Project Report entitled

"Movie Recommendation System"

Under subject

Big Data(COEN 242)

Submitted by

1	Astha Gupta	00001606877
2	Ruju Kiritkumar Patel	00001606924
3	Sneh Desai	00001616217

Guided by:

Prof. Zhiqiang Tao

TABLE OF CONTENTS

INTRODUCTION

Recommendation Systems are among the most important applications in the field of big data today. There are a number of areas where the recommendation Systems are used, such as online stores, music services or videos, books, furniture etc. Often, these systems are able to collect information about a user's choices, and can use this information to improve their suggestions in the future.

Based on the research we did in the past few weeks and the common interest shown by the team, we decided to move forward with the movie recommendation system. It is very difficult to come to a conclusion and decide upon a movie which would be appreciated and liked, whenever we plan to watch a movie.

The goal thus for the project is to build the recommendation system that would suggest movies to individuals with the highest accuracy that we could achieve based on our algorithms.

We will attempt at implementing a few recommendation algorithms (content based, popularity based) and try to build an ensemble of these models to come up with our final recommendation system.

This project specifically focuses only on **movie recommendation system**.
To reduce the efforts of people we can recommend them movies based on the criteria discussed in the next slide.

We will categorize the searches based on 2 algorithms:

1. Simple Recommendation System: The simple algorithm used in the project will help us to identify the movies with **highest ratings** and the ones which are **most popular**. This would result in a generic result for every user.
2. Content-based recommendation: It helps us to search the movie based on **Search history** of the user.

Dataset

In this we will be using a dataset that we found on web which consists of 26,000,000 ratings and 750,000 tag applications applied to 45,000 movies by 270,000 users. Includes tag genome data with 12 million relevance scores across 1,100 tags. And it also has many files in it and we will be using a few of the files as our historical or training and testing set.

adult	belongs_t	budget	genres	homepage	id	imdb_id	original_la	original_ti	overview	popularity
FALSE	{'id': 1019	30000000	[{'id': 16, '	http://toy	862	tt011470	en	Toy Story	Led by Wo	21.9469
FALSE		65000000	[{'id': 12, 'name': 'Ac		8844	tt011349	en	Jumanji	When sibl	17.0155
FALSE	{'id': 1190	0	[{'id': 10749, 'name'		15602	tt011322	en	Grumpier	A family w	11.7129
FALSE		16000000	[{'id': 35, 'name': 'Co		31357	tt011488	en	Waiting to	Cheated o	3.8595
FALSE	{'id': 9687	0	[{'id': 35, 'name': 'Co		11862	tt011304	en	Father of	Just when	8.38752
FALSE		60000000	[{'id': 28, 'name': 'Ac		949	tt011327	en	Heat	Obsessive	17.9249
FALSE		58000000	[{'id': 35, 'name': 'Co		11860	tt011431	en	Sabrina	An ugly du	6.67728
FALSE		0	[{'id': 28, 'name': 'Ac		45325	tt011230	en	Tom and I	A mischie	2.56116
FALSE		35000000	[{'id': 28, 'name': 'Ac		9091	tt011457	en	Sudden D	Internatio	5.23158
FALSE	{'id': 645,	58000000	[{'id': 12, '	http://ww	710	tt011318	en	GoldenEy	James Bor	14.686
FALSE		62000000	[{'id': 35, 'name': 'Co		9087	tt011234	en	The Amer	Widowed	6.31845
FALSE		0	[{'id': 35, 'name': 'Co		12110	tt011289	en	Dracula: D	When a la	5.43033
FALSE	{'id': 1176	0	[{'id': 10751, 'name'		21032	tt011245	en	Balto	An outcas	12.1407
FALSE		44000000	[{'id': 36, 'name': 'Hi		10858	tt011398	en	Nixon	An all-star	5.092
FALSE		98000000	[{'id': 28, 'name': 'Ac		1408	tt011276	en	Cutthroat	Morgan A	7.28448
FALSE		52000000	[{'id': 18, 'name': 'Dr		524	tt011264	en	Casino	The life of	10.1374
FALSE		16500000	[{'id': 18, 'name': 'Dr		4584	tt011438	en	Sense and	Rich Mr. D	10.6732

Libraries Used

There are some inbuilt libraries that we will be using to accomplish or project.

```
1. import pandas as pd
2. from tabulate import tabulate
3. import numpy as np
4. import sys
5. from ast import literal_eval
6. from sklearn.feature_extraction.text import TfidfVectorizer, CountVectorizer
7. from sklearn.metrics.pairwise import linear_kernel, cosine_similarity
8. from nltk.stem.snowball import SnowballStemmer
9. import warnings
10. from datetime import date
```

Here we have used

1. pandas for dataframe.
2. Tabulate for pretty output.
3. Numpy to do some operations
4. sklearn for matrix multiplication of TFIDF, cosine similarities etc.

