Social Network Analysis - Project Report

Risheek Rakshit S K
2018103580
Computer Science & Engineering
Anna University

Movie Recommendation System

Introduction

Recommendation systems are becoming increasingly important in today's extremely busy world. The purpose of a recommendation system basically is to search for content that would be interesting to an individual. Moreover, it involves a number of factors to create personalised lists of useful and interesting content specific to each user/individual. Recommendation systems are Artificial Intelligence based algorithms that skim through all possible options and create a customized list of items that are interesting and relevant to an individual.

These results are based on their profile, search/browsing history, what other people with similar traits/demographics are watching, and how likely are you to watch those movies. This is achieved through predictive modelling and heuristics with the data available.

Objective

- To develop various types of movie recommendation systems that would each satisfy a specific purpose.
- To recommend relevant movies to the users.

Base Paper

https://ijesc.org/upload/f0d1e3f5683da81c9018ff3308495420.A%20Movie%20Recommender%20System%20MOVREC%20using%20Machine%20Learning%20Techniques.pdf

Dataset

- *Wikipedia movie dataset*

  The dataset contains descriptions of 34,886 movies from around the world. Column descriptions are listed below: Release Year, Title, Origin/Ethnicity, Director, Plot, Genre, Wiki Page.

- *MovieLens dataset*

  The datasets describe ratings and free-text tagging activities from MovieLens, a movie recommendation service. It contains 20000263 ratings and 465564 tag applications across 27278 movies.

- *Movie metadata dataset*

  These files contain metadata for all 45,000 movies listed in the Full MovieLens Dataset. Data points include cast, crew, plot keywords, budget, revenue, posters, release dates, languages, production companies, countries, TMDB vote counts and vote averages. This dataset also has files containing 26 million ratings from 270,000 users for all 45,000 movies.

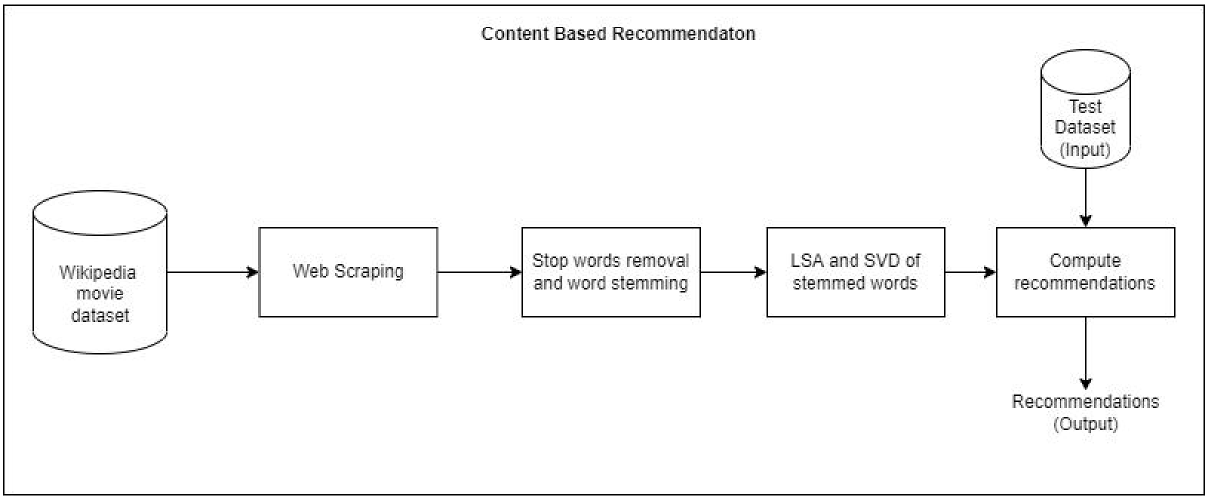
Algorithms

We are using three recommendation algorithms in this project, which are:

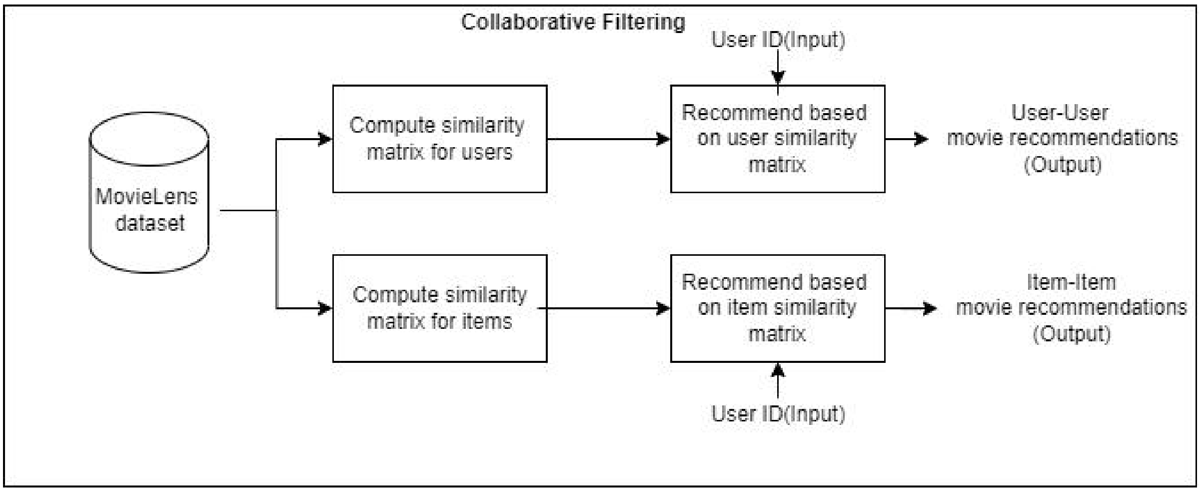
- Content Based
- Collaborative Filtering
- Popularity and Metadata (director, cast, keywords) Based

