澳門大學
UNIVERSIDADE DE MACAU
UNIVERSITY OF MACAU

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

2. First Crawler 3

3. Second Crawler 5

4. TF-IDF Model Building 7

4.1. Tokenize each article into an array. 7

4.2. Create term frequency vector. 7

4.3. Further combine tf vectors into tf matrix. 8

4.4. Using tf matrix, calculate inverse document frequency vector, then build tf-idf matrix. 9

5. Query Search & Problems 10

5.1. Cosine Similarity 10

5.2. Exception Handler: Unknown words. 11

5.3. Display Results 12

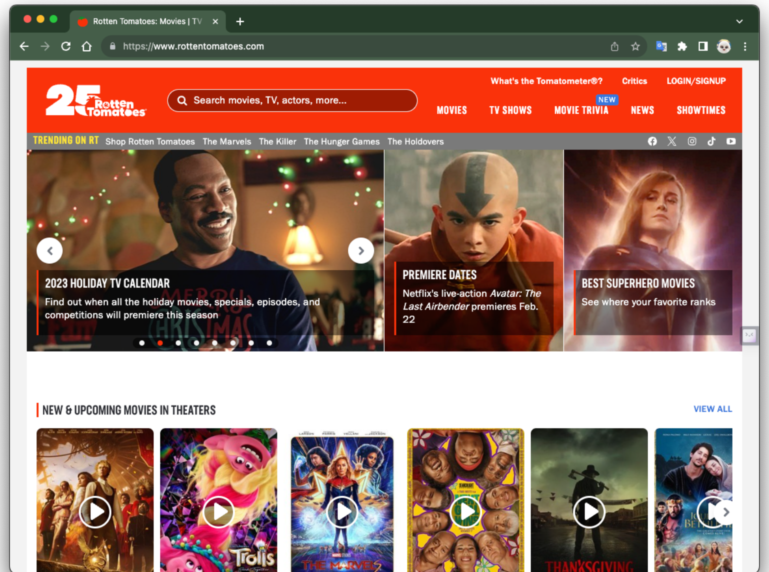
5.4. Problem Handler: Common Words Problem. 14

5.5. Search Results Examples 15

6. Code reviews 16

1. Introduction

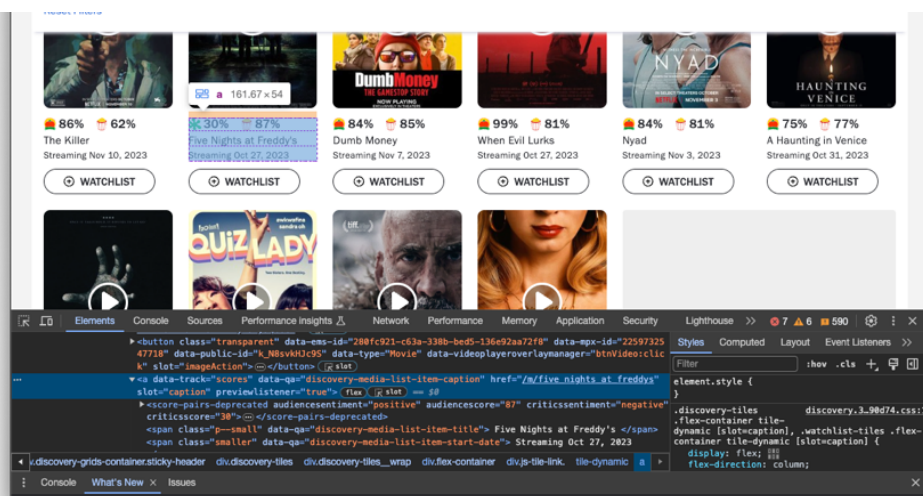
Rotten Tomatoes is a review-aggregation website for film and television in the U.S. It has its own ranking system of movies, with three tiers: Certified Fresh, Fresh, and Rotten. A screenshot of the *rottentomatoes.com* site:



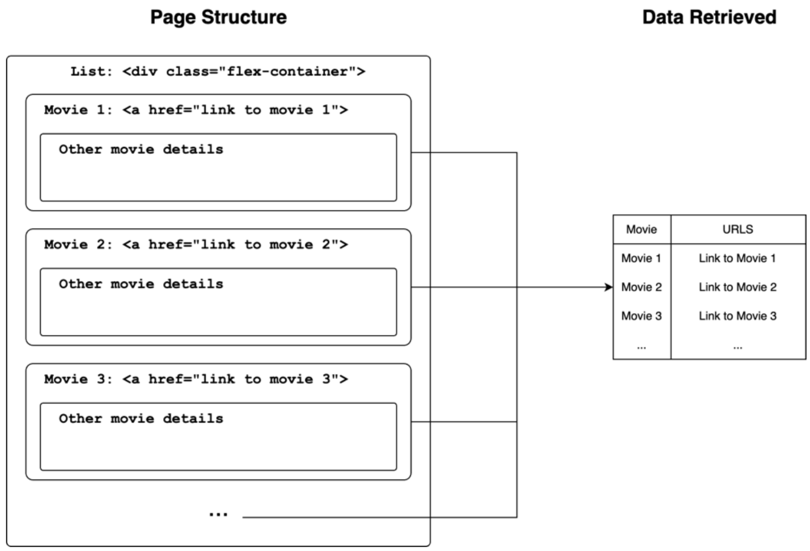
The goal of our project is to extract the main content of top 100+ list from RT, then make a query searcher based on the plot twists using TF-IDF model. This searcher allows users to search most relevant movies using simple queries.

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.



The abstract page structure and the idea of data retrieval is shown below:



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, first get the `<a>` tag, and then retrieve its attribute:

```
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`:

C29 /m/1112549-crossroads

	A	C	D	E	F	G
1	title	link	audience score	critics score	audience sentiment	critics sentiment
2	FiveNightsatFreddy's	/m/five_nights_at_freddys	88	26	1	0
3	PainHustlers	/m/pain_hustlers	70	23	1	0
4	NoHardFeelings	/m/no_hard_feelings_2023	87	71	1	1
5	TheExorcist:Believer	/m/the_exorcist_believer	59	22	0	0
6	TalktoMe	/m/talk_to_me_2023	82	94	1	1
7	SawX	/m/saw_x	89	79	1	1
8	TheBurial	/m/the_burial_2023	83	91	1	1
9	MilliVanilli	/m/milli_vanilli	85	100	1	1
10	ThePigeonTunnel	/m/the_pigeon_tunnel	75	96	1	1
11	WhenEvilLurks	/m/when_evil_lurks	57	99	0	1
12	TheSuperMarioBros.Movie	/m/the_super_mario_bros_movie	95	59	1	0
13	FairPlay	/m/fair_play_2023	52	86	0	1
14	TotallyKiller	/m/totally_killer	77	88	1	1
15	Reptile	/m/reptile_2023	71	44	1	0
16	TheNunII	/m/the_nun_ii	73	52	1	0
17	SuitableFlesh	/m/suitable_flesh	68	82	1	1
18	HauntedMansion	/m/haunted_mansion_2023	84	37	1	0
19	TheWonderfulStoryofHenrySugar	/m/the_wonderful_story_of_henry_s	82	95	1	1