Simple Recommendation System

The Simple Recommender offers generalized recommendations to every user based on movie popularity and (sometimes) genre. The basic idea behind this recommender is that movies that are more popular and more critically acclaimed will have a higher probability of being liked by the average audience. This model does not give personalized recommendations based on the user.

The implementation of this model is extremely trivial. All we have to do is sort our movies based on ratings and popularity and display the top movies of our list. As an added step, we can pass in a genre argument to get the top movies of a particular genre.

```
1.  # Read the .csv file to a variable
2.  full_data = pd.read_csv('movies_metadata.csv')
3.
4.  # View all the column in the output
5.  pd.set_option('display.max_columns', None)
6.
7.  # Data Cleaning step
8.  full_data['genres'] = full_data['genres'].fillna('[]').apply(literal_eval).apply(lambda x: [i['name'] for i in x] if isinstance(x, list)
    else [])
```

The above code will read all the data from the '.csv' and store all the value in the 'full_data' pandas Dataframe variable.

We used the TMDB Ratings to come up with our **Top Movies Chart.** We used IMDB's *weighted rating* formula to construct my chart. Mathematically, it is represented as follows:

Weighted Rating (WR) = ((v/v+m).R)+((m/v+m).C) 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 used **95th percentile** as our cut-off. 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.

We build our overall Top 250 Chart and will define a function to build charts for a particular genre.

```
1.  # Variable declared for vote count and average removing the null values
2.  vote_counts = full_data[full_data['vote_count'].notnull()]['vote_count'].astype('int')
3.  vote_averages = full_data[full_data['vote_average'].notnull()]['vote_average'].astype('int')
4.  C = vote_averages.mean()
5.  print(C)
```

```
5.244896612406511
```

```
m = vote_counts.quantile(0.95)
print("This is the 95th Percentile Limit of vote Count" + str(m))
```

```
434.0
```

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

# All the columns that we'll be showing in the output.
qualified = full_data[(full_data['vote_count'] >= m) & (full_data['vote_count'].notnull()) & (full_data['vote_average'].notnull())][['title', 'year', 'vote_count', 'vote_average', 'popularity', 'genres']]
qualified['vote_count'] = qualified['vote_count'].astype('int')
qualified['vote_average'] = qualified['vote_average'].astype('int')
print(str(qualified.shape))
```

```
(2274, 6)
```

```
# The weighted rating Function
def weighted_rating(x):
    # Calculates the rating base on 'vote count' and 'vote average'.
    v = x['vote_count']
    R = x['vote_average']
    return (v/(v+m) * R) + (m/(m+v) * C)
# Function Ends
```

```
qualified['weight_ratio'] = qualified.apply(weighted_rating, axis=1)

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

print(qualified.head(10))
```

title	year	vote_count	vote_average	popularity	genres	wr
Inception	2010	14075	8	29.1081	[Action, Thriller, Science Fiction, Mystery, A...	7.917588
The Dark Knight	2008	12269	8	123.167	[Drama, Action, Crime, Thriller]	7.905871
Interstellar	2014	11187	8	32.2135	[Adventure, Drama, Science Fiction]	7.897107
Fight Club	1999	9678	8	63.8696	[Drama]	7.881753
The Lord of the Rings: The Fellowship of the Ring	2001	8892	8	32.0707	[Adventure, Fantasy, Action]	7.871787
Pulp Fiction	1994	8670	8	140.95	[Thriller, Crime]	7.868660
The Shawshank Redemption	1994	8358	8	51.6454	[Drama, Crime]	7.864000
The Lord of the Rings: The Return of the King	2003	8226	8	29.3244	[Adventure, Fantasy, Action]	7.861927
Forrest Gump	1994	8147	8	48.3072	[Comedy, Drama, Romance]	7.860656
The Lord of the Rings: The Two Towers	2002	7641	8	29.4235	[Adventure, Fantasy, Action]	7.851924

We see that three Christopher Nolan Films, **Inception**, **The Dark Knight** and **Interstellar** occur at the very top of our chart. The chart also indicates a strong bias of TMDB Users towards particular genres and directors.

For this, we used default conditions to the **85th** percentile instead of 95.

