1 - Data Pre-Processing

movie['title']:

movie_id	1	2
title	Toy Story (1995)	GoldenEye (1995)
release_date	01-Jan-1995	01-Jan-1995

Interpretation: The movie title also has the year included.

Following code-snippet demonstrates the updated column names.

```
# Format 'title' i.e. remove 'year' from title
zz['title'] = zz['title'].astype(str).str[:-7]
```

```
zz.title.head()
```

```
0                    Kolya
1        Legends of the Fall
2    Hunt for Red October, The
3      Remains of the Day, The
4               Men in Black
Name: title, dtype: object
```

Interpretation: The attributes (column names) by default are self-explanatory. However, some of these are renamed to make it less confusing.

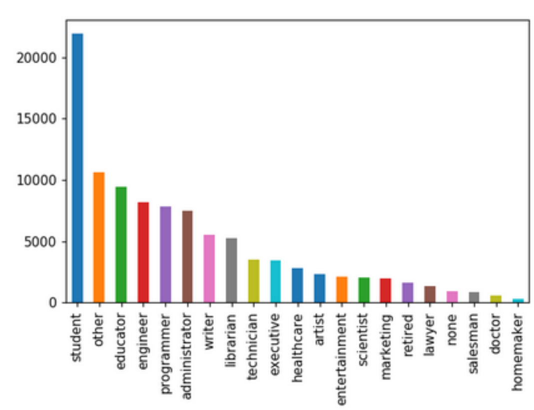
2 – Exploratory Data Analysis

2.1 Univariate Analysis

'occupation':

```
plt.gcf().subplots_adjust(bottom=0.25)
pd.value_counts(zz['occupation']).plot.bar()
```

```
<IPython.core.display.Javascript object>
```



Interpretation: Highest number of users are students.

2.2 Bivariate Analysis

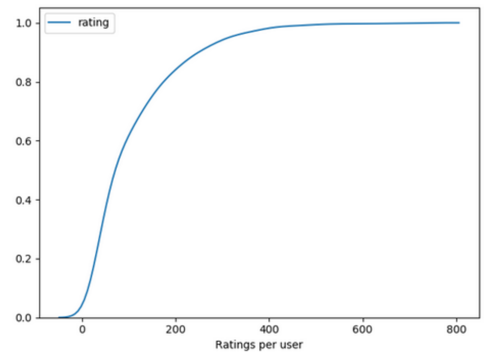
Ratings vs User - Cumulative Density Function

```
movies_per_user = zz.groupby(by='user_id')['rating'].count()
```

```
: movies_per_user = movies_per_user.sort_values(ascending=False)
movies_per_user.head()
```

```
: user_id
405    737
655    685
13     636
450    540
276    518
Name: rating, dtype: int64
```

```
# Cumulative Density Function
sns.kdeplot(movies_per_user, cumulative = True)
plt.xlabel('Ratings per user')
```



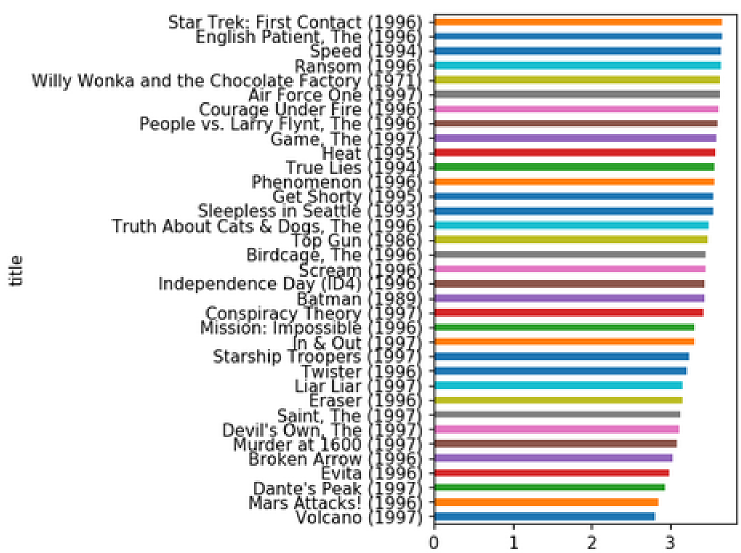
Interpretation: 82% of the users have made less than 200 ratings while 18% of the users have rated more than 200 of them.

