**CONTENTS**

**1. Abstract**

**1.1 Problem Statement  
 1.2 Client  
 1.3 Dataset**

**2. Data Merging**

**3. Data Inspection**

**4. Data Pre-Processing**

**5. Exploratory Data Analysis**

**5.1 Univariate Analysis  
 5.2 Bivariate Analysis  
 5.3 Transformations**

**6. Web Scraping**

**6.1 Beautiful Soup  
 6.2 TMDB Simple**

**7. Popularity Based Recommendation**

**8. Content Based Recommendation**

**8.1 Description Based  
 8.2 Metadata Based**

**9. Collaborative Filtering**

**9.1 Introduction  
 9.2 Performing Collaborative Filtering**

**10. Potential Next Steps**

**1. Abstract**

**1.1 Problem Statement:**

The need for building a good recommendation system for movies cannot be undermined, especially considering the huge increase in viewership of on-demand movies. The goal of this project is – Given a movies attributes and user ratings, design a robust recommendation system using content based and collaborative filtering approaches.

**1.2 Client:**

The data is related with User Ratings, of Movie Lens Data institution. GroupLens Research has collected over various periods of time and made available rating data sets on (<http://movielens.org>).

**1.3 Dataset:**

This dataset is collected from [Grouplens Website](https://grouplens.org/datasets/movielens/).

No demographic information of the user is included. Each user is represented by an id, and no other information is provided.

The data are contained in six files.

**tag** that contains tags applied to movies by users:

* userId
* movieId
* tag
* timestamp

**rating** that contains ratings of movies by users:

* userId
* movieId
* rating
* timestamp

**movie** that contains movie information:

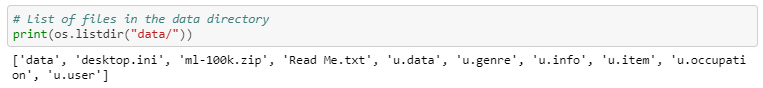
* movieId
* title
* genres

**link** that contains identifiers that can be used to link to other sources:

* movieId
* imdbId
* tmbdId

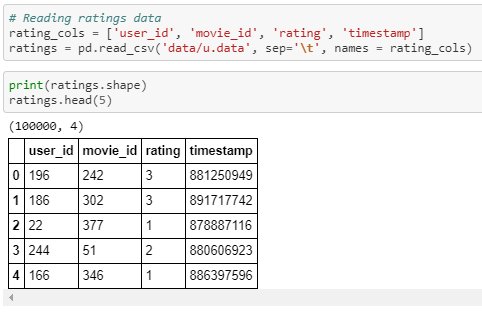
**2. Data Merging**

**os.listdir():**



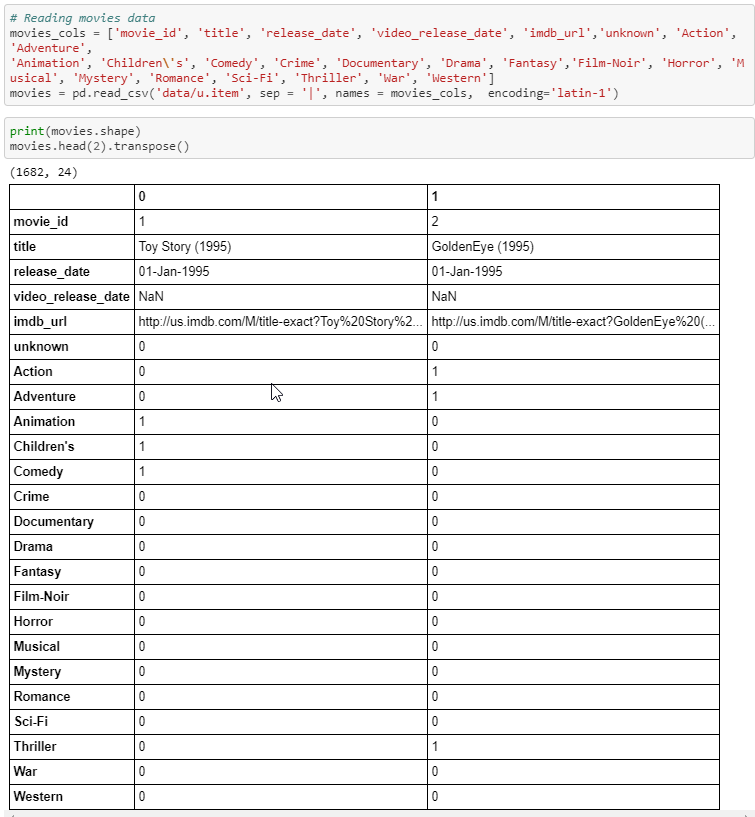
**Interpretation:** The data directory has the following files - data, genre, info, item, occupation and user.

**ratings:**



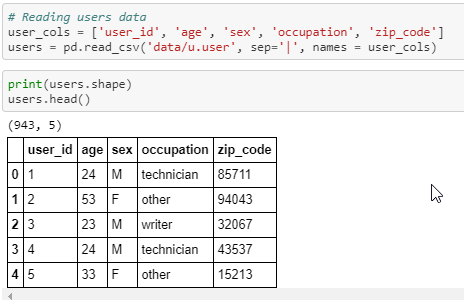
**Interpretation:** The ratings files contains the user id, movie id, rating and timestamp. Considering the recommendation system I’m building, timestamp can be excluded from data.

**movies:**



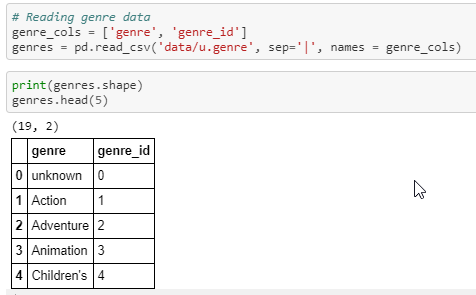
The movies file consists of movie id, title, date, url and movie genres. We can exclude dates, url and even the genres.

**users:**



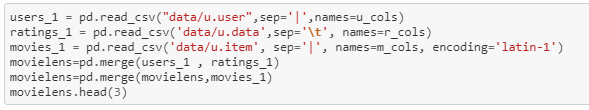
**Interpretation:** The user file consists of user id, age, sex, occupation and zip code.

**genres:**



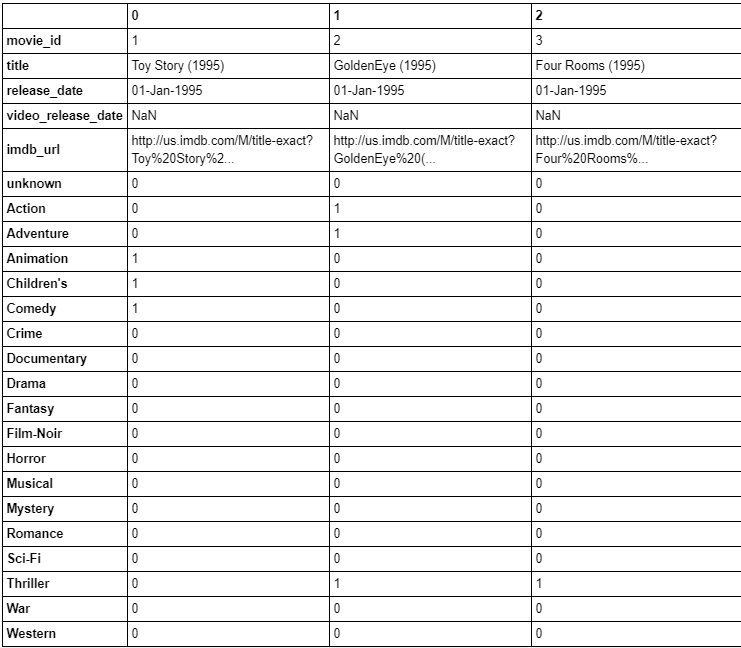
**Interpretation:** The genre file consists of genre id and genre name.

**pd.merge():**



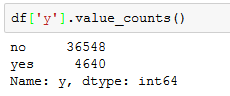
**Interpretation:** All individual dataframes are merged into a single dataframe based on the common columns.

This is the consolidated dataframe has the following columns



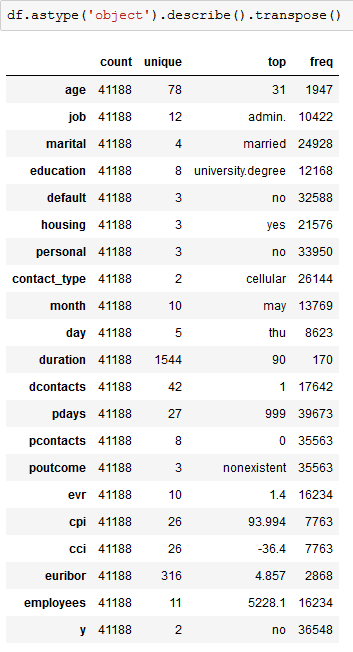
**df.response\_variable.value\_counts():**

The number of positive responses (yes) is largely fewer than the negative responses (no) implying that the dataset is significantly imbalanced.



