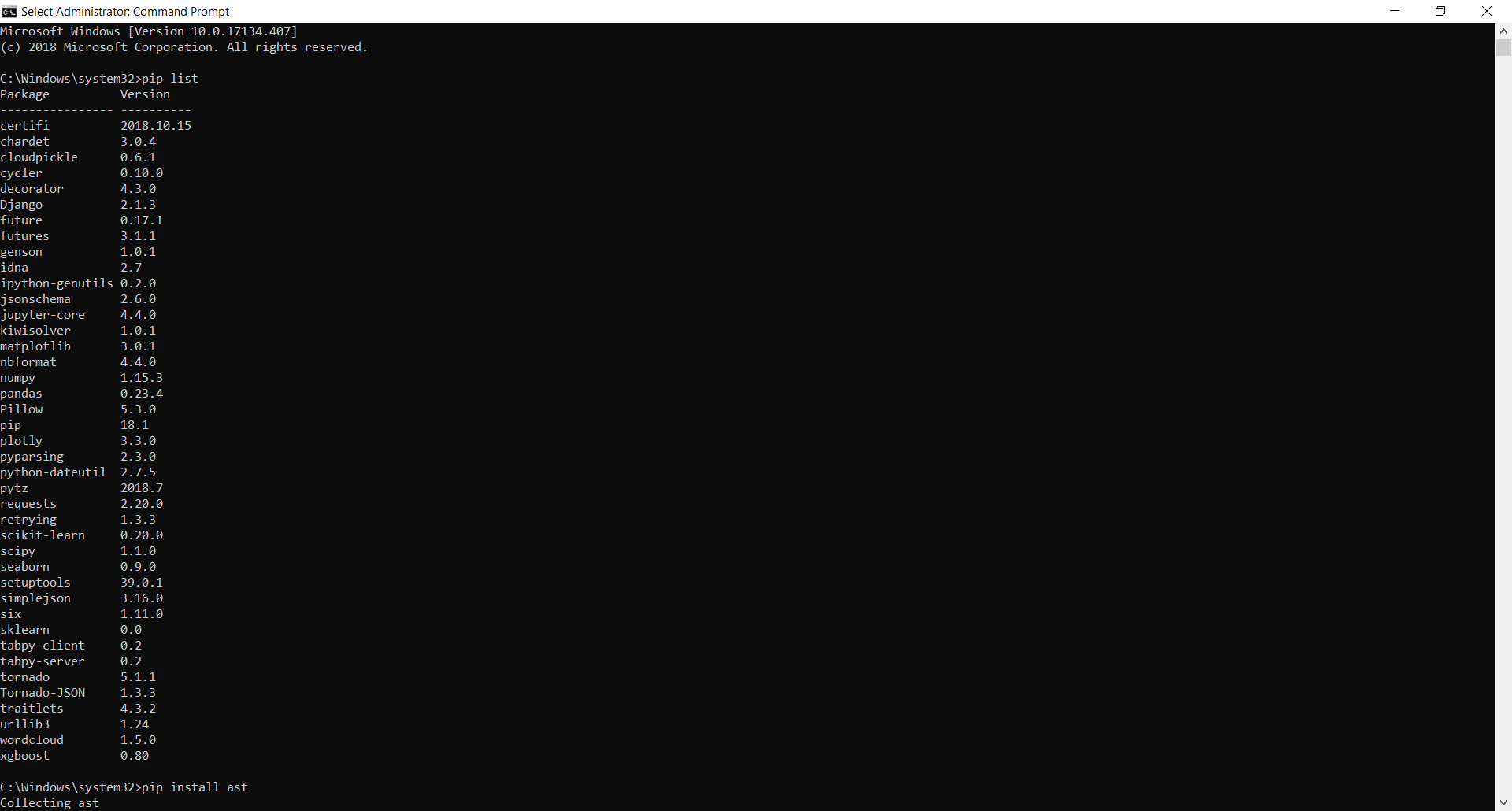