High rated movies (by rating)

Visual representation of highly rated movies.

```
dff_1[dff_1['rating']['size']> 200]['rating']['mean'].sort_values(ascending = True).head(35).plot(kind = 'barh')
```

```
:IPython.core.display.Javascript object>
```



Gender vs Rating vs Title

Gen dataframe has 'sex', 'title' and 'rating'

```
gen = z[['sex', 'title', 'rating']]
```

We pivot the dataframe with title as index, sex as columns and fill values with rating.

```
new_gen = gen.pivot_table(index = 'title', columns = 'sex', values = 'rating')
```

```
new_gen.head()
```

Now that we have a pivot table with average male and female ratings for each movie, we can go ahead and calculate their difference to find any interesting patterns in movie selection.

```
new_gen['diff'] = new_gen['M'] - new_gen['F']
```

```
new_gen.head()
```

sex	F	M	diff
title			
'Til There Was You (1997)	2.200000	2.500000	0.300000
1-900 (1994)	1.000000	3.000000	2.000000
101 Dalmatians (1996)	3.116279	2.772727	-0.343552
12 Angry Men (1957)	4.269231	4.363636	0.094406
187 (1997)	3.500000	2.870968	-0.629032

 # Top 10 movies highly rated by Females but not by Males

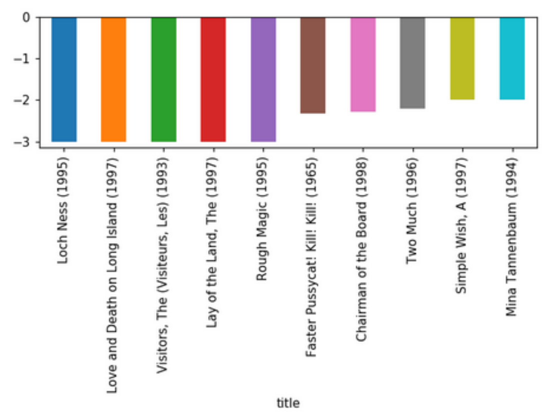
Negative values represent that females rated the movies higher than males on an average.

```
# Top 10 movies highly rated by Females but not by Males
new_gen.sort_values('diff').head(10)
```

sex	F	M	diff
title			
Loch Ness (1995)	4.0	1.000000	-3.000000
Love and Death on Long Island (1997)	4.0	1.000000	-3.000000
Visitors, The (Visiteurs, Les) (1993)	5.0	2.000000	-3.000000
Lay of the Land, The (1997)	4.0	1.000000	-3.000000
Rough Magic (1995)	4.0	1.000000	-3.000000
Faster Pussycat! Kill! Kill! (1965)	5.0	2.666667	-2.333333
Chairman of the Board (1998)	4.0	1.714286	-2.285714
Two Much (1996)	4.0	1.800000	-2.200000
Simple Wish, A (1997)	3.0	1.000000	-2.000000
Mina Tannenbaum (1994)	5.0	3.000000	-2.000000

```
new_gen.sort_values('diff').head(10)['diff'].plot(kind='bar')
```

<IPython.core.display.Javascript object>



Interpretation: We see that 'Loch Ness', 'Love Death and Long Island' are among the movies that have been rated highly by females than that of males.

 # Top 10 movies highly rated by Males but not by Females

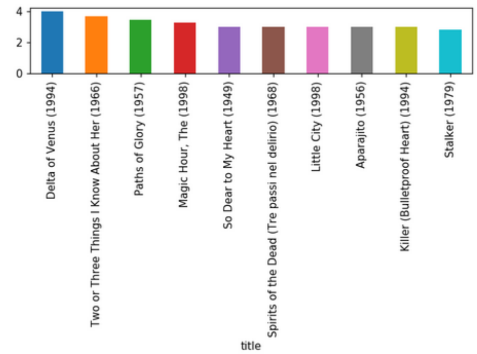
Positive values represent that females rated the movies higher than males on an average.

```
# Top 10 movies highly rated by Males but not by Females
new_gen.sort_values('diff', ascending=False).head(10)
```

sex	F	M	diff
title			
Delta of Venus (1994)	1.0	5.000000	4.000000
Two or Three Things I Know About Her (1966)	1.0	4.666667	3.666667
Paths of Glory (1957)	1.0	4.419355	3.419355
Magic Hour, The (1998)	1.0	4.250000	3.250000
So Dear to My Heart (1949)	1.0	4.000000	3.000000
Spirits of the Dead (Tre passi nel delirio) (1968)	1.0	4.000000	3.000000
Little City (1998)	2.0	5.000000	3.000000
Aparajito (1956)	1.0	4.000000	3.000000
Killer (Bulletproof Heart) (1994)	1.0	4.000000	3.000000
Stalker (1979)	1.0	3.800000	2.800000

```
new_gen.sort_values('diff', ascending=False).head(10)['diff'].plot(kind = 'bar')
```

<IPython.core.display.Javascript object>



Interpretation: We see that 'Loch Ness', 'Love Death and Long Island' are among the movies that have been rated highly by females than that of males.

