MOVIE RECOMMENDATION ALGORITHM BASED ON USER'S TOP MOVIE LIST

MCIS 5313 Data Structures and Algorithms

Abstract

There are a plethora of movies available to watch at any given time. So how does one navigate through all the possibilities to find a movie that one would enjoy? A movie recommendation algorithm is an easy way to find a list of movies that will match a users preferences to movies. The algorithm asks the user for at least five of their top favorite movies and constructs a recommendation list based on the actors, genres, and plots from the user top movie list, along with incorporating the average rating from other users.

Jonathan King

Table of Contents

Introduction

In the last 100 years, American and world culture has been significantly influenced by the film industry. While the exact number of movies that have been produced in that time frame is not known, the organization responsible for the rating system used by the US, the Motion Picture Association, has rated over 30,000 movies in the last 50 years (News, 2018). With such a large catalog of movies and a list that grows by hundreds each year (Stoll, 2021), how is one to even begin to find movies that fit one's unique tastes? To navigate this vast sea of options, a good recommendation algorithm can be used to find specific titles that align to an individual's preferences. In this paper, a movie recommendation algorithm will be constructed that will use inputs from a user to build a list of ten movies that should align to the user's preference in actor casting, genre type, over all movie plot, with also factoring a movie's overall rating from other individuals.

Movie Recommender Strategies

There are different types of approaches that can be used to build a recommendation algorithm. Some of these types are:

Simple Recommender (Sharma, 2020). This recommender gives non-individualized recommendations based on high popularity rankings or being critically acclaimed. An example of this would be a top movie list from Netflix or IMDB.

Content-Based (Le, 2018). In this approach, an individual's preferences are factors in determining if a movie would be recommended. Similarities in actors, genres, plots, or other movie details are found between what is known about the individual and potential movie titles.

Collaborative Filtering (Le, 2018). Here, past behavior in a user's movie likes and dislikes along with other people's movie likes and dislikes are used instead of content. "If user A likes movies 1, 2, 3 and user B likes movies 2, 3, 4, then they have similar interests and A should like movie 4 and B should like movie 1." (Le, 2018).

Project Recommendation Algorithm

A popular approach to building a movie recommendation is to base it around collaborative filtering. One downside to this is when new content (movies) is added to the catalog. The problem is, there are not enough interactions with the new movie to allow for recommendations. This is called a cold start problem (Deldjoo, 2019). The common approach to alleviate this issues is to switch over to a Content-Based approach. The strategy used to construct the recommendation algorithm for this project is an original creation, combining different approaches to recommend movies. This will be heavily reliant on a content-based approach, but still has influence by aggregate ratings from outside reviewers. This hybrid approach will fix the issue with any potential cold start problems with new movies that are released in the future instead, focusing on what the user likes in a movie along with factoring in other users opinions of movies. The Recommended Movie List (RML) generated from the algorithm for the user is scored based on movie attributes from the movies the user has chosen. The highest score being the movie that is most similar to the User's Top Movie List (UTML). The overall movie score is an equation that includes an Actor, Genre, Plot, and Rating score.

$$M = A * G * P * R$$

$$M = Overall\ Movie\ score$$

$$A = Actor\ Score$$

$$G = Genre\ Score$$

$$P = Plot\ Score$$

$$R = Rating\ score\ from\ other\ individuals$$

Two databases are used in the construction of the algorithm. The first is the Open Movie database (www.omdbapi.com) (Fritz, 2021). This database is used in retrieving movie details from the UTML. This database API is used because it easily lists the movie details like the headline actors along with the Rotten Tomatoes score for the movie (www.rottentomatoes.com) (About Rotten Tomatoes, 2021). The second movie database used in the algorithm is The Movie Database API (www.themoviedb.org) (The Movie Database API, 2021). This database API is used to collect information on the movies from the RML. This API was selected because it allows for easy searching of movies based on an actor's name as the search criteria.

The algorithm is broken down in to nine parts.