**Interpretation:** Business problems in financial, banking and healthcare industries often have datasets that are massively imbalanced. Considering the reality surrounding these problems, addressing the class balance anomaly is not a major priority, for now. However, later in this report, I use ‘upsampling’ and ‘downsampling’ to address class imbalance.

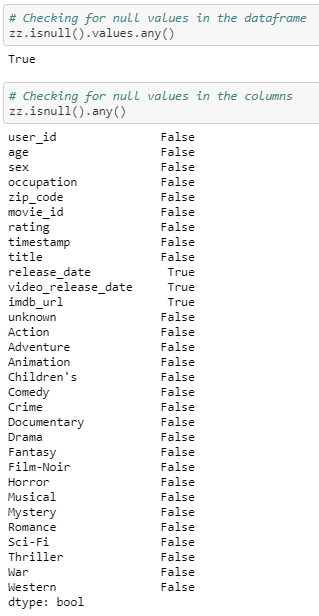
**df.astype('object').describe().transpose():**



**Interpretation:** Using the below code snippet, I can get an idea of most frequently occurring values in an attribute as well as their respective frequencies.

**3. Data Inspection**

**Check for Null values:**



**Interpretation:** Null values are in release\_date, video\_release\_date, imdb\_url; the features which are not significant for this project.

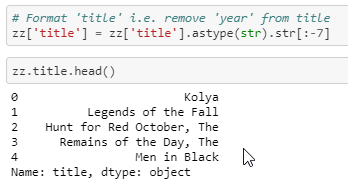
**4. Data Pre-Processing**

**movie[‘title’]:**

****

**Interpretation:** The movie title also has the year included.

Following code-snippet demonstrates the updated column names.

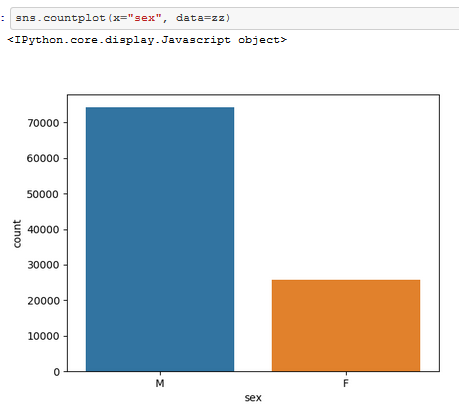


**Interpretation:** The attributes (column names) by default are self-explanatory. However, some of these are renamed to make it less confusing.

**5 – Exploratory Data Analysis**

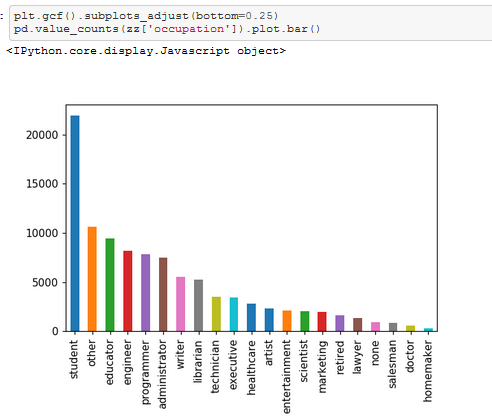
**5.1 Univariate Analysis**

**‘gender’:**



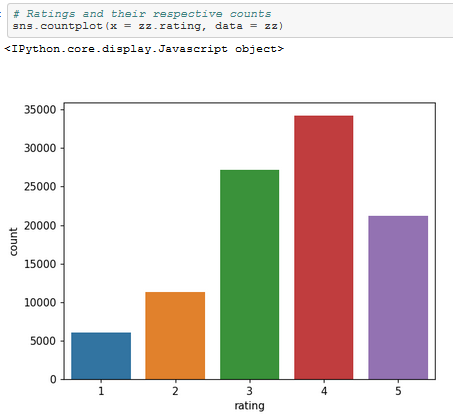
**Interpretation**: The dataset comprises of higher number of males than females.

**‘occupation’:**



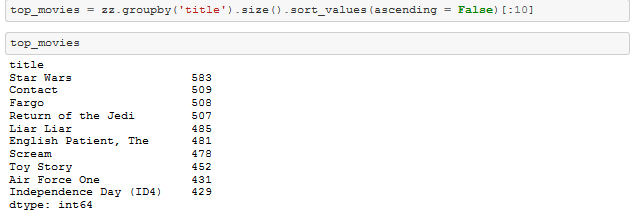
**Interpretation:** Highest number of users are students.

**‘rating’:**



**Interpretation:** From the above plot, it is apparent that most of the user ratings were either 3 or 4.

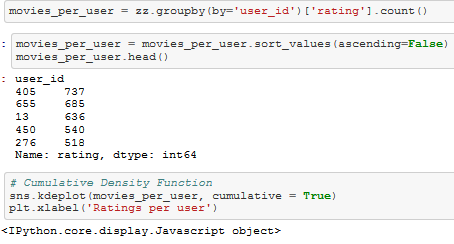
**‘top movies’:**

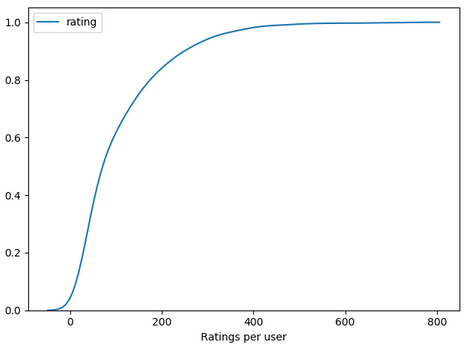


**Interpretation:** Theabove code displays the movies that are rated most.

**5.2 Bivariate Analysis**

**Ratings vs User - Cumulative Density Function**

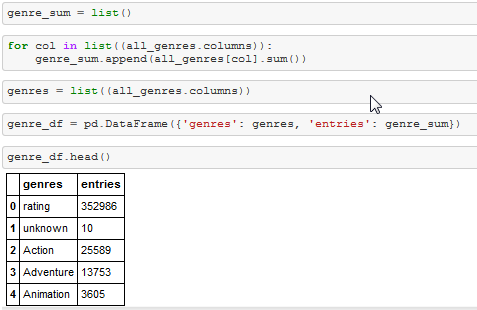
****

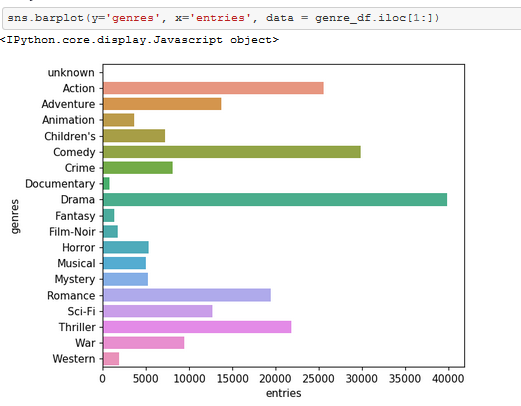


**Interpretation:** 82% of the users have made less than 200 ratings while 18% of the users have rated more than 200 of them.

**‘genre’ count:**

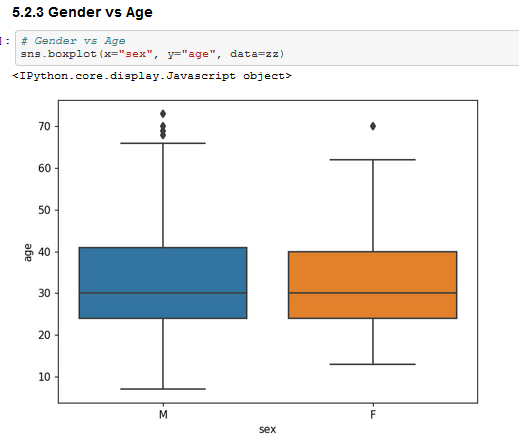
Let’s look at the genres that and their respective counts:





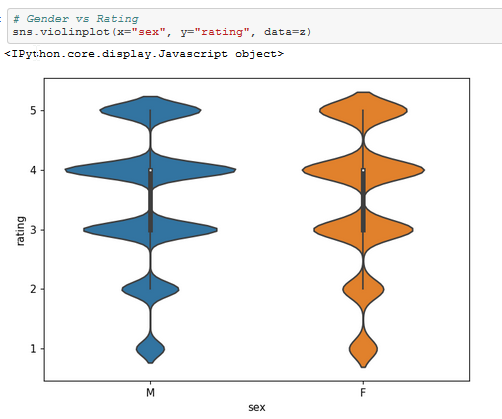
**Interpretation:** It is evident that the most movies belong to ‘drama’ and ‘comedy’ genre.