3 – Transformations

Replacing ratings with below_avg, avg and above_avg:

Ratings 1, 2 are replaced by 'below_average', while 3 is replaced as 'average' and 4, 5 are categorized as 'above_average'.

```
# Function to categorize 'rating'
def transformation_1(df):
    df['rating'].replace([1, 2, 3, 4, 5],
                         ['below_avg', 'below_avg', 'avg', 'above_avg', 'above_avg'],
                         inplace = True)
```

```
transformation_1(zz)
```

```
zz.rating.head(10)
```

```
0    above_avg
1    above_avg
2    above_avg
3    above_avg
4    above_avg
5    above_avg
6          avg
7    below_avg
8    above_avg
9          avg
Name: rating, dtype: object
```

4 - Web Scraping

4.1 Beautiful Soup:

Using Python's Beautiful Soup to get data from IMDB's Top 150 movies

```
url = 'http://www.imdb.com/search/title?release_date=2017&sort=num_votes,desc&page=1'
```

```
response = get(url)
print(response.text[:500])
```

```
<!DOCTYPE html>
<html
    xmlns:og="http://ogp.me/ns#"
    xmlns:fb="http://www.facebook.com/2008/fbml">
    <head>
```

I use html parser to convert html text into beautiful soup object.

```
html_soup = BeautifulSoup(response.text, 'html.parser')
type(html_soup)
```

```
bs4.BeautifulSoup
```

```
movie_containers = html_soup.find_all('div', class_ = 'lister-item mode-advanced')
print(type(movie_containers))
print(len(movie_containers))
```

```
<class 'bs4.element.ResultSet'>
50
```

```
first_movie = movie_containers[0]
```

This returns a prettified version of html text.

movie title:

```
first_name = first_movie.h3.a.text
first_name
```

```
'Logan'
```

movie year:

```
first_year = first_movie.h3.find('span', class_ = 'lister-item-year text-muted unbold').text
first_year
```

```
'(2017)'
```

imdb rating:

```
first_imdb = float(first_movie.strong.text)
first_imdb
```

```
8.1
```

Cons: This approach seems tedious and computationally expensive. Also, this requires revisiting the IMDB website once for every request.

4.2 Tmdbsimple:

Importing ‘tmdbsimple' and key in the credentials

```
import tmdbsimple as tmdb
```

```
tmdb.API_KEY = 'f875e3c0cde708e575e3b72bea080a66'
```

```
movie = tmdb.Movies(603)
```

```
movie
```

```
<tmdbsimple.movies.Movies at 0x54451d0>
```

```
movie = movie.info()
```

```
movie
```

```
{'adult': False,
 'backdrop_path': '/7u3pxc0K1wx32IleAkLv78MKgrw.jpg',
 'belongs_to_collection': {'backdrop_path': '/bRm2DEgUiYciDw3myHuYFInD7la.jpg',
  'id': 2344,
  'name': 'The Matrix Collection',
  'poster_path': '/lh4aGpd3U9rm9B8Oqr6CUgQLtZL.jpg'},
 'budget': 63000000,
 'genres': [{'id': 28, 'name': 'Action'},
  {'id': 878, 'name': 'Science Fiction'}],
 'homepage': 'http://www.warnerbros.com/matrix',
 'id': 603,
 'imdb_id': 'tt0133093',
```

Extracting Movie Attributes:

```
movie['overview']
```

```
'Set in the 22nd century, The Matrix tells the story of a computer hacker who joins a group of underground
insurgents fighting the vast and powerful computers who now rule the earth.'
```

Another way of accessing movie data is by passing the movie name to the argument ‘query’.

```
search = tmdb.Search()
response = search.movie(query='The Bourne')
```

```
for s in search.results:
    print(s['title'], s['id'], s['release_date'], s['popularity'])
```

```
The Bourne Identity 2501 2002-06-14 13.959
The Bourne Supremacy 2502 2004-07-23 13.047
The Bourne Legacy 49040 2012-08-08 12.672
The Bourne Ultimatum 2503 2007-08-03 12.482
Bette Bourne: It Goes with the Shoes 179304 2013-03-21 0.6
Jason Bourne 324668 2016-07-27 13.083
Untitled Jeremy Renner/Bourne Sequel 393640  0.806
```

Note: When we use tmdb.search() we do get the tmdb_id as well as the title. But using tmdb.Movies() yields much more information about the movie.

New Approach:

We can query TMDB API only using movie_ids and not by movie titles. (When queried, API throws a 404 Cleint Error) and also takes longer time to that of movie_id. However, Movie Lens dataset has its own movie_id which are quite different from that of TMDBs (tmdb_id)

Hence, we use the following approach:

- Get the movielens_id and title from movielens dataset
- Query TMDB API using movie title to get TMDB_IDs
- Use queried tmdb_id to get additional info about the movie

Based on this approach I web scrape using TMDB simple and get the metadata of the movie titles matching from movielens data.

