**IMPORT DATASETS**

#import datasets

credits = pd.read\_csv("C:\Users\SHUBHAM TOTLA\Desktop\Data analytics\movie dataset\credits.csv",engine='python')

keywords=pd.read\_csv("C:\Users\SHUBHAM TOTLA\Desktop\Data analytics\movie dataset\keywords.csv",engine='python')

metadata=pd.read\_csv("C:\Users\SHUBHAM TOTLA\Desktop\Data analytics\movie dataset\movies\_metadata.csv",engine='python')

**RECOMMEND MOVIES BASED ON THEIR PLOTS.**

The plot description is given in the overview feature in the metadata dataset. Let’s check out few plots of the movies.

print(metadata['overview'].shape)

The data is too large for the processor to process it. So we create a new dataset in which we eliminate the movies that are less than 90 percentile of the vote count

That is 90 percent of the films in the original metadata dataset have received less than 160 votes. In comparison, the film with the most votes received more than 14,000 votes and many others have vote counts in the thousands. So the 160 vote threshold is quite low for our purposes. Meanwhile, in the original dataframe, some films literally have zero votes and half of the films received less than 10 votes.

This makes our dataset small enough to be processed

Next, let's calculate the number of votes, *m*, received by a movie in the 90th percentile. The pandas library makes this task extremely trivial using the .quantile() method of a pandas

# Calculate the minimum number of votes required to be in the chart, m

m = metadata['vote\_count'].quantile(0.90)

print(m)

Filter the movies that qualify for the chart, based on their vote counts:

# Filter out all qualified movies into a new DataFrame

q\_movies = metadata.copy().loc[metadata['vote\_count'] >= m]

We use the .copy() method to ensure that the new q\_movies DataFrame created is independent of your original metadata DataFrame. In other words, any changes made to the q\_movies DataFrame does not affect the metadata

q\_movies.shape

#reset index

q\_movies = q\_movies.reset\_index(drop = True)

q\_movies.index

Now it is small enough to get processed

print(q\_movies['overview'].head(10))

In its current form, it is not possible to compute the similarity between any two overviews. To do this, you need to compute the word vectors of each overview or document, as it will be called from now on.

For this we’ll use TFIDF score which is a numerical statistic that is intended to reflect how important a word is to a [document](https://en.wikipedia.org/wiki/Document) in a collection or [corpus](https://en.wikipedia.org/wiki/Text_corpus).

Computing TFIDF for overview will give you a matrix where each column represents a word in the overview vocabulary (all the words that appear in at least one document) and each column represents a movie, as before.

from sklearn.feature\_extraction.text import TfidfVectorizer

#Define a TF-IDF Vectorizer Object. Remove all english stop words such as 'the', 'a'

tfidf = TfidfVectorizer(stop\_words='english')

#Replace NaN with an empty string

q\_movies['overview'] = q\_movies['overview'].fillna('')

#Construct the required TF-IDF matrix by fitting and transforming the data

tfidf\_matrix = tfidf.fit\_transform(q\_movies['overview'])

#Output the shape of tfidf\_matrix

print(tfidf\_matrix.shape)

You see that over 19,695 different words were used to describe the 4,555 movies in your dataset.

With this matrix in hand, we can now compute a similarity score.

We will be using the cosine similarity to calculate a numeric quantity that denotes the similarity between two movies. We use the cosine similarity score since it is independent of magnitude and is relatively easy and fast to calculate (especially when used in conjunction with TF-IDF scores, which will be explained later). Mathematically, it is defined as follows:

Since we have used the TF-IDF vectorizer, calculating the dot product will directly give you the cosine similarity score. Therefore, we will use sklearn's linear\_kernel() instead of cosine\_similarities() since it is faster.