**‘gender’ vs ‘age’:**



**Interpretation:** The data is distributed between similar age groups for both genders.

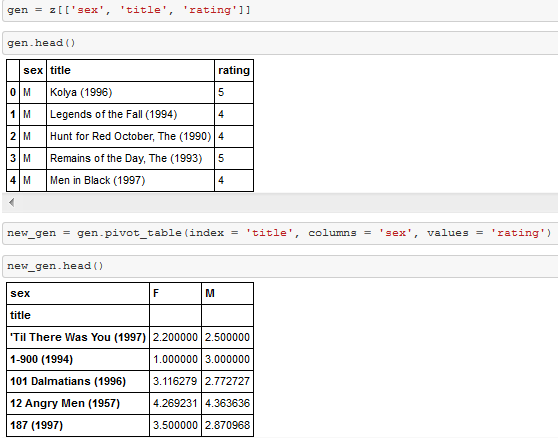
**‘gender’ vs ‘rating’:**



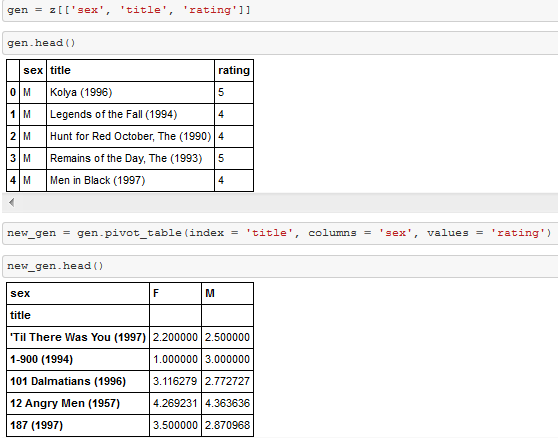
**Interpretation:** It can be noted that males rated movies slightly more generously.

**Gender vs Rating vs Title**

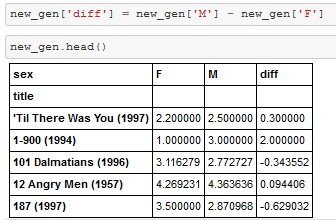
Gen dataframe has ‘sex’, ‘title’ and ‘rating’



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

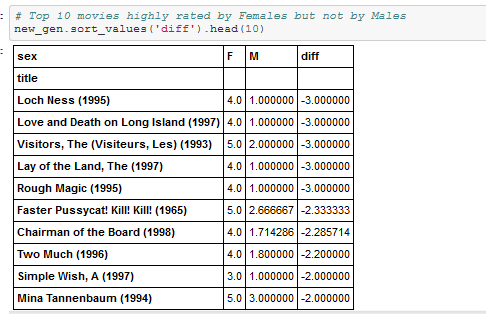


Now that we have a pivot table with average male and female ratings for each movie, we can go ahead and calculate their difference to find any interesting patterns in movie selection.

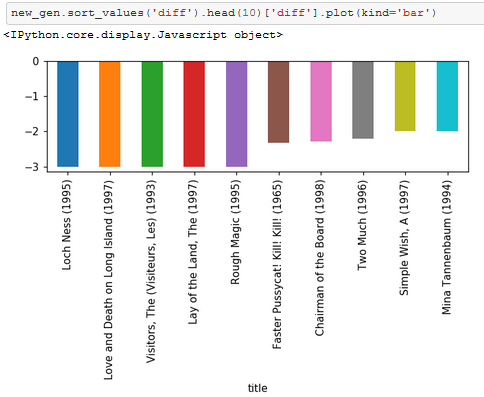


**# Top 10 movies highly rated by Females but not by Males**

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



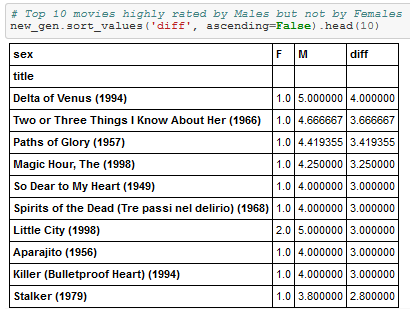
Visual representation of movies:



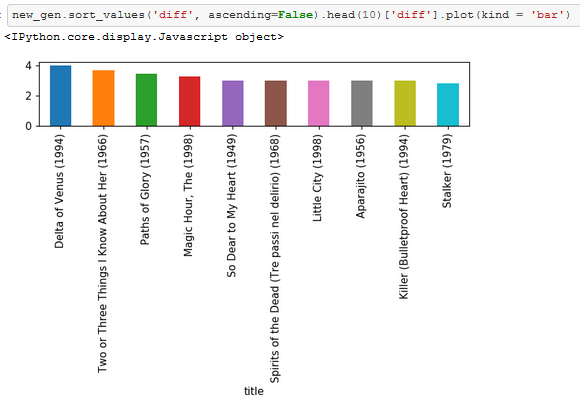
**Interpretation:** We see that ‘Loch Ness’, ‘Love Death and Long Island’ are among the movies that have been rated highly by females than that of males.

**# 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.



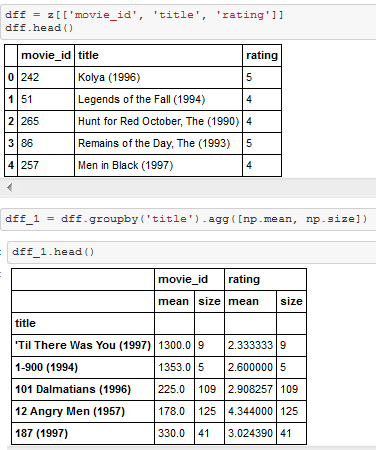
Visual representation of movies:



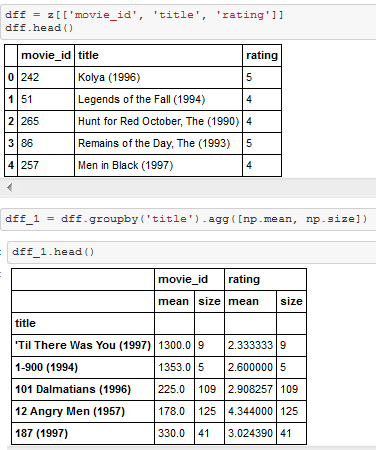
**Interpretation:** We see that ‘Loch Ness’, ‘Love Death and Long Island’ are among the movies that have been rated highly by females than that of males.

**Rating vs Title**

Create a new dataframe with movie\_ids, title and rating.

****

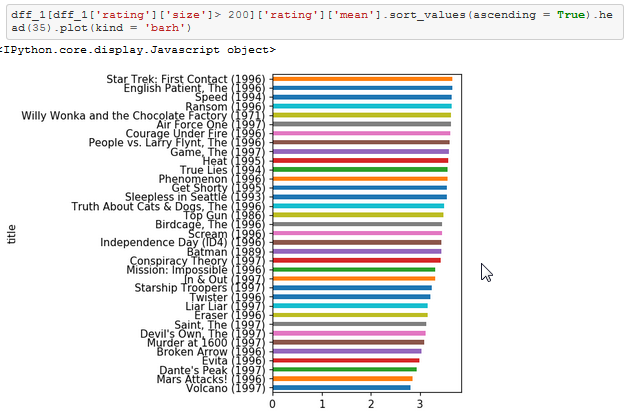
Group the new dataframe by title and aggregate by mean and size as values.

****

**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.

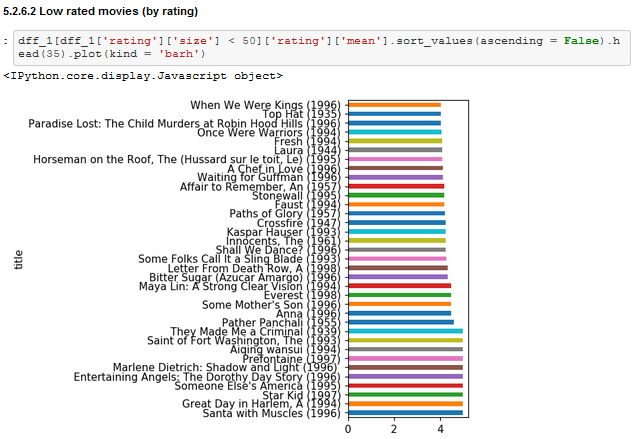
**High rated movies (by rating)**

Visual representation of highly rated movies.



