Boost parameters.
- **System 3:** System without Boost Fields.
- **System 4:** System with the standard query parameters, defined in the previous section.
- **System 5:** System with a higher weight on the Title (10) than the Plot (6) and Overview(6) fields, and a higher boost on the 'excellent' and 'recommend*' words.

The process follows always the same logic. First, we search each query that represents an information need and evaluate each result as relevant or not.
For example, for the query "toys that have life", we will consider relevant any film that is based on a story or just a segment in which inanimate objects (toys) come to life, with that meaning acquiring human capabilities, such as running or talking. Based on the relevance, we then calculate the precision, which measures the fraction of relevant retrieved documents, and the recall, that measures the fraction of retrieved relevant documents, obtaining a precision-recall curve by interpolating these values. This curve allow us to visually get an idea of the effectiveness of our system.

3.4.1 Precision

In this section, we'll demonstrate two tables, one for the Average-Precision and the other one to the Mean Average-Precision. Each one of the tables contains the values calculated for each Information Need defined in the section above and for each System Configuration also described above.

	IN1	IN2	IN3	IN4	IN5
System 1	1	0,23	0,98	0,85	0,78
System 2	0,95	0,33	0,95	0,98	0,83
System 3	0,83	0,87	0,83	1	0,46
System 4	0,94	0,29	0,83	1	0,70
System 5	0,6	0,92	0,74	1	0,57

Table 3: Average-Precision table.

	System 1	System 2	System 3	System 4	System 5
MAP	0,77	0,80	0,80	0,75	0,76

Table 4: Mean Average-Precision table.

We can conclude, based on Average-Precision and Mean Average-Precision tables, that the System 2 and System 3 configuration offer better performance.

3.4.2 Precision-Recall Curves

In this section, we present different Precision-Recall curves for each one of the queries that represent one of the Information Needs defined above. The colors in the graphics represent the different System Configurations, i.e. Dark blue - System 1, Orange - System 2, Grey - System 3, Yellow - System 4, Light blue - System 5.

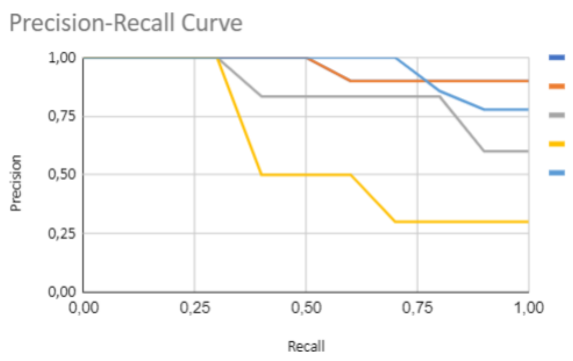


Figure 6: Precision-Recall Curve for the query "toys that have life".

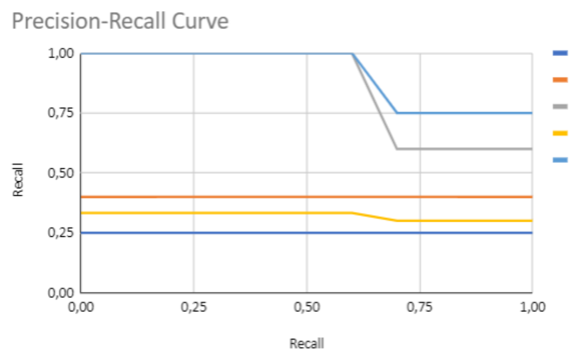


Figure 7: Precision-Recall Curve for the query "kid on trip to see santa claus".

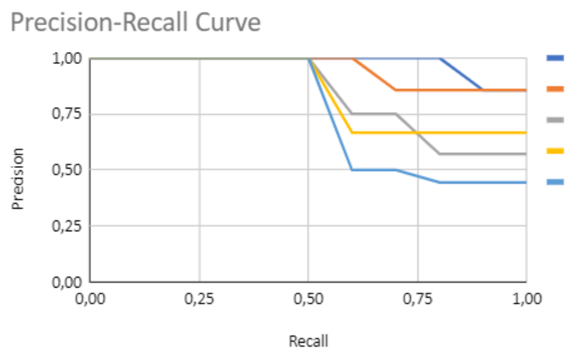


Figure 8: Precision-Recall Curve for the query "ogre voices by Mike Myers".