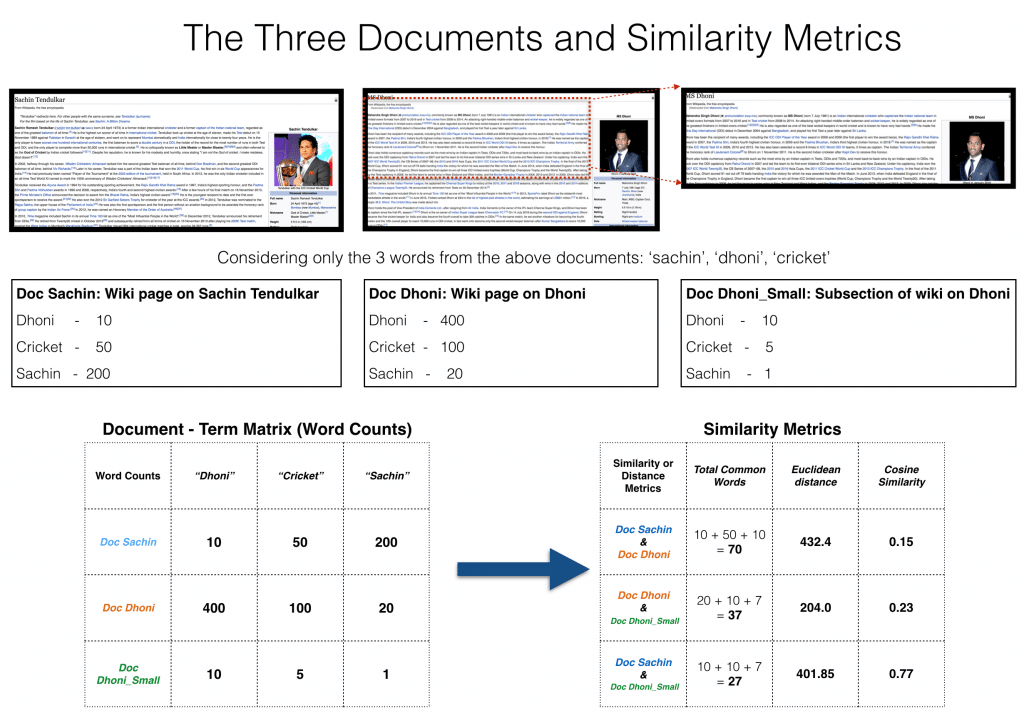
# Import linear\_kernel

from sklearn.metrics.pairwise import linear\_kernel

# Compute the cosine similarity matrix

cosine\_sim = linear\_kernel(tfidf\_matrix, tfidf\_matrix)

**COSINE SIMILARITY**



The cosine\_sim matrix contains columns of all the words and rows of all the documents

Now, we are going to define a function that takes movie title as input and print a list of 10 most similar movies

For this we need to know the index of the movie in the metadata dataset given its title

#Construct a reverse map of indices and movie titles

indices = pd.Series(q\_movies.index,index=q\_movies['title']).drop\_duplicates()

//We create a series data structure that contains indices of the movies in the q\_movies dataset with index redefined as movie title

**Title Index**

Toy Story 0

Jumanji 1

Heat 2

GoldenEye 3

Balto 4

Casino 5

Sense and Sensibility 6

Four Rooms 7

Ace Ventura: When Nature Calls 8

Assassins 9

Leaving Las Vegas 10

Twelve Monkeys 11

Babe 12

Dead Man Walking 13

Clueless 14

Mortal Kombat 15

Se7en 16

Pocahontas 17

The Usual Suspects 18

Friday 19

From Dusk Till Dawn 20

Broken Arrow 21

La Haine 22

Happy Gilmore 23

The Bridges of Madison County 24

Braveheart 25

Taxi Driver 26

Bad Boys 27

The Basketball Diaries 28

Apollo 13 29

...

Fifty Shades Darker 2708

To the Bone 2709

Fallen 2710

Life 271

mjp= Series([5,4,3,2,1])*# a simple series*

print mjp.values *# similar to dictionary. ".values" command returns values in a series*

0 5

1 4

2 3

3 2

4 1

print mjp.index *# returns the index values of the series*

([0, 1, 2, 3, 4], dtype='int64')

jeeva = Series([5,4,3,2,1,-7,-29], index =['a','b','c','d','e','f','h']) *# The index is specified*

print jeeva *# try jeeva.index and jeeva.values*\

a 5

b 4

c 3

d 2

e 1

f -7

h -29

To define your recommendation function these are the following steps you'll follow:

* Get the index of the movie given its title.
* Get the list of cosine similarity scores for that particular movie with all movies. Convert it into a list of tuples where the first element is its position and the second is the similarity score.
* Sort the aforementioned list of tuples based on the similarity scores; that is, the second element.
* Get the top 10 elements of this list. Ignore the first element as it refers to self (the movie most similar to a particular movie is the movie itself).
* Return the titles

# Function that takes in movie title as input and outputs most similar movies

def get\_recommendations(title, cosine\_sim=cosine\_sim):

# Get the index of the movie that matches the title

idx = indices[title]

# Get the pairwsie similarity scores of all movies with that movie and convert it into a list

sim\_scores = list(enumerate(cosine\_sim[idx]))