1) **User data collection**: Here, the user is prompted to enter at least five of their top overall movies. Details from these movies will be used to generate the RML that is returned to the user.
2) **Confirmation of exact movie title**: Because of sequels, remakes, and sometimes ambiguity of movie titles, it is necessary to get the exact movie title that the user originally entered. The algorithm will take the movie titles that the user entered and search out possible movie titles from the OMDB database (Fritz, 2021). The user is given a selection of titles and year of release. From the list the user selects the movie title that matches their original intended movie. This is done for each movie the user entered and will generate the final UTML.
3) **Collection of movie detail**: From UTML, the movie attributes of headlining actors, genres, and plot are collected and used to generate the RML. As an aside, the Rotten Tomatoes score (About Rotten Tomatoes, 2021) for each movie is displayed along with the average for all the movies entered to tell if the user has good taste in movies.
4) **Generating recommended movie list:** To keep from having to look at every movie title in the database, the list of possible movies is limited to those only starring actors that are headline actors in the UTML. This limits the overall possible recommendation list to only a few thousand at most, depending on how many movies the actors from the UTML starred in. The Movie Database (The Movie Database API, 2021) API is used to generate the RML to which a score will be given to each movie.
5) **Calculating the Actor score:** The Actor score is calculated by factoring in how many of the actors from UTML are listed in the movie credits for each movie in the RML. Also, if an actor stars in more than one of the movies in the UTML, that actor will have more influence in the actor score for that movie than an actor that is only listed once. An example of this would be if the UTML included the movies "Star Wars" and "Raiders of the Lost Ark," the actor Harrison Ford would appear twice. This would show that the user prefers movies starring Harrison Ford and thus

movies with Harrison Ford would have a higher Actor score than other movies in the RML. The formula for calculating the Actors score is:

$$A = \sum_{i=a} \frac{Na}{T}$$

$$A = Actor\ Score$$

$$N = Number\ of\ times\ the\ actor\ appears\ in\ the\ UTML$$

$$T = Total\ number\ of\ actors\ in\ the\ UTML$$

$$a = actor\ in\ the\ UTML\ appering\ in\ the\ movie\ in\ the\ RML$$

6) **Calculating the Genres score:** The issues with comparing genres between the UTML and the RML is that genres are just a collection of words. So how are a set of words from two different sources compared in their raw form? To do this comparison, the word vectors for each set of words must be computed. Word vectors are vectorized representation of words in a set. To calculate these word vectors, the a process known as Term Frequency-Inverse Document Frequency (TF-IDF) is used. "TF-IDF is a statistical measure that evaluates how relevant a word is to a document in a collection of documents. This is done by multiplying two metrics: how many times a word appears in a document, and the inverse document frequency of the word across a set of documents" (Stecanella, 2019). The Term Frequency is a raw count of the number of instances a word appears in a document (Stecanella, 2019). The Inverse Document Frequency is how common or rare a word is in the document set. The closer to 0 the more common a word is. "This metric can be calculated by taking the total number of documents, dividing it by the number of documents that contain a word, and calculating the logarithm" (Stecanella, 2019). The vectors are calculated between the UTML and each movie's genres in the RML. This is done by creating a single string of genre types from all the movies in the UTML. Next, it is compared to a single string of the genres for each movie in the RML. Below is the formula to calculate the TF-IDF score: (Le, 2018):

$$\mathbf{tfidf}_{i,j} = \mathbf{tf}_{i,j} \times \log\left(\frac{\mathbf{N}}{\mathbf{df}_i}\right)$$

$\mathbf{tf}_{i,j}$ = total number of occurences of i in j
$\mathbf{df}_i$ = total number of documents (speeches) containing i
N = total number of documents (speeches)

Once the TF-IDF vectors are calculated, the cosine similarity is used to calculate a value that denotes the similarity between the UTML genres and the genres on the movie in the RML. The cosine similarity comes from measuring the angle between the two vectors of the TF-IDF. The lower the angle between the two, the higher the cosine value will be (Nixon, 2020). The higher the similarity values, the more likely the movie from the RML is to the movies in the UTML. The cosine value is the Genre score. The formula for the similarity is below: (Nixon, 2020):

$$\text{similarity} = \cos(\theta) = \frac{\mathbf{A} \cdot \mathbf{B}}{\|\mathbf{A}\|\|\mathbf{B}\|} = \frac{\sum_{i=1}^{n} A_i B_i}{\sqrt{\sum_{i=1}^{n} A_i^2}\sqrt{\sum_{i=1}^{n} B_i^2}},$$

7) **Calculating the Plot score:** Comparing the plots from the UTML and RML has the same issues as in comparing genres, comparing one set of words to another set. Therefore, the same approach of utilizing the TF-IDF and cosine similarity to score the plots of each movie in the RML is used. Again, the higher the cosine values the more the plot of the movie in the RML is similar to the plots of the movies in the UTML. The cosine value is the Plot score.
8) **Calculating the Rating score:** Even though a movie from the RML may share many of the same movie attributes from the UTML, it does not guarantee that movie should be included in the RML. Therefore, introducing a movie rating from an outside source into the overall movie score helps to filter out bad movies. The Rating score uses the voter rating from the TMDB database (The Movie Database API, 2021). Their rating system is based on a 0 – 10 rating (higher the rating the better the movie) voted by their members. The vote rating for each movie in the RML is divided by 10 to get the Rating score.
9) **Total score and sort movies in the RML**: The scores from the different areas are compiled into the Overall Movie score for each movie in the RML. The movies are then sorted from highest to lowest, with highest being a movie that is most like that of those in the UTML. Because the movies in the UTML will also be in the RML, these movie titles need to be removed from the RML. The top ten of the remaining movies in the RML is then presented to the user as the recommended movies based on the movie title that were supplied to the algorithm by the user.

