



# 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

<i>Introduction.....</i>	<i>1</i>
<i>Movie Recommender Strategies .....</i>	<i>1</i>
<i>Project Recommendation Algorithm.....</i>	<i>1</i>
<i>Algorithm Code .....</i>	<i>4</i>
<i>Example of Input and Output.....</i>	<i>14</i>
<i>Conclusion .....</i>	<i>15</i>
<i>References.....</i>	<i>16</i>

## *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.

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

$M = \text{Overall Movie score}$

$A = \text{Actor Score}$

$G = \text{Genre Score}$

$P = \text{Plot Score}$

$R = \text{Rating score from other individuals}$

Two databases are used in the construction of the algorithm. The first is the Open Movie database ([www.omdbapi.com](http://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](http://www.rottentomatoes.com)) (About Rotten Tomatoes, 2021). The second movie database used in the algorithm is The Movie Database API ([www.themoviedb.org](http://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:

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

$A = \text{Actor Score}$

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

$T = \text{Total number of actors in the UTML}$

$a = \text{actor in the UTML appering in the movie in the RML}$

- 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):

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

$\text{tf}_{i,j}$  = total number of occurences of i in j

$\text{df}_i$  = 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.

```
import omdb as omdb
import numpy as np
import pandas as pd
import requests
from sklearn.feature_extraction.text import TfidfVectorizer # to do
TF-IDF calculations
from sklearn.metrics.pairwise import cosine_similarity # to do Cosine
comparison

omdb.set_default('apikey', 'XXXXXX') #OMDB key, You will need your own
key
TMDBKey = 'XXXXXX' #TMDB key, You will need your own key
Users_list = [] # Movie list from user
final_list = [] # Exact movie list to get ratings
score_array = np.array([]) # Array for Rotten tomato score
movie_score_list = [] # List of movie's data
fav_actors = {} # Favorite actor list
TMDB_Actor_score = {} # score of the movie by actor
TMDB_Genre_score = {} # score of the movie by Genre
TMDB_plot_score = {} # score of the movie by plot
TMDB_vote_score = {} # score of the movie by user voting
name_id_xref = {} # Name to TMDB's actor id cross reference
dictionary
Recommendation_score = {} # total recommendation score for the movies
Top_rec_list = [] # Top movie recommendations
user_IMDB_id = [] # IMDB ids from user complied list
TMDB_total_movie_score = {} # Total movie score
actor_list = [] # actor from users movie list to find recommended
movies
plot_words = '' # words for plots of favorite movies used to rank
recommended movies used for score calculations
genre_list = '' # genre from user movie list used to rank recommended
movies used for score calculations
TMDB_genre_list = '' # genre from TMDB movie
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.

```
# Genre id cross reference
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).

```
# Function to add user movies to the top Users_list
# and make sures the title is not an empty space
def addmovie(movie): # O(1)
    if movie == '' or movie == ' ':
        print('Not a good movie title')
        return
    if movie == 'x':
        return
    Users_list.append(movie)
    return

# function to calculate the Td-Ifd matrix and cosine similarity
def Movie_score(user, recomen): # O(1)
    vectorizer = TfidfVectorizer(analyzer='word', min_df = 0,
stop_words = 'english')
    vectors = vectorizer.fit_transform([user, recomen])
    feature_names = vectorizer.get_feature_names_out()
    dense = vectors.todense()
    denselist = dense.tolist()
    Tdifd_matrix = pd.DataFrame(denselist, columns=feature_names)
    cosine = cosine_similarity(Tdifd_matrix)
    cosine_matrix = pd.DataFrame(cosine)
    score = cosine_matrix[0][1]
    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.

