

Information Retrieval

Retrieval System on Movies Database

December 17, 2023

Group Members

Name: Saartje Herman

Student ID: S0192618

E-mail: Saartje.Herman@student.uantwerpen.be

Name: Annila Munsaf

Student ID: S0221564

E-mail: Annila.Munsaf@student.uantwerpen.be

Name: Arbesa Jashanica

Student ID: S0193598

E-mail: Arbesa.Jashanica@student.uantwerpen.be

1 Scope

The retrieval system we've built can be used to search for movies. Our movie retrieval system offers two search options. The first allows users to search for movies by entering a description or keywords. The second approach recommends similar movies based on the genre of a specified movie title. Our project can be found in this GitHub repository: <https://github.com/AnnilaMunsaf/InformationRetrievalProject/tree/master>

2 Overview

2.1 Database

Our movie retrieval system leverages the extensive database from the Wikipedia Movie Plots dataset. This rich source encompasses a diverse collection of movie plots, providing a foundation for comprehensive searches.

The dataset consists of multiple columns:

1. Release Year - Year in which the movie was released
2. Title - Movie title
3. Origin/Ethnicity - Origin of movie (i.e. American, Bollywood, Tamil, etc.)
4. Director - Director(s)
5. Plot - Main actor and actresses
6. Genre - Movie Genre(s)
7. Wiki Page - URL of the Wikipedia page from which the plot description was scraped
8. Plot - Long form description of movie plot

For our retrieval system we only used the columns **Title**, **Release Year**, **Plot** (long form description) and **Genre**.

2.2 Algorithms

In our information retrieval system, we employ several key algorithms to enhance the search and recommendation functionalities.

2.2.1 Inverted Indexing

Inverted Indexing serves as the backbone of our system, creating an efficient data structure that maps terms to the documents containing them. This technique enables swift searches and contributes to space efficiency by indexing only unique terms. It plays a pivotal role in providing users with quick access to relevant movie information.

2.2.2 Word2Vec

We explored a state-of-the-art natural language processing method known as Word2Vec. This technique converts words into numerical vectors, capturing their meanings and relationships. While we conducted tests to observe its outcomes, we opted not to incorporate it into our final code due to results that lacked relevance to the entered queries. Nevertheless, despite its capability to represent words in a continuous vector space, the results from our experiments with Word2Vec were not promising. This model has the ability to capture semantic similarities among words, enhancing contextual understanding, but these potential benefits did not manifest in our specific experiments.

2.2.3 Levenstein Distance

Levenshtein Distance, is a metric that quantifies the similarity between two strings by measuring the minimum number of single-character edits (insertions, deletions, or substitutions) required to transform one string into the other. It provides a versatile and efficient method for quantifying the dissimilarity between strings, making it a valuable tool in diverse fields where string matching or correction is pivotal.

2.2.4 Cosine Similarity

Cosine Similarity is employed to measure the similarity between vectors in high-dimensional spaces, making it invaluable for comparing the content of movie documents. Its scale-invariant nature and wide applicability make it an effective choice for assessing the similarity between movies in our retrieval system.

2.2.5 Jaccard Similarity

Jaccard Similarity is utilized for set-based comparisons, particularly relevant when considering the overlap between sets of terms. This algorithm excels in scenarios where the order of elements is not crucial, providing an intuitive measure of set similarity and dissimilarity. It significantly contributes to our recommendation system, helping identify movies with similar descriptions to the description or keywords given by the user.

2.2.6 TF-IDF

TF-IDF is a statistical measure adopted for evaluating the importance of terms in a document relative to their prevalence across a collection. Leveraging weighted representation, TF-IDF enables our system to prioritize key terms, making it a robust tool for ranking document relevance in response to user queries.

These algorithms collectively empower our information retrieval system, contributing to its efficiency in searching for movies based on user queries and recommending similar movies tailored to individual preferences. The thoughtful integration of these techniques ensures a robust and user-friendly experience for our platform's users.