# Sort the movies based on the similarity scores from second element since first one is itself

sim\_scores = sorted(sim\_scores, key=lambda x: x[1], reverse=True)

# Get the scores of the 10 most similar movies leaving first element

sim\_scores = sim\_scores[1:11]

# Get the movie indices

movie\_indices = [i[0] for i in sim\_scores]

# Return the top 10 most similar movies

print(q\_movies['title'].iloc[movie\_indices])

ENUMERATE:

A lot of times when dealing with iterators, we also get a need to keep a count of iterations. Python eases the programmers’ task by providing a built-in function enumerate() for this task.  
Enumerate() method adds a counter to an iterable and returns it in a form of enumerate object. This enumerate object can then be used directly in for loops or be converted into a list of tuples using list() method.

get\_recommendations('The Dark Knight Rises')

We see that, while our system has done a decent job of finding movies with similar plot descriptions, the quality of recommendations is not that great. "The Dark Knight Rises" returns all Batman movies while it more likely that the people who liked that movie are more inclined to enjoy other Christopher Nolan movies.

**Recommendation based on cast, director, genre, keyword, imdb rating**

# Convert IDs to int. Required for merging

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

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

q\_movies['id'] = q\_movies['id'].astype('int')

# Merge keywords and credits into your main metadata dataframe

q\_movies = q\_movies.merge(credits, on='id')

q\_movies = q\_movies.merge(keywords, on='id')

print(q\_movies.shape)

From your new features, cast, crew and keywords, we need to extract the three most important actors, the director and the keywords associated with that movie. Right now, your data is present in the form of "stringified" lists. You need to convert them into a form that is usable for you.

# Get the director's name from the crew feature. If director is not listed, return NaN

def get\_director(x):

for i in x:

if i['job'] == 'Director':

return i['name']

return np.nan

# Returns the list of top 3 actors

def get\_list(x):

if isinstance(x, list):

names = [i['name'] for i in x]

#Check if more than 3 actors exist. If yes, return only first three. If no, return entire list.

if len(names) > 3:

names = names[:3]

return names

#Return empty list in case of missing/malformed data

return []

#get directors using the function and create its respective column

q\_movies['director']=q\_movies['crew'].apply(get\_director)

#get actors using the function and create its respective column

q\_movies['actors']=q\_movies['cast'].apply(get\_list)

#get keywords using the function and create its respective column

q\_movies['keywords']=q\_movies['keywords'].apply(get\_list)

#get genres using the function and create its respective column

q\_movies['genres']=q\_movies['genres'].apply(get\_list)

**# Print the new features of the first 3 films**

q\_movies[['title', 'cast', 'director', 'keywords', 'genres']].head(3)

The next step would be to convert the names and keyword instances into lowercase and strip all the spaces between them. This is done so that your vectorizer doesn't count the Johnny of "Johnny Depp" and "Johnny Galecki" as the same. After this processing step, the aforementioned actors will be represented as "johnnydepp" and "johnnygalecki" and will be distinct to your vectorizer.

[::-1]

Extended slice offers to put a “step” field as**[start,stop,step]**, and giving no field as start and stop indicates default to 0 and string length respectively and “**-1**” denotes starting from end and stop at the start, hence reversing string.

list1 = ['1','2','3','4']

s = "-"

s =s.join(list1)

print(s)

**1-2-3-4**

list1 = ['g','e','e','k', 's']

print("".join(list1))

**geeks**

text = 'geeks for geeks'

# Splits at space

print(text.split())

word = 'geeks, for, geeks'

# Splits at ','

print(word.split(', '))

word = 'geeks:for:geeks'

# Splitting at ':'

print(word.split(':'))

word = 'CatBatSatFatOr'

# Splitting at 3

print([word[i:i+3] for i in range(0, len(word), 3)])

**['geeks', 'for', 'geeks']**

**['geeks', 'for', 'geeks']**

**['geeks', 'for', 'geeks']**

**['Cat', 'Bat', 'Sat', 'Fat', 'Or']**

# Function to convert all strings to lower case and strip names of spaces

def clean\_data(x):

if isinstance(x, list):

return [str.lower(i.replace(" ", "")) for i in x]

else:

#Check if director exists. If not, return empty string

if isinstance(x, str):

return str.lower(x.replace(" ", ""))

else:

return ''

# Apply clean\_data function to your features.

q\_movies['director']=q\_movies['director'].apply(clean\_data)

q\_movies['actors']=q\_movies['actors'].apply(clean\_data)

q\_movies['keywords']=q\_movies['keywords'].apply(clean\_data)

q\_movies['genres']=q\_movies['genres'].apply(clean\_data)

print(q\_movies[['title', 'cast', 'director', 'keywords', 'genres']].head(3))

You are now in a position to create your "metadata new", which is a string that contains all the metadata that you want to feed to your vectorizer (namely actors, director and keywords).