Algorithm Code

The code used to construct the algorithm contains several packages that need to be installed. This helps in ease and simplicity of coding the algorithm. There are several packages installed in Python to help in simplifying the code. Here is the list of those packages:

Pandas (pandas v 1.2.3, 2021): Formats the printout of the movie list into easy to read tables.

NumPy (Oliphant, 2020): Creates an array to store the Rotten Tomatoes score from the UTML and easily calculate the average Rotten Tomatoes score for the user.

Requests (Reitz, 2019): Accesses and retrieves movie information from The Movie Database API.

Scikit-learn (Scikit Learn, 2021): Simplifies the coding for the TF-IDF and Cosine Similarity calculation need for the Genre and Plot score.

OMDB (Gilland, 2018): Simplifies the accessing and retrieving of movie information from the OMDB API.

The data structure used to retain the scores for the different attributes of the movie are dictionaries. These are needed to keep the movie score for that attribute connected to the movie ID

using the Key -Value pair, where the Key is the movie and the Value is the movie's score for that attribute.

A list is used to store the user's movie list. As the user enters the movie titles at the prompt, they are appended to the list.

```
1. import omdb as omdb
2. import numpy as np
3. import pandas as pd
4. import requests
5. from sklearn.feature_extraction.text import TfidfVectorizer  # to do
   TI-IDF calculations
6. from sklearn.metrics.pairwise import cosine_similarity  # to do Cosine
   comparison
7.
8. omdb.set_default('apikey', 'XXXX')
9. Users_list = []  # Movie list from user
10.  final_list = []  # Exact movie list to get ratings
11.  score_array = np.array([])  # Array for Rotten tomato score
12.  movie_score_list = []  # List of movie's data
13.  fav_actors = {}  # Favorite actor list
14.  TMDB_Actor_score = {}  # score of the movie by actor
15.  TMDB_Genre_score = {}  # score of the movie by Genre
16.  TMDB_plot_score = {}  # score of the movie by plot
17.  TMDB_vote_score = {}  # score of the movie by user voting
18.  name_id_xref = {}  # Name to TMDB's actor id cross reference
   dictionary
19.  Recommendation_score = {}  # total recommendation score for the movies
20.  Top_rec_list = []  # Top movie recommendations
21.  user_IMDB_id = []  # IMDB ids from user complied list
22.  TMDB_total_movie_score = {}  # Total movie score
23.  actor_list = []  # actor from users movie list to find recommended
   movies
24.  plot_words = ''  # words for plots of favorite movies used to rank
   recommended movies used for score calculations
25.  genre_list = ''  # genre from user movie list used to rank recommended
   movies used for score calculations
26.  TMDB_genre_list = ''  # genre from TMDB movie
27.  TMDB_movie_actor_count = {}  # list of movies and number of match
   favorite actors to score for recommendation
```

Because the TMDB database classifies their genres by an ID in the movie detail when listing the genres with the movie data in the API, a cross reference dictionary was created to match up the genre types between the OMDB and TMDB databases.

```
30.  genre_id_xref = {28: 'Action', 12: 'Adventure', 16: 'Animation', 35:
   'Comedy', 80: 'Crime', 99: 'Documentary',18: 'Drama',10751: 'Family',
   14: 'Fantasy', 36: 'History', 27: 'Horror', 10402: 'Music', 9648:
   'Mystery',10749: 'Romance',878: 'Sci-Fi', 10770: 'TV Movie', 53:
   'Thriller', 10752: 'War', 37: 'Western'}
```