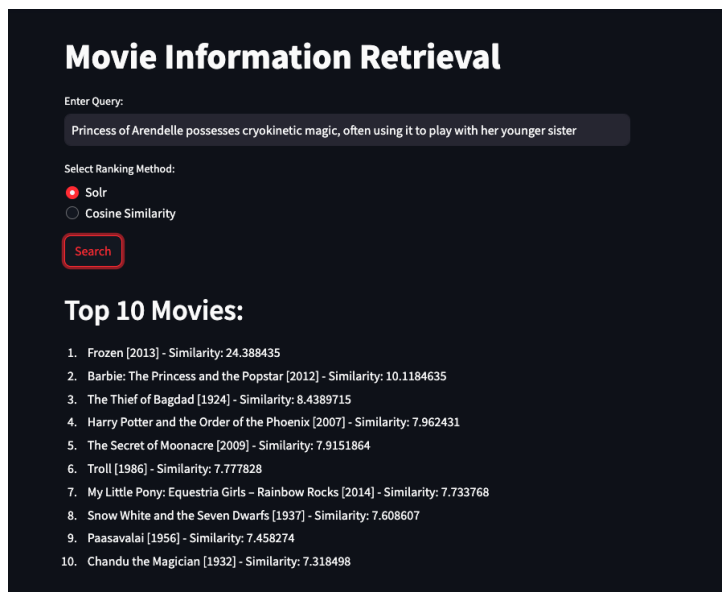
2.3 Graphical User Interface GUI

For the user interface we built a small GUI using Streamlit. Streamlit is an open-source Python library that is designed to make it easy to create web applications for data science and machine learning. It allows developers and data scientists to turn data scripts into interactive web apps with minimal effort. Streamlit is particularly popular for its simplicity and ease of use.

Upon launching the application, an initial preprocessing phase is initiated on our data. This preprocessing involves applying steps to both the Plot and Title. The creation of an inverted index is exclusive to the plot-based retrieval system. Notably, the inverted index is not utilized for the genre-based retrieval system.

We feature two distinct graphical user interfaces (GUIs): one for the Plot-Based Retrieval system and another for the Genre-Based Retrieval system.

In the first case, where we retrieve movies based on plot, the users start by entering some keywords of the movie. Then, they pick a ranking method, Solr or Cosine similarity, and based on their selection, a list of the top 10 movies is displayed. These 10 movies are the top 10 results that most closely matches the given keywords. Cosine similarity is the similarity algorithm on which our retrieval system is based and Solr is used for the evaluation part.



Movie Information Retrieval

Enter Query:

Princess of Arendelle possesses cryokinetic magic, often using it to play with her younger sister

Select Ranking Method:

☒ Solr

☐ Cosine Similarity

Search

Top 10 Movies:

1. Frozen [2013] - Similarity: 24.388435
2. Barbie: The Princess and the Popstar [2012] - Similarity: 10.1184635
3. The Thief of Bagdad [1924] - Similarity: 8.4389715
4. Harry Potter and the Order of the Phoenix [2007] - Similarity: 7.962431
5. The Secret of Moonacre [2009] - Similarity: 7.9151864
6. Troll [1986] - Similarity: 7.777828
7. My Little Pony: Equestria Girls - Rainbow Rocks [2014] - Similarity: 7.733768
8. Snow White and the Seven Dwarfs [1937] - Similarity: 7.608607
9. Paasavalai [1956] - Similarity: 7.458274
10. Chandu the Magician [1932] - Similarity: 7.318498

Figure 1: Plot-based Retrieval System (Solr)

Movie Information Retrieval

Enter Query:

Princess of Arendelle possesses cryokinetic magic, often using it to play with her younger sister

Select Ranking Method:

☐ Solr

☒ Cosine Similarity

Search

Top 10 Movies:

1. Frozen [2013] - Similarity: 0.1914
2. Ithaya Geetham [1950] - Similarity: 0.1881
3. Everything Is Rhythm [1936] - Similarity: 0.1185
4. The Slim Princess [1920] - Similarity: 0.1081
5. The Thief of Bagdad [1924] - Similarity: 0.1076
6. The Desert Hawk [1950] - Similarity: 0.1058
7. Vijayakumari [1950] - Similarity: 0.0946
8. Fiddlers Three [1948] - Similarity: 0.0906
9. Vanangamudi [1957] - Similarity: 0.0903
10. Karma [1933] - Similarity: 0.0899

Figure 2: Plot-based Retrieval System (Cosine Similarity)

In the second situation, where we recommend movies that share a genre similar to a specified movie title, the user starts by entering the movie title. Then, they pick a ranking method, and based on their selection, a list of the top 20 movies is displayed. Users can choose their preferred movie from this list. Following that, a handpicked selection of the top 10 movies with a genre similar to the chosen one is presented. Cosine and Jaccard similarity are the similarity algorithms on which our retrieval system is based and Solr is used for the evaluation part.

Movie Information Retrieval

Enter Query:

frozen

Select Ranking Method:

☐ Solr

☒ Cosine Similarity

☐ Jaccard Similarity

Search

Select a movie:

Frozen (2010) (Similarity: 1.0, distance: 1)

Selected Movie: Frozen (2010) (Similarity: 1.0, distance: 1)

Genre of Movie: horror

Show Similar Movies

10 Similar Recommended Movies:

1. The Frozen Ghost (1945)

Figure 3: Genre-Based Retrieval System

3 Implementation details

In developing our movie retrieval system, we strategically divide the implementation into three steps to ensure efficiency and effectiveness. The initial stage involves preprocessing, where we check for missing values, tokenize the words, remove stopwords, filter out short words, and applies lemmatization. Following this, text indexing is our second step where we apply techniques like inverted indexing to create structured representations of the textual data. Finally, the ranking step employs advanced algorithms, including Word2Vec, Cosine and Jaccard Similarity, and TF-IDF to prioritize and present search results based on relevance. This tripartite approach ensures a comprehensive and streamlined system, adept at delivering precise and contextually relevant movie recommendations to users.

As described in the scope, our retrieval system offers 2 search options. The first allows users to search for movies by entering a description or keywords, therefore we apply all three steps on the **Plot** column of our dataset. For the second search option which recommends similar movies based on the genre of a specified movie title, the three steps are applied on the **Title** column of our dataset.

3.1 Preprocessing steps

In the initial phase of implementing our movie retrieval system, some preprocessing steps are employed to guarantee the cleanliness and relevance of textual data. The process begins with checking for missing values, ensuring a robust foundation for subsequent analyses. Following this, we implement a series of text refinement steps.

3.1.1 Tokenization

To prepare the text for further analysis, we employ the `word_tokenize` function from the NLTK library in Python. This powerful tool breaks down the text into individual words, enabling us to extract key phrases and generate concise summaries with greater ease.

3.1.2 Stopword Removal

Common stopwords, which contribute little semantic value, are systematically eliminated from the textual data. This enhances the focus on significant terms during information retrieval. To accomplish this step, we import the stopwords from *gensim.parsing.preprocessing* and remove the stopwords from the corresponding column of the dataset depending on the search option.

3.1.3 Length Filtering

To streamline the dataset, words with fewer than three characters are excluded. This step mitigates the inclusion of overly generic or inconsequential terms in the analysis.

3.1.4 Lemmatization

The next step in our text preparation process involves lemmatization. This linguistic technique brings words back to their original form, essentially collapsing different variations of a word into a single, unified representation. This normalization process helps us gain a more holistic understanding of the underlying meaning of the text. We utilize the WordNetLemmatizer, a tool provided by the *nltk.stem* library, to effectively implement this step.