#Create a new row that contains all the above info

def create\_row(x):

return ' '.join(x['keywords']) + ' ' + ' '.join(x['actors']) + ' ' + x['director'] + ' ' + ' '.join(x['genres'])

q\_movies['new'] = q\_movies.apply(create\_row, axis=1)

The next steps are the same as what you did with your plot description based recommender. One important difference is that you use the CountVectorizer() instead of TF-IDF. This is because you do not want to down-weight the presence of an actor/director if he or she has acted or directed in relatively more movies. It doesn't make much intuitive sense.

i.e.,

When recommending movies on plot basis there are many words that may come in all plots(like: The, and etc). So here we use tfidf vectorizer so that only the unique words for a particular movie plot is found out and can used for recommendation

Whereas while recommending movies on keywords, actors, genre basis we use count vectorizer because we can recommend other movies of the particular actor or genre or director

**# Import CountVectorizer and create the count matrix**

from sklearn.feature\_extraction.text import CountVectorizer

count = CountVectorizer(stop\_words='english')

count\_matrix = count.fit\_transform(q\_movies['new'])

**# Compute the Cosine Similarity matrix based on the count\_matrix**

from sklearn.metrics.pairwise import cosine\_similarity

cosine\_sim2 = cosine\_similarity(count\_matrix, count\_matrix)

**#Reset index and generate title as index**

q\_movies = q\_movies.reset\_index()

indices = pd.Series(q\_movies.index, index=q\_movies['title'])

**#Recommend movies using the function**

get\_recommendations('The Dark Knight Rises', cosine\_sim2)

**Popularity and Ratings**

One thing that we notice 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 will add a mechanism to remove bad movies and return movies which are popular and have had a good critical response.

I 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.

One of the most basic metrics you can think of is the rating. However, using this metric has a few caveats. For one, it does not take into consideration the popularity of a movie. Therefore, a movie with a rating of 9 from 10 voters will be considered 'better' than a movie with a rating of 8.9 from 10,000 voters.

On a related note, this metric will also tend to favor movies with smaller number of voters with skewed and/or extremely high ratings. As the number of voters increase, the rating of a movie regularizes and approaches towards a value that is reflective of the movie's quality. It is more difficult to discern the quality of a movie with extremely few voters.

Taking these shortcomings into consideration, it is necessary that you come up with a weighted rating that takes into account the average rating and the number of votes it has garnered. Such a system will make sure that a movie with a 9 rating from 100,000 voters gets a (far) higher score than a YouTube Web Series with the same rating but a few hundred voters.

Since you are trying to build a clone of IMDB's Top 250, you will use its weighted rating formula as your metric/score. Mathematically, it is represented as follows:



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; And
* *C* is the mean vote across the whole report

You already have the values to *v* (vote\_count) and *R* (vote\_average) for each movie in the dataset. It is also possible to directly calculate *C* from this data.

We know that q\_movies features all the movies in the charts, that have more votes than at least 90% of the movies in the list.

As a first step, let's calculate the value of C, the mean rating across all movies:

C = metadata['vote\_average'].mean()

print(C)

**5.61820721513**

The average rating of a movie on IMDB is around 5.6, on a scale of 10.

Next, let's calculate the number of votes, m, received by a movie in the 90th percentile.

m = metadata['vote\_count'].quantile(0.94)

q\_movies.shape

You see that there are 4555 movies which qualify to be in this list. Now, you need to calculate your metric for each qualified movie. To do this, you will define a function, weighted\_rating() and define a new feature score, of w

# Function that computes the weighted rating of each movie

def weighted\_rating(x, m=m, C=C):

v = x['vote\_count']

R = x['vote\_average']

# Calculation based on the IMDB formula

return (v/(v+m) \* R) + (m/(m+v) \* C)

# Define a new feature 'score' and calculate its value with `weighted\_rating()`

q\_movies['score'] = q\_movies.apply(weighted\_rating, axis=1)

def improved\_recommendations(title, cosine\_sim=cosine\_sim2):

# Get the index of the movie that matches the title

idx = indices[title]

