|  |  |  |
| --- | --- | --- |
| A logo on a black background  Description automatically generated | | |
| **Faculty of Science and Technology**  **CISC3014 – Information Retrieval and Web Search** | | |
| **Project Title: Plot Search Using TF-IDF Model from Popular list of Rotten Tomatoes** | | |
| **Group Members:** | | |
| Huang Yanzhen | | DC126732 |
|  | Chen Zirui | DC127901 |

Table of Contents

[1. Introduction 3](#_Toc151420622)

[2. First Crawler 3](#_Toc151420623)

[3. Second Crawler 5](#_Toc151420624)

[4. TF-IDF Model Building 6](#_Toc151420625)

[4.1. Tokenize each article into an array. 6](#_Toc151420626)

[4.2. Create term frequency vector. 7](#_Toc151420627)

[4.3. Further combine tf vectors into tf matrix. 7](#_Toc151420628)

[4.4. Using tf matrix, calculate inverse document frequency vector, then build tf-idf matrix. 9](#_Toc151420629)

[5. Query Search & Problems 10](#_Toc151420630)

[5.1. Cosine Similarity 10](#_Toc151420631)

[5.2. Exception Handler: Unknown words. 11](#_Toc151420632)

[5.3. Display Results 11](#_Toc151420633)

[5.4. Problem Handler: Common Words Problem. 13](#_Toc151420634)

[5.5. Search Results Examples 14](#_Toc151420635)

[6. Code reviews 16](#_Toc151420636)

# 1. Introduction

Rotten Tomatoes is a review-aggregation website for film and television in the U.S. It has its own ranking system of movies, with three tiers: Certified Fresh, Fresh, and Rotten. A screenshot of the *rottentomatoes.com* site:

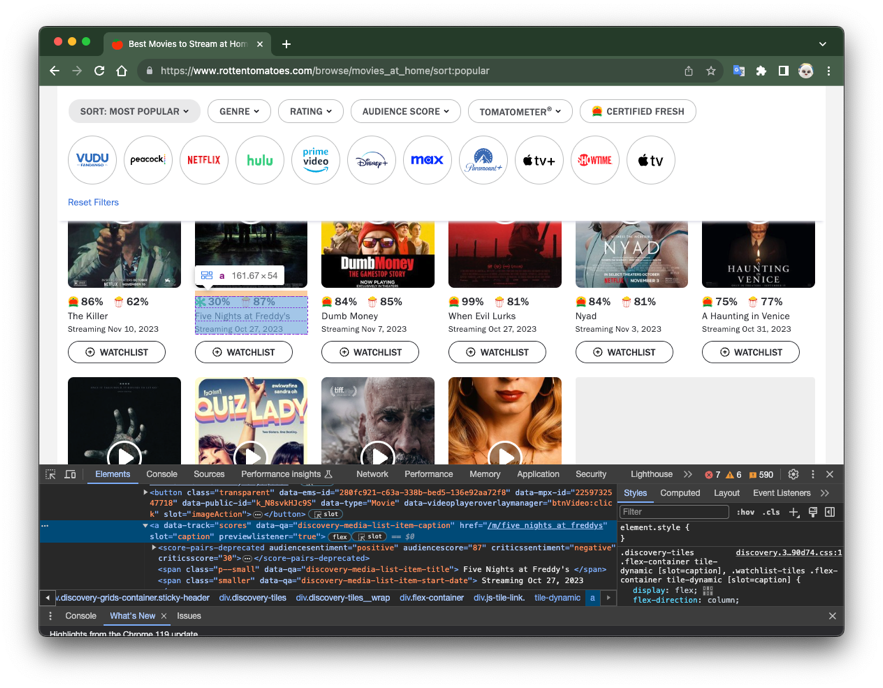
A screenshot of a computer

Description automatically generated

The goal of our project is to extract the main content of top 100+ list from RT, then make a query searcher based on the plot twists using TF-IDF model. This searcher allows users to search most relevant movies using simple queries.