20	NoOneWillSaveYou	/m/no_one_will_save_you	56	82	0	1
21	Mission:Impossible-DeadReckoning,PartOne	/m/mission_impossible_dead_recko	94	96	1	1
22	AHauntinginVenice	/m/a_haunting_in_venice	77	75	1	1
23	Huesera:TheBoneWoman	/m/huesera_the_bone_woman	67	97	1	1
24	Barbie	/m/barbie	83	88	1	1
25	PastLives	/m/past_lives	92	98	1	1
26	TheEqualizer3	/m/the_equalizer_3	94	75	1	1
27	LongShot	/m/long_shot_2019	74	82	1	1
28	FiftyShadesofGrey	/m/fifty_shades_of_grey	41	25	0	0
29	Crossroads	/m/1112549-crossroads	40	15	0	0
30	Attachment	/m/attachment	59	95	0	1
31	Influencer	/m/influencer	73	92	1	1
32	M3GAN	/m/m3gan	78	93	1	1
33	X	/m/x_2022	75	94	1	1
34	SoundofFreedom	/m/sound_of_freedom	99	58	1	0
35	TheBoogeyman	/m/the_boogeyman	66	60	1	1
36	GetOut	/m/get_out	86	98	1	1
37	TheNightmareBeforeChristmas	/m/nightmare_before_christmas	92	95	1	1
38	BlueBeetle	/m/blue_beetle	92	78	1	1
39	EvilDeadRise	/m/evil_dead_rise	76	84	1	1
40	Hereditary	/m/hereditary	70	90	1	1
41	Willy'sWonderland	/m/willys_wonderland	68	61	1	1
42	Spider-Man:AcrosstheSpider-Verse	/m/spider_man_across_the_spider_v	94	96	1	1

Sheet1

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 `get_movie_url()` settled in `__settings__.py` 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.

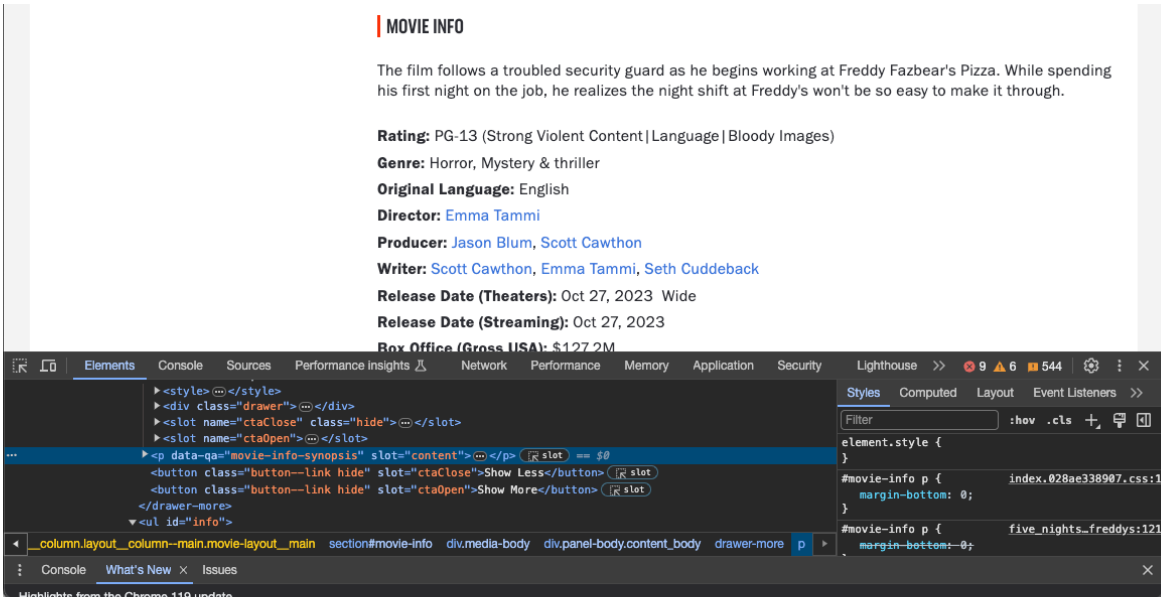
```
def get_movie_url():
    # Read Excel file and do analysis
    file_path = './movie_list/movie_data.xlsx'
    movie_data = pd.read_excel(file_path)
    # Get URL
    url_series = movie_data['link']
    sub_urls = url_series.values