# Get the pairwsie similarity scores of all movies with that movie and convert it into a list

sim\_scores = list(enumerate(cosine\_sim[idx]))

# Sort the movies based on the similarity scores from second element since first one is itself

sim\_scores = sorted(sim\_scores, key=lambda x: x[1], reverse=True)

# Get the scores of the 10 most similar movies leaving first element

sim\_scores = sim\_scores[1:11]

# Get the movie indices

movie\_indices = [i[0] for i in sim\_scores]

# Return the top 10 most similar movies

movies = q\_movies.iloc[movie\_indices][['title', 'score']]

movies = movies.sort('score', ascending=False)

print(movies.head(10))

**improved\_recommendations('The Dark Knight Rises')**

**COLLABORATIVE BASED RECOMMENDATION SYSTEM**

Our content based engine suffers from some severe limitations. It is only capable of suggesting movies which are *close* to a certain movie. That is, it is not capable of capturing tastes and providing recommendations across genres.

Also, the engine that we built is not really personal in that it doesn't capture the personal tastes and biases of a user. Anyone querying our engine for recommendations based on a movie will receive the same recommendations for that movie, regardless of who s/he is.

Therefore, in this section, we will use a technique called **Collaborative Filtering** to make recommendations to Movie Watchers. Collaborative Filtering is based on the idea that users similar to a me can be used to predict how much I will like a particular product or service those users have used/experienced but I have not.

I will not be implementing Collaborative Filtering from scratch. Instead, I will use the **Surprise** library that used extremely powerful algorithms like **Singular Value Decomposition (SVD)** to minimise RMSE (Root Mean Square Error) and give great recommendations.

ratings = pd.read\_csv('data/ratings\_small.csv')

ratings.head()

**from** **surprise** **import** Reader, Dataset, SVD, evaluate

reader = Reader()

data = Dataset.load\_from\_df(ratings[['userId', 'movieId', 'rating']], reader)

data.split(n\_folds=5)

svd = SVD()

evaluate(svd, data, measures=['RMSE', 'MAE'])

We get a mean **Root Mean Sqaure Error** of 0.8963 which is more than good enough for our case. Let us now train on our dataset and arrive at predictions

trainset = data.build\_full\_trainset()

svd.fit(trainset)

Let us pick user 5000 and check the ratings s/he has given.

ratings[ratings['userId'] == 1]

We can now predict ratings by directly calling the [**predict()**](https://surprise.readthedocs.io/en/stable/algobase.html#surprise.prediction_algorithms.algo_base.AlgoBase.predict) method. Let’s say you’re interested in user 1 and item 302 (make sure they’re in the trainset!), and you know that the true rating rui=4rui=3

svd.predict(1, 302, 3)

For movie with ID 302, we get an estimated prediction of **2.686**. One startling feature of this recommender system is that it doesn't care what the movie is (or what it contains). It works purely on the basis of an assigned movie ID and tries to predict ratings based on how the other users have predicted the movie.

# Reader class

The Reader class is used to parse a file containing ratings.

**build\_full\_trainset()**

Do not split the dataset into folds and just return a trainset as is, built from the whole dataset.

## Hybrid Recommender

In this section, I will try to build a simple hybrid recommender that brings together techniques we have implemented in the content based and collaborative filter based engines. This is how it will work:

* **Input:** User ID and the Title of a Movie
* **Output:** Similar movies sorted on the basis of expected ratings by that particular user.

**def** convert\_int(x):

**try**:

**return** int(x)

**except**:

**return** np.nan

In [63]:

id\_map = pd.read\_csv('data/links\_small.csv')[['movieId', 'tmdbId']]

id\_map['tmdbId'] = id\_map['tmdbId'].apply(convert\_int)

id\_map.columns = ['movieId', 'id']

id\_map = id\_map.merge(smd[['title', 'id']], on='id').set\_index('title')

*#id\_map = id\_map.set\_index('tmdbId')*

In [64]:

indices\_map = id\_map.set\_index('id')

In [65]:

**def** hybrid(userId, title):

idx = indices[title]

tmdbId = id\_map.loc[title]['id']

*#print(idx)*

movie\_id = id\_map.loc[title]['movieId']

sim\_scores = list(enumerate(cosine\_sim[int(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', 'id']]

movies['est'] = movies['id'].apply(**lambda** x: svd.predict(userId, indices\_map.loc[x]['movieId']).est)

movies = movies.sort\_values('est', ascending=**False**)

**return** movies.head(10)