5. Popularity Based Recommendation

Simple Recommendation System (Popularity based - Ratings)

Ratings matrix with movie_id as columns and user_id as rows and ratings as values

```
ratings_matrix = ratings.pivot_table(index = ['movie_id'], columns = ['user_id'], value
s = 'rating').reset_index(drop = True)

# Fill nans with 0
ratings_matrix.fillna(0, inplace = True)
ratings_matrix.head()
```

user_id	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26
0	5.0	4.0	0.0	0.0	4.0	4.0	0.0	0.0	0.0	4.0	0.0	0.0	3.0	0.0	1.0	5.0	4.0	5.0	0.0	3.0	5.0	0.0	5.0	0.0	5.0	3.0
1	3.0	0.0	0.0	0.0	3.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	3.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	2.0	0.0	0.0	0.0	0.0
2	4.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	[illegible]	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
3	3.0	0.0	0.0	0.0	0.0	0.0	5.0	0.0	0.0	4.0	0.0	5.0	5.0	0.0	0.0	5.0	0.0	3.0	4.0	0.0	0.0	5.0	0.0	0.0	0.0	0.0
4	3.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	1.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	2.0	0.0	0.0	0.0	0.0	0.0

The above matrix has:

Rows – Users
Columns – Movies
Values – Ratings

```python
def pop_rec_system_new(user_input, metricc):

    if metricc == "cosine":
        movie_similarity = 1 - pairwise_distances(ratings_matrix.as_matrix(), metric = "cosine")
    elif metricc == "euclidean":
        movie_similarity = 1 - pairwise_distances(ratings_matrix.as_matrix(), metric = "euclidean")
    elif metricc == "manhattan":
        movie_similarity = 1 - pairwise_distances(ratings_matrix.as_matrix(), metric = "manhattan")
    elif metricc == "correlation":
        movie_similarity = 1 - pairwise_distances(ratings_matrix.as_matrix(), metric = "correlation")

    np.fill_diagonal(movie_similarity, 0)
    cosine_similarity_matrix = pd.DataFrame(movie_similarity)

    if (any(movies.title == user_input)):

            inp = movies[movies['title']==user_input].index.tolist() # Index of the user impput (movie)
            inp = inp[0]                                             # Index of the user imput (movie)

            similar_movies = movies[['movie_id', 'title']]           # similar Movies [dataframe with id,
title]
            # 'similarity' column contains cosine values of each movie with user input
            similar_movies['similarity'] = cosine_similarity_matrix.iloc[inp]
            similar_movies.columns = ['movie_id', 'title', 'similarity'] # rename columns

            # Reccommended Movies
            print("Reccommended movies")
            print("-------------------")
            print(similar_movies.sort_values( ["similarity"], ascending = False )[1:10])

    # If movie is not in existing dataframe
    else:
        print("Movie doesn't exist in the database")
```

The above code calculates pairwise distances using various metrics to return movies.

Cosine Similarity:

Results for the movie ‘Golden Eye’ using cosine similarity as a metric.

```python
pop_rec_system_new('GoldenEye', 'cosine')
```

```
Reccommended movies
-------------------
     movie_id                         title  similarity
160       161                       Top Gun    0.623544
384       385                    True Lies     0.617274
402       403                        Batman    0.616143
61         62                      Stargate    0.604969
575       576                  Cliffhanger     0.601960
225       226                    Die Hard 2    0.597083
230       231                Batman Returns    0.595684
549       550     Die Hard: With a Vengeance   0.590124
95         96     Terminator 2: Judgment Day   0.584100
```

Note: This recommendation system is solely based on popularity. The movies returned with cosine, euclidean and manhattan distance are quite similar to each other. However, they are not so much when the recommendation system uses pearson correlation.

Limitation: This recommendation system suggests movies IRRSPECTIVE OF USER PREFERENCES.

6 – Content Based Recommendation

6.1 Description Based Recommendation:

6.1.1 Recommendation Engine using 'overview':

First recommendation engine considers only the 'overview' of the movie. 'Overview' stands for the descriptive text that is outlined for a movie in 'IMDB' official site.

```
zz_metadata = metadata[metadata['id'].isin(zz['movie_id'])]
```

```
# tf-idf vectorizer
tf = TfidfVectorizer(analyzer = 'word', ngram_range = (1, 2), min_df = 0, stop_words = 'english')
tfidf_matrix = tf.fit_transform(zz_metadata['overview'])                          # Fit Transform 'ov
erview'
cosine_sim = linear_kernel(tfidf_matrix, tfidf_matrix)                            # Cosine Similarity
 of td-idf matrix

zz_metadata_1 = zz_metadata.reset_index()                                         # Reset Index
titles = zz_metadata_1['title']                                                   # Titles
indices = pd.Series(zz_metadata_1.index, index = zz_metadata_1['title'])          # Indices
```

```
# Recommendation Engine
def recommendations_overview(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[1:10]
    movie_indices = [i[0] for i in sim_scores]
    return titles.iloc[movie_indices]
```

```
recommendations_overview('The Dark Knight')
```

```
21                   Batman Forever
233                  Batman Returns
71                           Batman
427                             JFK
843                   Batman Begins
248                 Batman & Robin
324                 A Few Good Men
435    Teenage Mutant Ninja Turtles
261             Tomorrow Never Dies
Name: title, dtype: object
```

Interpretation: This model provides robust recommendations using metadata ['overview'].