Block Diagram

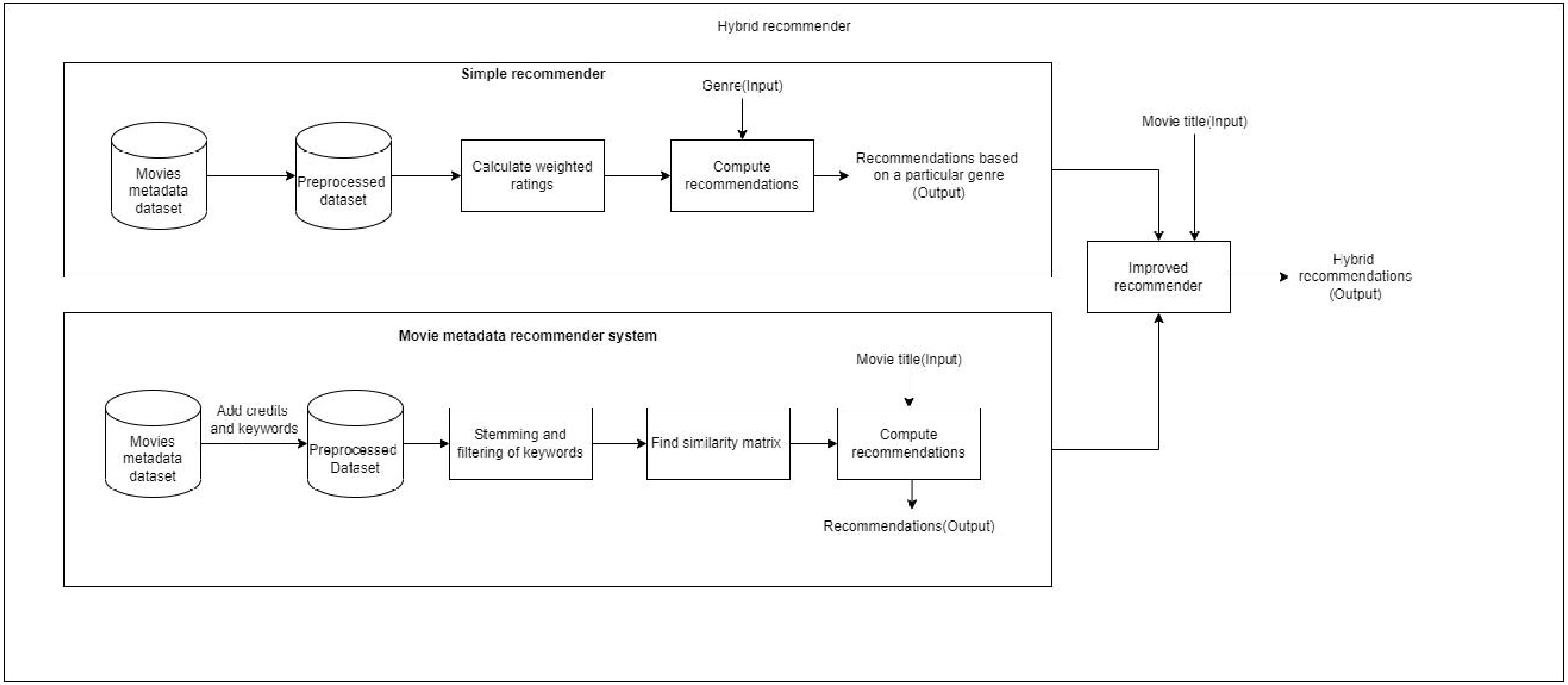
Content Based Recommendation



Collaborative Filtering



Hybrid Recommender



Modules

Content Based Recommendation

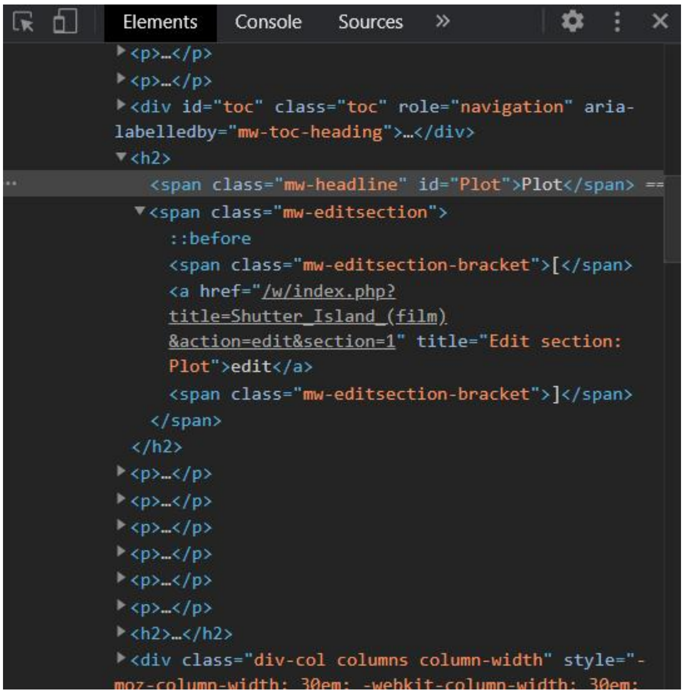
1.Web Scraping

Input: csv file with movie title, release year, genre and other columns (Wikipedia movie dataset).

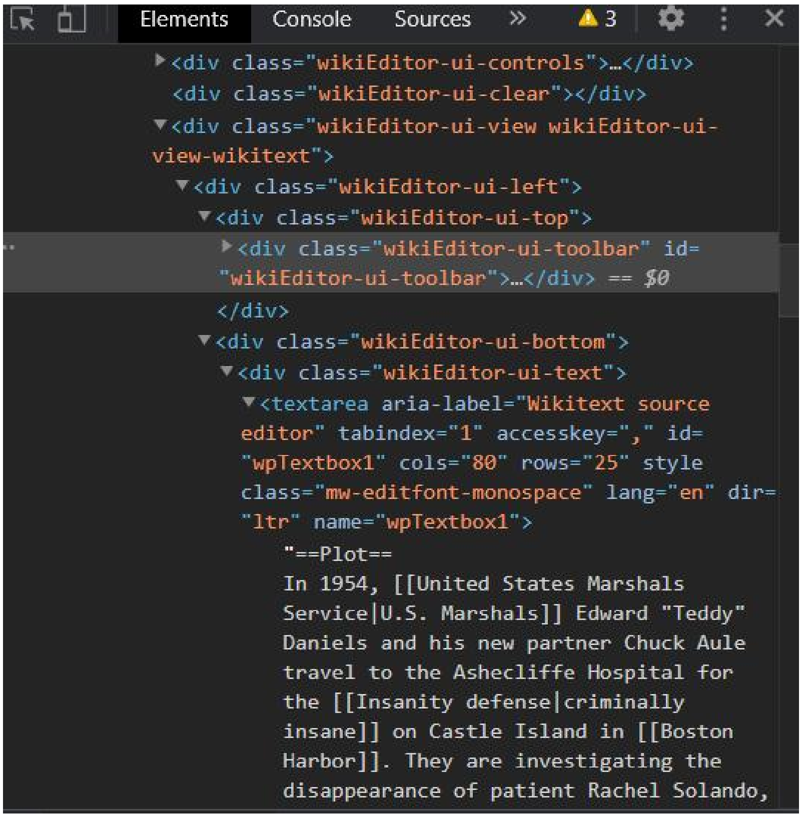
Output: Respective movie plots scraped from Wikipedia.

A csv file with movie title, release year, genre as columns is the input of this module. Most of the Wikipedia URL for a movie is in the form of en.wikipedia.org/ followed by the movie name, title cased and spaces replaced by underscore (_). So, a column was created as a 'Wiki Page' in which the URL of the movie is stored.

Then from the URL obtained we will have to scrape the plot. For most of the movies the story will be under the plot section. Moreover, Wikipedia provides privilege to edit each section. So, if we somehow are able to reach the section, we can obtain the entire plot of the movie which was uploaded. All these scraping works are done by using Python's Beautiful Soup.



We can see from the above picture that the edit section is below the Plot’s <span>.We will fetch the edit section’s URL by finding for the <a>’s href. Then we will have to fetch the required plot from the edit section.



The entire plot is in the <textarea>’s inner Text. It has lots and lots of noisy words which are Wikipedia related syntaxes , moreover all those words were mostly redirects and also nouns which are nowhere useful in our context, so they’ll have to be removed.

Like the same we will be removing the text enclosed in {{}} , <<>> etc… because the text within those braces will be a noun which is not required for us in finding the similarity.

The above code will, to some extent, remove the noisy words. Then the output is stored in the ‘Plot’ column . It is repeated for all the records in the table then it is later stored into the file from which the input was extracted.

Example output for one movie:

```
Plot :  == Plot ==An ambulance driver, a broker, a hospital worker, and a surgeon are abducted at dawn, which is traced to Dr.
Maaran, a doctor from  known for providing treatment to all at just . Maaran is arrested and interrogated by Inspector Ratnavel
"Randy", who is assigned the case. Maaran explains his motive for the abductions; the four were responsible for the death of an
auto driver's daughter and subsequent suicide of his wife due to their greed for money and negligence in providing proper healt
hcare. He gives Randy the locations of his hostages but asserts that he had already killed them simultaneously.Maaran reveals t
hat he is not Maaran but his  Vetri, a . Vetri was also responsible for the death of Dr. Arjun Zachariah, a corrupt doctor kill
ed during a stage performance in Paris two years ago. Dr. Daniel Arokiaraj, another corrupt doctor and the head of the state's
medical council, sees Maaran's inexpensive healthcare as a threat to his flourishing hospital business and decides to kill Maar
an using his goons but is saved in the nick of time by Vetri; he knocks Maaran unconscious and swaps places with him, providing
clues to the police which led to his arrest. Maaran is rescued, while Vetri, manages to escape from Randy. Later, Maaran confro
nts Vetri, believing him to be the cause of all his problems. Vadivu, Maaran's compounder, and also Vetri's assistant, interven
es and explains to Maaran why Vetri is targeting doctors indulging in corrupt medical practices and Daniel and Arjun in particu
lar.Maaran is the elder son of Aishwarya "Aishu" and Vetrimaaran, a village wrestler and chieftain from the 1970s. Vetrimaaran
has an  nature and he decides to build a temple in his area and holds a large festive event. However, a fire breaks out, injuri
ng many and killing two children due to lack of mobility. By the advice of Aishu, Vetrimaaran establishes a hospital in his vil
lage Manoor in  with Daniel and Arjun being made chief doctors, while Vetrimaaran managed the hospital. However, it soon turned
out that Daniel and Arjun were money-minded  and performed a  on Aishu when she was in labour with her second child, to extract
more money from Vetrimaaran. Aishu with an overdose of  and loss of blood, dies while the child was declared . When Vetrimaaran
found out how Aishu had died, he goes to confront Daniel, but was attacked. Vetrimaaran fights but in the end gets stabbed to d
eath by Daniel's assistant Kasi, but before he died, he placed an unconscious Maaran safely on a Chennai-bound lorry. Daniel re
veals that he was the one who set fire to the temple so that Vetrimaaran will build a hospital for their business profits. Vetr
imaaran swears that one day Daniel will pay for his misdeeds. Maaran lost his memory and was unable to recollect anything that
happened before passing out. After killing Vetrimaaran, Daniel and Arjun  money in medical service to consolidate their dictato
rship over the years, while creating a massive  from beyond.Meanwhile, Vetrimaaran's second child miraculously survived, and th
is child was Vetri. Vadivu, their paternal uncle took care of him. Both were adopted by a famous magician Salim Ghosh, and it w
as from him that Vetri learned all his magic tricks. Vetri had his anger and sense of justice, and Maaran had his sense of self
```