Combining these preprocessing steps results in a refined and standardized textual dataset that serves as a foundation for various downstream algorithms, including TF-IDF ranking, Word2Vec embedding, Cosine Similarity, and Jaccard Similarity. The careful curation of the dataset ensures that our movie retrieval system delivers accurate and relevant results, fostering a seamless and effective user experience. The output of preprocessing step is given below:

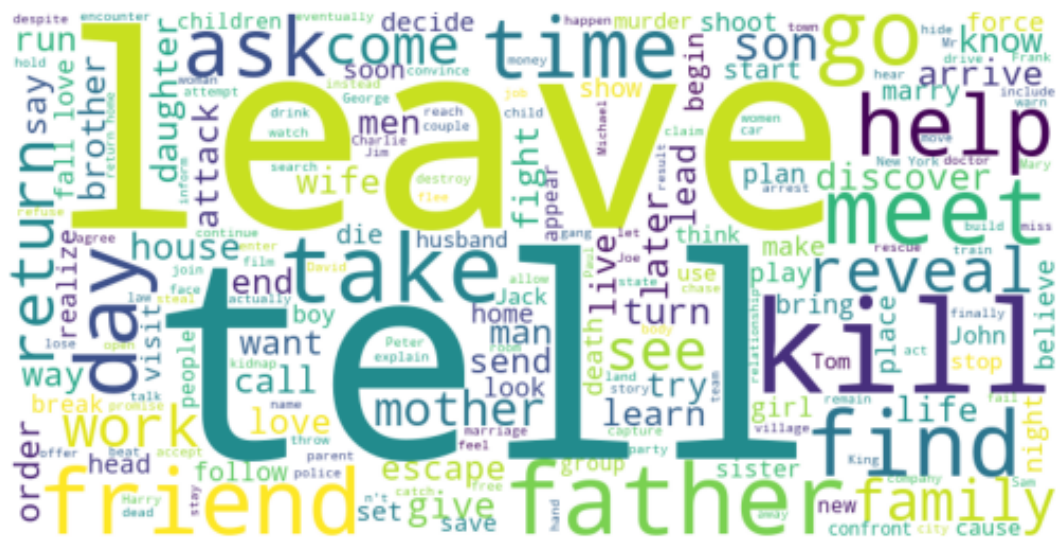


Figure 4: Output Preprocess (Plot)



÷	Release Year ÷	Title	÷ Genre ÷
25016	1977	Jadu Tona	unknown
13883	2001	K-PAX	drama
18038	2017	Noor Jahaan	romance
4129	1945	A Tale of Two Mice	animated
32791	2017	Vunnadhi Okate Zindagi	drama
12670	1996	Don't Be a Menace to South Central...	comedy
32705	2016	Lacchimdeviki O Lekkundi	comedy-thriller
31837	1984	Srivariki Premalekha	unknown
4590	1948	A-Lad-In His Lamp	animated