```
s = full_data.apply(lambda x: pd.Series(x['genres']),axis=1).stack().reset_index(level=1, drop=True)
s.name = 'genre'
gen_md = full_data.drop('genres', axis=1).join(s)


# Function for particular Genre
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', 'vote_count', 'vote_average', 'popularity']]
    qualified['vote_count'] = qualified['vote_count'].astype('int')
    qualified['vote_average'] = qualified['vote_average'].astype('int')

    qualified['weight_ratio'] = qualified.apply(
        lambda x: (x['vote_count'] / (x['vote_count'] + m) * x['vote_average']) + (m / (m + x['vote_count']) * C),
        axis=1)
    qualified = qualified.sort_values('weight_ratio', ascending=False).head(250)

    return qualified
# Function Ends
```

It would only give all the thriller and omit out other genre movies.

```
1. print(tabulate(build_chart('Thriller').head(10),headers='keys',tablefmt='psql'))
```

The Top thriller is Inception according to the list.

```
+-------+--------------------------+--------+--------------+----------------+--------------+----------------+
|       | title                    |   year |   vote_count |   vote_average |   popularity |   weight_ratio |
|-------+--------------------------+--------+--------------+----------------+--------------+----------------|
| 15480 | Inception                |   2010 |        14075 |              8 |     29.1081  |        7.95646 |
| 12481 | The Dark Knight          |   2008 |        12269 |              8 |    123.167   |        7.95016 |
|   292 | Pulp Fiction             |   1994 |         8670 |              8 |    140.95    |        7.93    |
|    46 | Se7en                    |   1995 |         5915 |              8 |     18.4574  |        7.89857 |
| 24860 | The Imitation Game       |   2014 |         5895 |              8 |     31.5959  |        7.89824 |
|   586 | The Silence of the Lambs |   1991 |         4549 |              8 |      4.30722 |        7.86954 |
| 11354 | The Prestige             |   2006 |         4510 |              8 |     16.9456  |        7.86846 |
|   289 | Leon: The Professional   |   1994 |         4293 |              8 |     20.4773  |        7.86214 |
|  4099 | Memento                  |   2000 |         4168 |              8 |     15.4508  |        7.85822 |
|  1213 | The Shining              |   1980 |         3890 |              8 |     19.6116  |        7.84863 |
+-------+--------------------------+--------+--------------+----------------+--------------+----------------+
(9099, 25)
(9099, 268124)
```

Content-Based Recommendation System

The recommender we built in the previous section suffers some severe limitations. For one, it gives the same recommendation to everyone, regardless of the user's personal taste. If a person who loves romantic movies (and hates action) were to look at our Top 10 Chart, s/he wouldn't probably like most of the movies. If s/he were to go one step further and look at our charts by genre, s/he wouldn't still be getting the best recommendations.

For instance, consider a person who loves *Dilwale Dulhania Le Jayenge*, *My Name is Khan* and *Kabhi Khushi Kabhi Gham*. One inference we can obtain is that the person loves the actor Shahrukh Khan and the director Karan Johar. Even if s/he were to access the romance chart, s/he wouldn't find these as the top recommendations.

To personalise our recommendations more, we build an engine that computes similarity between movies based on certain metrics and suggests movies that are most similar to a particular movie that a user liked. Since we will be using movie metadata (or content) to build this engine, this also known as **Content Based Filtering.**

We build two Content Based Recommenders based on:

- Movie Overviews and Taglines
- Movie Cast, Crew, Keywords and Genre

Also, as mentioned in the introduction, we have used a subset of all the movies available to us due to limiting computing power available to us.

```
1.  links_small = pd.read_csv('links_small.csv')
2.  links_small = links_small[links_small['tmdbId'].notnull()]['tmdbId'].astype('int')
3.  full_data = full_data.drop([19730, 29503, 35587])
4.
5.  full_data['id'] = full_data['id'].astype('int')
6.
7.  smd = full_data[full_data['id'].isin(links_small)]
8.  print(smd.shape)
```

```
(9099, 25)
```

We have **9099** movies available in our small movies metadata dataset which is 5 times smaller than our original dataset of 45000 movies.

Movie Description Based Recommendation

We first tried to build a recommender using movie descriptions and taglines. We do not have a quantitative metric to judge our machine's performance so this will have to be done qualitatively.