The "addmovie" function adds movies from the initial prompt by appending them to the "Users_list". This is then used to search the OMDB database to find the exact movie title. It also prevents the user

from not entering a movie title or only entering a *space* for a movie title. If an 'x' is entered after five movie titles have been entered, it will exit the prompt. The "moviescore" function is used to calculate the score using the TI-IDF and cosine similarity. Here the Scikit-learn package simplifies the code for doing all the calculations (Scikit Learn, 2021).

```
39.  # Function to add user movies to the top Users_list
40.  # and make sures the title is not an empty space
41.  def addmovie(movie): # O(1)
42.      if movie == '' or movie == ' ':
43.          print('Not a good movie title')
44.          return
45.      if movie == 'x':
46.          return
47.      Users_list.append(movie)
48.      return
49.
50.  # function to calculate the Td-Ifd matrix and cosine similarity
51.  def Movie_score(user, recomen):  # O(1)
52.      vectorizer = TfidfVectorizer(analyzer='word', min_df = 0,
   stop_words = 'english')
53.      vectors = vectorizer.fit_transform([user, recomen])
54.      feature_names = vectorizer.get_feature_names()
55.      dense = vectors.todense()
56.      denselist = dense.tolist()
57.      Tdifd_matrix = pd.DataFrame(denselist, columns=feature_names)
58.      cosine = cosine_similarity(Tdifd_matrix)
59.      cosine_matrix = pd.DataFrame(cosine)
60.      score = cosine_matrix[0][1]
61.      return score
```

This section loops until the user enters at least five movie titles. Then the user is prompted to either add more titles or exit.

```
64.  print('''This program will judge your taste in movies
65.  by judging them to the Rotten Tomatoes score\n
66.  Enter at least your top 5 movies.\n''')
67.
68.  # loop to enter the first 5 movies
69.  while len(Users_list) < 5: # O(n)
70.      top_5 = input('Enter Movie Title ' + str(len(Users_list) + 1) + ':
   ')
71.      addmovie(top_5)
72.
73.  print('\nYou can enter more movies to the list if you like or type "x"
   to exit\n')
74.
75.  # Adds more titles if the user wishes
76.  more_movie = ''
77.  while more_movie != 'x': # O(n)
78.      more_movie = input('Enter another movie: ')
79.      addmovie(more_movie)
80.  print('\n')
```

The code uses the OMDB API to retrieve possible matches for each movie the user entered. If there are multiple movies with the same title, the user will be given choices to select the exact movie. The selected movie title is then added to the UTML by appending it to the “final_list”.

If the user entered a movie title that could not be found, they will be prompted to enter a new movie title. The program will continue to loop until all movies that the user entered have found a matching title in the OMDB database.

```
82.  # ----------------------Find exact matches of the movies in the OMDB
   database-----------------#
83.  # Loop through all the users movie list
84.  while len(Users_list) > 0:  # O(n^3)
85.
86.      for movie in Users_list:  # Pulls the movie list from the user and
   pulls possible matches from OMDB
87.          choice_list = []  # empty list to put exact movies choices
   into
88.          res = omdb.request(s=movie, type='movie')
89.          title = res.json()
90.
91.          if title['Response'] == 'True':  # A match on the data base
   was found
92.              movie_choices = title['Search']
93.              for choice in movie_choices:  # Formats the possible moves
   that came back from OMDB
94.                  title = choice['Title']  # official movie name
95.                  year = choice['Year']  # release date
96.                  m_num = choice['imdbID']  # IMDB's ID number
97.                  choice_list.append([title, year, m_num])
98.
99.              # shows the different options of movies to pick from
100.             print('Here are the exact Movie titles from your list with
   release year ')
101.             for option in range(len(choice_list)):
102.                 num = choice_list[option]
103.                 print(option, num[0:2])
104.
105.             # User selects the exact movie and test to make sure that
   it is a valid answer
106.             Title_select = input('Select the number of the exact Movie
   you were thinking about \n')
107.             z = True
108.             while z:
109.                 try:
110.                     Title_select = int(Title_select)
111.                     selected_movie = choice_list[Title_select]
112.                     final_list.append(selected_movie)  # adds the
   movie to the final movie list
113.                     Users_list.remove(movie)  # removes the user movie
   from the user list so that it is not duplicated
114.                     z = False
115.                 except:
116.                     print('Not a valid choice')
117.                     Title_select = input('Select the number of the
   exact Movie you were thinking about \n')
```

```
118.
119.         else:  # movie title was not found on the database must pick a
   new movie title
120.             print('Movie ' + movie + ' is not in the database please
   enter another movie title')
121.             new_movie = input('New movie title: ')
122.             Users_list.remove(movie)  # removes the bad movie title
   and replaces it with a new one
123.             Users_list.append(new_movie)
```

Movie details from the UTML are retrieved from the OMDB API. All genres from the movies are combined into one string, representing the genres the user prefers. A string is need because this is the structure that is used by the Scikit-learn package for in the TF-IDF, Cosine Similarity test.