**Low rated movies (by rating)**

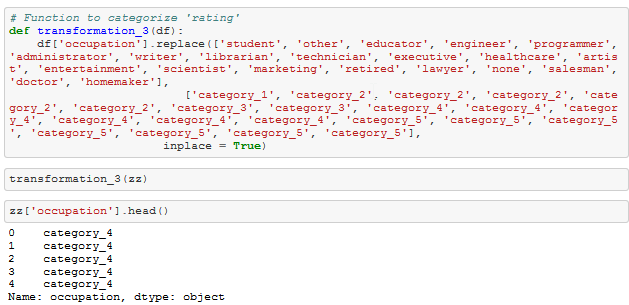
Visual representation of low rated movies.



**6 – Transformations**

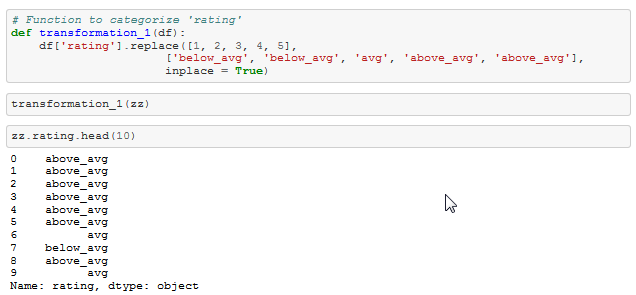
**Replacing occupation with categories:**

Occupation is categorized based on the number of entries



**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’.



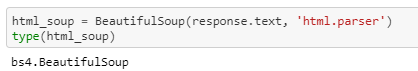
**6. Web Scraping**

**6.1 Beautiful Soup:**

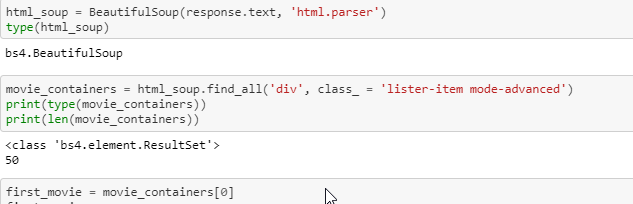
Using Python's Beautiful Soup to get data from IMDB's Top 150 movies



The above code snippet returns an unprettified html text.

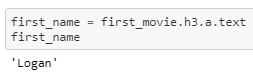


I use html parser to convert html text into beautiful soup object.

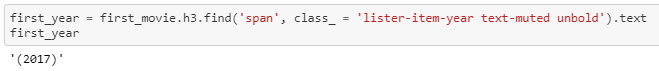


This returns a prettified version of html text.

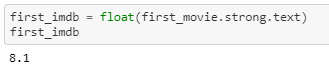
**movie title:**



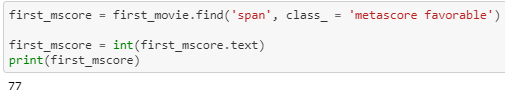
**movie year:**

****

**imdb rating:**

****

**metascore:**

****

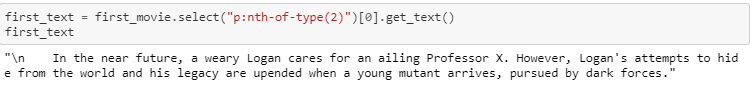
**total votes:**

****

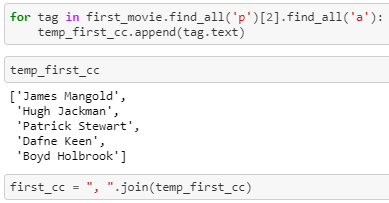
**runtime:**

****

**text:**

****

**cast:**

****

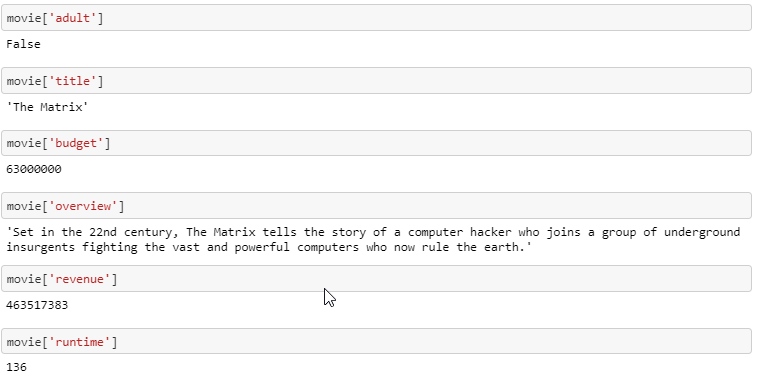
'James Mangold, Hugh Jackman, Patrick Stewart, Dafne Keen, Boyd Holbrook'

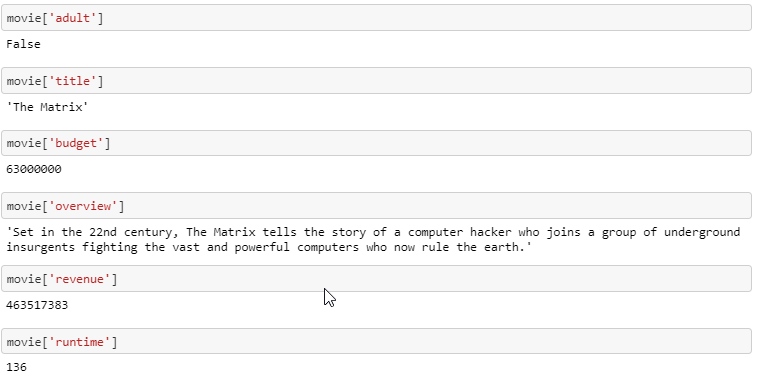
**Cons:** This approach seems tedious and computationally expensive. Also, this requires revisiting the IMDB website once for every request.

**6.2 Tmdbsimple:**

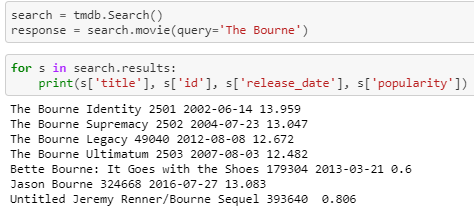
Importing ‘tmdbsimple' and key in the credentials

****

****

****

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



**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) Also, quering with titles take singnificantly 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:

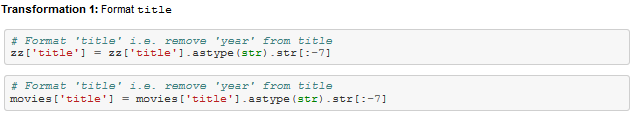
* Get the movielens\_id and title from movielens dataset
* Query TMDB API using movie title to get TMDB\_IDs
* Use queried tmdb\_id to get additional info about the movie

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

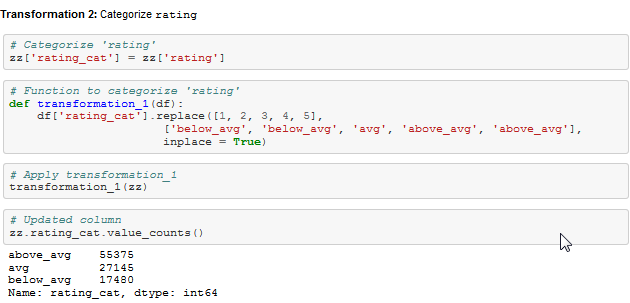
**7. Popularity Based Recommendation**

**7.1.1 Transformations**

**Transformation 1:** Format title



**Transformation 2:** Categorize rating

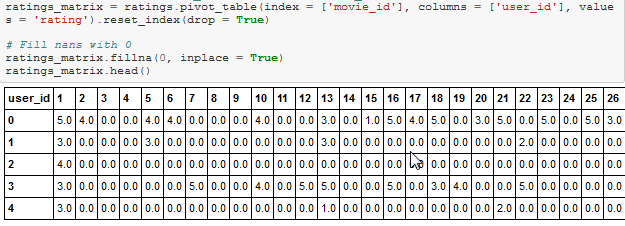


**Transformation 3:** Categorize occupation



**7.1.2 Simple Recommendation System (Popularity based - Ratings)**

Ratings matrix with movie\_id as columns and user\_id as rows and ratings as values



The above matrix has:

Rows – Users  
Columns – Movies  
Values – Ratings



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

**Code Explanation:**

Step 1 –Calculate pairwise distance with respective metric for all the movies

Step 2 – Fill diagonals with 0s

Step 3 – Convert the results (matrix) into a dataframe

Step 4 – Get the index of the movie provided as input.