```
1.  smd['tagline'] = smd['tagline'].fillna('')
2.  smd['description'] = smd['overview'] + smd['tagline']
3.  smd['description'] = smd['description'].fillna('')
4.
5.  tf = TfidfVectorizer(analyzer='word',ngram_range=(1, 2),min_df=0, stop_words='english')
6.  tfidf_matrix = tf.fit_transform(smd['description'])
7.
8.  print(tfidf_matrix.shape)
```

```
(9099, 268124)
```

Cosine Similarity

Cosine similarity is a method to measure the difference between two non zero vectors of an inner product space.

We have used the Cosine Similarity to calculate a numeric quantity that denotes the similarity between two movies. Mathematically, it is defined as follows:

cosine(x,y) = $(x.y^T)$ / (||x||.||y||)

Theoretically, the cosine similarity can be any number between -1 and +1 because of the image of the cosine function, but in this case, there will not be any negative movie rating so the angle θ will be between 0º and 90º bounding the cosine similarity between 0 and 1. If the angle θ = 0º =>cosine similarity = 1, if θ = 90º => cosine similarity =0.

Here Cosine similarity is found using the Matrix Multiplication.

Since we have used the TF-IDF Vectorizer, calculating the Dot Product will directly give us the Cosine Similarity Score. Therefore, we will use sklearn's **linear_kernel** instead of cosine_similarities since it is much faster.

```
cosine_sim = linear_kernel(tfidf_matrix, tfidf_matrix)

smd = smd.reset_index()
titles = smd['title']
indices = pd.Series(smd.index, index=smd['title'])
```

```
# Function Get Recommendation
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)
    sim_scores = sim_scores[1:31]
    movie_indices = [i[0] for i in sim_scores]
    return titles.iloc[movie_indices]
# Function Ends
```

```
print(tabulate(get_recommendations('The Dark Knight').head(10), headers='keys', tablefmt='psql'))
```

We see that for **The Dark Knight**, our system is able to identify it as a Batman film and subsequently recommend other Batman films as its top recommendations. But unfortunately, that is all this system can do at the moment. This is not of much use to most people as it doesn't take into considerations very important features such as cast, crew, director and genre, which determine the rating and the popularity of a movie. Someone who liked **The Dark Knight** probably likes it more because of Nolan and would hate **Batman Forever** and every other substandard movie in the Batman Franchise.

Therefore, we used much more suggestive metadata than **Overview** and **Tagline**.

Metadata Based Recommendation System:

To build our standard metadata-based content recommender, we had to merge our current dataset with the crew and the keyword datasets.

```
credits = pd.read_csv('credits.csv')
keywords = pd.read_csv('keywords.csv')

keywords['id'] = keywords['id'].astype('int')
credits['id'] = credits['id'].astype('int')
full_data['id'] = full_data['id'].astype('int')

print(full_data.shape)
```

```
(45463, 25)
```

```
full_data = full_data.merge(credits, on='id')
full_data = full_data.merge(keywords, on='id')

smd = full_data[full_data['id'].isin(links_small)]
print(smd.shape)
```

```
(9219, 28)
```

- **Crew:** From the crew, we only picked the director as our feature since the others don't contribute that much to the *feel* of the movie.
- **Cast:** Choosing Cast is a little trickier. Lesser-known actors and minor roles do not really affect people's opinion of a movie. Therefore, we only selected the major characters and their respective actors. Arbitrarily we choose the top 3 actors that appear in the credits list.

```
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))
```

```
# Function to get Director
def get_director(x):
    for i in x:
        if i['job'] == 'Director':
            return i['name']
    return np.nan
# Function Ends
```

```
smd['director'] = smd['crew'].apply(get_director)
```

```
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)
```

```
3.
4.  smd['keywords'] = smd['keywords'].apply(lambda x: [i['name'] for i in x] if isinstance(x, list) else [])
```

Our approach to build the recommender was to be extremely *hacky*. What we planned was we created a metadata dump for every movie which consists of **genres, director, main actors and keywords.** Then use a **Count Vectorizer** to create 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 steps which we followed 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.

```
1.  smd['cast'] = smd['cast'].apply(lambda x: [str.lower(i.replace(" ", "")) for i in x])
2.
3.  smd['director'] = smd['director'].astype('str').apply(lambda x: str.lower(x.replace(" ", "")))
4.  smd['director'] = smd['director'].apply(lambda x: [x,x, x])
```

We also did a small amount of pre-processing of our keywords before putting them to any use. As a first step, we calculate the frequent counts of every keyword that appears in the dataset.

```
1.  s = smd.apply(lambda x: pd.Series(x['keywords']),axis=1).stack().reset_index(level=1, drop=True)
2.  s.name = 'keyword'
3.
4.  s = s.value_counts()
5.  print(s[:5])
```

```
independent film        610
woman director          550
murder                  399
duringcreditsstinger    327
based on novel          318
Name: keyword, dtype: int64
```

Keywords occur in frequencies ranging from 1 to 610. We do not have any use for keywords that occur only once. Therefore, we removed them. Finally, we converted every word to its stem so that words such as *Dogs* and *Dog* are considered the same.