Movie Plot scrapped from Wikipedia for the movie Mersal

Code Snippet:

Plot Extraction from Wiki

In [7]:
```
def ExtractPlot(df):
    for i in df.index:
        url = df.loc[i]['Wiki Page']
        html = urlopen(url)
        soup = BeautifulSoup(html, 'html.parser')
        try:
            url = "https://en.wikipedia.org"+soup.find('a',{'title':'Edit section: Plot'})['href']
        except:
            continue
        html = urlopen(url)
        soup = BeautifulSoup(html, 'html.parser')
        plot = soup.find('textarea').text
        try:
            i = plot.index('==')
            plot = plot[i+len('=='):]
            i = plot.index('==')
            plot = plot[i+len('=='):]
        except:
            plot = plot
            df.loc[i,'Plot'] = 'a'
        plot = func(plot)
        df.loc[i,'Plot'] = plot
```

2.Stop words removal and word stemming

Input: Plot of the movie.

Output: Stemmed words.

We will have to split the plot into words using nltk's word tokenizer. The first thing we will have to remove is the character names, places in which the story happens, generalising it we need to remove the nouns.

After removal of nouns we will remove punctuations and in those places we will add space. Then we will have to remove the stop words like 'is', 'was' etc…. We have a certain list of stop words predefined in nltk we will use that to remove the stop words. Then the stop words removed list is stemmed using nltk's Porter Stemmer algorithm.

Code Snippet:

```
In [9]:
def SWRandStem(plot):
    p = ''

    for i in plot:
        if(i!='\n'):
            p += i
    plot = p
    stop_words = set(stopwords.words('english'))
    tagged_sentence = nltk.tag.pos_tag(plot.split())
    edited_sentence = [word for word,tag in tagged_sentence if tag != 'NNP' and tag != 'NNPS']
    example_sentence = ' '.join(edited_sentence)
    punctuations = '''!()-[]{};:'"\,<>./?@#$%^&*_~0123456789'''
    no_punct = ""
    for char in example_sentence:
       if char not in punctuations:
           no_punct = no_punct + char
       else:
            no_punct += ' '
    example_sentence = no_punct
    tagged_sentence = nltk.tag.pos_tag(example_sentence.split())
    edited_sentence = [word for word,tag in tagged_sentence if tag not in ['NNP','NNPS','NNS','NN']]
    example_sentence = ' '.join(edited_sentence)
    example_sentence = example_sentence.lower()
    stop_words = set(stopwords.words('english'))

    word_tokens = word_tokenize(example_sentence)

    filtered_sentence = [w for w in word_tokens if not w in stop_words]
    filt = [ps.stem(filtered_sentence[i]) for i in range(len(filtered_sentence))]
    filt = ' '.join(filt)
    return filt
```

3.LSA and SVD

Input: Space separated stemmed words.

Output: An Array of genre related words.

In this module we will construct a matrix with documents (movie plots) as rows and the words in those documents as columns. In linear algebra, the **singular value decomposition** (**SVD**) is a factorization of a real or complex matrix that generalizes the eigen decomposition of a square normal matrix to any m*n matrix via an extension of the polar decomposition.

Where the m*n matrix is reduced to a n*n matrix and thus the number of related words. For finding the related words we use TFIDF (Term Frequency Inverse Document Frequency) vectorizer of sklearn's feature extraction.

Code Snippet:

```
In [10]:
gen = ['sci_fi','fantasy','crime','adventure','music']
cons = pd.DataFrame()
def SelectTopic(genre,gen_name):
    global cons
    genre = genre[genre['Plot_reduction']!= np.nan]
    th_plot = genre['Plot_reduction']
    th_plot = th_plot.dropna()

    th_plot_80,th_plot_20 = train_test_split(th_plot,train_size = int(len(th_plot)*0.8),\
                                             test_size = len(th_plot)-int(len(th_plot)*0.8))

    if(gen_name in gen):
        vectorizer = TfidfVectorizer(min_df = 50,stop_words='english',max_df = 0.9,max_features = 500)
    else:
        vectorizer = TfidfVectorizer(min_df = 200,stop_words='english',max_df = 0.9,max_features = 500)
    th_plot = th_plot_80.append(pd.Series(th_plot_20))

    bagofwords = vectorizer.fit_transform(th_plot)
    feature_names = vectorizer.get_feature_names()
    dense = bagofwords.todense()
    denselist = dense.tolist()
    df = pd.DataFrame(denselist, columns=feature_names)
    rat = df.iloc[df.shape[0]-len(th_plot_20):]

    ncomp = 15
    svd = TruncatedSVD(n_components = ncomp)
    lsa = svd.fit_transform(bagofwords)

    diction30 = vectorizer.get_feature_names()

    topic = ['Topic '+str(i) for i in range(1,ncomp+1)]

    encoding_matrix = pd.DataFrame(svd.components_,index = topic).T
    encoding_matrix['words'] = diction30
    diction30 = encoding_matrix.sort_values(by=['Topic 1'],ascending = False)
    diction30.index = diction30['words']
    diction30 = diction30.drop(['words'],axis = 1)
#    cons = diction30
    rat = rat.T
    rat.index.set_names(['words'],inplace = True)
    comp = pd.concat([rat,diction30], axis=1, join='inner').T
#    cons = comp
    c = cosine_similarity(comp,comp)
    final = pd.DataFrame(c[len(c[0])-ncomp:,:comp.shape[0]-ncomp],index = topic)


    final['sum'] = final.sum(axis = 1,skipna = True)
    cons = final['sum']
    return final.idxmax(axis = 0,skipna = True)['sum']
```

```
In [11]:
gen = ['sci_fi','fantasy','crime','adventure','music']
ncomp =15
def FindSimilarity(genre,summa,gen_name,title):
    global cons
    global cons1
    genre = genre[genre['Plot_reduction']!= np.nan]
    th_plot = genre['Plot_reduction']
    th_plot = th_plot.append(pd.Series(summa))

    if(gen_name in gen):
        vectorizer = TfidfVectorizer(min_df = 50,stop_words='english',max_df = 0.9,max_features = 500)
    else:
        vectorizer = TfidfVectorizer(min_df = 200,stop_words='english',max_df = 0.9,max_features = 500)
    bagofwords = vectorizer.fit_transform(th_plot)
    feature_names = vectorizer.get_feature_names()
    dense = bagofwords.todense()
    denselist = dense.tolist()
    df2 = pd.DataFrame(denselist, columns=feature_names)
    cons1 = df2
    rat = df2.iloc[df2.shape[0]-len(summa):]
    rat = rat.T
    rat.index.set_names(['words'],inplace = True)

    svd = TruncatedSVD(n_components = ncomp)
    lsa = svd.fit_transform(bagofwords)
    diction30 = vectorizer.get_feature_names()

    top = SelectTopic(genre,gen_name)
    topic = ['Topic '+str(i) for i in range(1,ncomp+1)]
    cons = feature_names
```

```python
    encoding_matrix = pd.DataFrame(svd.components_,index = topic).T
    cons = encoding_matrix
    encoding_matrix['words'] = diction30
    diction30 = encoding_matrix.sort_values(by=['Topic 1'],ascending = False)
    diction30.index = diction30['words']
    topic.remove(top)
    diction30 = diction30.drop(topic,axis = 1)
    diction30 = diction30.drop(['words'],axis = 1)

    comp = pd.concat([rat,diction30], axis=1, join='inner').T
    comp = comp.drop(['tell','ask','come','want','make','say','young'], axis=1, errors='ignore')

    if gen_name not in ['horror','thriller','action']:
        comp = comp.drop(['kill'],axis = 1,errors = 'ignore')

    c = cosine_similarity(comp,comp)

    temp = pd.DataFrame(c[len(c[0])-1],index = title)
    temp.rename(columns = {0:gen_name},inplace = True)
    return temp.T
```