Limitation: But there are few not-so meaningful recommendations. Example: (Teenage Mutant Ninja Turtles, Tomorrow Never Dies)

6.1.2 Recommendation Engine using 'tagline':

Second recommendation engine considers only the 'tagline' of a movie. 'Tagline' stands for the extended movie title which certain movies have.

Example: 'Die Hard 3: With a Vengeance'

Title of the movie is 'Die Hard 3' while the tagline is 'With a Vengeance'.

```python
# Dropping null values using index
zz_metadata = zz_metadata.drop(list(zz_metadata[zz_metadata['tagline'].isnull()]['id'].index))
```

```python
# tf-idf vectorizer
tf = TfidfVectorizer(analyzer = 'word', ngram_range = (1, 2), min_df = 0, stop_words = 'english')
tfidf_matrix = tf.fit_transform(zz_metadata['tagline'])                          # Fit Transform 'over
view'
cosine_sim = linear_kernel(tfidf_matrix, tfidf_matrix)                            # Cosine Similarity o
f td-idf matrix

zz_metadata_1 = zz_metadata.reset_index()                                         # Reset Index
titles = zz_metadata_1['title']                                                   # Titles
indices = pd.Series(zz_metadata_1.index, index=zz_metadata_1['title'])            # Indices
```

```python
# Recommendation Engine
def recommendations_tagline(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[1:10]
    movie_indices = [i[0] for i in sim_scores]
    return titles.iloc[movie_indices]
```

```python
recommendations_tagline('The Dark Knight')
```

```
1                  GoldenEye
2           Cutthroat Island
3                     Casino
4                 Four Rooms
5          Leaving Las Vegas
6    The City of Lost Children
7              Twelve Monkeys
8                 To Die For
9                      Se7en
Name: title, dtype: object
```

Interpretation: The model built with respect to 'tagline' is not as robust as the previous model. It is apparent that the first model (using metadata ['overview']) provides highly similar movies than the model using 'taglines'.

6.1.3 Recommendation Engine using metadata ['overview'] + metadata ['tagline']:

Final recommendation engine using description considers both the ‘overview’ and the ‘tagline’ of a movie. These two columns are concatenated to form a new column ‘description’.

```python
# Filling nans with empty strings
zz_metadata['tagline'] = zz_metadata['tagline'].fillna('')

# Create a new column 'description' = 'overview' + 'tagline'
zz_metadata['description'] = zz_metadata['overview'] + zz_metadata['tagline']

# Filling nans with empty strings
zz_metadata['description'] = zz_metadata['description'].fillna('')
```

```
# tf-idf vectorizer
tf = TfidfVectorizer(analyzer = 'word', ngram_range = (1, 2), min_df = 0, stop_words = 'english')
tfidf_matrix = tf.fit_transform(zz_metadata['description'])                          # Fit Transform
 'overview'
cosine_sim = linear_kernel(tfidf_matrix, tfidf_matrix)                                 # Cosine Simila
rity of td-idf matrix

zz_metadata_1 = zz_metadata.reset_index()                                              # Reset Index
titles = zz_metadata_1['title']                                                        # Titles
indices = pd.Series(zz_metadata_1.index, index=zz_metadata_1['title'])                 # Indices
```

```
# Recommendation Engine
def recommendations_description(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[1:10]
    movie_indices = [i[0] for i in sim_scores]
    return titles.iloc[movie_indices]
```

```
recommendations_description('The Dark Knight')
```

```
19                    Batman Forever
210                   Batman Returns
61                            Batman
392                              JFK
697                    Batman Begins
223                   Batman & Robin
399    Teenage Mutant Ninja Turtles
236              Tomorrow Never Dies
506                         48 Hrs.
Name: title, dtype: object
```

Interpretation: This model provides similar recommendations to that of the initial model (using metadata ['overview']). We can infer that 'tagline' is not the best feature to consider building a recommendation system.

6.2 Metadata Based Recommendation System

After scraping data from the web for the movie ids in the merged dataframe (‘movielens’), we can now use the metadata to build the recommendation system.

```
# Sample version of the full dataset
links_small = pd.read_csv('data_full/links_small.csv')
```

```
links_small.head(1)
```

	movieId	imdbId	tmdbId
0	1	114709	862.0

Movie lens data has a file ‘links’ that consists of ‘movie id’, ‘imdb id’ and ‘tmdb id’ using which the data was scraped from the web.