```
1.  s = s[s > 1]
2.
3.  stemmer = SnowballStemmer('english')
```

```
# Function to Filter
def filter_keywords(x):
    words = []
    for i in x:
        if i in s:
            words.append(i)
    return words
# Function Ends
```

```
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])

smd['soup'] = smd['keywords'] + smd['cast'] + smd['director'] + smd['genres']
smd['soup'] = smd['soup'].apply(lambda x: ' '.join(x))

count = CountVectorizer(analyzer='word',ngram_range=(1, 2),min_df=0, stop_words='english')
count_matrix = count.fit_transform(smd['soup'])

cosine_sim = cosine_similarity(count_matrix, count_matrix)

smd = smd.reset_index()
titles = smd['title']
indices = pd.Series(smd.index, index=smd['title'])
```

We reused the get_recommendations function that we had written earlier. Since our cosine similarity scores have changed, we expect it to give us different (and probably better) results.

```
print(get_recommendations('The Dark Knight').head(10))
```

```
7931                       The Dark Knight Rises
132                              Batman Forever
1113                             Batman Returns
8227    Batman: The Dark Knight Returns, Part 2
7565                 Batman: Under the Red Hood
524                                      Batman
7901                          Batman: Year One
2579               Batman: Mask of the Phantasm
2696                                        JFK
8165    Batman: The Dark Knight Returns, Part 1
Name: title, dtype: object
```

Popularity and Ratings

One thing about our recommendation system is that it recommends movies regardless of ratings and popularity. It is true that **Batman and Robin** has a lot of similar characters as compared to **The Dark Knight** but it was a terrible movie that shouldn't be recommended to anyone.

Therefore, we added a mechanism to remove bad movies and return movies which are popular and have had a good critical response.

We took 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 calculated the weighted rating of each movie using IMDB's formula like we did in the Simple Recommender section.

```
# Function For a better Recommendation
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]
    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
# Function End


print(tabulate(improved_recommendations('The Dark Knight'), headers='keys', tablefmt='psql'))
```

title	vote_count	vote_average	year	wr
Inception	14075	8	2010	7.917588
Interstellar	11187	8	2014	7.897107
The Prestige	4510	8	2006	7.758148
Memento	4168	8	2000	7.740175
The Dark Knight Rises	9263	7	2012	6.921448
Batman Begins	7511	7	2005	6.904127
Batman Returns	1706	6	1992	5.846862
Batman Forever	1529	5	1995	5.054144
Batman v Superman: Dawn of Justice	7189	5	2016	5.013943
Batman & Robin	1447	4	1997	4.287233

Essential Questions:

1. Use of parallelization in project?

   In data parallelism we partition the big dataset into a number of small datasets among multiple processing unit. Each node operates on the assigned chunk of data.

   We have used a function to optimize a parallel code for computation of a similarity measure between every sample set of a large number of high-dimensional vectors such that the code scales up the performance and executes the program faster, since the multiple chunks of data are running in parallel.

2. Matrix multiplication?

   We have used matrix for tfidf vectorization. We have computed the size of matrix using .shape() function and performed vectorization on that. We have compared the 2 matrices.

   Matrix multiplication is used to find the cosine similarity between these 2 matrix. All these operations have been performed in parallel to improve the efficiency of our code and reduce the running time of program.

   We have used the dot product to directly get the cosine similarity score between 2 samples.

3. What is cosine Similarity?

   Cosine similarity is a metric used to measure how the common data in 2 sample sets irrespective of their size. Cosine similarity measures the cosine of the angle between two vectors projected in a multi-dimensional space.

   It computes the values in parallel. We use cosine similarity only for non-null dimensions. When we call the function, it will output the overlapping items of the given conditions.

   $\text{cosine}(x,y) = (x.y^T) / (||x||.||y||)$

Conclusion

Two different recommendation engines based on different ideas and algorithms are generated. These are as follows:

Simple Recommender: This system used overall TMDB Vote Count and Vote Averages to build Top Movies Charts, in general and for a specific genre. The IMDB Weighted Rating System was used to calculate ratings on which the sorting was finally performed.

Content Based Recommender: We built two content-based engines; one that took movie overview and taglines as input and the other which took metadata such as cast, crew, genre and keywords to come up with predictions. We also devised a simple filter to give greater preference to movies with more votes and higher ratings.