The plots from the movies are also combined into one string. This will represent the types of plots the user prefers. As with the genres, the plots must be in a string format so that it can fed in to the TF-IDF, Cosine Similarity test. The actors are added to the "fav_actors" dictionary and the total number of times the actor appears is counted and stored in the dictionary. A dictionary is used because of the ease of pairing the actor with the number of times they appear. The actor's name is the Key and the number of times they appear as the Value.

```
127. # goes though final user list and pulls title, genre, actors, plot
128. for final in final_list: O(n^2)
129.     ids = final[2]
130.     res = omdb.request(i=ids)
131.     Movie_info = res.json()
132.     title = Movie_info['Title']  # collect Official Movie title name
133.
134.     genres = Movie_info['Genre']  # collect Genre
135.     genre_list = genre_list + ',' + genres  # Creates genre string for
   Td-Ifd matrix and cosine similarity
136.
137.     actors = Movie_info['Actors'].split(',')  # collect actors
138.     # collect actors in to actors dictionary and keep count how many
   times they appear
139.     for name in actors:
140.         actor = name.lstrip()
141.         if actor not in fav_actors:
142.             fav_actors[actor] = 1
143.         else:
144.             fav_actors[actor] += 1
145.
146.     plot = Movie_info['Plot']  # Get plots text and splits it
147.     plot_words = plot_words + ',' + plot  # Creates plots string for
   Td-Ifd matrix and cosine similarity
```

This part of the code looks to see if there is a "Rotten Tomatoes" (About Rotten Tomatoes, 2021) score. If a "Rotten Tomatoes" is not available, it will use the Internet Movie Database score (www.imdb.com) (Ratings FAQ, 2021) as the rating score. Each movie's score is appended to the "movie_score_list" array to track the score.

```
149. ratings = Movie_info['Ratings']  # collect ratings
150.     RTper = 0 # initialized RTper to 0
151.     for rating in ratings:
152.         # Finds the Rotten Tomatoes score and turns it into a %
153.         if rating['Source'] == 'Rotten Tomatoes':
154.             score = rating['Value'].replace('%', '')
155.             RTper = int(score) / 100
156.         else:
157.             # Finds the IMDB score and turns it into a % if there is
    no Rotten Tomatoes score
158.             if rating['Source'] == 'Internet Movie Database':
159.                 IMDB_rating = rating['Value']
160.                 IMDBper = float(IMDB_rating[:-3]) / 10
161.     if RTper>0: # test to see if there is a Rotten tomatoes score
162.         movie_score_list.append([title, RTper])
163.     else:
164.         movie_score_list.append([title, IMDBper])
```

The program reports back to the user what the ratings are for the movies in the UTML and gives the average score. Based on the score, it will let them know if they have good taste in movies or not. The range for the average score and judgment is:

- 100% – 90%: The user has great taste in movies.
- 80% - 89%: The user has good taste in movies.
- 70% - 79%: The user has average taste in movies.
- < 70%: The user has poor taste in movies.

The program then lets the user know that it will make recommendations of other movie titles based on the UTML.

```
166. # Displays rating of movies from user final list
167. ml = pd.DataFrame(movie_score_list)
168. ml.columns = ["Movie", "Rating"]
169. ml['Rating'] = ml['Rating'].map('{:,.0%}'.format)
170. print(ml.iloc[:, [0, 1]])
171.
172. print('\n')
173. for score in movie_score_list: # O(n)
174.     RT_percent = score[1]
175.     score_array = np.append(score_array, RT_percent)
176.
177. RT_mean = score_array.mean()
178. print('Your Rotten Tomatoes Score is {:.0%}'.format(RT_mean))
179.
180. if RT_mean >= .9: # O(1)
181.     print("You have great taste in movies")
182. elif .9 > RT_mean >= .8:
183.     print("You have good taste in movies")
184. elif .8 > RT_mean >= .7:
185.     print("You have average taste in movies")
186. else:
187.     print("You have poor taste in movies")
188.
```

```
189. print('\nBased on the movie selections you made here are some
   recommended movie titles.\n')
```

The program now takes the actors in the "fav_actors" dictionary and uses the TMDB API (The Movie Database API, 2021) to retrieve the actor's unique TMDB ID and adds it to the "name_id_xref" dictionary. This is needed because the TMDB API requires the actor ID to search all movies that an actor has appeared in by their unique TMDB ID.

```
192. Actor_url =
   'https://api.themoviedb.org/3/search/person?api_key=XXXX=en-US'
193. # Creates actor TMDB cross reference dictionary
194. for i in fav_actors.keys(): # O(n)
195.     name = {"query": i}
196.     Actor_res = requests.get(Actor_url, params=name)
197.     actor = Actor_res.json()
198.     try:  # if a name on OMDB is not in TMDB it will skip the name
199.         x = actor["results"][0]["id"]
200.         if i not in name_id_xref:
201.             name_id_xref[i] = x
202.     except:
203.         continue
```