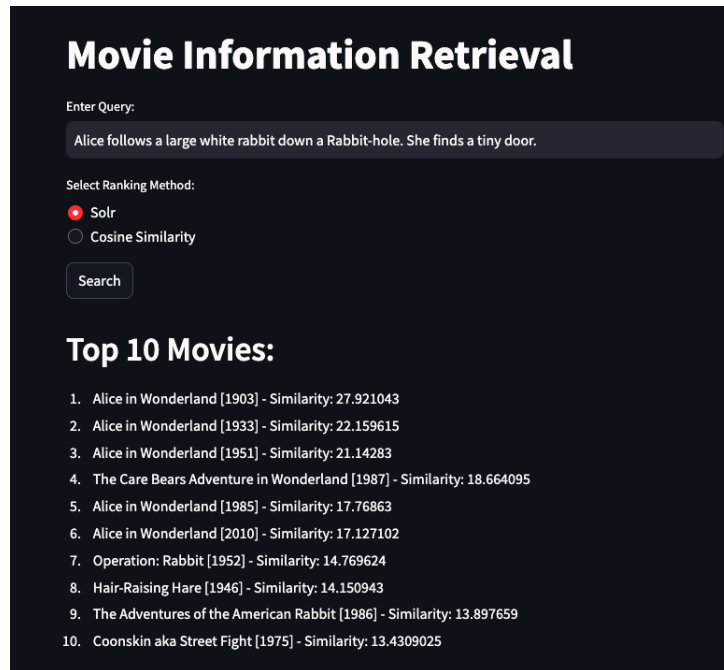
Figure 6: Word2Vec Result

3.3.2 TF-IDF with Cosine Similarity using Inverted Index (on Plot)

This is the main implementation of plot plot-based retrieval system. In this part, we take user query, the dataset with preprocessed plot and inverted index. The steps are given below:

1. First the function tokenizes the preprocessed user input using the `word_tokenize` function. Tokenization involves breaking the input text into individual words or tokens.
2. For each token in the user's input, it retrieves the corresponding indices from an inverted index.
3. The next step involves fitting the TF-IDF (Term Frequency-Inverse Document Frequency) vectorizer on a relevant subset of the data. The vectorizer is trained on the preprocessed plots of relevant movies, learning the vocabulary and document frequencies
4. The user's preprocessed input is transformed into a TF-IDF vector using the previously fitted vectorizer. This creates a numerical representation of the user's input based on the learned TF-IDF weights
5. At the final stage, Cosine similarity is calculated between the TF-IDF vector of the user's input and the TF-IDF vectors of the relevant movies' preprocessed plots. The similarity scores quantify the resemblance between the user's input and each relevant movie, helping to rank them based on similarity
6. A new column named 'Similarity (TF-IDF)' is created in the `relevant_movies` DataFrame, and it is populated with the calculated similarity scores.
7. The DataFrame `relevant_movies` is sorted based on the 'Similarity (TF-IDF)' column in descending order. Sorting the data frame in descending order allows you to prioritize movies with higher similarity scores. This arrangement facilitates the identification of the most relevant recommendations at the top of the list.

This code has been incorporated into the Streamlit Application. The outcomes for the query "Alice follows a large white rabbit down a rabbit hole. She finds a tiny door." are displayed below.



Movie Information Retrieval

Enter Query:

Alice follows a large white rabbit down a Rabbit-hole. She finds a tiny door.

Select Ranking Method:

☒ Solr

☐ Cosine Similarity

Search

Top 10 Movies:

1. Alice in Wonderland [1903] - Similarity: 27.921043
2. Alice in Wonderland [1933] - Similarity: 22.159615
3. Alice in Wonderland [1951] - Similarity: 21.14283
4. The Care Bears Adventure in Wonderland [1987] - Similarity: 18.664095
5. Alice in Wonderland [1985] - Similarity: 17.76863
6. Alice in Wonderland [2010] - Similarity: 17.127102
7. Operation: Rabbit [1952] - Similarity: 14.769624
8. Hair-Raising Hare [1946] - Similarity: 14.150943
9. The Adventures of the American Rabbit [1986] - Similarity: 13.897659
10. Coonskin aka Street Fight [1975] - Similarity: 13.4309025

Figure 7: Solr Result



Movie Information Retrieval

Enter Query:

Alice follows a large white rabbit down a Rabbit-hole. She finds a tiny door.