# 2. First Crawler

In *rottentomatoes.com,* the movies collection is presented as a grid view of <div> container of attribute class="flex-container". Within each container, there's an <a> tab containing an @href attribute that stores the sub-link to the movie details.



Intuitively, we craw the entire list of movies by xpath:

|  |
| --- |
| movie\_list = response.xpath("//div[@class='flex-container']") |

Iterate this list. Within each list item, we first get the tag, and then retrieve its attributes:

|  |
| --- |
| score\_container = movie.xpath(".//a[@data-track='scores']")  score\_link = score\_container.xpath("@href").get() |

Lastly, encapsulate this data into a data frame, and store in an excel file.

|  |
| --- |
| data = {  "title": movie\_title,  "stream\_time": stream\_time,  'link': score\_link,  "audience score": aud\_score,  "critics score": critics\_score,  "audience sentiment": bin\_aud\_sentiment,  "critics sentiment": bin\_critics\_sentiment,  }    *# Store the data in a new Excel file*  self.start\_urls.append("https://www.rottentomatoes.com/" + data['link'])  **if** self.custom\_settings['SAVE\_DATA']:  \_\_save\_data\_\_.save\_data\_to\_excel(data) |

The excel file is movie\_data.xls:

A screenshot of a computer

Description automatically generated

# 3. Second Crawler

Having the URL list, we have the second crawler to crawl movie contents of each movie. One movie, having one URL, leads to one movie content, i.e., plots. We wrote a special function to read excel file and form an array of movie URLs. These URLs are encapsulated into a URL array that's ' used as the start\_urls attribute of the second crawler.

|  |
| --- |
| **def** get\_movie\_url():  *# Read Excel file and do analysis*  file\_path = './movie\_list/movie\_data.xlsx'  movie\_data = pd.read\_excel(file\_path)  *# Get URL*  url\_series = movie\_data['link']  sub\_urls = url\_series.values    *# Header URL*  header\_url = 'https://www.rottentomatoes.com'  urls = np.array(sub\_urls, dtype=object)  urls\_with\_header = header\_url + urls    **print**(len(urls\_with\_header))  **return** urls\_with\_header |

The second crawler calls this function to store the urls to be crawled. After this, it calls a start\_requests() function to parse every URL with the for-loop contained in it.

|  |
| --- |
| **def** start\_requests(self):  **for** url **in** self.start\_urls:  headers = {  'User-Agent': self.get\_random\_user\_agent()  }  **yield** scrapy.Request(url=url, headers=headers, callback=self.parse) |

The rest of the crawling shares the same idea with the first crawler. The plot twist is stored in a <p> element with parameter slot= “content”. We retrieve the title & contents of each crawler and store them into an Excel file.

A screen shot of a computer

Description automatically generated

Before storing each plot twist, we remove all the return and tab characters. The excel file is named movie\_content.xls:

A screenshot of a computer

Description automatically generated

Evidently, each movie corresponds to its own plot twists. These articles will then be used to build a tf-idf search model. The parsing function is shown below:

|  |
| --- |
| **def** parse(self, response):  title = response.xpath("//h1[@class='title']/text()").get()  genre = response.xpath("//span[@class='genre']/text()").get()  genre = re.sub(r'**\n**|**\s**|**\t**', '', genre)    content = response.xpath("//p[@slot='content']/text()").get()  content = re.sub(r'**\n**|**\t**', '', content)    data = {  "title": title,  "genre": genre,  "content": content,  }    *# Save data*  **if** self.custom\_settings['SAVE\_DATA']:  \_\_save\_data\_\_.save\_data\_to\_excel(data, file\_name="movie\_content")    **print**("**\n** title:" + title)  **print**("**\n** genre:" + genre)  **print**("**\n** content:**\n**" + content) |

# 4. TF-IDF Model Building

## 4.1. Tokenize each article into an array.

This function parses a continuous sentence into an array of words. An English sentence is basically words separated by delimiters like spaces, commas, periods, and many other symbols. The function compiles a regular expression rule of parsing using these delimiters. For example, the sentence “The cake tastes like cake.” will be parsed into array ["the", "cake", "tastes", "like", "cake"].