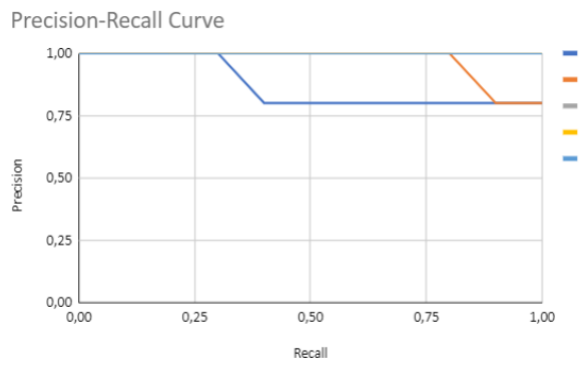


Figure 9: Precision-Recall Curve for the query "animals ice sid".

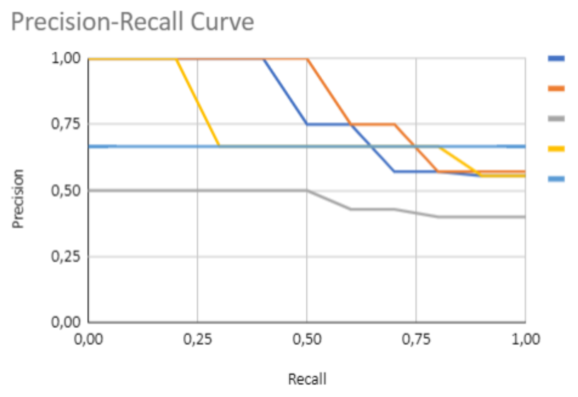


Figure 10: Precision-Recall Curve for the query "central park zoo animals".

As we can visualize in the above Precision-Recall curves, for the majority of the Information Needs, the system configuration that performed better was System 2. For different Information Needs, depending on the relevant results obtained, the Precision-Recall curves will vary. As an example, for the query "animals ice sid", almost all the results were relevant so almost all the systems are very effective, which doesn't happen with the query "kid on trip to see santa claus".

4 Search System

The final version of our search system includes not only enhancements to the previous version, as well as Cross-Language support and a Graphical User Interface. On the next sections, each one of this updates is described with further detail.

4.1 Enhancements

The final search system includes a set of enhancements that result in corrections and modifications to the previous version. On the next sections, there's a brief explanation for each one of the enhancements performed.

4.1.1 Correction of duplicate results error

The was an error before, in which some movies appeared on the results more than once. The problem was that the collection had more than one entry for those movies. The problem was solved by removing the duplicates.

4.1.2 Lexemes

To overcome the problem of the user query sometimes not matching a specific string due to be written with a slightly different English variation of the word(s), we've imported lexemes, specifically a dictionary including different grammar/lexical forms for each lexical entry, from Wikidata [13] to Solr. This was achieved by using a Python script obtained from a GitHub repository [5].

4.1.3 Indexing only the searchable fields

Following a question raised by one of our colleagues on the last presentation, we concluded that it wouldn't make sense to index all the fields if we then only made a sub-set of those searchable. This way, we end up with indexing only the searchable fields: Title, Rate, Director(s), Writer(s), Actor(s), Plot, Language, Country, Awards, Production and Reviews.

4.2 Graphical User Interface

To facilitate the interaction with our search system, we created a Graphical User Interface, using the Python Flask framework [4]. The GUI includes two pages: one for the user to input the search string and the other to see the results for a specific query.
By clicking on the search button, a Python script created for the purpose gets the user input from the HTML form and then calls the Solr API with the parameters defined previously, checking in first place if the user typed something (i.e. if the input string is not empty) and the language of the input string. If the language of the input string is different from English, the code will also convert it to English, which means that the search system works for any user input language.
After getting the JSON response with the list of the movies for the user query through the Solr API, the second HTML, receiving the response itself, creates a visually-appellant page showing the movies and its information.
The following two pictures illustrate an use of the search system GUI.