Missing Values:

The metadata has column values in dictionary. This can be trickier to handle. Instead of using the dictionary to operate on, I convert the dictionary to a list.

```
# Converting genre dictionary to list
def dict_to_list(x):
    ls = []
    for i in literal_eval(x):
        ls.append(i['name'])
    return ls
```

```
# Apply 'dict_to_list' method
for col in ['cast', 'crew', 'keywords']:
    metadata_full[col] = metadata_full[col].apply(dict_to_list)
```

Checking for null values in 'tagline' and 'overview':

```
# Null values in tagline = 2137
print(links_small_new['tagline'].isnull().sum())

# Null values in tagline = 12
print(links_small_new['overview'].isnull().sum())
```

```
2137
12
```

Note: Since there are null values in 'tagline' and 'overview', we cannot simply join them together to create a new column ('description').

Solution: Strip off the white spaces.

```
# Strip off white spaces from 'tagline'
links_small_new['tagline'] = links_small_new['tagline'].fillna('')

# Create new column 'description' = 'overview' + 'tagline'
links_small_new['description'] = links_small_new['overview'] + links_small_new['tagline']

# Strip off white spaces from 'description', if any
links_small_new['description'] = links_small_new['description'].fillna('')
```

Note: So far, `links_small_new` has cast, crew, credits and genres. But we do not need all the data in them. To efficiently use them, I clean each column further.

Creating new columns 'cast_size' and 'crew_size':

```
# Creating new features 'cast_size' and 'crew_size'
links_small_new['cast_size'] = links_small_new['cast'].apply(lambda x: len(x))
links_small_new['crew_size'] = links_small_new['crew'].apply(lambda x: len(x))
```

```
# Cast of a movie
links_small_new['cast'][0]
```

```
['Tom Hanks',
 'Tim Allen',
 'Don Rickles',
 'Jim Varney',
 'Wallace Shawn',
 'John Ratzenberger',
 'Annie Potts',
 'John Morris',
 'Erik von Detten',
 'Laurie Metcalf',
 'R. Lee Ermey',
 'Sarah Freeman',
 'Penn Jillette']
```

Note: Cast can include actors and actress that are both famous and infamous. However, famous artists are most likely to play a significant role in affecting the user's opinion than others.

Solution: Select 4 artists [lead actor 1, lead actor 2, supporting actor 1, supporting actor 2] rather than considering all.

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

1. **Strip Spaces and Convert to Lowercase** from all our features. This way, engine will not confuse between **Johnny Depp** and **Johnny Galecki.**
2. **Mention Director 2 times** to give it more weight relative to the entire cast.

```
# Strip spaces from 'cast' and convert to lowercase
links_small_new['cast'] = links_small_new['cast'].apply(lambda x: [str.lower(i.replace(" ", "")) for i in x])
```

```
# Strip spaces from 'director'
links_small_new['director'] = links_small_new['director'].astype('str').apply(lambda x: str.lower(x.replace(" ", "")))

# Adding weight to 'director'
links_small_new['director'] = links_small_new['director'].apply(lambda x: [x,x])
```

Keywords:

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

```
links_small_new['keywords'][:3]
```

```
0    [jealousy, toy, boy, friendship, friends, riva...
1    [board game, disappearance, based on children'...
2    [fishing, best friend, duringcreditsstinger, o...
Name: keywords, dtype: object
```

Not all words could prove significant.

```
# Stacking all words from 'keywords'
w = links_small_new.apply(lambda x: pd.Series(x['keywords']), axis = 1).stack().reset_index(level = 1, dro
p = True)
w.name = 'keyword'
```

```
# Value counts
w = w.value_counts()
w[:5]
```

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

Note: Keywords occur in frequencies ranging from 1 to 610. We do not have any use for keywords that occur only once.
Interpretation: Keywords that occur just once.

```
w = w[w > 1]
```

<u>Stemming</u>:

Words like ‘play’, ‘played’ and ‘playing’ can be stemmed to the word ‘play’. This process is called stemming.