|  |
| --- |
| **def** tokenize(input\_str):  *# Define the splitting delimiters using regular expression*  rule = r'[**\s\~\`\!\@\#\$\%\^\&\\*\(\)\-\\_\+\=\{\}\[\]\;\:\'\"\,\<\.\>\/\?\\**|]+'  re.compile(rule)    *# Turn all letters in the string into lowercase*  *# This may contain empty member ''*  terms\_ = []  terms\_ = terms\_ + re.split(rule, input\_str.lower())    *# Remove the empty member ''*  terms = []  **for** term **in** terms\_:  **if** term != '':  terms.append(term)    last\_word = terms[-1]  *# print("last\_word: " + last\_word)*  **return** terms |

## 

## 4.2. Create term frequency vector.

The get\_term\_freq() function takes an input of a tokenized array of strings, counts duplicates within the array using the Counter() function, and merge them together, yielding a term-frequency number for each term. For example, the query ["the", "cake", "tastes", "like", "cake"] would be merged as {"the":1, "cake":2, "tastes":1, "like":1}.

|  |
| --- |
| **def** get\_term\_freq(movie\_item):  title = movie\_item['title']  content = movie\_item['content']    *# Split content article into word array.*  term\_array = tokenize(content)    *# Using word array, count term frequency.*  *# Term frequency: term:key -> frequency:value*  movie\_tf = Counter(term\_array)  movie\_tf = dict(movie\_tf)    *# Console logs*  **if** \_\_settings\_\_.custom\_settings['CONSOLE\_LOG\_PROCESS']:  **print**("**\n**>> " + title)  *# print(term\_array)*  **print**(movie\_tf)    **return** movie\_tf, title |

## 4.3. Further combine tf vectors into tf matrix.

We would first build the index of the matrix, named vocabulary, which is an array of all the word that's ever existed in the queries. This requires performing a set operation on all the tf vectors.

|  |
| --- |
| **def** create\_vocabulary(path="./movie\_list/movie\_content.xlsx"):  movie\_content = xls\_to\_df(file\_path=path)    *# Initialize vocabulary set and term frequency array.*  vocab = set()  term\_freqs = []  titles = []    *# For ALL movies:*  **for** index, row **in** movie\_content.iterrows():  tf, title = get\_term\_freq(row) *# Get its term frequency vector.*  titles.append(title) *# Merge terms into vocabulary first*  vocab.update(tf.keys())  term\_freqs.append(tf) *# Store in a unified term frequency matrix.*  vocab = list(vocab)    *# Console Log*  **if** \_\_settings\_\_.custom\_settings['CONSOLE\_LOG\_PROCESS']:  **print**("**\n**>>>> Vocab")  **print**(vocab)  **return** vocab, term\_freqs, titles |

Besides, create\_vocabulary() also preserves the sequence of movie titles, which will be used to match movie by their sequence IDs.

Having the index, we just insert data into the matrix. For each plot twist, i.e., each tf vector, for each word within the vector, traverse the index until the word is found, then insert it.

|  |
| --- |
| **def** create\_tf\_mat(path="./movie\_list/movie\_content.xlsx"):  *# First, extract vocabulary & term frequency 2D vector.*  vocab, term\_freqs, titles = create\_vocabulary(path=path)    *# Initializes term frequency matrix.*  term\_freq\_mat = pd.DataFrame(np.zeros((len(vocab), len(term\_freqs))), index=vocab)    *# Insert data into the matrix.*  **for** index, term\_freq **in** enumerate(term\_freqs):  **for** key, value **in** term\_freq.items():  term\_freq\_mat.loc[key, index] = value    **if** \_\_settings\_\_.custom\_settings['CONSOLE\_LOG\_PROCESS']:  **print**('**\n**>>>> Term Frequency Matrix')  **print**(term\_freq\_mat)    **return** term\_freq\_mat, titles |