Select Ranking Method:

☐ Solr

☒ Cosine Similarity

Search

Top 10 Movies:

1. Alice in Wonderland [1903] - Similarity: 0.5642
2. Coonskin aka Street Fight [1975] - Similarity: 0.4845
3. Rabbit, Run [1970] - Similarity: 0.4747
4. Night of the Lepus [1972] - Similarity: 0.4533
5. Alice in Wonderland [1951] - Similarity: 0.3981
6. Alice in Wonderland [1985] - Similarity: 0.3882
7. The Adventures of the American Rabbit [1986] - Similarity: 0.3791
8. Alice in Wonderland [1933] - Similarity: 0.3600
9. Winnie the Pooh: Seasons of Giving [1999] - Similarity: 0.3237
10. The Care Bears Adventure in Wonderland [1987] - Similarity: 0.2979

Figure 8: Cosine Result

3.3.3 TF-IDF with Cosine Similarity and Levenstein Distance (on Title)

This is the main implementation of title title-based retrieval system of TF-IDF with Cosine Similarity and Levenstein Distance. In this part, we take user query and the

dataset with preprocessed titles. The steps are given below:

1. The first step is to calculate the Levensteins distance between the user query and every title of the preprocessed titles.
2. A new column named 'Levenstein distance' is created in the `relevant_movies` DataFrame, and it is populated with the calculated distances.
3. The next step involves fitting the TF-IDF (Term Frequency-Inverse Document Frequency) vectorizer on a relevant subset of the data. The vectorizer is trained on the preprocessed titles of relevant movies, learning the vocabulary and document frequencies
4. The user's input is transformed into a TF-IDF vector using the previously fitted vectorizer. This creates a numerical representation of the user's input based on the learned TF-IDF weights
5. At the final stage, **Cosine similarity** is calculated between the TF-IDF vector of the user's input and the TF-IDF vectors of the relevant movies' preprocessed titles. The similarity scores quantify the resemblance between the user's input and each relevant movie, helping to rank them based on similarity.
6. A new column named '**Similarity (TF-IDF)**' is created in the `relevant_movies` DataFrame, and it is populated with the calculated similarity scores.
7. The DataFrame `relevant_movies` is sorted based on the '**Similarity (TF-IDF)**' column in descending order and on the Levenstein distance in ascending order. Sorting the data frame in descending order on similarity and ascending order on distance allows you to prioritize movies with higher similarity scores and lower Levenstein distance. This arrangement facilitates the identification of the most relevant recommendations at the top of the list.
8. After the user selects a movie from the most relevant results based on the given title, the genre of this selected movie is taken. Lastly, 10 similar movies are selected with the same genre as the selected movie.

3.3.4 TF-IDF with Jaccard Similarity and Levenstein Distance (on Title)

The implementation of TF-IDF with Jaccard Similarity and Levenstein Distance is almost identical as with Cosine Similarity. The only changes happen in steps 5 to 7 and uses Jaccard Similarity instead of Cosine Similarity. The steps are given below:

1. The first step is to calculate the Levensteins distance between the user query and every title of the preprocessed titles.
2. A new column named 'Levenstein distance' is created in the `relevant_movies` DataFrame, and it is populated with the calculated distances.
3. The next step involves fitting the TF-IDF (Term Frequency-Inverse Document Frequency) vectorizer on a relevant subset of the data. The vectorizer is trained on the preprocessed titles of relevant movies, learning the vocabulary and document frequencies

4. The user's input is transformed into a TF-IDF vector using the previously fitted vectorizer. This creates a numerical representation of the user's input based on the learned TF-IDF weights
5. At the final stage, **Jaccard similarity** is calculated between the TF-IDF vector of the user's input and the TF-IDF vectors of the relevant movies' preprocessed titles. The similarity scores quantify the resemblance between the user's input and each relevant movie, helping to rank them based on similarity.
6. A new column named '**Similarity (Jaccard)**' is created in the `relevant_movies` DataFrame, and it is populated with the calculated similarity scores.
7. The DataFrame `relevant_movies` is sorted based on the '**Similarity (Jaccard)**' column in descending order and on the Levenstein distance in ascending order. Sorting the data frame in descending order on similarity and ascending order on distance allows you to prioritize movies with higher similarity scores and lower Levenstein distance. This arrangement facilitates the identification of the most relevant recommendations at the top of the list.
8. After the user selects a movie from the most relevant results based on the given title, the genre of this selected movie is taken. Lastly, 10 similar movies are selected with the same genre as the selected movie.