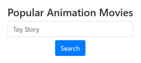


Figure 11: Search page with the input string "Toy Story".

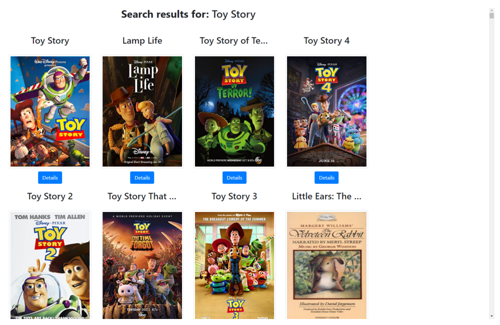
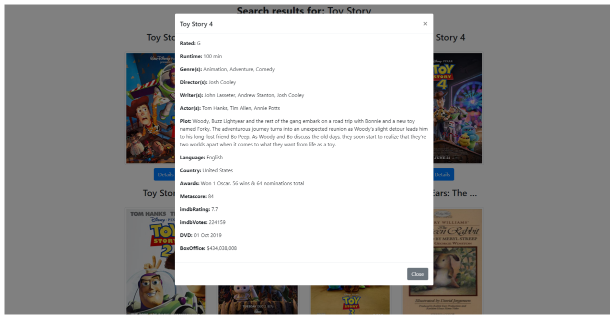


Figure 12: Results page for the input string "Toy Story".


Toy Story 4

Rated: G

Runtime: 100 min

Genre(s): Animation, Adventure, Comedy

Director(s): Josh Cooley

Writer(s): John Lasseter, Andrew Stanton, Josh Cooley

Actor(s): Tom Hanks, Tim Allen, Annie Potts

Plot: Woody, Buzz Lightyear and the rest of the gang embark on a road trip with Bonnie and a new toy named Forky. The adventurous journey turns into an unexpected reunion as Woody's slight detour leads him to his long-lost friend Bo Peep. As Woody and Bo discuss the old days, they soon start to realize that they're two worlds apart when it comes to what they want from life as a toy.

Language: English

Country: United States

Awards: Won 1 Oscar. 56 wins & 64 nominations total

Metascore: 84

imdbRating: 7.7

imdbVotes: 224159

DVD: 01 Oct 2019

BoxOffice: $434,038,008

Figure 13: Details for the movie "Toy Story 4".

4.3 Cross-language Retrieval

Using the Google Translate API for Python [6], our search system is capable of receiving queries in 106 different languages. After receiving the user search input, it is able to recognize automatically the language used from the list of supported languages and translate it to English, which is the language of our movie collection.

5 Conclusions

The objective of the project was achieved by developing the final version of the search system, starting from the data processing to data analysis and then finalizing with the information retrieval, which included the indexation, the queries, the evaluation and the development of additional features, such as the cross-language support and the GUI.
The current state of the search system allows the user to use the GUI to search for almost every animation movie in the world, using any language, obtaining the results in a displayed user-friendly interface.
Eventual future work would include the application of the same translation functions used for the user query to the translation of the results so that the GUI could be accessed and displayed in a language chosen by the user.

6 Revisions Introduced

The following list contains all the revisions introduced to the previous version of this report.

- Introduction of the subsection "Information Needs" under the section "Information Retrieval".
- New parameter used for the query process: rows.
- The "Evaluation" section was reformulated and updated according to the feedback received after the last project presentation.
- Update of the description of the 'qs', 'ps' and 'tie' eDismax parameters and the 'ps' value from 2 to 3.