    # 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.



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

title	genre	
Milli Vanilli	Documentary,Biography,Music	The bizarre untold truth behind the greatest con in music history -- Milli Vanilli.
No Hard Feelings	Romance,Comedy	On the brink of losing her childhood home, Maddie (Jennifer Lawrence) discovers an intriguin
Saw X	Horror,Mystery&thriller	John Kramer (Tobin Bell) is back. The most chilling installment of the SAW franchise yet explo
The Exorcist: Believer	Horror,Mystery&thriller	Since the death of his pregnant wife in a Haitian earthquake 12 years ago, Victor Fielding (Ton
The Burial	Drama	Inspired by true events, when a handshake deal goes sour, funeral home owner Jeremiah O'Ke
Five Nights at Freddy's	Horror,Mystery&thriller	The film follows a troubled security guard as he begins working at Freddy Fazbear's Pizza. Whil
Talk to Me	Horror,Mystery&thriller	When a group of friends discover how to conjure spirits using an embalmed hand, they becon
The Nun II	Horror,Mystery&thriller	1956 -- France. A priest is murdered. An evil is spreading. The sequel to the worldwide smash h
Totally Killer	Horror,Comedy	Thirty-five years after the shocking murder of three teens, the infamous "Sweet Sixteen Killer"
The Pigeon Tunnel	Documentary	Academy Award-winning documentarian Errol Morris pulls back the curtain on the storied lif
Fair Play	Drama	When a coveted promotion at a cutthroat financial firm arises, once supportive exchanges be
Reptile	Crime,Drama	Following the brutal murder of a young real estate agent, a hardened detective attempts to ur
The Super Mario Bros. Movie	Kids&family,Comedy,Adventure,	Mario and Luigi go on a whirlwind adventure through Mushroom Kingdom, uniting with a ca
The Wonderful Story of Henry S	Comedy,Adventure,Short	The Wonderful Story of Henry Sugar:A rich man learns about a guru who can see without usin
Haunted Mansion	Fantasy,Comedy	A woman and her son enlist a motley crew of so-called spiritual experts to help rid their home
A Haunting in Venice	Holiday,Mystery&thriller,Drama	"A Haunting in Venice" is set in eerie, post-World War II Venice on All Hallows' Eve and is a terr
Suitable Flesh	Horror,Mystery&thriller	Psychiatrist Elizabeth Derby becomes obsessed with helping a young patient suffering extrem
Huesera: The Bone Woman	Horror,Mystery&thriller	Valeria's joy at becoming a first-time mother is quickly taken away when she's cursed by a sinis
When Evil Lurks	Horror	When brothers Pedro (Ezequiel Rodríguez) and Jimmy (Demián Salomón) discover that a dem
No One Will Save You	Mystery&thriller,Horror,Sci-fi	"No One Will Save You" introduces Brynn Adams (Kaitlyn Dever), a creative and talented youn