Bagofwords holds the list of selected words the tfidf scores of those words can be selected using **svd.components_.** TruncatedSVD is used for selection of various topics from the document array. Number of topics can be passed as an argument via n_components .

encoding_matrix consists of scores of the words in the get_feature_names().The textual similarity between dataset and input plot is calculated and it is returned.

Out[18]:

	ratchasan	96	santhosh subramaniyam	kutram 23	vtv	thozha	silence	anandham	zodiac	the silence of the lambs	...	Viswasam	Harry Potter and the Philosopher's Stone	Maya
action	0.458806	0.247055	0.274224	0.437196	0.256206	0.276975	0.528012	0.280074	0.405475	0.487408	...	0.550614	0.405831	0.351979
adventure	0.242600	0.190242	0.246525	0.302848	0.216153	0.277365	0.363746	0.238982	0.283763	0.316657	...	0.341311	0.361398	0.289157
crime	0.302169	0.170987	0.217991	0.328189	0.236445	0.235912	0.357079	0.238227	0.267117	0.299495	...	0.270394	0.283523	0.289781
drama	0.188556	0.235667	0.453189	0.247129	0.308780	0.571308	0.282007	0.485333	0.234671	0.240226	...	0.405210	0.243782	0.239550
fantasy	0.451376	0.298064	0.350654	0.421919	0.376725	0.383334	0.486407	0.372243	0.411706	0.519554	...	0.464542	0.743536	0.392137
horror	0.355500	0.171024	0.220470	0.400027	0.284929	0.250987	0.532003	0.271527	0.446992	0.519433	...	0.334007	0.441606	0.615502
romance	0.263091	0.683040	0.676998	0.281375	0.687418	0.437195	0.433516	0.567790	0.300601	0.296649	...	0.523907	0.318640	0.351447
sci_fi	0.290915	0.172670	0.234728	0.372594	0.303082	0.307951	0.414208	0.281295	0.335515	0.349897	...	0.359967	0.414276	0.358748
thriller	0.745720	0.275500	0.358863	0.705330	0.383990	0.290501	0.721022	0.359951	0.664420	0.735625	...	0.429655	0.559172	0.487563

9 rows × 26 columns

For our 25 input movies we have got the similarity score in that particular genre . We can see that Ratchasan, which is a ‘psycho thriller’ movie, is more correlated in the thriller genre rather than other genres.

We compute our recommendation score by taking three values a1 (the film’s similarity with that particular genre), a2 (similarity score between the film and the recommended film) , a3 (the recommended film’s similarity score with that genre) . Then we take a softmax of all the three . Then we find the cumulative average and then multiply with the weight which is the similarity score of the movie in a particular genre multiplied by 100.

In [17]:
```
def FindAccuracy(ans):
    cumu_sum = 0
    for ind,col in enumerate(ans.columns):
        wt = [ans[col][i] * 100 for i in range(len(ans[col]))]
        sumi = 0

        for i in range(first_h):
            sumj = 0
            for j in range(len(genre_df)):
                a1=FindSimilarity(genre_df[j],\
                      df[df['Title'] == fin[ind].index[i]][:1]['Plot_reduction'].values,\
                      genre_nm[j],[fin[ind].index[i],'Topic1']).iloc[0,0]
                a2,a3 = ans[col][genre_nm[j]],fin[ind].values[i]
                a1,a2,a3 = softmax(np.array([a1,a2,a3]))
                mx = max(a1,a2,a3)
                sumj += (a1/mx + a2/mx + a3/mx)/3*wt[j]
            sumi += (sumj/sum(wt))
        cumu_sum += (sumi*100)
        print(ind,end = ' ')
    format_c = '{:.2f}'.format(cumu_sum/(len(ans.columns))/10)
    print('\nRecomendation Score : '+format_c+'/100')
    return cumu_sum/10/(len(ans.columns))

```

Recommendation score:

```
FindAccuracy(ans)
```

```
0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23
Recomendation Score : 87.05/100
```

```
87.05123549841228
```

Recommendation for the movie Ratchasan:

Out[21]:
```
Title
Pawn                                         0.778800
Shocker                                      0.777774
Deadfall                                     0.766957
Ee Adutha Kalathu (ഈ അടുത്ത കാലത്ത്)      0.764180
Australia                                    0.759452
Freelance                                    0.740670
Tattoo                                       0.740149
Naku Penta Naku Taka                         0.739304
Hammett                                      0.733938
Man-Thing                                    0.723099
Name: ratchasan, dtype: float64
```

Collaborative Filtering

1.Compute similarity matrix

Input: MovieLens Dataset

Output: Similarity matrices

Code Snippet:

In [19]:
```

for i in items['movie id']:
    if(len(ratings[ratings['movie_id'] == i]) <= 10):
        items.drop(i-1,inplace = True)
        ratings.drop(ratings[ratings['movie_id'] == i].index,inplace = True)

def predict(ratings, similarity, type='user'):

    if type == 'user':
        mean_user_rating = ratings.mean(axis=1).reshape(-1,1)

        ratings_diff = (ratings - mean_user_rating)
        pred = mean_user_rating + similarity.dot(ratings_diff) / np.array([np.abs(similarity).sum(axis=1)]).T

    elif type == 'item':
        pred = ratings.dot(similarity) / np.array([np.abs(similarity).sum(axis=1)])

    return pred
```

In [20]:
```
items = items.reset_index()
```

In [21]:
```
n_users = ratings.user_id.unique().shape[0]
n_items = ratings.movie_id.unique().shape[0]
print(n_users,n_items)
```

```
943 1119
```

In [22]:
```
data_matrix = np.zeros((n_users, n_items))
for line in ratings.itertuples():
    data_matrix[line[1]-1, items[items['movie id'] == line[2]].index.values[0] ] = line[3]

from sklearn.metrics.pairwise import pairwise_distances
user_similarity = pairwise_distances(data_matrix, metric='cosine')
user_prediction = predict(data_matrix, user_similarity, type='user')
user_pred = pd.DataFrame(user_prediction)

item_similarity = pairwise_distances(data_matrix.T, metric='cosine')
item_prediction = predict(data_matrix, item_similarity, type='item')
item pred = pd.DataFrame(item prediction)
```

Out[23]:

	0	1	2	3	4	5	6	7	8	9	...	1109	1110	1111	1112	11
0	2.235147	0.905309	0.802637	1.179952	0.813373	0.649508	1.953717	1.332476	1.684132	0.876699	...	0.592456	0.600405	0.605281	0.603287	0.6187
1	1.721155	0.341706	0.153312	0.689435	0.183163	-0.039437	1.451273	0.834104	1.065142	0.218675	...	-0.095335	-0.083947	-0.078534	-0.081460	-0.0663
2	1.733892	0.265929	0.095588	0.621250	0.110092	-0.099101	1.425780	0.772978	1.072625	0.173057	...	-0.162878	-0.153909	-0.148191	-0.149763	-0.1355
3	1.646599	0.208889	0.042018	0.560581	0.056682	-0.147971	1.352342	0.712612	1.012281	0.126791	...	-0.211812	-0.201886	-0.198297	-0.199491	-0.1847
4	1.833881	0.493106	0.394702	0.800791	0.399657	0.236958	1.585258	0.946607	1.331683	0.478224	...	0.170942	0.178991	0.178287	0.178943	0.1957
...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	
938	1.634921	0.304397	0.135463	0.648047	0.157835	-0.038829	1.387420	0.789011	1.028555	0.220079	...	-0.100817	-0.090443	-0.086544	-0.089210	-0.0713
939	1.834629	0.430399	0.298027	0.726663	0.305992	0.117922	1.526185	0.863983	1.206493	0.370406	...	0.054573	0.065137	0.069030	0.068257	0.0800
940	1.512465	0.196274	0.023087	0.545502	0.054860	-0.148657	1.241485	0.686661	0.955857	0.113594	...	-0.210165	-0.201302	-0.196955	-0.198497	-0.1827
941	1.811383	0.405065	0.276039	0.726800	0.281799	0.087434	1.551089	0.850030	1.206129	0.343305	...	0.021760	0.033368	0.033057	0.036058	0.0423
942	1.913402	0.554451	0.460539	0.854916	0.463838	0.316787	1.638901	1.021269	1.364795	0.563002	...	0.249043	0.257431	0.261032	0.258860	0.2782

943 rows × 1119 columns

User-User Similarity Matrix

Out[24]:

	0	1	2	3	4	5	6	7	8	9	...	1109	1110	1111	1112	1113
0	0.695108	0.732261	0.768520	0.691876	0.776530	0.804270	0.693761	0.715756	0.729759	0.776204	...	0.827081	0.822338	0.826941	0.826365	0.830749
1	0.173203	0.209972	0.194437	0.200298	0.202680	0.192151	0.176658	0.193793	0.171772	0.186274	...	0.203740	0.205072	0.208352	0.205157	0.205723
2	0.140326	0.147867	0.138660	0.147642	0.141411	0.136493	0.136492	0.145628	0.136947	0.138308	...	0.142039	0.137547	0.143169	0.141055	0.140575
3	0.076324	0.084774	0.081256	0.083995	0.082965	0.083616	0.075500	0.083675	0.078705	0.084403	...	0.086742	0.085210	0.086677	0.085318	0.086304
4	0.349602	0.352207	0.399179	0.352867	0.393322	0.433752	0.362238	0.366299	0.397460	0.413498	...	0.427081	0.422173	0.403575	0.412337	0.426602
...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...
938	0.151608	0.183636	0.174363	0.183868	0.177861	0.186784	0.161154	0.179896	0.158921	0.185671	...	0.190875	0.190198	0.190900	0.185610	0.194984
939	0.269168	0.298175	0.310883	0.270590	0.308917	0.318010	0.266209	0.268786	0.276119	0.304603	...	0.320448	0.324534	0.323808	0.326377	0.318625
940	0.052898	0.072610	0.067910	0.070933	0.074907	0.077477	0.053558	0.069217	0.063336	0.074252	...	0.081450	0.078837	0.080494	0.079843	0.081172
941	0.258395	0.280760	0.299015	0.268342	0.293586	0.294498	0.269593	0.254769	0.268526	0.283870	...	0.291866	0.299990	0.287385	0.298291	0.280405
942	0.402151	0.391831	0.443970	0.392732	0.437508	0.505460	0.405244	0.424028	0.436678	0.488501	...	0.493139	0.487811	0.488952	0.482933	0.504754

943 rows × 1119 columns

Item-Item Similarity Matrix

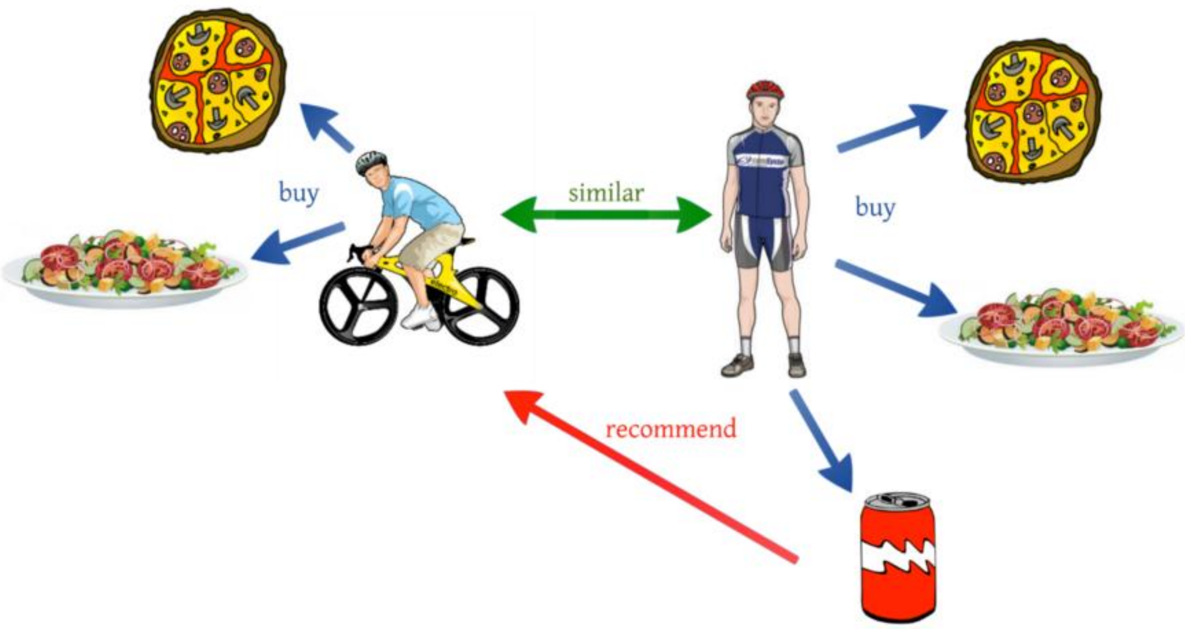
2.Recommendations based on similarity matrix

Input: User ID

Output: Movie recommendations

User-User recommendation

This algorithm first finds the similarity score between users. Based on this similarity score, it then picks out the most similar users and recommends products which these similar users have liked or bought previously.



In terms of our movies example from earlier, this algorithm finds the similarity between each user based on the ratings they have previously given to different movies. The prediction of an item for a user *u* is calculated by computing the weighted sum of the user ratings given by other users to an item *i*.

The prediction *Pu,i* is given by:

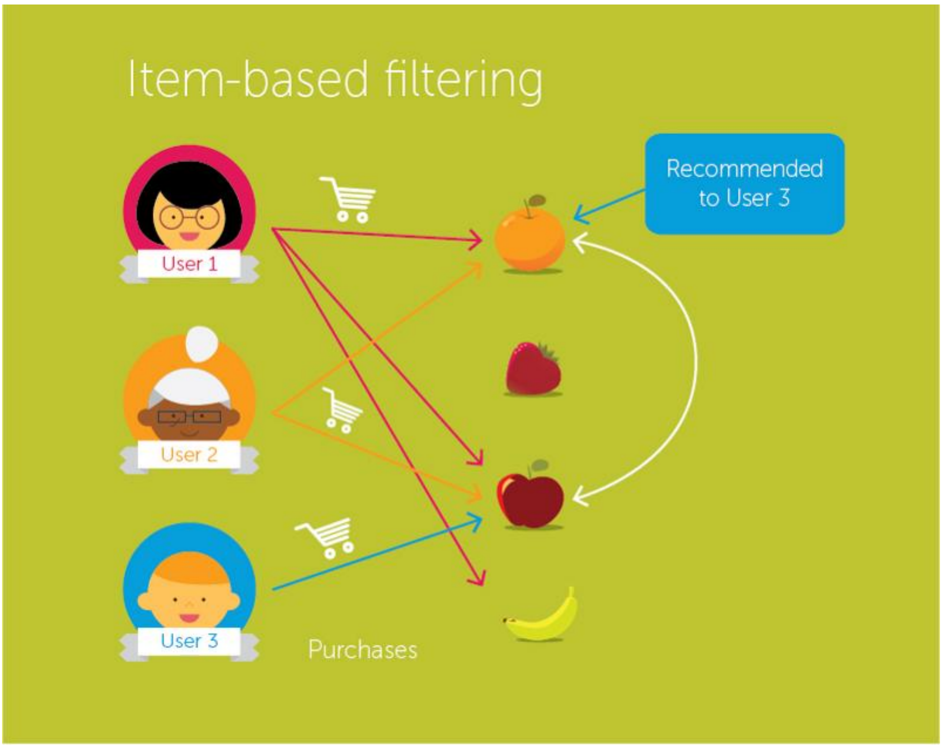
$$P_{u,i} = \frac{\sum_v (r_{v,i} * s_{u,v})}{\sum_v s_{u,v}}$$

Here,

- *Pu,i* is the prediction of an item
- *Rv,i* is the rating given by a user *v* to a movie *i*
- *Su,v* is the similarity between users

Item-Item recommendation

In this algorithm, we compute the similarity between each pair of items.



