Plot Search using TF-IDF Model

1. Introduction to Rotten Tomatoes

Image 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. 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.

2. First Crawler `__get_movies__.py`

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.

Image

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

```
movie_list = response.xpath("//div[@class='flex-container']")
```

Iterate this path. Within each path, 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,
    }

    # Append movie data... not really useful, cuz i plug it into excel at
each loop anyways
    self.movie_data.append(data)

    # 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`: Image

3. Second Crawler `__get_movie_detail__.py`

Having the url list, we have the second crawler to crawl movie contents of each movie. One movie, with one url, which 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)
```

Image

The following is quite the same. 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`:

Image

Evidently, each movie corresponds to its own plot twists, in other words, articles. These articles will then be used to build a tf-idf search model.

```
    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.

```
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. For each array, remove duplicates and form a term-frequency vector.

The main part is the `Counter()` function, which counts duplicates and merge them together. 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, which is an array of all the word that's ever existed in the queries. This requires to perform 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, and
        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 of the matrix, we just insert data into the matrix. For each plot twist, i.e., each tf vector, for each word in 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.

```
>>>> 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.

Inverse document frequency can be retrieved by:

$$[\text{{IDF}}(t, D) = \log \left(\frac{{N}}{{\text{{df}}(t, D) + 1}} \right)]$$

which yields to be:

```
>>>> Inverse Document Frequency
[[2.38108697]
 [2.38108697]
 [2.38108697]
 ...
 [2.38108697]
 [2.38108697]
 [1.58739131]]
```

Timing the idf vector term-wise with each column of the tf matrix would yield the tf-idf matrix.

```
def create_tfidf_mat(term_freq_mat):
    # Inverse document frequency.
    # idf(term) = log(movie number) / 1 + (numer of movies containing this
term)
    def calc_idf(term_freq_mat):
        doc_num = term_freq_mat.shape[1]                    # Number of
movies
        freq = np.count_nonzero(term_freq_mat, axis=1)      # Doc
frequency

        # Inverse document frequency
        idf = np.log(doc_num) / (1 + freq)
        idf = np.reshape(idf, (len(idf), 1))

        # Filter words that are very common.
        # I can use nltk, but this is simpler.
        if __settings__.custom_settings['RM_COMMON_WORDS']:
            min_idf = np.log(doc_num) / (1 + doc_num)
            idf[idf == min_idf] = 0

        # Console Log
        if __settings__.custom_settings['CONSOLE_LOG_PROCESS']:
            print("\n>>>> Inverse Document Frequency")
            print(idf)
        return idf

    # Inverse Document Frequency
    idf_vector = calc_idf(term_freq_mat)
    # tf-idf matrix
    tfidf_mat = term_freq_mat * idf_vector
    # Console Log
    if __settings__.custom_settings['CONSOLE_LOG_PROCESS']:
        print("\n>>>> tf-idf Matrix")
        print(tfidf_mat)
    return tfidf_mat, idf_vector
```

```
>>>> 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:

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

The cosine similarity geometrically is the cosine value of the angle between two sample vectors in the N-dimensional vector space. Hence the distance (i.e., how long the query or the article is) is not considered. Given a query and a tf-idf matrix, a score indexed by movies is given by this 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
```

It is important to turn query into a tf-idf vector first.

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:

```
    # 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
```

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.

5.3. Display Results

The key of displaying results is to sort the score array while still preserving its index. The index is an integer pointing to the `titles` array that stores the movie titles. This function will return the indexes (not values) of the top x scored movies.

```
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, it can't be easier to find the movie titles:

```
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)   # Sort from
high to low, return index.
        top_x_names = get_top_x_names(similarity_scores, top_x, titles)
```

```
# Use index to retrieve title.

        # 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, the `search()` function prints an error message instead of raises an error:

```
    # 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
```

To make a consistent user interface, we uses a while-loop to constantly takes user input:

```
    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)  # 
Perform search
```

This loop will 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 search 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), when the term frequency in one plot twist is large enough, it is still able to contribute 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:

$$[\text{{IDF}}(t, D) = \log \left(\frac{{N}}{{\text{{0.5N}} + 1}} \right)]$$

This can be performed when calculating idf:

```
    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.

An example of a common search result:

```
>> What do you want to search? year old
Searched for: "year old"
Top 5 relevant:
1.
ID: 69
Title: Halloween
Sim score: 0.15211712289647808
2.
ID: 45
Title: Cuties
Sim score: 0.10380713227139456
3.
ID: 10
Title: Totally Killer
Sim score: 0.0679638359039251
4.
ID: 44
Title: Dark Harvest
Sim score: 0.05955330512572505
5.
ID: 60
Title: Hocus Pocus
Sim score: 0.053317720342773836
Search is complete!
```