The rows are the frequency vector of each term, and the columns are frequency vector of each term in a specific plot twist. The abbreviated term frequency matrix is shown below. It is obvious that there are lots of fragmentation in the matrix.

|  |
| --- |
| >>>> Term Frequency Matrix  0 1 2 3 4 5 6 ... 110 111 112 113 114 115 116  zone 0.0 0.0 0.0 0.0 0.0 0.0 0.0 ... 0.0 0.0 0.0 0.0 0.0 0.0 0.0  renfield 0.0 0.0 0.0 0.0 0.0 0.0 0.0 ... 0.0 0.0 0.0 0.0 0.0 0.0 0.0  temple 0.0 0.0 0.0 0.0 0.0 0.0 0.0 ... 0.0 0.0 0.0 0.0 0.0 0.0 0.0  storied 0.0 0.0 0.0 0.0 0.0 0.0 0.0 ... 0.0 0.0 0.0 0.0 0.0 0.0 0.0  issue 0.0 0.0 0.0 0.0 0.0 0.0 0.0 ... 0.0 0.0 0.0 0.0 0.0 0.0 0.0  ... ... ... ... ... ... ... ... ... ... ... ... ... ... ... ...  scarecrow 0.0 0.0 0.0 0.0 0.0 0.0 0.0 ... 0.0 0.0 0.0 0.0 0.0 0.0 0.0  puzzle 0.0 0.0 0.0 0.0 0.0 0.0 0.0 ... 0.0 0.0 0.0 0.0 0.0 0.0 0.0  leaving 0.0 0.0 0.0 0.0 0.0 0.0 0.0 ... 0.0 0.0 0.0 0.0 0.0 0.0 0.0  sappy 0.0 0.0 0.0 0.0 0.0 0.0 0.0 ... 0.0 0.0 0.0 0.0 0.0 0.0 0.0  enjoying 0.0 0.0 0.0 0.0 0.0 0.0 0.0 ... 0.0 0.0 0.0 0.0 0.0 0.0 1.0 |

## 4.4. Using tf matrix, calculate inverse document frequency vector, then build tf-idf matrix.

Given the term frequency matrix, the inverse document frequency is given by:

The idf vector is shown below. The index of the idf vector is the index of the tf matrix.

|  |
| --- |
| >>>> Inverse Document Frequency  [[2.38108697]  [2.38108697]  [2.38108697]  ...  [2.38108697]  [2.38108697]  [1.58739131]] |

In a term-wise manner, time the idf vector to each column of the tf matrix. The result will be the tf-idf matrix. This is done by the create\_tfidf\_mat() function. Besides the tf-idf matrix, this function also hands out the idf\_vector it creates.

|  |
| --- |
| **def** create\_tf\_mat(path="./movie\_list/movie\_content.xlsx"):  *# First, extract vocabulary & term frequency 2D vector.*  vocab, term\_freqs, titles = create\_vocabulary(path=path)    *# Initializes term frequency matrix.*  term\_freq\_mat = pd.DataFrame(np.zeros((len(vocab), len(term\_freqs))), index=vocab)    *# Insert data into the matrix.*  **for** index, term\_freq **in** enumerate(term\_freqs):  **for** key, value **in** term\_freq.items():  term\_freq\_mat.loc[key, index] = value    **if** \_\_settings\_\_.custom\_settings['CONSOLE\_LOG\_PROCESS']:  **print**('**\n**>>>> Term Frequency Matrix')  **print**(term\_freq\_mat)    **return** term\_freq\_mat, titles |

The abbreviated tf-idf matrix is shown below.