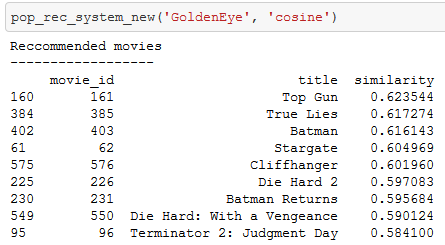
Step 6 – Extract movies and titles and save it as ‘similar movies’

Step 7 – Add a new column ‘similarity’ that has similarity scores based on the metric mentioned

Step 8 – Sort and display top 10 movies.

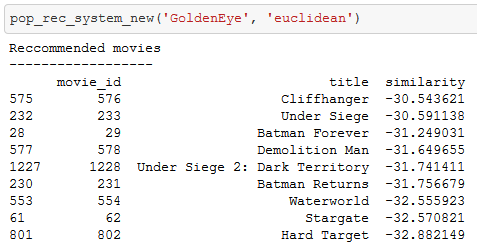
**Cosine Similarity:**

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



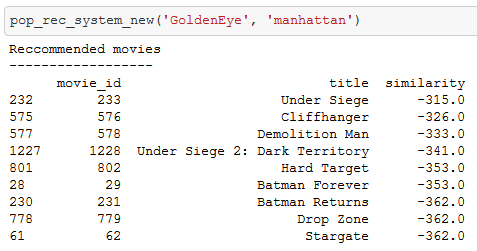
**Euclidean Distance:**

Results for the movie ‘Golden Eye’ using Euclidean distance as a metric.



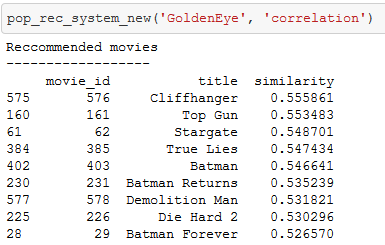
**Manhattan Distance:**

Results for the movie ‘Golden Eye’ using Euclidean distance as a metric.



**Correlation:**

Results for the movie ‘Golden Eye’ using Euclidean distance as a metric.



**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.

**7.1.3 More about metrics used to calculate pairwise distances:**

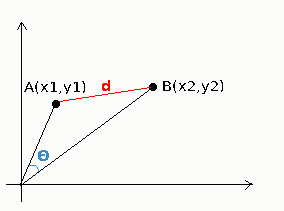
Euclidean distance between two points is calculated by:



Cosine similarity function is defined as:



Considering the below picture:



This is a visual representation of euclidean distance (d) & cosine similarity (θ). While cosine looks at the **angle** between vectors; euclidean distance is like using a ruler to actually measure the distance.

Euclidean distance is very rarely a good distance to choose in Machine Learning and this becomes more obvious in higher dimensions. This is because most of the time in Machine Learning you are not dealing with a Euclidean Metric Space, but a Probabilistic Metric Space and therefore you should be using probabilistic and information theoretic distance functions, e.g. entropy-based ones.

Cosine similarity is generally used as a metric for measuring distance when the magnitude of the vectors does not matter. This happens for example when working with text data represented by word counts. We could assume that when a word (e.g. science) occurs more frequent in document 1 than it does in document 2, that document 1 is more related to the topic of science. However, it could also be the case that we are working with documents of uneven lengths (Wikipedia articles for example). Then, science probably occurred more in document 1 just because it was way longer than document 2. Cosine similarity corrects for this.

Pearson correlation and Cosine Similarity are invariant to scaling, i.e. multiplying all elements by a nonzero constant. Pearson correlation is also invariant to adding any constant to all elements.

For example, if you have two vectors X1 and X2, and your Pearson correlation function is pearson(X1, X2) == pearson(X1, 2 \* X2 + 3).

This is a pretty important property because we often care about if vectors are similar or not but overlook if they **vary in the same way**.

**8 – Content Based Recommendation**

**8.1 Description Based Recommendation:**

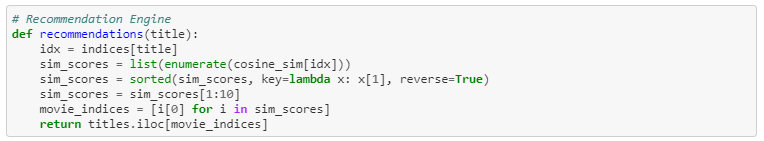
We use three columns for our description-based recommendation:

* overview
* tagline
* description (overview + tagline)

The following code snippet consolidates the individual data and merges into one dataframe.



Python code for Recommendation Engine looks like this:



Step 1 – Extract the indices of the title passed as argument to recommendations (title)

Step 2 – Calculate cosine similarities with respect to the movie index.

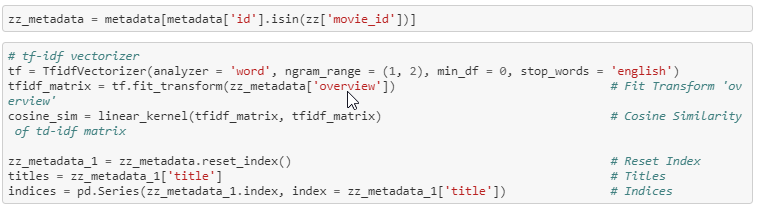
Step 3 – Sorts similarity scores.

Step 4 – Returns the titles of the indices with highest similarity scores.

Creation of the series ‘titles’ and ‘indices’ is show in the below sections. Above steps are revisited using different approaches in the following three subsections.

**8.1.1 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.



The above code snippets demonstrates:

Step 1 – Instantiate a Tf-Idf Vectorizer object

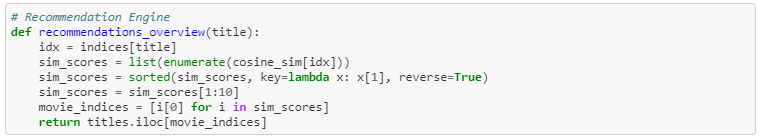
Step 2 – Fit-Transform the tf-idf object to metadata[‘overview’]

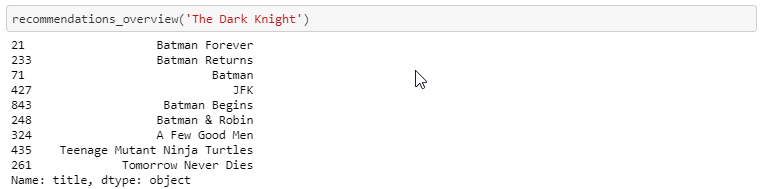
Step 3 – Calculate the consine similarities using linear kernel.

Step 4 – Reset Index

Step 5 – Create a series of titles

Step 6 – Create a series of indices.





**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)

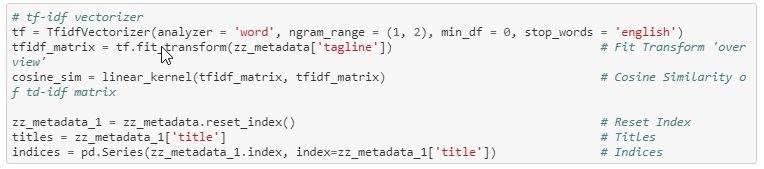
**8.1.2 Recommendation Engine using ‘tagline’:**

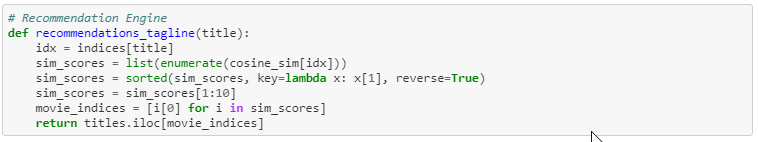
Second recommendation engine considers only the ‘tagline’ of a movie. ‘Tagline’ stands for the extended movie title which certain movies have.

Example: ‘Die Hard 3: With a Vengeance’

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







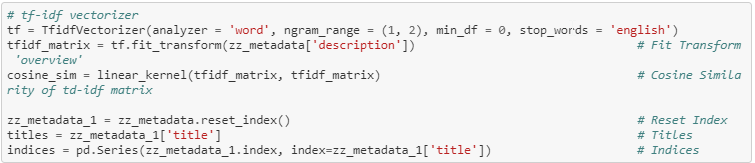


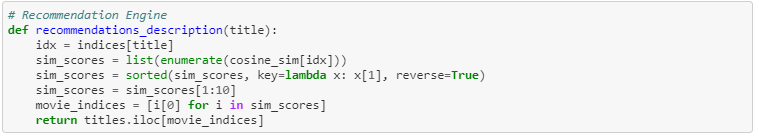
**Interpretation:** The model built with respect to 'tagline' is not as robust as the previous model. It is apparent that the first model (using metadata ['overview']) provides highly similar movies than the model using 'taglines'.