So in our case we will find the similarity between each movie pair and based on that, we will recommend similar movies which are liked by the users in the past. This algorithm works similar to user-user collaborative filtering with just a little change — instead of taking the weighted sum of ratings of “user-neighbours”, we take the weighted sum of ratings of “item-neighbours”. The prediction is given by:

$$P_{u,i} = \frac{\sum_N (s_{i,N} * R_{u,N})}{\sum_N (|s_{i,N}|)}$$

Code Snippet:

```
In [ ]:
first_h = 10
cols = items.columns[6:]

def UserUserRecommendation(userid):
    user_indv = ratings[ratings['user_id'] == userid]['movie_id'].values
    user_pred_indv = items['movie id'][user_pred.iloc[userid-1].sort_values(ascending = False).index.values]
    diff = np.setdiff1d(user_pred_indv, user_indv)
    diff = [items[items['movie id'] == diff[i]].index.values[0] for i in range(len(diff))]
    rec_mov = user_pred.iloc[0,diff].sort_values(ascending = False)[:first_h]
    li = [ list(cols[(items.iloc[rec_mov.index.values,6:].values == 1)[i]].values) for i in range(first_h)]
    li = [','.join(i) for i in li]
    return [i+' -- '+j for i,j in zip(list(items.iloc[rec_mov.index.values]['movie title']),li)]
```

User-User recommendation

```
In [ ]:
def ItemItemRecommendation(userid):
    pred = None
    for i in ratings[ratings['user_id'] == userid].sort_values(by = ['unix_timestamp'])['movie_id']:
        ind = items[items['movie id'] == i].index.values[0]
        k = item_df[ind]
        k = k.sort_values(ascending = False)
        k = k[:10]
        if type(pred) != pd.Series:
            pred = k
        else:
            pred.append(k)
            pred = pred.sort_values(ascending = False)[:10]

    rec_mov = pred
    lis = [ list(cols[(items.iloc[rec_mov.index.values,6:].values == 1)[i]].values) for i in range(first_h)]
    lis = [','.join(i) for i in lis]
    return [i+' -- '+j for i,j in zip(list(items.iloc[rec_mov.index.values]['movie title']),lis)]
```

Item-Item recommendation

```
In [31]: UserUserRecommendation(6)

Out[31]: ['Return of the Jedi (1983) -- Action,Adventure,Romance,Sci-Fi,War',
          'Scream (1996) -- Horror,Thriller',
          'Air Force One (1997) -- Action,Thriller',
          'Titanic (1997) -- Action,Drama,Romance',
          'Independence Day (ID4) (1996) -- Action,Sci-Fi,War',
          'Empire Strikes Back, The (1980) -- Action,Adventure,Drama,Romance,Sci-Fi,War',
          'Star Trek: First Contact (1996) -- Action,Adventure,Sci-Fi',
          'Indiana Jones and the Last Crusade (1989) -- Action,Adventure',
          'Conspiracy Theory (1997) -- Action,Mystery,Romance,Thriller',
          'Saint, The (1997) -- Action,Romance,Thriller']
```

User-User Movie Recommendations for User ID 6

```
In [34]: ItemItemRecommendation(943)

Out[34]: ['Truman Show, The (1998) -- Drama',
          "Gone Fishin' (1997) -- Comedy",
          'Half Baked (1998) -- Comedy',
          'Telling Lies in America (1997) -- Drama',
          'Four Days in September (1997) -- Drama',
          'Palmetto (1998) -- Film-Noir,Mystery,Thriller',
          'One Night Stand (1997) -- Drama',
          'Switchback (1997) -- Thriller',
          'Oscar & Lucinda (1997) -- Drama,Romance',
          'Welcome To Sarajevo (1997) -- Drama,War']
```

Item-Item Movie Recommendations for User ID 943

Recommendation Score

```
In [38]: format_c = '{:.2f}'.format(ReccomScore('user'))
         print('Recommendation Score for user-user recommendation: '+format_c+'/100')

Recommendation Score for user-user recommendation: 67.29/100
```

```
In [39]: format_c = '{:.2f}'.format(ReccomScore('item'))
         print('Recommendation Score for item-item recommendation: '+format_c+'/100')

Recommendation Score for item-item recommendation: 66.45/100
```

Simple recommender

1.Pre-processing

Input: Movie Metadata dataset

Output: Pre-processed dataset

Various pre-processing steps have been performed to obtain the desired format.

Code Snippet:

```
In [4]: md['genres'] = md['genres'].fillna('[]').apply(literal_eval).apply(lambda x: [i['name'] for i in x] if isinstance(x, list) e
```

```
In [6]:
vote_counts = md[md['vote_count'].notnull()]['vote_count'].astype('int')
print("votes counts\n",vote_counts)
vote_averages = md[md['vote_average'].notnull()]['vote_average'].astype('int')
print("vote average\n",vote_averages)
C = vote_averages.mean()
```

```
In [9]:
md['year'] = pd.to_datetime(md['release_date'], errors='coerce').apply(lambda x: str(x).split('-')[0] if x != np.nan else np
```

Out[16]:

	title	year	vote_count	vote_average	popularity	genres	wr
15480	Inception	2010	14075	8	29.108149	[Action, Thriller, Science Fiction, Mystery, A...	7.917588
12481	The Dark Knight	2008	12269	8	123.167259	[Drama, Action, Crime, Thriller]	7.905871
22879	Interstellar	2014	11187	8	32.213481	[Adventure, Drama, Science Fiction]	7.897107
2843	Fight Club	1999	9678	8	63.869599	[Drama]	7.881753
4863	The Lord of the Rings: The Fellowship of the Ring	2001	8892	8	32.070725	[Adventure, Fantasy, Action]	7.871787
292	Pulp Fiction	1994	8670	8	140.950236	[Thriller, Crime]	7.868660
314	The Shawshank Redemption	1994	8358	8	51.645403	[Drama, Crime]	7.864000
7000	The Lord of the Rings: The Return of the King	2003	8226	8	29.324358	[Adventure, Fantasy, Action]	7.861927
351	Forrest Gump	1994	8147	8	48.307194	[Comedy, Drama, Romance]	7.860656
5814	The Lord of the Rings: The Two Towers	2002	7641	8	29.423537	[Adventure, Fantasy, Action]	7.851924
256	Star Wars	1977	6778	8	42.149697	[Adventure, Action, Science Fiction]	7.834205
1225	Back to the Future	1985	6239	8	25.778509	[Adventure, Comedy, Science Fiction, Family]	7.820813
834	The Godfather	1972	6024	8	41.109264	[Drama, Crime]	7.814847
1154	The Empire Strikes Back	1980	5998	8	19.470959	[Adventure, Action, Science Fiction]	7.814099
46	Se7en	1995	5915	8	18.45743	[Crime, Mystery, Thriller]	7.811669