4 Evaluation Criteria

In our evaluation, **Apache Solr** serves as a reliable benchmark to assess the quality and effectiveness of our recommendation system. To use Solr, we created an index that includes movie details such as title, release year, plot, genre, and more. Once the index is set up, Solr allows for quick and efficient searches to retrieve specific movie information.

4.1 Cosine/Jaccard Similarity Comparison:

Evaluation Metric: Compare Cosine similarity (and Jaccard similarity for second search option) scores between our system and Solr.

Rationale: Understanding the alignment and relative distribution of similarity scores ensures a qualitative assessment of our recommendation system compared to an industry-standard search engine. While absolute score values may differ due to normalization or scoring mechanisms, the relative consistency and order of scores are essential for evaluating the effectiveness of our recommendations.

4.2 Top-K Comparison:

Evaluation Metric: Retrieve the top 10 results for each query from both systems and compare the lists.

Rationale: Focusing on the top results allows to assess the accuracy of the most relevant recommendations, as users often pay more attention to the top-ranked items.

4.3 Consistency in Top Recommendations:

Evaluation Metric: Examine the consistency of top recommendations between our system and Solr, considering the position of movies in the ranked lists.

Rationale: A high degree of consistency enhances user trust in your recommendation system. Evaluate if movies ranked at the top by Solr are also positioned prominently in our system’s recommendations.

5 Quantitative results

Now, let’s discuss the outcomes of the two different cases. Our objective is to assess whether our ranking methods, namely Cosine and Jaccard similarity, outperform Solr.

For the first case where we want to get the top results based on plot, the following table (table 1) is given. Here we have 4 columns, the title column is the movie the user wants and the query is the plot or keywords the user gives. Lastly we have Cosine and Solr similarity. These last two columns always indicate the score and position of the film that the user wants.

Our aim is to achieve results that are either equal to or better than those produced by Solr. Upon analysis, it becomes evident that Solr consistently provides superior results and is more effective at identifying the correct movies. While our systems also yield satisfactory outcomes, the rankings may vary. This discrepancy is attributed to our approach, which relies on token overlap between user input and movie plots. In situations where there is limited token overlap between movie plots and user queries, the TF-IDF approach may encounter challenges in capturing relevant similarities, resulting in less accurate recommendations.

Title	Query	Cosine	Solr
Alice in Wonderland (1903)	Alice follows a large white rabbit down a Rabbithole. She finds a tiny door.	Score: 0,56 Place 1	Score: 27,9 Place 1
Frozen (2013)	Princess of Arendelle possesses cryokinetic magic, often using it to play with her younger sister	Score: 0,19 Place 1	Score: 24,39 Place 1
Harry Potter and Sorcerer’s Stone (2001)	Harry is the orphaned son of two wizards who met their demise at the hands of Lord Voldemort	Score: 0,36 Place 6	Score: 22,21 Place 1

Table 1: Results Retrieval System based on Plot

For the second case where we want to get the top results based on title, the following table (table 2) is given. Here we have 6 columns, the title column is the movie the user wants and the query is the title the user gives. We assume that the user sometimes doesn't know the whole title or that there are spelling mistakes. The column distance stands for the Levenstein distance and lastly we have Cosine, Jaccard and Solr similarity. These last three columns always indicate the score and position of the film that the user wants.

We can conclude that in most cases our system is doing better than Solr, based on the place of the movie we want. When comparing Jaccard Similarity with Solr, Jaccard gives better or equal results as Solr. For Cosine Similarity we also get mostly better or equal results as Solr, except for the last query is Cosine performing worse.

Overall, based on these results, we can say that our retrieval system is performing quite well.