**8.1.3 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’.









**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.

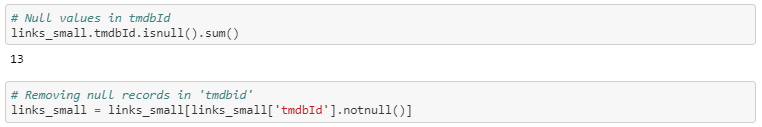
**8.2 Metadata Based Recommendation System**

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.

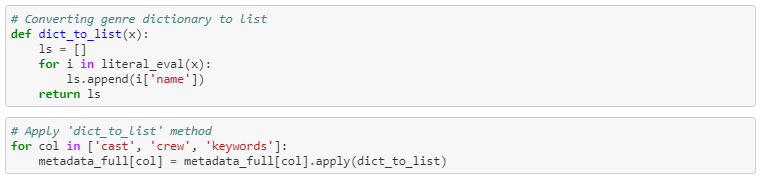


Movie lens data has a file ‘links’ that consists of ‘movie id’, ‘imdb id’ and ‘tmdb id’ using which the data was scraped from the web.

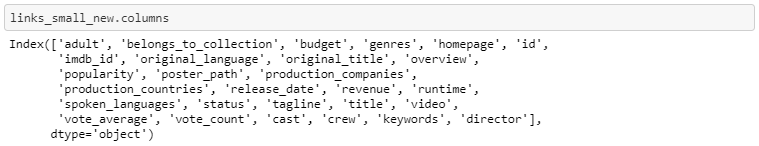
**Missing Values:**



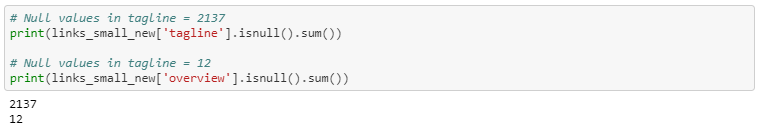
The metadata has column values in dictionary. This can be trickier can handle. Instead of using the dictionary to operate on, I convert the dictionary to a list.



Updated columns after merging movie titles and metadata:

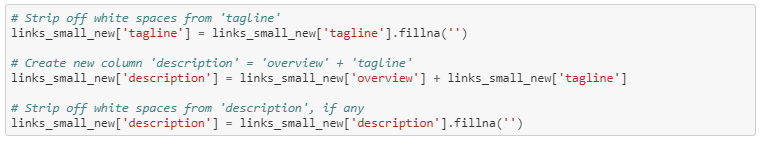


Checking for null values in ‘tagline’ and ‘overview’:



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

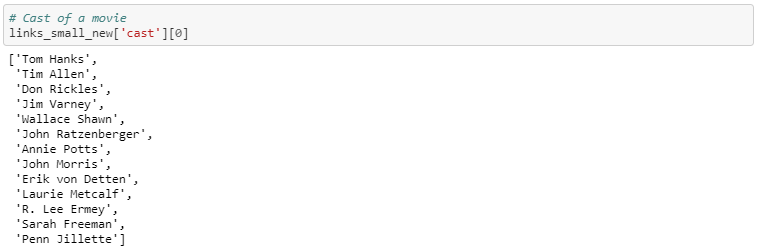
**Solution:** Strip off the white spaces.



**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’:





**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.

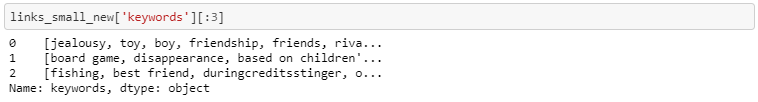
These are steps I follow in the preparation of genres and credits data:

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.

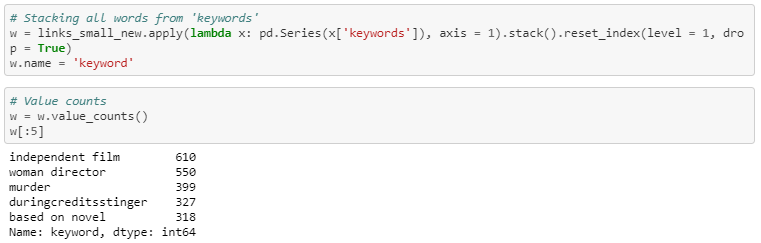


**Keywords:**

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.



Not all words could prove significant.



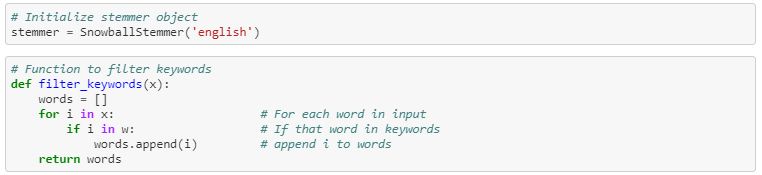
**Note:** Keywords occur in frequencies ranging from 1 to 610. We do not have any use for keywords that occur only once.   
**Interpretation:** Keywords that occur just once.



**Stemming:**

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

Code to perform stemming.



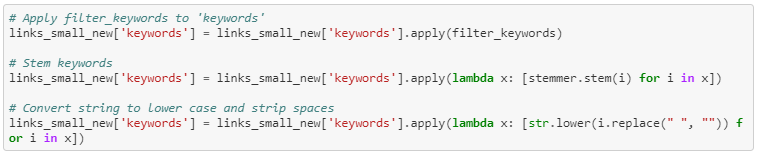
**Preprocess ‘keywords’ column:**

Step 1 – Apply filter words function

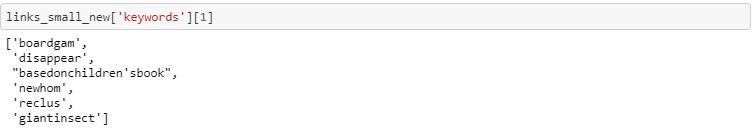
1 – Loop for each word in the input  
 2 – If the word in keywords  
 3 – Add to the temporary list words  
 4 – Returns temporary list

Step 2 – Stem all words

Step 3 – Remove blank spaces

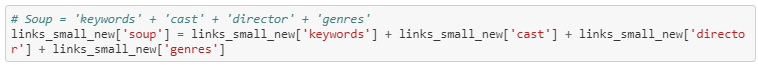


Updated ‘keywords’ column:

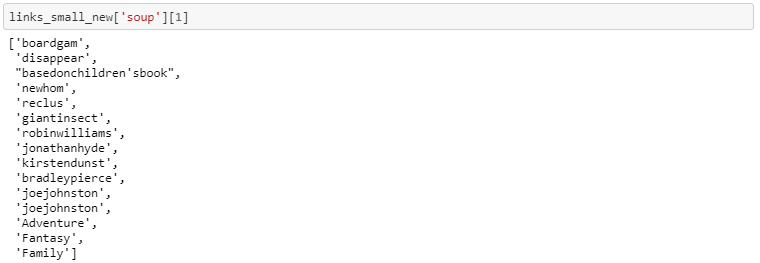


**Soup:**

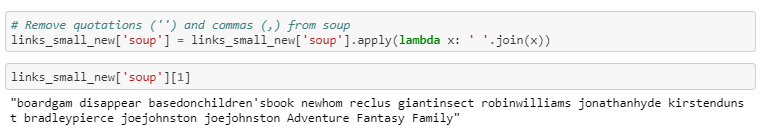
Soup is the metadata of genres, director, cast and keywords.



Soup contains genres, director, cast and keywords.

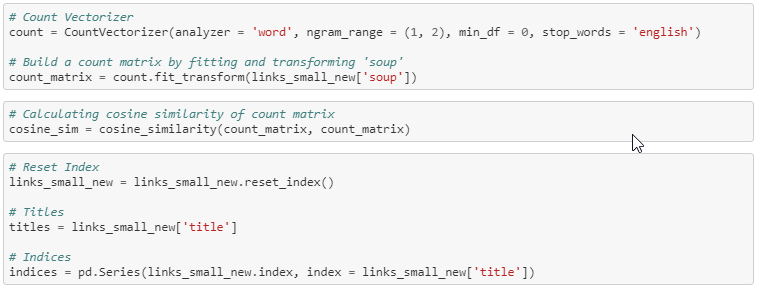


Removing quotations and commas:

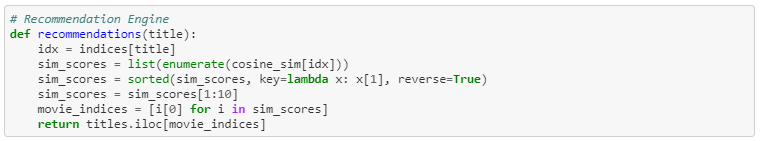


**Count Vectorizer:**

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



Python code for recommendation engine



Recommendations:



**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:**

**IMDB's *weighted rating* formula:**  
[Weighted Rating](https://math.stackexchange.com/questions/169032/understanding-the-imdb-weighted-rating-function-for-usage-on-my-own-website) (WR) =

where,

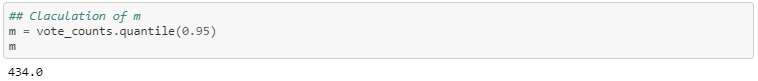
* *v* - number of ratings for the movie
* *m* - number of ratings needed to qualify (usually mean)
* *R* - average rating of the movie
* *C* - mean rating of the population (whole dataset)

Before we could use the above weighted average formula, m and C should be determined.