Code to perform stemming.

```
# Initialize stemmer object
stemmer = SnowballStemmer('english')
```

```
# Function to filter keywords
def filter_keywords(x):
    words = []
    for i in x:                     # For each word in input
        if i in w:                  # If that word in keywords
            words.append(i)         # append i to words
    return words
```

<u>Preprocess ‘keywords’ column:</u>

```
# Apply filter_keywords to 'keywords'
links_small_new['keywords'] = links_small_new['keywords'].apply(filter_keywords)

# Stem keywords
links_small_new['keywords'] = links_small_new['keywords'].apply(lambda x: [stemmer.stem(i) for i in x])

# Convert string to lower case and strip spaces
links_small_new['keywords'] = links_small_new['keywords'].apply(lambda x: [str.lower(i.replace(" ", "")) f
or i in x])
```

```
links_small_new['keywords'][1]
```

```
['boardgam',
 'disappear',
 "basedonchildren'sbook",
 'newhom',
 'reclus',
 'giantinsect']
```

Soup:

Soup is the metadata of genres, director, cast and keywords.

```
# Soup = 'keywords' + 'cast' + 'director' + 'genres'
links_small_new['soup'] = links_small_new['keywords'] + links_small_new['cast'] + links_small_new['directo
r'] + links_small_new['genres']
```

Soup contains genres, director, cast and keywords.

```
links_small_new['soup'][1]
```

```
['boardgam',
 'disappear',
 "basedonchildren'sbook",
 'newhom',
 'reclus',
 'giantinsect',
 'robinwilliams',
 'jonathanhyde',
 'kirstendunst',
 'bradleypierce',
 'joejohnston',
 'joejohnston',
 'Adventure',
 'Fantasy',
 'Family']
```

Count Vectorizer:

Create a count matrix and calculate the cosine similarities to find movies that are most similar.

```
# Count Vectorizer
count = CountVectorizer(analyzer = 'word', ngram_range = (1, 2), min_df = 0, stop_words = 'english')

# Build a count matrix by fitting and transforming 'soup'
count_matrix = count.fit_transform(links_small_new['soup'])
```

```
# Calculating cosine similarity of count matrix
cosine_sim = cosine_similarity(count_matrix, count_matrix)
```

```
# Reset Index
links_small_new = links_small_new.reset_index()

# Titles
titles = links_small_new['title']

# Indices
indices = pd.Series(links_small_new.index, index = links_small_new['title'])
```

Python code for recommendation engine

```
# Recommendation Engine
def 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[1:10]
    movie_indices = [i[0] for i in sim_scores]
    return titles.iloc[movie_indices]
```

Recommendations:

```
recommendations('The Dark Knight').head(10)
```

```
8031         The Dark Knight Rises
6218                 Batman Begins
7659    Batman: Under the Red Hood
6623                  The Prestige
1134                Batman Returns
5943                      Thursday
8927       Kidnapping Mr. Heineken
1260                Batman & Robin
2085                     Following
Name: title, dtype: object
```

Limitation: This recommendation system returns only the movies based on soup. It does not consider popularity.

Solution: We use the results returned from our Count Vectorizer (indices) and return the movies that are popular based on the IMDB's weighted average. Additionally, I use three different criteria to cut-off the movies (75% percentile, Mean and No Cut-Off criteria)

Weighted Average:

Function to calculate weighted average:

```
# Function to calculate 'weighted_rating'
def weighted_rating(x):
    v = x['vote_count']
    R = x['vote_average']
    return (v/(v+m) * R) + (m/(m+v) * C)
```

```
# Apply weighted rating method to qualified_perc, qualified_mean, new_qualified, sm_df, metadata
for df in [qualified_perc, qualified_mean, new_qualified, sm_df, metadata, metadata_full]:
    df['weighted_rating'] = df.apply(weighted_rating, axis=1)
```

```
# Columns for qualified movies
col_list = ['title', 'release_date', 'vote_count', 'vote_average', 'popularity', 'genres']

# qualification criteria
qualified_perc = metadata[(metadata_full['vote_count'] >= m)
                 & (metadata_full['vote_count'].notnull())
                 & (metadata_full['vote_average'].notnull())][col_list]

# converting vote_count and vote_average colums to integer
qualified_perc['vote_count'] = qualified['vote_count'].astype('int')
qualified_perc['vote_average'] = qualified['vote_average'].astype('int')

qualified_perc.shape
```

Getting qualified movies (cutoff: 95%)

Code for recommendation system with movies cutoff 95%

```
# Better recommendation engine
def better_recommendations_percentile_popularity(title):
    idx = indices[title]                                    # Considers indices of the previous recommedat
ion system
    sim_scores = list(enumerate(cosine_sim[idx]))
    sim_scores = sorted(sim_scores, key=lambda x: x[1], reverse=True)
    sim_scores = sim_scores[1:51]
    movie_indices = [i[0] for i in sim_scores]

    improved_movies = links_small_new.iloc[movie_indices][['title', 'vote_count', 'vote_average']]
    vote_counts = improved_movies[improved_movies['vote_count'].notnull()]['vote_count'].astype('int')
    vote_averages = improved_movies[improved_movies['vote_average'].notnull()]['vote_average'].astype('in
t')

    C = vote_averages.mean()
    m = vote_counts.quantile(0.75)

    qualified = improved_movies[(improved_movies['vote_count'] >= m) & (improved_movies['vote_count'].notn
ull()) & (improved_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
```