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.

```python
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:

$$idf(term) = log\left(\frac{N}{df(term) + 1}\right)$$

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:

$$cosine(q, d) = \frac{q^T \cdot d}{|q| \cdot |d|}$$

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.

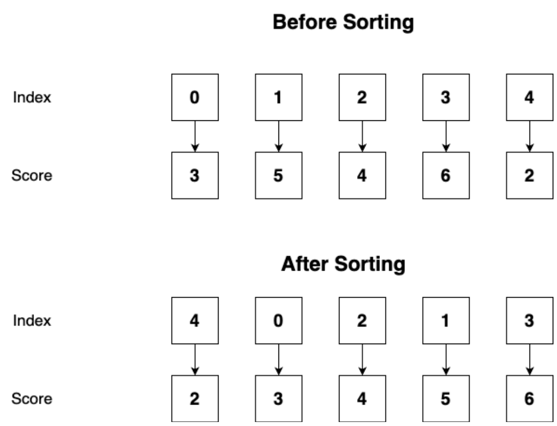
One drawback of the tf-idf model is that it won't recognize any query word that doesn't exist in the index of the tf-idf matrix. Giving a new term in the query will cause an exception that the length of the query tf-idf vector will become longer than the index of the matrix, resulting a shape-unmatch. 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("\033[31m$ Warning: Word not exist, try another one.\033[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("\033[32m" + index + "Searched for: \"" + query + "\"\n" +
              "Top " + str(top_x) + " relevant:" + "\033[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("\033[33m\n$ Warning: No more related movies!!\033[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("\033[32mSearch is complete!\033[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:

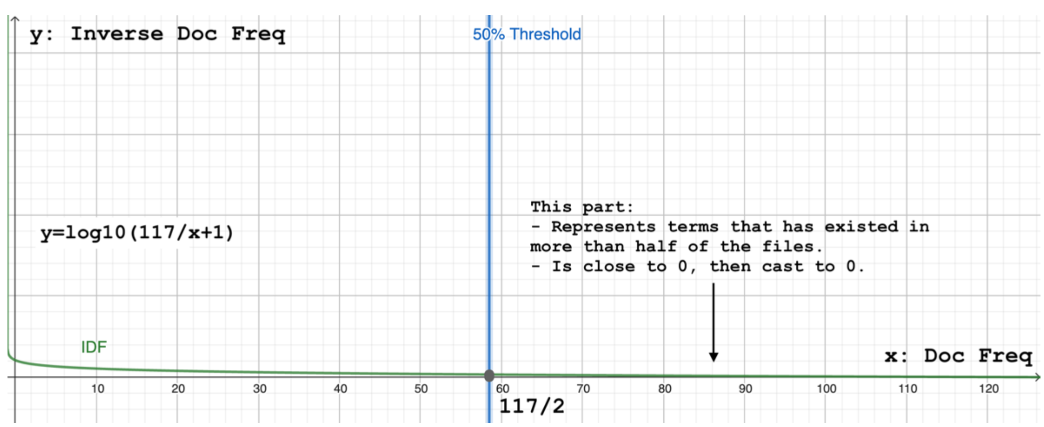
```
1.
ID: 69
Title: Halloween
Sim score: 0.15211712289647808
```

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. This minor problem reveals that, 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:

$$minidf(term) = log\left(\frac{N}{0.5N + 1}\right)$$

The idea is demonstrated by this diagram:



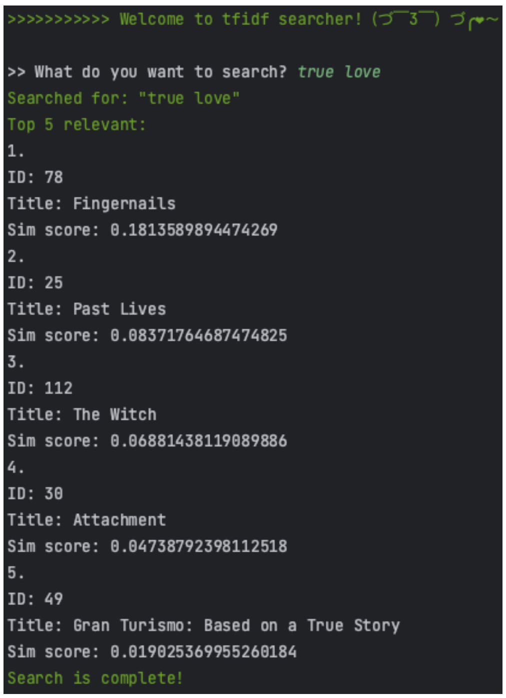
This can be performed when calculating inverse document frequency by:

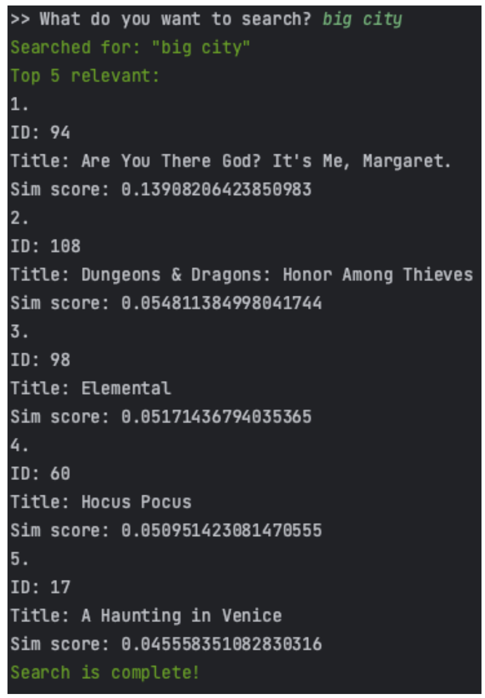
```
    if __settings__.custom_settings['RM_COMMON_WORDS']:
        min_idf = np.log(doc_num) / (1 + doc_num)
        idf[idf == min_idf] = 0
```

This allows us to prevent any word that is to some extent too common among plot twists from contributing to the search result. In our project, we've managed to do this in a least effective way (i.e. to cast terms that has existed in every document).

5.5. Search Results Examples

Regular search results (retrieve Top 5):





```
>> What do you want to search? theme park
Searched for: "theme park"
Top 5 relevant:
1.
ID: 87
Title: Nope
Sim score: 0.3735403181190305
2.
ID: 115
Title: Strays
Sim score: 0.03241281664819874

$ Warning: No more related movies!!
Search is complete!
```

```
>> What do you want to search? big car house
Searched for: "big car house"
Top 5 relevant:
1.
ID: 113
Title: Beetlejuice
Sim score: 0.08536689374232999
2.
ID: 43
Title: Willy's Wonderland
Sim score: 0.07982732272297352
3.
ID: 69
Title: Halloween
Sim score: 0.07723988680862427
4.
ID: 50
Title: Barbarian
Sim score: 0.06262129214414133
5.
ID: 94
Title: Are You There God? It's Me, Margaret.
Sim score: 0.049267287119874215
Search is complete!
```

An exception handling result:

```
>> What do you want to search? typoExample
$ Warning: Word not exist, try another one.
```

6. Code reviews

Please kindly refer to this link for source code:
https://github.com/YanzhenHuang/CISC3014-IR-and-WebSearch-Project.git