Title	Query	Distance	Cosine	Jaccard	Solr
Frozen	frozen	1	Score: 1,0 Place 1	Score: 1,0 Place 1	Score: 5,04 Place 1
Harry Potter and The Philosopher's Stone	harry potter stone	14	Score: 0,83 Place 2	Score: 1,0 Place 2	Score: 6,66 Place 2
Pirates of the Caribbean: The Curse of the Black Pearl	pirates caribbean pearl	17	Score: 0,67 Place 1	Score: 0,67 Place 1	Score: 3,44 Place 6
Interstellar	interstellar	1	Score: 1,0 Place 1	Score: 1,0 Place 1	Score: 5,85 Place 1
Toy Story	Toy	6	Score: 0,82 Place 2	Score: 1,0 Place 4	Score: 4,19 Place 4
The Shawshank Redemption	shaw redemption	7	Score: 0,69 Place 1	Score: 0,5 Place 1	Score: 4,17 Place 1
Forrest Gump	forest gump	3	Score: 0,56 Place 6	Score: 0,5 Place 1	Score: 5,13 Place 1

Table 2: Results Retrieval System based on Title

6 Limitations

1. In the system, the preprocessing step, encompassing tasks such as stopword removal, tokenization, and the creation of an inverted index, contributes to a slowdown in the retrieval process based on movie plots. The impact is particularly noticeable when users input lengthy queries, as the system takes more time to calculate cosine similarity between the user's input and movie plot embeddings. This results in increased waiting times for users, making the overall retrieval system less efficient. Therefore, the system's performance may be a limiting factor, especially when handling extensive user queries, leading to potential user dissatisfaction due to prolonged response times.
2. Another limitation of our system is its dependency on a specific database structure imported from Kaggle, relying on predefined column names. If another database with different column names is introduced, the system may not function properly, as it expects specific attributes for processing. This lack of flexibility to adapt to

varying database structures could pose a challenge in integrating diverse datasets, restricting the system's compatibility and scalability with datasets beyond the initially utilized Kaggle source.

3. The efficiency of the inverted index depends on its size. If the movie dataset is extensive, the inverted index could become large, potentially impacting memory usage and retrieval times.
4. The function relies on token overlap between the user input and movie plots. In cases where movie plots have limited token overlap with user queries, the TF-IDF approach may struggle to capture relevant similarities, leading to less accurate recommendations.
5. The function uses a static TF-IDF model based on the relevant subset of the data. If the dataset evolves or updates over time, the model may become outdated, affecting the system's adaptability to changes in the underlying data.

7 References/Links

1. Project on Github: <https://github.com/AnnilaMunsaf/InformationRetrievalProject/tree/master>
2. <https://www.kaggle.com/datasets/jrobischon/wikipedia-movie-plots/>
3. <https://medium.com/web-mining-is688-spring-2021/content-based-movie-recommendation-system-72f122641eab>
4. <https://towardsdatascience.com/using-cosine-similarity-to-build-a-movie-recommendation-system-ae7f20842599>
5. <https://medium.com/data-science-at-microsoft/search-and-ranking-for-information-retrieval-ir-5f9ca52dd056>
6. <https://marketbrew.ai/search-engine-indexing-understanding-the-inner-workings-of-google-and-other-search-engines>
7. Book: "Introduction to Information Retrieval" by Christopher D. Manning, Prabhakar Raghavan, and Hinrich Schütze
8. <https://chat.openai.com>
9. Course lecture slides

7.1 Used Libraries

1. TF-IDF: https://scikit-learn.org/stable/modules/generated/sklearn.feature_extraction.text.TfidfVectorizer.html
2. Cosine Similarity: https://scikit-learn.org/stable/modules/generated/sklearn.metrics.pairwise.cosine_similarity.html

3. Jaccard Similarity: https://scikit-learn.org/stable/modules/generated/sklearn.metrics.jaccard_score.html
4. Levenstein Distance: <https://pypi.org/project/python-Levenshtein/>