|  |
| --- |
| >>>> tf-idf Matrix  0 1 2 3 4 5 ... 111 112 113 114 115 116  zone 0.0 0.0 0.0 0.0 0.0 0.0 ... 0.0 0.0 0.0 0.0 0.0 0.000000  renfield 0.0 0.0 0.0 0.0 0.0 0.0 ... 0.0 0.0 0.0 0.0 0.0 0.000000  temple 0.0 0.0 0.0 0.0 0.0 0.0 ... 0.0 0.0 0.0 0.0 0.0 0.000000  storied 0.0 0.0 0.0 0.0 0.0 0.0 ... 0.0 0.0 0.0 0.0 0.0 0.000000  issue 0.0 0.0 0.0 0.0 0.0 0.0 ... 0.0 0.0 0.0 0.0 0.0 0.000000  ... ... ... ... ... ... ... ... ... ... ... ... ... ...  scarecrow 0.0 0.0 0.0 0.0 0.0 0.0 ... 0.0 0.0 0.0 0.0 0.0 0.000000  puzzle 0.0 0.0 0.0 0.0 0.0 0.0 ... 0.0 0.0 0.0 0.0 0.0 0.000000  leaving 0.0 0.0 0.0 0.0 0.0 0.0 ... 0.0 0.0 0.0 0.0 0.0 0.000000  sappy 0.0 0.0 0.0 0.0 0.0 0.0 ... 0.0 0.0 0.0 0.0 0.0 0.000000  enjoying 0.0 0.0 0.0 0.0 0.0 0.0 ... 0.0 0.0 0.0 0.0 0.0 1.587391  [2898 rows x 117 columns] |

# 5. Query Search & Problems

## 5.1. Cosine Similarity

Cosine similarity will be performed to measure the similarity between the query and a specific plot twist, which is a column in the tf-idf matrix. It is given by:

## 

The geometric expression of the cosine simlarity score is the cosine value of the angle between the two query vectors in the N-dimensiuonal vector space. Hence the distance (i.e. how long the query or the plot twist is) is not considered. Given a query and a tf-idf matrix, a score indexed by movies is gtiven by this cosine\_compare() function:

|  |
| --- |
| **def** cosine\_compare(query, idf\_vector, tfidf\_mat):  *# Cosine Similarity*  **def** cosine(q, d):  q = q.T *# Transpose vector to fit the dot op.*  cos\_sim = np.dot(q, d) / (np.linalg.norm(q) \* np.linalg.norm(d))  **return** cos\_sim.item()    *# Query tf-idf Vector*  **def** create\_query\_tfidf\_vector(query, idf\_vector):  *# Tokenizes query into term 2D vector*  q\_term = tokenize(query)  q\_term\_freq = Counter(q\_term) *# remove duplicates, make into dictionary*  q\_term\_freq = dict(q\_term\_freq)    *# Query tf vector*  q\_tf\_vector = pd.DataFrame(np.zeros((len(idf\_vector), 1)), index=tfidf\_mat.index)  **for** key, value **in** q\_term\_freq.items():  q\_tf\_vector.loc[key] = value    *# Query tfidf vector*  *# Error handling: Size doesn't mach*  **if** q\_tf\_vector.shape[0] != idf\_vector.shape[0] **or** q\_tf\_vector.shape[1] != idf\_vector.shape[1]:  **return** q\_tf\_vector, False  *# Size matches*  q\_tfidf\_vector = q\_tf\_vector \* idf\_vector  **return** q\_tfidf\_vector, True    *# Compare query tfidf vector with all columns of tfidf\_mat*  q\_tfidf\_vector, is\_success = create\_query\_tfidf\_vector(query, idf\_vector)  *# Error handling: Size don't match*  **if** **not** is\_success:  **return** [], False  *# Size matches, continue.*  similarity\_scores = []  **for** doc **in** tfidf\_mat.columns:  doc\_vector = tfidf\_mat[doc]  similarity\_scores.append(cosine(q\_tfidf\_vector, doc\_vector))    **return** similarity\_scores, True |

We first turn query into a tf-idf vector by timing it term-wisely with the idf vector. Then, we compare the query with each column of the tf-idf matrix, i.e. a plot twist. If no exception occurs, the function will return an array of similarity scores. The index of the array is the movies. To find the movie title, we need to match the corresponding index with the titles array returned by the create\_tf\_mat() function.