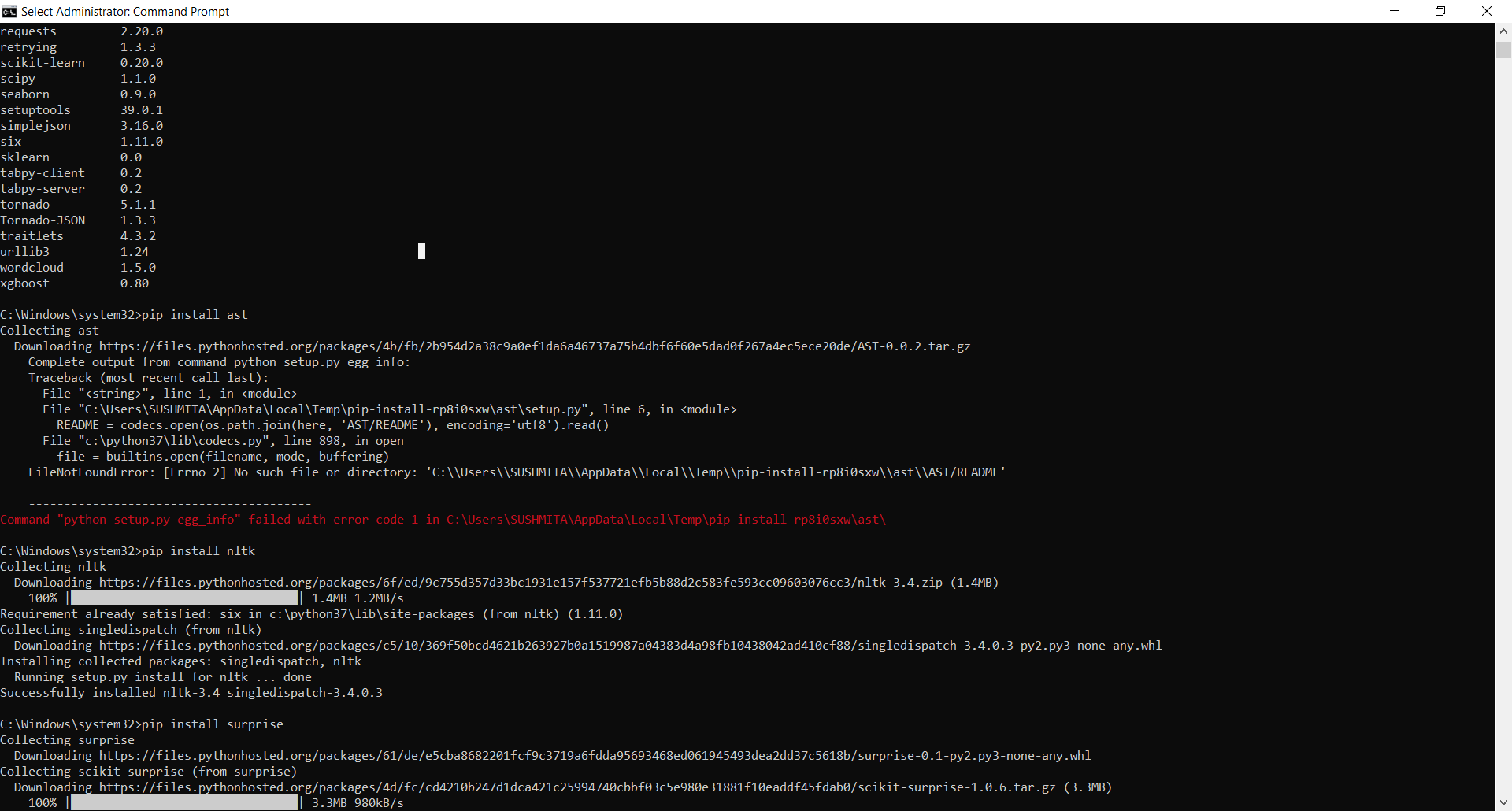
Check list of packages installed: -