Calculate c:



Calculate m:

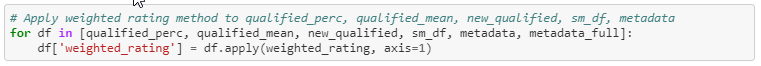


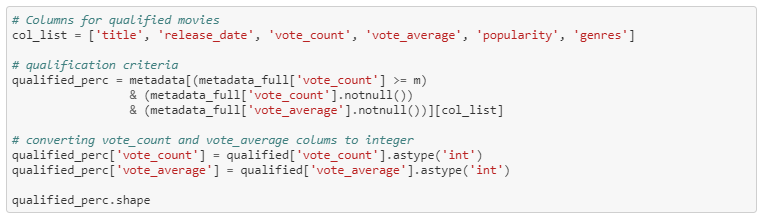
Function to calculate weighted average:



I try three different cutoff criteria:

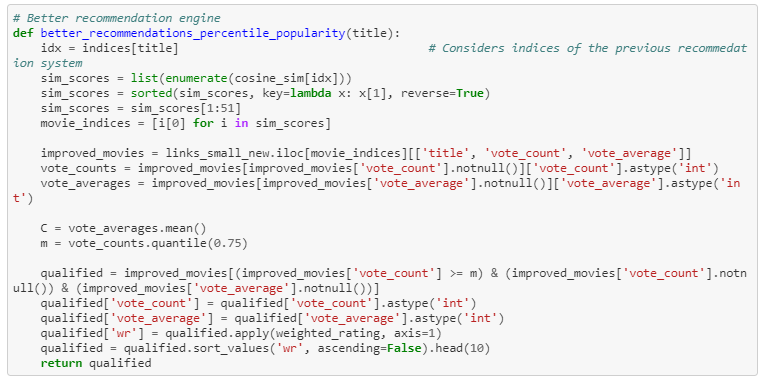
1. 95th percentile
2. Mean
3. No cut-off



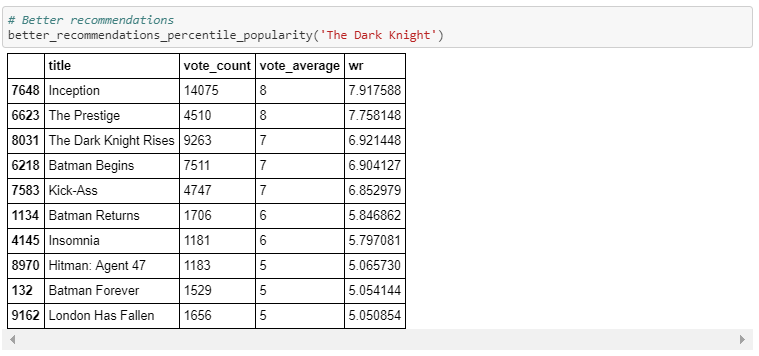


1. **Getting qualified movies (cutoff: 95%)**

Code for recommendation system with movies cutoff 95%

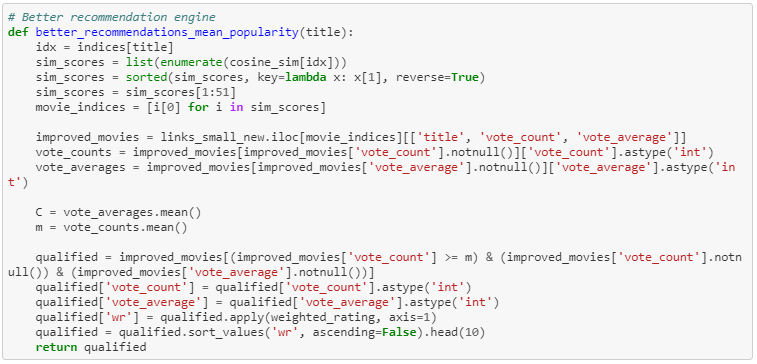


We see that the movies recommended by the engine highly emphasized on the crew (director).

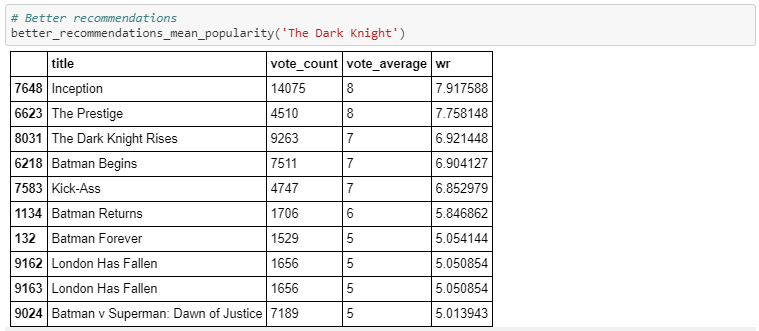


1. **Getting qualified movies (cutoff: mean)**

Code for recommendation system with mean cutoff for movies.

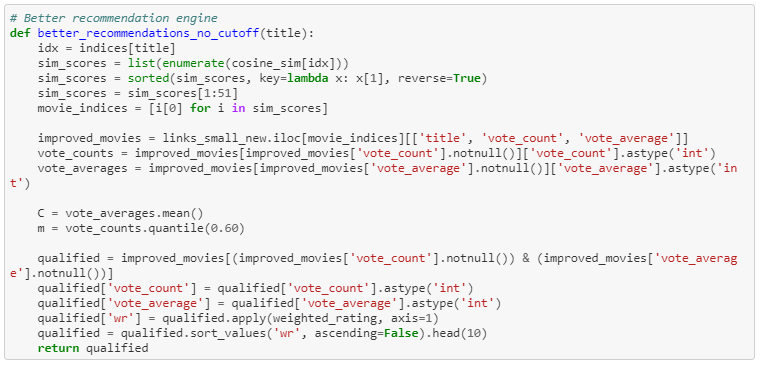


We see that the movies recommended by the new engine includes.

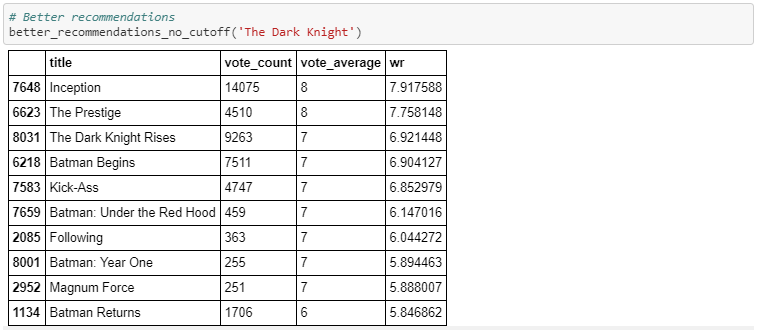


1. **Getting qualified movies (cutoff: none)**

Code for recommendation system with no cutoff for movies.

****

It is evident that the top 5 movies returned by the recommendation system are the same across all three criteria.

****

**9 – Collaborative Filtering**

**9.1 Introduction:**

The Collaborative Filtering Recommender is entirely based on the past behavior and not on the context. More specifically, it is based on the similarity in preferences, tastes and choices of two users. It analyses how similar the tastes of one user is to another and makes recommendations on the basis of that.

For instance, if user A likes movies 1, 2, 3 and user B likes movies 2,3,4, then they have similar interests and A should like movie 4 and B should like movie 1. This makes it one of the most commonly used algorithm as it is not dependent on any additional information.

In general, collaborative filtering is the workhorse of recommender engines. The algorithm has a very interesting property of being able to do feature learning on its own, which means that it can start to learn for itself what features to use. It can be divided into **Memory-Based Collaborative Filtering** and **Model-Based Collaborative filtering**. In this post, I'll only focus on the Memory-Based Collaborative Filtering technique.