The code next loops through all the actor IDs and retrieves all the movies that the actor was in, regardless of, if it was a headlining role or supporting actor. For each movie, the unique movie ID, genre ID, plot overview and voter rating is retrieved.

```
205. # ---------loops the actor name ids to get the movies they are in and
   adds them to the scoring dictionaries------------
206.
207. # loop through actor id dictionary to get all movies the actors were
   in
208. for id in name_id_xref: # O(n^3)
209.     actor_multiplier = fav_actors[id]  # get the number of times that
   the actor appeared in the user movie list
210.     Actor_id = name_id_xref[id]
211.     Actor_id = str(Actor_id)
212.     Act_id_url = 'https://api.themoviedb.org/3/person/' + Actor_id + \
213.                  '/movie_credits?api_key=XXXXX&language=en-US'
214.     Act_id_res = requests.get(Act_id_url)
215.     movie = Act_id_res.json()
216.
217.     # loop through all the movies the favorite actors have been in and
   pull movie details
218.     for titleNo in range(len(movie['cast'])):
219.
220.         recommend_movie_id = movie['cast'][titleNo]['id']  # movie
   title id
221.         recommend_movie_genre = movie['cast'][titleNo]['genre_ids']  #
   movie Genre types
222.         recommend_movie_vote = movie['cast'][titleNo]['vote_average']
   # Movie user votes
223.         recommend_movie_plot = movie['cast'][titleNo]['overview']  #
   Movie plot
```

As the algorithm loops through the different actors, each movie that they appeared in is added to the "TMDB_movie_actor_count" dictionary. If the actor appeared in more than one of the movies in the UTML, that number is accounted for in the "actor_ multiplier." This will give these movies a higher Actor score because they have a more favorable actor in them.

```
225.         # adds movie id the TMDB_movie_actor_count dictionary and
   keeps count of how many of the favorite actors are
226.         # in the movie
227.         if recommend_movie_id not in TMDB_movie_actor_count:
228.             TMDB_movie_actor_count[recommend_movie_id] = 1 *
   actor_multiplier
229.         else:
230.             TMDB_movie_actor_count[recommend_movie_id] =
   TMDB_movie_actor_count[recommend_movie_id] + (
231.                     1 * actor_multiplier)
```

The genre and plot score are calculated using the TF-IDF and Cosine Similarity. The function "Movie_score" is called to calculate the Plot and Genre score and adds them to their respective dictionaries.

```
233.         # adds the Genre score to the TMDB_Genre_score dictionary
234.         if recommend_movie_id not in TMDB_Genre_score:
235.             for genre_id in recommend_movie_genre:  # Iterates through
   the different Genre ids
236.                 genre = genre_id_xref[genre_id]
237.                 TMDB_genre_list = TMDB_genre_list + ',' + genre
238.             g_score = Movie_score(genre_list, TMDB_genre_list)  #
   Genre cosine sore from TF-IDF for the movie
239.             TMDB_Genre_score[recommend_movie_id] = g_score
240.             TMDB_genre_list = ''  # reset the genre list for each
   movie
241.
242.         # adds the Plot score to the TMDB_plot_score dictionary
243.         if recommend_movie_id not in TMDB_plot_score:
244.             plot_score = Movie_score(plot_words, recommend_movie_plot)
   # Plot cosine sore from TF-IDF for the movie
245.             TMDB_plot_score[recommend_movie_id] = plot_score
```

The Rating score is computed using the TMDB rating which is based on a 0-10 scale. The TMDB rating for each movie was divided by 10 to get a number from 0-1 to match the format of the other scoring factors.

```
247.         # adds the Vote score to the TMDB_Vote_score dictionary
248.         if recommend_movie_id not in TMDB_vote_score:  #
249.             Vote_score = float(recommend_movie_vote / 10)  # Converts
   the TMDB voter score to a decimal
250.             TMDB_vote_score[recommend_movie_id] = Vote_score
```

The Actor score is derived by taking the actor count for each movie in the RML and dividing it by the total number of actors from the movies in the UTML.

```
actor_count = 0  # initialize total number of actors from user top list
# get total number actors from user list movie
for count in fav_actors: # O(n)
    actor_count = actor_count + fav_actors[count]

# get the total favorite actors count from recommended movie and divides it by the total number of favorite actors
for actor_name in TMDB_movie_actor_count: # O(n)
    count = TMDB_movie_actor_count[actor_name]
    score = count / actor_count
    if actor_name not in TMDB_Actor_score:
        TMDB_Actor_score[actor_name] = score
```