```
print('''This program will judge your taste in movies
by judging them to the Rotten Tomatoes score\n
Enter at least your top 5 movies.\n''')
```

```
# loop to enter the first 5 movies
```



```

while len(Users_list) < 5: # O(n)
    top_5 = input('Enter Movie Title ' + str(len(Users_list) + 1) + ':')
    addmovie(top_5)

print('\nYou can enter more movies to the list if you like or type "x" to exit\n')

# Adds more titles if the user wishes
more_movie = ''
while more_movie != 'x': # O(n)
    more_movie = input('Enter another movie: ')
    addmovie(more_movie)
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.

```

# Loop through all the users movie list
while len(Users_list) > 0: # O(n^3)

    for movie in Users_list: # Pulls the movie list from the user and
        pulls possible matches from OMDB
        choice_list = [] # empty list to put exact movies choices
        into
        res = omdb.request(s=movie, type='movie')
        title = res.json()

        if title['Response'] == 'True': # A match on the data base
            was found
            movie_choices = title['Search']
            for choice in movie_choices: # Formats the possible moves
                that came back from OMDB
                title = choice['Title'] # official movie name
                year = choice['Year'] # release date
                m_num = choice['imdbID'] # IMDB's ID number
                choice_list.append([title, year, m_num])

            # shows the different options of movies to pick from
            print('Here are the exact Movie titles from your list with
            release year ')
            for option in range(len(choice_list)):

```

```

        num = choice_list[option]
        print(option, num[0:2])

        # User selects the exact movie and test to make sure that
        it is a valid answer
        Title_select = input('Select the number of the exact Movie
you were thinking about \n')
        z = True
        while z:
            try:
                Title_select = int(Title_select)
                selected_movie = choice_list[Title_select]
                final_list.append(selected_movie) # adds the
movie to the final movie list
                Users_list.remove(movie) # removes the user movie
from the user list so that it is not duplicated
                z = False
            except:
                print('Not a valid choice')
                Title_select = input('Select the number of the
exact Movie you were thinking about \n')

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

```

# goes though final user list and pulls title, genre, actors, plot
for final in final_list: # O(n^2)
    ids = final[2]
    res = omdb.request(i=ids)
    Movie_info = res.json()

```

```

title = Movie_info['Title'] # collect Official Movie title name

genres = Movie_info['Genre'] # collect Genre
genre_list = genre_list + ',' + genres # Creates genre string for
Td-Ifd matrix and cosine similarity

actors = Movie_info['Actors'].split(',') # collect actors
# collect actors in to actors dictionary and keep count how many
times they appear
for name in actors:
    actor = name.lstrip()
    if actor not in fav_actors:
        fav_actors[actor] = 1
    else:
        fav_actors[actor] += 1

plot = Movie_info['Plot'] # Get plots text and splits it
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.

```

ratings = Movie_info['Ratings'] # collect ratings
RTper = 0 # initialized RTper to 0
for rating in ratings:
    # Finds the Rotten Tomatoes score and turns it into a %
    if rating['Source'] == 'Rotten Tomatoes':
        score = rating['Value'].replace('%', '')
        RTper = int(score) / 100
    else:
        # Finds the IMDB score and turns it into a % if there is no
        Rotten Tomatoes score
        if rating['Source'] == 'Internet Movie Database':
            IMDB_rating = rating['Value']
            IMDBper = float(IMDB_rating[:-3]) / 10
if RTper>0: # test to see if there is a Rotten tomatoes score
    movie_score_list.append([title, RTper])
else:
    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.

```
# Displays rating of movies from user final list
ml = pd.DataFrame(movie_score_list)
ml.columns = ["Movie", "Rating"]
ml['Rating'] = ml['Rating'].map('{:,.0%}'.format)
print(ml.iloc[:, [0, 1]])

print('\n')
for score in movie_score_list: # O(n)
    RT_percent = score[1]
    score_array = np.append(score_array, RT_percent)

RT_mean = score_array.mean()
print('Your Rotten Tomatoes Score is {:,.0%}'.format(RT_mean))

if RT_mean >= .9: # O(1)
    print("You have great taste in movies")
elif .9 > RT_mean >= .8:
    print("You have good taste in movies")
elif .8 > RT_mean >= .7:
    print("You have average taste in movies")
else:
    print("You have poor taste in movies")

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.

```
Creates actor TMDB cross reference dictionary
for i in fav_actors.keys(): # O(n)
    name = {"query": i}
    Actor_res = requests.get(Actor_url, params=name)
```

```

actor = Actor_res.json()
try:  # if a name on OMDb is not in TMDB it will skip the name
    x = actor["results"][0]["id"]
    if i not in name_id_xref:
        name_id_xref[i] = x
except:
    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.