Types of collaborative filtering:

1. Item-Based Collaborative Filtering (IB-CF)

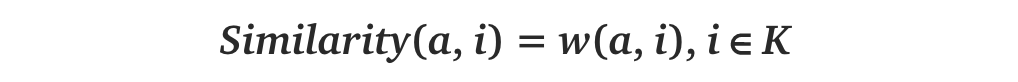
2. User-Based Collaborative Filtering (UB-CF)

**User-Based Collaborative Filtering (UB-CF)**

Imagine that we want to recommend a movie to our friend *Stanley*. We could assume that similar people will have similar taste. Suppose that me and *Stanley* have seen the same movies, and we rated them all almost identically. But Stanley hasn’t seen *‘The Godfather: Part II’*and I did*.*If I love that movie, it sounds logical to think that he will too. With that, we have created an artificial rating based on our similarity.

Well, UB-CF uses that logic and recommends items by finding similar users to the *active user* (to whom we are trying to recommend a movie). A specific application of this is the user-based [Nearest Neighbor algorithm](https://en.wikipedia.org/wiki/K-nearest_neighbor_algorithm). This algorithm needs two tasks:

1. Find the K-nearest neighbors (KNN) to the user ***a,*** using a similarity function ***w*** to measure the distance between each pair of users:



2. Predict the rating that user ***a*** will give to all items the ***k*** neighbors have consumed but ***a*** has not. We Look for the item ***j*** with the best predicted rating.

In other words, we are creating a User-Item Matrix, predicting the ratings on items the active user has not see, based on the other similar users. This technique is **memory-based**.

Filling the blanks

**PROS:**

* Easy to implement.
* Context independent.
* Compared to other techniques, such as content-based, it is more accurate.

**CONS:**

* Sparsity: The percentage of people who rate items is really low.
* Scalability: The more ***K*** neighbors we consider (under a certain threshold), the better my classification should be. Nevertheless, the more users there are in the system, the greater the cost of finding the nearest K neighbors will be.
* Cold-start: New users will have no to little information about them to be compared with other users.
* New item: Just like the last point, new items will lack of ratings to create a solid ranking (More of this on [‘How to sort and rank items’](https://medium.com/@cfpinela/recommender-systems-ranking-and-classifications-3abae71d6fbf)).

**Item-Based Collaborative Filtering (IB-CF)**

Back to *Stanley*. Instead of focusing on his friends, we could focus on what items from all the options are more similar to what we know he enjoys. This new focus is known as Item-Based Collaborative Filtering (IB-CF).

We could divide IB-CF in two sub tasks:

1. Calculate similarity among the items:

* Cosine-Based Similarity
* Correlation-Based Similarity
* Adjusted Cosine Similarity
* 1-Jaccard distance

2. Calculation of Prediction:

* Weighted Sum
* Regression

The difference between UB-CF and this method is that, in this case, we directly pre-calculate the similarity between the co-rated items, skipping K-neighborhood search.

**9.2 Performing Collaborative Filtering:**

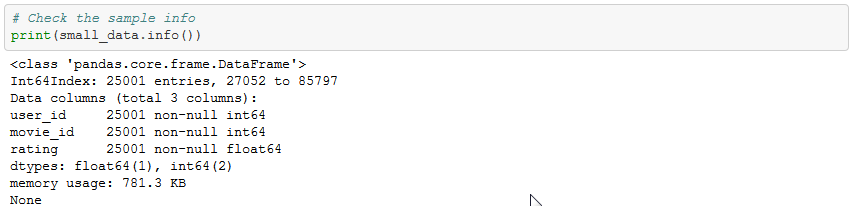


**Note:** Python throws 'Memory' Error when I use the full dataset. Hence, I pick 25% of the dataset and perform collborative filtering on it.

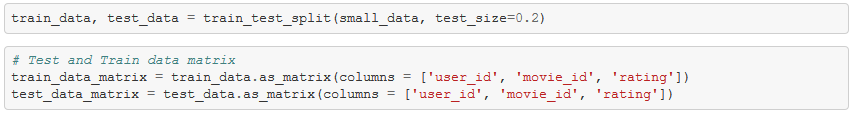
I pick only 25% of the data.



Checking the sample info:

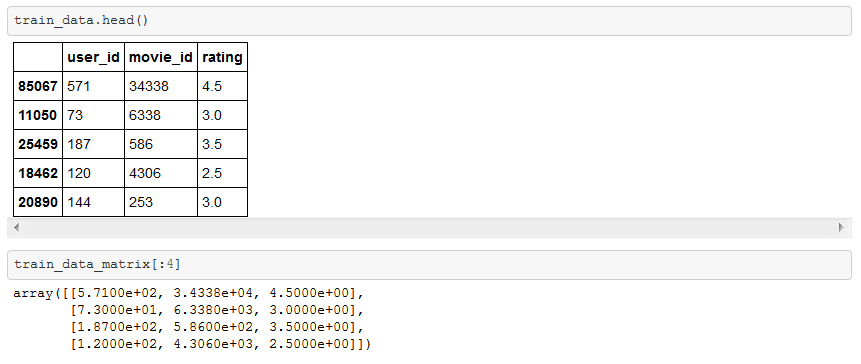


Dividing the data into train and test set:



The train and test dataframes are converted to arrays using .as\_matrix()

Below are the dataframe and matrix versions of train dataset.

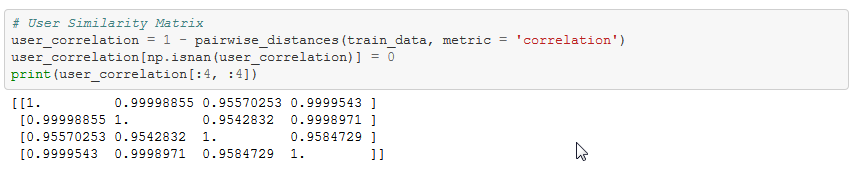


**Idea behind user and item similarity:**

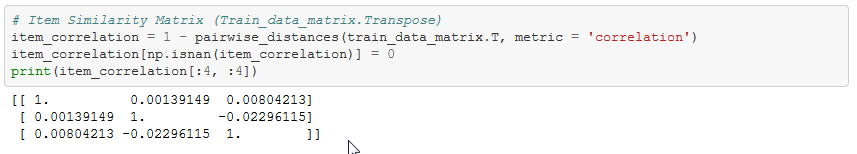
User similarity can be calculated by measuring 'pairwise distances' between ratings datset.

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

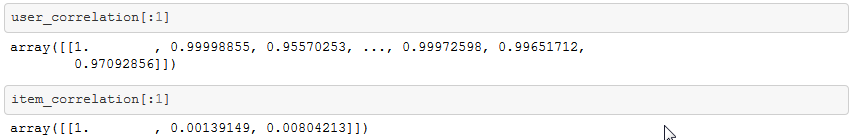
User Similarity Matrix:



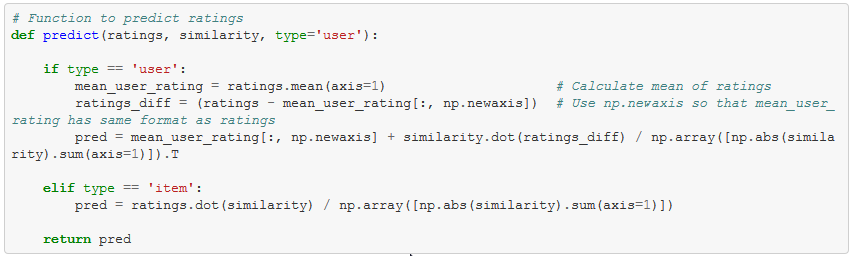
Item Similarity Matrix:



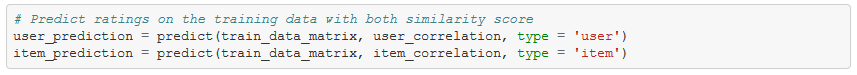
User Correlation and Item Correlation:



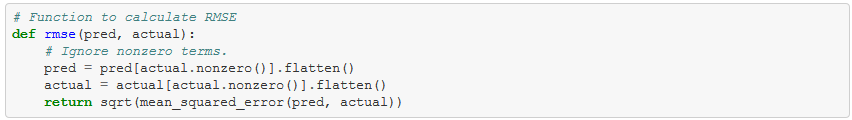
Function to predict ratings



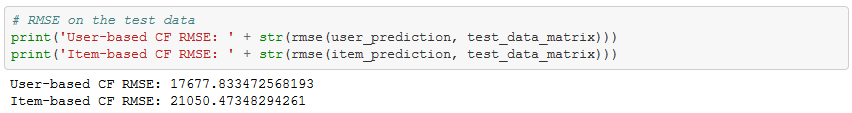
Calling predict function:



Calculate Root Mean Squared Error:



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



**10. 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