Here the final Overall score for each movie in the RML is calculated and sorted from highest score to lowest in the "TMDB_total_movie_score" dictionary using the sorted function.

```
# Calculates the total movie score from all the different components
for movie_id in TMDB_Actor_score:  # O(n)
    plot_movie_score = TMDB_plot_score[movie_id]
    genre_movie_score = TMDB_Genre_score[movie_id]
    actor_movie_score = TMDB_Actor_score[movie_id]
    vote_movie_score = TMDB_vote_score[movie_id]
    total_score = plot_movie_score * genre_movie_score * actor_movie_score * vote_movie_score
    TMDB_total_movie_score[movie_id] = total_score

# Sorts the TMDB_total_movie_score dictionary from highest value to lowest
# O(n log n)
Recommendation_score_sort = dict(sorted(TMDB_total_movie_score.items(), key=lambda item: item[1], reverse=True))
```

The code then finds the top ten recommended movies from the RML to display to the user. For each movie, the TMDB API is used to retrieve the IMDB unique movie ID to compare it to the movies from the UTML. If the movie from the RML matches a movie from the UTML the movie title is ignored and proceeds to the next movie. There is also a check to make sure that the movie from the RML had at least $1,000 in revenue to prevent "Making of…" titles and non-theatrical release movies from showing up on the list. This is repeated until a total of ten movies have been selected.

```
# Gets list of IMDB movie ids from user entered movie list
for i in final_list:  # O(n)
    user_IMDB_id.append(i[2])

count = 0  # initialized counter to limit the number of recommendations

# loop through the recommended movies and find the top 10
for TMDB_Movie_id in Recommendation_score_sort: # O(n)
    TMDB_Movie_id = str(TMDB_Movie_id)  # Gets TMDB id that goes with title
```

```
287.     # Retrieves the IMDB Number for the movie
288.     Movie_id_url = 'https://api.themoviedb.org/3/movie/' +
   TMDB_Movie_id + \
289.                    '?api_key=XXXX0&language=en-US'
290.     movie_id_res = requests.get(Movie_id_url)
291.     movie_details = movie_id_res.json()
292.     IMDBid = movie_details['imdb_id']
293.     Recommened_Title = movie_details['original_title']
294.     movie_year = movie_details['release_date']
295.     box_office = movie_details['revenue']
296.
297.     if box_office > 1000:  # movie has to have been released in the
   theater
298.
299.         # checks to make sure the movie the user entered does not come
   back as a recommended movie
300.         if IMDBid not in user_IMDB_id:
301.             Top_rec_list.append([Recommened_Title, movie_year[:4]])
302.             if count < 9:
303.                 count += 1
304.             else:
305.                 break
306.
307. # Formats the top list
308. Top_list_format = pd.DataFrame(Top_rec_list)
309. Top_list_format.columns = ["Movie", "Year"]
310. print(Top_list_format)
311.
```

Example of Input and Output

This is an example of inputs that a user would enter:

```
Enter Movie Title 1: Blazing saddles
Enter Movie Title 2: Rio Bravo
Enter Movie Title 3: The empire strikes back
Enter Movie Title 4: Iron Man
Enter Movie Title 5: Ronin
```

Here are the results of the Rotten Tomatoes scores and judgement of the users taste in movies:

```
                                          Movie Rating
0                                Blazing Saddles    88%
1  Star Wars: Episode V - The Empire Strikes Back    94%
2                                          Ronin    68%
3                                      Rio Bravo   100%
4                                       Iron Man    94%


Your Rotten Tomatoes Score is 89%
You have good taste in movies

Based on the movie selections you made here are some recommended movie
titles.
```

This is the output of the top ten recommended movie list based on the movies that the user entered:

```
                              Movie  Year
0                          Star Wars  1977
1                 Return of the Jedi  1983
2  Star Wars: The Rise of Skywalker  2019
3            Avengers: Age of Ultron  2015
4                          Iron Man 3  2013
5                   Avengers: Endgame  2019
6       Star Wars: The Force Awakens  2015
7           Star Wars: The Last Jedi  2017
8             Spider-Man: Homecoming  2017
9                          Iron Man 2  2010
```

Conclusion

The algorithm worked as designed, producing the top ten recommended movies based on the UTML. Depending on the types of movies the user enters, there is a noticeable bias on the movies that are listed in the top ten recommended movies. If there are any movies on the UTML that are part of a series, trilogy or cinematic universe, the top ten movie list is heavily populated by the other movies from that series. This is consistent with matching the movies from the UTML, however, if the user liked one of the movies from a particular series or cinematic universe, they probably have already seen recommended movies.