Pre-processed dataset

2.Calculate weighted ratings

Input: Pre-processed dataset

Output: Weighted ratings

We are using IMDB's *weighted rating* formula to construct the chart. Mathematically, it is represented as follows:

$$\text{Weighted Rating (WR)} = \left(\frac{v}{v+m} . R\right) + \left(\frac{m}{v+m} . C\right)$$

where,

- v is the number of votes for the movie
- m is the minimum votes required to be listed in the chart
- R is the average rating of the movie
- C is the mean vote across the whole report

The next step is to determine an appropriate value for *m*, the minimum votes required to be listed in the chart. We will use 95th percentile as our cutoff. In other words, for a movie to feature in the charts, it must have more votes than at least 95% of the movies in the list.

```
In [7]:  m = vote_counts.quantile(0.95)
         m

Out[7]: 434.0
```

Therefore, to qualify to be considered for the chart, a movie has to have at least **434 votes.**

Code Snippet:

```
In [13]: def weighted_rating(x):
             v = x['vote_count']
             R = x['vote_average']
             return (v/(v+m) * R) + (m/(m+v) * C)
```

3.Compute recommendations

Input: Genre

Output: Movie recommendations

Code Snippet:

```
In [18]: def build_chart(genre, percentile=0.85):
             df = gen_md[gen_md['genre'] == genre]
             vote_counts = df[df['vote_count'].notnull()]['vote_count'].astype('int')
             vote_averages = df[df['vote_average'].notnull()]['vote_average'].astype('int')
             C = vote_averages.mean()
             m = vote_counts.quantile(percentile)

             qualified = df[(df['vote_count'] >= m) & (df['vote_count'].notnull()) & (df['vote_average'].notnull())][['title', 'year'
             qualified['vote_count'] = qualified['vote_count'].astype('int')
             qualified['vote_average'] = qualified['vote_average'].astype('int')

             qualified['wr'] = qualified.apply(lambda x: (x['vote_count']/(x['vote_count']+m) * x['vote_average']) + (m/(m+x['vote_co

             qualified = qualified.sort_values('wr', ascending=False).head(250)

             return qualified
```

We can also get the top films with certain genre. We separate the genres for each film and make it a new film with that genre. Now we view the top 15 horror movies by applying the same simple recommender as shown above. The result is given below.

```
In [90]: build_chart('Horror').head(15)
```

Out[90]:

	title	year	vote_count	vote_average	popularity	wr
1213	The Shining	1980	3890	8	19.611589	7.901294
1176	Psycho	1960	2405	8	36.826309	7.843335
1171	Alien	1979	4564	7	23.37742	6.941936
41492	Split	2016	4461	7	28.920839	6.940631
14236	Zombieland	2009	3655	7	11.063029	6.927969
1158	Aliens	1986	3282	7	21.761179	6.920081
21276	The Conjuring	2013	3169	7	14.90169	6.917338
42169	Get Out	2017	2978	7	36.894806	6.912248
1338	Jaws	1975	2628	7	19.726114	6.901088
8147	Shaun of the Dead	2004	2479	7	14.902948	6.895426
8230	Saw	2004	2255	7	23.508433	6.885580
1888	The Exorcist	1973	2046	7	12.137595	6.874560
39097	The Conjuring 2	2016	2018	7	14.767317	6.872920
6353	28 Days Later	2002	1816	7	17.656951	6.859688
12277	Sweeney Todd: The Demon Barber of Fleet Street	2007	1745	7	10.038401	6.854358

Movie metadata recommender system

1.Pre-processing

Input: Movies metadata dataset

Output: Pre-processed dataset

Code Snippet:

```
In [23]:
md = md.drop([19730, 29503, 35587])
md['id'] = md['id'].astype('int')
```

```
In [24]:
keywords['id'] = keywords['id'].astype('int')
credits['id'] = credits['id'].astype('int')
md['id'] = md['id'].astype('int')
```

```
In [27]:
links_small = pd.read_csv('links_small.csv')
links_small = links_small[links_small['tmdbId'].notnull()]['tmdbId'].astype('int')
links_small
```

```
In [29]:
smd['tagline'] = smd['tagline'].fillna('')
smd['description'] = smd['overview'] + smd['tagline']
smd['description'] = smd['description'].fillna('')
```

```
In [31]:
smd['cast'] = smd['cast'].apply(literal_eval)
smd['crew'] = smd['crew'].apply(literal_eval)
smd['keywords'] = smd['keywords'].apply(literal_eval)
smd['cast_size'] = smd['cast'].apply(lambda x: len(x))
smd['crew_size'] = smd['crew'].apply(lambda x: len(x))
```

```python
In [34]: def get_director(x):
             for i in x:
                 if i['job'] == 'Director':
                     return i['name']
             return np.nan
```

```python
In [35]: smd['director'] = smd['crew'].apply(get_director)
```

```python
In [38]: smd['cast'] = smd['cast'].apply(lambda x: [i['name'] for i in x] if isinstance(x, list) else [])
         smd['cast'] = smd['cast'].apply(lambda x: x[:3] if len(x) >=3 else x)
```

```python
In [41]: smd['keywords'] = smd['keywords'].apply(lambda x: [i['name'] for i in x] if isinstance(x, list) else [])
```

```python
In [44]: smd['cast'] = smd['cast'].apply(lambda x: [str.lower(i.replace(" ", "")) for i in x])
```

```python
In [47]: smd['director'] = smd['director'].astype('str').apply(lambda x: str.lower(x.replace(" ", "")))
         smd['director'] = smd['director'].apply(lambda x: [x,x, x])
```

Our approach is to create a metadata dump for every movie which consists of genres, director, main actors and keywords. We then use a Count vectorizer to create our count matrix as we did in the Description Recommender. The remaining steps are similar to what we did earlier: we calculate the cosine similarities and return movies that are most similar.

These are the steps we follow in the preparation of genres and credits data:

1. Strip Spaces and Convert to Lowercase from all our features. This way, our engine will not confuse between Johnny Depp and Johnny Galecki.

2. Mention Director 3 times to give it more weight relative to the entire cast.

```
Out[109]: 0                [johnlasseter, johnlasseter, johnlasseter]
          1                   [joejohnston, joejohnston, joejohnston]
          2                [howarddeutch, howarddeutch, howarddeutch]
          3          [forestwhitaker, forestwhitaker, forestwhitaker]
          4                [charlesshyer, charlesshyer, charlesshyer]
                                          ...
          40952         [greggchampion, greggchampion, greggchampion]
          41172    [tinusureshdesai, tinusureshdesai, tinusureshd...
          41225    [ashutoshgowariker, ashutoshgowariker, ashutos...
          41391               [hideakianno, hideakianno, hideakianno]
          41669                     [ronhoward, ronhoward, ronhoward]
          Name: director, Length: 9219, dtype: object
```

```
In [49]: smd
```

Out[49]:

	adult	belongs_to_collection	budget	genres	homepage	id	imdb_id	original_language	original_title	overview	...
0	False	{'id': 10194, 'name': 'Toy Story Collection', ...	30000000	[Animation, Comedy, Family]	http://toystory.disney.com/toy-story	862	tt0114709	en	Toy Story	Led by Woody, Andy's toys live happily in his ...	...
1	False	NaN	65000000	[Adventure, Fantasy, Family]	NaN	8844	tt0113497	en	Jumanji	When siblings Judy and Peter discover an encha...	...
2	False	{'id': 119050, 'name': 'Grumpy Old Men Collect...	0	[Romance, Comedy]	NaN	15602	tt0113228	en	Grumpier Old Men	A family wedding reignites the ancient feud be...	...
3	False	NaN	16000000	[Comedy, Drama, Romance]	NaN	31357	tt0114885	en	Waiting to Exhale	Cheated on, mistreated and stepped on, the wom...	...
4	False	{'id': 96871, 'name': 'Father of the Bride Col...	0	[Comedy]	NaN	11862	tt0113041	en	Father of the Bride Part II	Just when George Banks has recovered from his ...	...