```

# -----loops the actor name ids to get the movies they are in and
# adds them to the scoring dictionaries-----

# loop through actor id dictionary to get all movies the actors were
# in
for id in name_id_xref: # O(n^3)
    actor_multiplier = fav_actors[id]  # get the number of times that
    the actor appeared in the user movie list
    Actor_id = name_id_xref[id]
    Actor_id = str(Actor_id)
    Act_id_url = 'https://api.themoviedb.org/3/person/' + Actor_id + \
        '/movie_credits?api_key='+TMDBKey+'&language=en-US'
    Act_id_res = requests.get(Act_id_url)
    movie = Act_id_res.json()

    # loop through all the movies the favorite actor's have been in
    and pull movie details
    for titleNo in range(len(movie['cast'])):

        recommend_movie_id = movie['cast'][titleNo]['id']  # movie
        title id
        recommend_movie_genre = movie['cast'][titleNo]['genre_ids']  #
        movie Genre types
        recommend_movie_vote = movie['cast'][titleNo]['vote_average']
        # Movie user votes
        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.

```

# adds movie id to the TMDB_movie_actor_count dictionary and keeps
count of how many of the favorite actors are
# in the movie
if recommend_movie_id not in TMDB_movie_actor_count:
    TMDB_movie_actor_count[recommend_movie_id] = 1 * actor_multiplier
else:
    TMDB_movie_actor_count[recommend_movie_id] =
TMDB_movie_actor_count[recommend_movie_id] + (
    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.

```

# adds the Genre score to the TMDB_Genre_score dictionary
if recommend_movie_id not in TMDB_Genre_score:
    for genre_id in recommend_movie_genre: # Iterates through the
different Genre ids
        genre = genre_id_xref[genre_id]
        TMDB_genre_list = TMDB_genre_list + ',' + genre
        g_score = Movie_score(genre_list, TMDB_genre_list) # Genre cosine
score from TF-IDF for the movie
        TMDB_Genre_score[recommend_movie_id] = g_score
        TMDB_genre_list = '' # reset the genre list for each movie

# adds the Plot score to the TMDB_plot_score dictionary
if recommend_movie_id not in TMDB_plot_score:
    plot_score = Movie_score(plot_words, recommend_movie_plot) # Plot
cosine score from TF-IDF for the movie
    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.

```

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

```

# _____compile top 10 movies recommendations-----
-----
# 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
    # Retrieves the IMDB Number for the movie
    Movie_id_url = 'https://api.themoviedb.org/3/movie/' +
    TMDB_Movie_id + \

    '?api_key=a9c03ccd5aefa83d5c72d86bf0fb9cf0&language=en-US'
    movie_id_res = requests.get(Movie_id_url)
    movie_details = movie_id_res.json()
    IMDBid = movie_details['imdb_id']
    Recommended_Title = movie_details['original_title']
    movie_year = movie_details['release_date']
    box_office = movie_details['revenue']

    if box_office > 1000: # movie has to have been released in the
theater

        # checks to make sure the movie the user entered does not come
back as a recommended movie
        if IMDBid not in user_IMDB_id:
            Top_rec_list.append([Recommended_Title, movie_year[:4]])
            if count < 9:
                count += 1
            else:
                break

# Formats the top list
Top_list_format = pd.DataFrame(Top_rec_list)
Top_list_format.columns = ["Movie", "Year"]
print(Top_list_format)

```

## 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://markneedham.com/blog/2016/07/27/scikit-learn-tfidf-and-cosine-similarity-for-computer-science-papers/>
- News. (2018, October 29). Retrieved from Motion Picture Association: <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 Movie Database: [https://help.imdb.com/article/imdb/track-movies-tv/ratings-faq/G67Y87TFYYP6TWAV?ref\\_=helpms\\_helpart\\_inline#](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](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>