Part of the desire to create a recommendation list is to expose the user to options that they may not have thought of or known about. With the algorithm's bias to recommend the same movies in a series or cinematic universe, it greatly reduces the potential for the user to be exposed to unfamiliar movie titles. A possible feature that could be added, would be to prompt the user to either include or exclude movies from the same series as movies from the UTML, thus giving more variety to the top recommended movies for the user to see.

Overall, the combination of user input and factoring outside voter ratings produces a well-rounded recommendation list that caters to the user preference and will possibly expose them to movies that they may not have thought of or would have given a second thought to, ultimately resulting to a future of many happy hours of movie watching.

References

About Rotten Tomatoes. (2021). Retrieved from Rotten Tomatoes: https://www.rottentomatoes.com/about#whatisthetomatometer

Deldjoo, Y. D. (2019). Movie genome: alleviating new item cold start in movie recommendation. *User Modeling and User-Adapted Interaction*(29), 291–343. doi:https://doi.org/10.1007/s11257-019-09221-y

Fritz, B. (2021). Retrieved from OMDb API: http://www.omdbapi.com/

Gilland, D. (2018, December 18). *Python wrapper for OMDb API*. Retrieved from Pypi: https://pypi.org/project/omdb/

Kavita Ganesan, P. (n.d.). *Tutorial: Extracting Keywords with TF-IDF and Python's Scikit-Learn*. Retrieved from Kavita Ganesan: https://kavita-ganesan.com/extracting-keywords-from-text-tfidf/#.YFj42a9KiMq

Le, J. (2018, May 1). *The 4 Recommendation Engines That Can Predict Your Movie Tastes*. Retrieved from Towards Data Science: https://towardsdatascience.com/the-4-recommendation-engines-that-can-predict-your-movie-tastes-109dc4e10c52

Maklin, C. (2019, May 5). *TF IDF | TFIDF Python Example*. Retrieved from Towards data science: https://towardsdatascience.com/natural-language-processing-feature-engineering-using-tf-idf-e8b9d00e7e76

Needham, M. (2016, July 27). *scikit-learn: TF/IDF and cosine similarity for computer science papers*. Retrieved from Mark Needham: https://markhneedham.com/blog/2016/07/27/scitkit-learn-tfidf-and-cosine-similarity-for-computer-science-papers/

News. (2018, October 29). Retrieved from Motion Pictuce Assocaition: https://www.motionpictures.org/press/mpaa-celebrates-50-years-of-film-ratings/#:~:text=Over%20its%2050%2Dyear%20history,a%20total%20of%2029%2C791%20films.

Nixon, A. E. (2020, August 28). *Building a movie content based recommender using tf-idf*. Retrieved from Towards Data Science: https://towardsdatascience.com/content-based-recommender-systems-28a1dbd858f5

Oliphant, T. (2020). *NumPy v1.20.0*. Retrieved from NumPy: https://numpy.org/

pandas v 1.2.3. (2021). Retrieved from pandas: https://pandas.pydata.org/

Ratings FAQ. (2021). Retrieved from Internet Moive Database: https://help.imdb.com/article/imdb/track-movies-tv/ratings-faq/G67Y87TFYYP6TWAV?ref_=helpms_helpart_inline#

Reitz, K. (2019). *Request v2.25.1*. Retrieved from Requests: HTTP for Humans: https://docs.python-requests.org/en/master/

Sang-Min Choi, S.-K. K.-S. (2012). A movie recommendation algorithm based on genre correlations. *Elsevier*, 8079-8085. Retrieved from http://toc.yonsei.ac.kr/~emmous/papers/ESWA7426.pdf

Scikit Learn. (2021). Retrieved from sklearn.feature_extraction.text.TfidfVectorizer: https://scikit-learn.org/stable/modules/generated/sklearn.feature_extraction.text.TfidfVectorizer.html

Sharma, A. (2020, May 29). *Beginner Tutorial: Recommender Systems in Python*. Retrieved from Datacamp: https://www.datacamp.com/community/tutorials/recommender-systems-python

Stecanella, B. (2019, May 10). *What is TF-IDF?* Retrieved from MonkeyLearn: https://monkeylearn.com/blog/what-is-tf-idf/#:~:text=TF%2DIDF%20is%20a%20statistical,across%20a%20set%20of%20documents.

Stoll, J. (2021, January 18). *Movie releases in North America from 2000-2020*. Retrieved from Statista: https://www.statista.com/statistics/187122/movie-releases-in-north-america-since-2001/

The Movie Database API. (2021). Retrieved from The Movie Database: https://developers.themoviedb.org/3/getting-started/introduction