Pre-processed dataset

2.Stemming and filtering of keywords

Input: Pre-processed dataset

Output: Stemmed and filtered keywords

Code Snippet:

```
In [50]:
s = smd.apply(lambda x: pd.Series(x['keywords']),axis=1).stack().reset_index(level=1, drop=True)
s.name = 'keyword'
```

```
In [57]:
def filter_keywords(x):
    words = []
    for i in x:
        if i in s:
            words.append(i)
    return words
```

```
In [58]:
stemmer = SnowballStemmer('english')
smd['keywords'] = smd['keywords'].apply(filter_keywords)
smd['keywords'] = smd['keywords'].apply(lambda x: [stemmer.stem(i) for i in x])
smd['keywords'] = smd['keywords'].apply(lambda x: [str.lower(i.replace(" ", "")) for i in x])
```

Keywords occur in frequencies ranging from 1 to 610. We do not have any use for keywords that occur only once. Therefore, these can be safely removed. Finally, we will convert every word to its stem so that words such as *Dogs* and *Dog* are considered the same. Then we make a filter for words and apply the count and stemmers to all the words and join them as shown below:

```
Out[121]: 0        jealousi toy boy friendship friend rivalri boy...
          1        boardgam disappear basedonchildren'sbook newho...
          2        fish bestfriend duringcreditssting waltermatth...
          3        basedonnovel interracialrelationship singlemot...
          4        babi midlifecrisi confid age daughter motherda...
                                         ...
          40952    friendship sidneypoitier wendycrewson jayo.san...
          41172    bollywood akshaykumar ileanad'cruz eshagupta t...
          41225    bollywood hrithikroshan poojahegde kabirbedi a...
          41391    monster godzilla giantmonst destruct kaiju hir...
          41669    music documentari paulmccartney ringostarr joh...
          Name: soup, Length: 9219, dtype: object
```

3.Find Similarity Matrix

Input: Stemmed and filtered keywords

Output: Similarity Matrix

Code Snippet:

We apply the count vectorization and apply the cosine similarity function to detect the word similarities. We apply the below mentioned function to get the cosine similarity.

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

```
Out[128]: array([[1.        , 0.02441931, 0.02738955, ..., 0.        , 0.        ,
                 0.        ],
                [0.02441931, 1.        , 0.        , ..., 0.02973505, 0.02500782,
                 0.        ],
                [0.02738955, 0.        , 1.        , ..., 0.03335187, 0.        ,
                 0.        ],
                ...,
                [0.        , 0.02973505, 0.03335187, ..., 1.        , 0.08700222,
                 0.        ],
                [0.        , 0.02500782, 0.        , ..., 0.08700222, 1.        ,
                 0.        ],
                [0.        , 0.        , 0.        , ..., 0.        , 0.        ,
                 1.        ]])
```

Similarity Matrix

4.Compute recommendations

Input: Movie title

Output: Movie recommendations

Code Snippet:

```
In [85]:
def get_recommendations(title):
    idx = indices[title]
    sim_scores = list(enumerate(cosine_sim[idx]))

    sim_scores = sorted(sim_scores, key=lambda x: x[1], reverse=True)
    print("sc",sim_scores)
    sim_scores = sim_scores[1:31]
    print("sc",sim_scores)

    sum_of_scores = 0
    for elem in sim_scores[:10]:
        print(elem)
        sum_of_scores += elem[1]
    print((sum_of_scores/10)*100)
    print("max correlation value = ",sim_scores[0][1])
    #acc_score = (sum(map(float,sim_scores[1:31])))
    movie_indices = [i[0] for i in sim_scores]
    return titles.iloc[movie_indices]
```

Now we modify the title as the index and also use the content based recommender based on the cosine values to get the recommendation. We see the top 10 recommendations and their similarity value for the given movie.

We can see the recommendations along with their correlation value below.

```
In [71]:  get_recommendations('Mean Girls').head(10)

(3319, 0.4622501635210242)
(4763, 0.38865016537877745)
(1329, 0.3846153846153846)
(6277, 0.3846153846153846)
(7905, 0.37511724246026645)
(7332, 0.36854868854485945)
(6959, 0.3103164454170876)
(8883, 0.15384615384615383)
(6698, 0.12955005512625914)
(7377, 0.12503908082008883)
30.82548764345286
max correlation value =  0.4622501635210242

Out[71]: 3319                Head Over Heels
4763                  Freaky Friday
1329               The House of Yes
6277               Just Like Heaven
7905           Mr. Popper's Penguins
7332      Ghosts of Girlfriends Past
6959       The Spiderwick Chronicles
8883                        The DUFF
6698           It's a Boy Girl Thing
7377         I Love You, Beth Cooper
Name: title, dtype: object
```

Hybrid Recommender

One thing that we notice about our recommendation system is that it recommends movies regardless of ratings and popularity.

We will add a mechanism to remove bad movies and return movies which are popular and have had a good critical response.

We will take the top 25 movies based on similarity scores and calculate the vote of the 60th percentile movie.

Then, using this as the value of m, we will calculate the weighted rating of each movie using IMDB's formula like we did in the Simple Recommender section.

We will combine the simple and meta data based recommender to get this new recommender.

Input: Movie title

Output: Movie recommendations

Code Snippet:

In [72]:

```
def improved_recommendations(title):
    idx = indices[title]
    sim_scores = list(enumerate(cosine_sim[idx]))
    sim_scores = sorted(sim_scores, key=lambda x: x[1], reverse=True)
    sim_scores = sim_scores[1:26]
    sum_of_scores = 0
    for elem in sim_scores:
        print(elem)
        sum_of_scores += elem[1]
    print((sum_of_scores/25)*100)
    print("max correlation value = ",sim_scores[0][1])

    movie_indices = [i[0] for i in sim_scores]

    movies = smd.iloc[movie_indices][['title', 'vote_count', 'vote_average', 'year']]
    vote_counts = movies[movies['vote_count'].notnull()]['vote_count'].astype('int')
    vote_averages = movies[movies['vote_average'].notnull()]['vote_average'].astype('int')
    C = vote_averages.mean()
    m = vote_counts.quantile(0.60)
    qualified = movies[(movies['vote_count'] >= m) & (movies['vote_count'].notnull()) & (movies['vote_average'].notnull())]
    qualified['vote_count'] = qualified['vote_count'].astype('int')
    qualified['vote_average'] = qualified['vote_average'].astype('int')
    qualified['wr'] = qualified.apply(weighted_rating, axis=1)
    qualified = qualified.sort_values('wr', ascending=False).head(10)
    return qualified
```

In [74]:

```
improved_recommendations('Mean Girls')
```

```
(3319, 0.4622501635210242)
(4763, 0.38865016537877745)
(1329, 0.3846153846153846)
(6277, 0.3846153846153846)
(7905, 0.37511724246026645)
(7332, 0.368548688544485945)
(6959, 0.3103164454170876)
(8883, 0.153846153846153846153383)
(6698, 0.12955005512625914)
(7377, 0.12503908082008883)
(3712, 0.12209657262637608)
(7494, 0.11149893466761211)
(5542, 0.10826639239215334)
(5163, 0.10529962529853128)
(5092, 0.09245003270420485)
(1547, 0.08920515501750789)
(2005, 0.0863667034175061)
(8844, 0.08627959628145762)
(5152, 0.08362420100070908)
(7084, 0.07897471897389846)
(7436, 0.07897471897389846)
(7688, 0.07897471897389846)
(4996, 0.07692307692307693)
(6449, 0.07692307692307693)
(390, 0.07502344849205331)
17.733718948044988
max correlation value =  0.4622501635210242
```

Out[74]:

	title	vote_count	vote_average	year	wr
1547	The Breakfast Club	2189	7	1985	6.709602
390	Dazed and Confused	588	7	1993	6.254682
8883	The DUFF	1372	6	2015	5.818541
3712	The Princess Diaries	1063	6	2001	5.781086
4763	Freaky Friday	919	6	2003	5.757786
6277	Just Like Heaven	595	6	2005	5.681521
6959	The Spiderwick Chronicles	593	6	2008	5.680901
7494	American Pie Presents: The Book of Love	454	5	2009	5.119690
7332	Ghosts of Girlfriends Past	716	5	2009	5.092422
7905	Mr. Popper's Penguins	775	5	2011	5.087912

Hybrid movie recommendations