References

[1] *Apache Solr*. URL: https://solr.apache.org/. Last Accessed: 13.12.2021.
[2] *Apache Solr - The DisMax Query Parser*. URL: https://solr.apache.org/guide/8_11/the-dismax-query-parser.html. Last Accessed: 13.12.2021.

[3] *Apache Solr - The Extended DisMax (eDismax) Query Parser.* URL: https://solr.apache.org/guide/8_11/the-extended-dismax-query-parser.html. Last Accessed: 13.12.2021.
[4] *Flask - Python Framework.* URL: https://flask.palletsprojects.com/en/2.0.x/. Last Accessed: 16.01.2022.
[5] *GitHub - OpenSemanticsSearch - Lexemes.* URL: https://github.com/opensemanticsearch/lexemes. Last Accessed: 17.01.2022.
[6] *Google Translator API - Python.* URL: https://py-googletrans.readthedocs.io/en/latest/. Last Accessed: 17.01.2022.
[7] *IMDB.* URL: http://www.imdb.com/. Last Accessed: 14.11.2021.
[8] *OMDB API - The Open Movie Database API.* URL: http://www.omdbapi.com/. Last Accessed: 14.11.2021.
[9] *TMDB - The Movie Database.* URL: https://www.themoviedb.org/. Last Accessed: 14.11.2021.
[10] *TMDB API - The Movie Database - About.* URL: https://www.themoviedb.org/about. Last Accessed: 13.12.2021.
[11] *TMDB API - The Movie Database - Contribution Rules.* URL: https://www.themoviedb.org/bible/general/. Last Accessed: 13.12.2021.
[12] *TMDB API - The Movie Database API.* URL: https://www.themoviedb.org/documentation/api. Last Accessed: 14.11.2021.
[13] *Wikidata.* URL: https://www.wikidata.org/wiki/Wikidata:Main_Page. Last Accessed: 17.01.2022.

7 Annexes

The following tables represent the Relevance tables for each one of the defined Information Needs. Note: **1** corresponds to a relevant document and **0** corresponds to a not relevant document.

	System 1	System 2	System 3	System 4	System 5
1	1	1	1	1	1
2	1	1	1	1	0
3	1	1	0	1	0
4	1	1	1	1	1
5	1	1	1	1	0
6	1	0	1	0	0
7	1	1	0	1	0
8	1	1	0	0	0
9	1	1	0	1	0
10	0	1	1	0	1

Table 5: Relevance table for the first Information Need.

	System 1	System 2	System 3	System 4	System 5
1	0	0	1	0	1
2	0	0	1	0	1
3	0	0	0	0	0
4	0	1	0	1	1
5	1	1	1	0	0
6	0	0	0	1	0
7	0	0	0	0	0
8	1	0	0	0	0
9	0	0	0	0	0
10	0	0	0	1	0

Table 6: Relevance table for the second Information Need.

	System 1	System 2	System 3	System 4	System 5
1	1	1	1	1	1
2	1	1	1	1	1
3	1	1	0	1	0
4	1	1	1	0	0
5	1	0	0	0	0
6	0	1	0	1	1
7	1	1	1	0	0
8	0	0	0	1	0
9	0	0	0	1	1
10	0	0	0	0	0

Table 7: Relevance table for the third Information Need.

	System 1	System 2	System 3	System 4	System 5
1	1	1	1	1	1
2	1	1	1	1	1
3	1	1	1	1	1
4	0	1	1	1	1
5	1	1	1	1	1
6	0	1	1	1	1
7	1	1	1	1	1
8	1	0	1	1	0
9	1	0	0	0	0
10	1	1	0	0	0

Table 8: Relevance table for the fourth Information Need.

	System 1	System 2	System 3	System 4	System 5
1	1	1	0	1	0
2	1	1	1	0	1
3	0	0	0	1	0
4	1	1	1	0	1
5	0	0	0	1	1
6	0	0	0	1	1
7	1	1	1	0	0
8	0	0	0	0	0
9	1	0	0	1	0
10	0	0	1	0	0

Table 9: Relevance table for the fifth Information Need.