We see that the movies recommended by the engine highly emphasized on the crew (director).

```
# Better recommendations
better_recommendations_percentile_popularity('The Dark Knight')
```

	title	vote_count	vote_average	wr
7648	Inception	14075	8	7.917588
6623	The Prestige	4510	8	7.758148
8031	The Dark Knight Rises	9263	7	6.921448
6218	Batman Begins	7511	7	6.904127
7583	Kick-Ass	4747	7	6.852979
1134	Batman Returns	1706	6	5.846862
4145	Insomnia	1181	6	5.797081
8970	Hitman: Agent 47	1183	5	5.065730
132	Batman Forever	1529	5	5.054144
9162	London Has Fallen	1656	5	5.050854

7 – Collaborative Filtering

7.1 Collaborative Filtering:

I pick only 25% of the data.

```
# Randomly sample 25% of the ratings dataset
small_data = ratings_small.sample(frac=0.25)
```

Dividing the data into train and test set:

```
train_data, test_data = train_test_split(small_data, test_size=0.2)
```

```
# Test and Train data matrix
train_data_matrix = train_data.as_matrix(columns = ['user_id', 'movie_id', 'rating'])
test_data_matrix = test_data.as_matrix(columns = ['user_id', 'movie_id', 'rating'])
```

The train and test dataframes are converted to arrays using .as_matrix()

Idea behind user and item similarity:

User similarity can be calculated by measuring 'pairwise distances' between ratings datset.

However, if you have to calculate the 'item similarity', we have to transpose the 'ratings' data and then calculate the pairwise distances.

User Similarity Matrix:

```
# User Similarity Matrix
user_correlation = 1 - pairwise_distances(train_data, metric = 'correlation')
user_correlation[np.isnan(user_correlation)] = 0
print(user_correlation[:4, :4])
```

```
[[1.         0.99998855 0.95570253 0.9999543 ]
 [0.99998855 1.         0.9542832  0.9998971 ]
 [0.95570253 0.9542832  1.         0.9584729 ]
 [0.9999543  0.9998971  0.9584729  1.        ]]
```

Item Similarity Matrix:

```
# Item Similarity Matrix (Train_data_matrix.Transpose)
item_correlation = 1 - pairwise_distances(train_data_matrix.T, metric = 'correlation')
item_correlation[np.isnan(item_correlation)] = 0
print(item_correlation[:4, :4])
```

```
[[ 1.          0.00139149  0.00804213]
 [ 0.00139149  1.         -0.02296115]
 [ 0.00804213 -0.02296115  1.        ]]
```

User Correlation and Item Correlation:

```
user_correlation[:1]
```

```
array([[1.        , 0.99998855, 0.95570253, ..., 0.99972598, 0.99651712,
        0.97092856]])
```

```
item_correlation[:1]
```

```
array([[1.        , 0.00139149, 0.00804213]])
```

Function to predict ratings

```
# Function to predict ratings
def predict(ratings, similarity, type='user'):

    if type == 'user':
        mean_user_rating = ratings.mean(axis=1)                   # Calculate mean of ratings
        ratings_diff = (ratings - mean_user_rating[:, np.newaxis])  # Use np.newaxis so that mean_user_
rating has same format as ratings
        pred = mean_user_rating[:, np.newaxis] + similarity.dot(ratings_diff) / np.array([np.abs(simila
rity).sum(axis=1)]).T

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

    return pred
```

Calling predict function:

```
# Predict ratings on the training data with both similarity score
user_prediction = predict(train_data_matrix, user_correlation, type = 'user')
item_prediction = predict(train_data_matrix, item_correlation, type = 'item')
```

Calculate Root Mean Squared Error:

```
# Function to calculate RMSE
def rmse(pred, actual):
    # Ignore nonzero terms.
    pred = pred[actual.nonzero()].flatten()
    actual = actual[actual.nonzero()].flatten()
    return sqrt(mean_squared_error(pred, actual))
```

Calling RMSE() to calculate error on user based and item based predictions:

```
# RMSE on the test data
print('User-based CF RMSE: ' + str(rmse(user_prediction, test_data_matrix)))
print('Item-based CF RMSE: ' + str(rmse(item_prediction, test_data_matrix)))
```

```
User-based CF RMSE: 17677.833472568193
Item-based CF RMSE: 21050.47348294261
```

8. Potential Next Steps:

Suggestions for Content-Based filtering from other data scientists I met during the meet-up:

1. Use weighted average on each movie:
    - How about multiplying rating count and average rating.
    - For a linear column, there can be huge variance. [Try normalize and standardize]
2. Use metadata td-idf matrix (cosine similarity) rather than just the movies.
    - Use 'word2vec'
3. For collaborative filtering - try 'movie-movie' similarity and 'user-user' similarity (Computationally Expensive)
4. Try to build a Hybrid Recommender