## 5.2. Exception Handler: Unknown words.

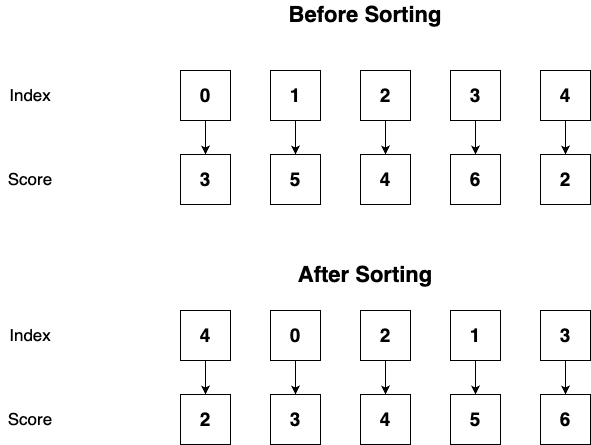
One drawback of the tf-idf model is that it won't recognize any query word that doesn't exist in the index of the tf-idf matrix. Giving a new term in the query will cause an exception that the length of the query tf-idf vector will become longer than the index of the matrix, resulting a shape-unmatch. To prevent python from halting, the exception handler is place as a guardian:

|  |
| --- |
| **if** q\_tf\_vector.shape[0] != idf\_vector.shape[0] **or** q\_tf\_vector.shape[1] != idf\_vector.shape[1]:  **return** q\_tf\_vector, False |

It basically just detects a shape-unmatch in advance and skip the following code that's meant to be failing. Once a failure is detected, a False value will be returned by the cosine\_compare() function.

## 5.3. Display Results

The key of displaying results is to sort the score array while still preserving its corresponding index. The index is an integer pointing to the titles array that stores the movie titles. The idea is shown below:



This get\_top\_x\_id() function will return the indexes (not values) of the top x scored movies. It sorts an array into a descending order using the argsort() feature of numpy and extract the index of the top x (x is a variable) records.

|  |
| --- |
| **def** get\_top\_x\_id(similarity\_scores, top\_x):  *# Fetch top x most relevant.*  *# Sort array into descending order. Keep the original index.*  sorted\_similarity\_scores = np.argsort(similarity\_scores)[::-1]  top\_x\_id = sorted\_similarity\_scores[:top\_x]  **return** top\_x\_id |

Having the indexes to movies, we can match it to the titles array to retrieve the movie titles. This is perfomed by the get\_top\_x\_names() function.

|  |
| --- |
| **def** get\_top\_x\_names(similarity\_scores, top\_x, titles):  top\_x\_id = get\_top\_x\_id(similarity\_scores, top\_x)  top\_x\_names = []  **for** id **in** top\_x\_id:  top\_x\_names.append(titles[id])  **return** top\_x\_names |

The search() function takes an input of a search query, and calls the cosine\_compare() function to perform the scoring. Besides printing the result, it also prints the similarity scores. To find the score, we should first find the top x indexes, and then use the index to find the corresponding score.

|  |
| --- |
| **def** search(search\_queries, idf\_vector, tfidf\_mat, titles, top\_x):  **if** len(search\_queries) > 1:  **print**("------------- Totally " + str(len(search\_queries)) + " search attempts! -------------")    **for** index\_search, query **in** enumerate(search\_queries):  *# Scores, in sequence of movies*  similarity\_scores, is\_success = cosine\_compare(query, idf\_vector, tfidf\_mat)  *# Exception: Query size doesn't fit!*  **if** **not** is\_success:  **print**("**\0**33[31m$ Warning: Word not exist, try another one.**\0**33[0m**\n**")  **return**    top\_x\_id = get\_top\_x\_id(similarity\_scores, top\_x)  top\_x\_names = get\_top\_x\_names(similarity\_scores, top\_x, titles)    *# Print Results*  *# Title*  index = str(index\_search) + ". " **if** len(search\_queries) > 1 **else** ""  **print**("**\0**33[32m" + index + "Searched for: **\"**" + query + "**\"\n**" +  "Top " + str(top\_x) + " relevant:" + "**\0**33[0m"  )    *# Topx x result!*  **for** index\_top **in** range(0, len(top\_x\_id)):  *# index\_top -> top\_x\_id -> score*  this\_similarity\_score = similarity\_scores[top\_x\_id[index\_top]]  **if** this\_similarity\_score == 0:  **print**("**\0**33[33m**\n**$ Warning: No more related movies!!**\0**33[0m")  **break**  **print**(str(index\_top + 1) + ".**\n**" +  "ID: " + str(top\_x\_id[index\_top] + 2) + "**\n**" +  "Title: " + str(top\_x\_names[index\_top]) + "**\n**" +  "Sim score: " + str(this\_similarity\_score)  )  **print**("**\0**33[32mSearch is complete!**\0**33[0m") |

On receiving a False message from cosine\_compare(), instead of raising a python error, search() is designed to handle this by printing a prompt message.

Lastly, to make a consistent user interface, we uses a while-loop to constantly takes user input. This prevents a process-restart and a re-build of the tf-idf matrix.

|  |
| --- |
| **while** True:  query\_arr\_encap = []  input\_query = input("**\n**>> What do you want to search? ") *# User input a search query.*  *# Read user inputs.*  **if** input\_query == \_\_settings\_\_.special\_scripts['BREAK\_WHILE\_LOOP']:  *# A means to halt the while-loop.*  **break**  **if** input\_query == \_\_settings\_\_.special\_scripts['LIST\_ALL']:  *# List all movies*  **for** index, title **in** enumerate(titles):  **print**("ID: " + str(index+2) + "**\n**" + "Title: " + title + "**\n**")  **continue**    *# Search the user input query*  query\_arr\_encap.append(input\_query)  search(query\_arr\_encap, idf\_vector, tfidf\_mat, titles, \_top\_x) |

This loop is designed to only halt when the user types in the pre-set string break(). Searcher can also list all the movies by typing the ls command. A shape-unmatch exception caused by an unknown word won't terminate it because it is handled in the search() function.

## 5.4. Problem Handler: Common Words Problem.

A problem caused by common word is discovered during project development. A great example is that, the top search result for query "six year old" is:

|  |
| --- |
| 1.  ID: 79  Title: Host  Sim score: 0.17905439885819746 |

While the top seasrch result for query “year old” is:

|  |
| --- |
| 1.  ID: 69  Title: Halloween  Sim score: 0.15211712289647808 |

The problem is that the word "six" has been in the plot twist for "Host" for so many times, so that it contributes more than expected to our search. In fact, even if the idf value of some word is very small (hence they are very common and shouldn't dominate the search result), yet the term frequency in one plot twist is large enough, it is still able to contribute sufficiently to the search result.

Our solution is idf casting. To be frank, it just cuts the word that's existed for a certain amount. For example, if I don't want the word that has existed in more than 50% of the document, I can just assign a 0 to any idf that is equal or below to this value:

This can be performed when calculating inverse document frequency by:

|  |
| --- |
| **if** \_\_settings\_\_.custom\_settings['RM\_COMMON\_WORDS']:  min\_idf = np.log(doc\_num) / (1 + doc\_num)  idf[idf == min\_idf] = 0 |

This allows us to prevent any word that's to some extent too common among plot twists from contributing to the search result.

## 5.5. Search Results Examples

Regular search results:

|  |  |
| --- | --- |
| A screenshot of a computer  Description automatically generated | A screenshot of a computer  Description automatically generated |
|  |  |
| A screenshot of a computer  Description automatically generated |  |

An exception handling result:

A black background with white text

Description automatically generated

# 6. Code reviews

Please kindly refer to this link for source code:

<https://github.com/YanzhenHuang/CISC3014-IR-and-WebSearch-Project.git>