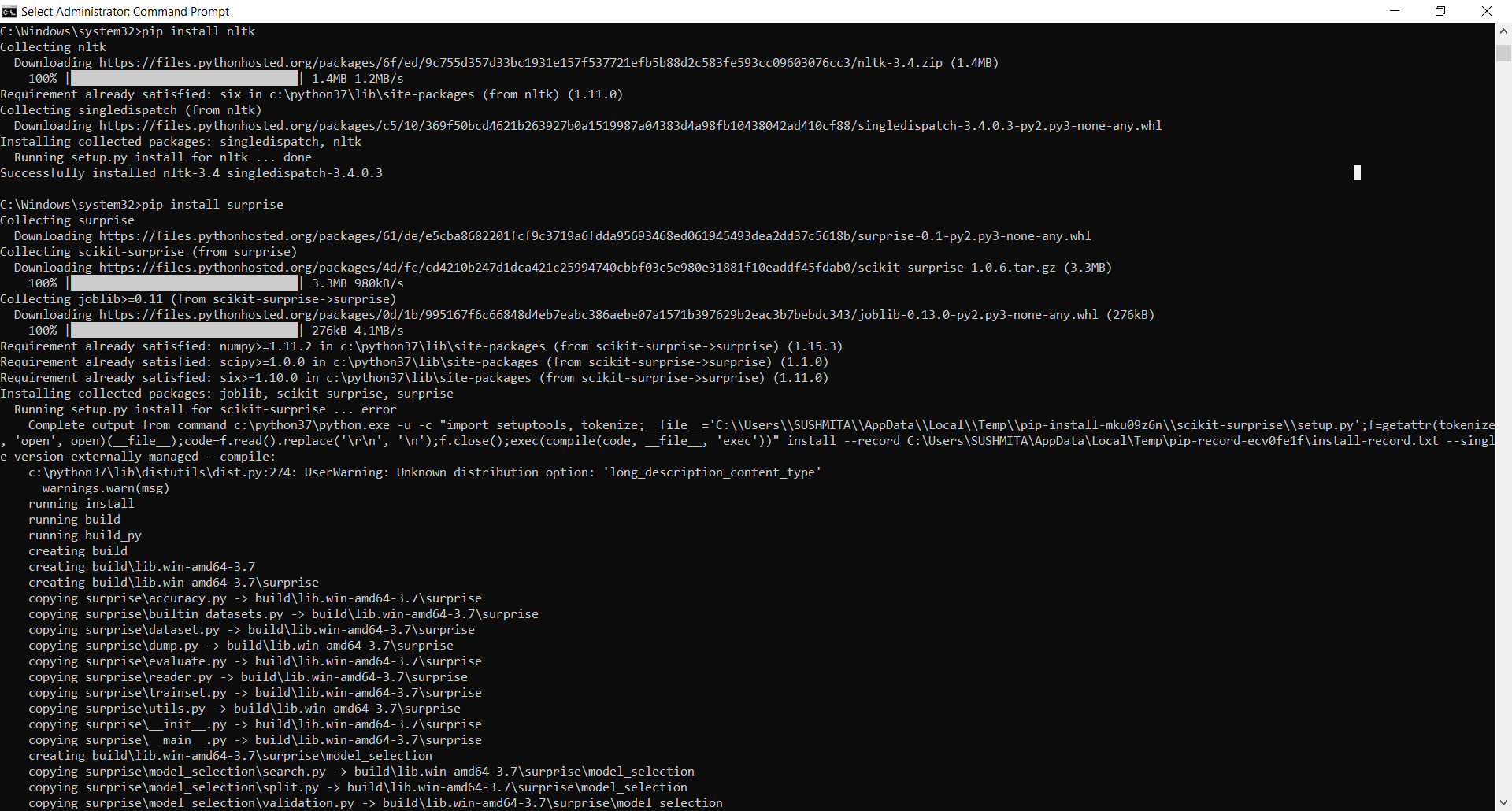
pip list



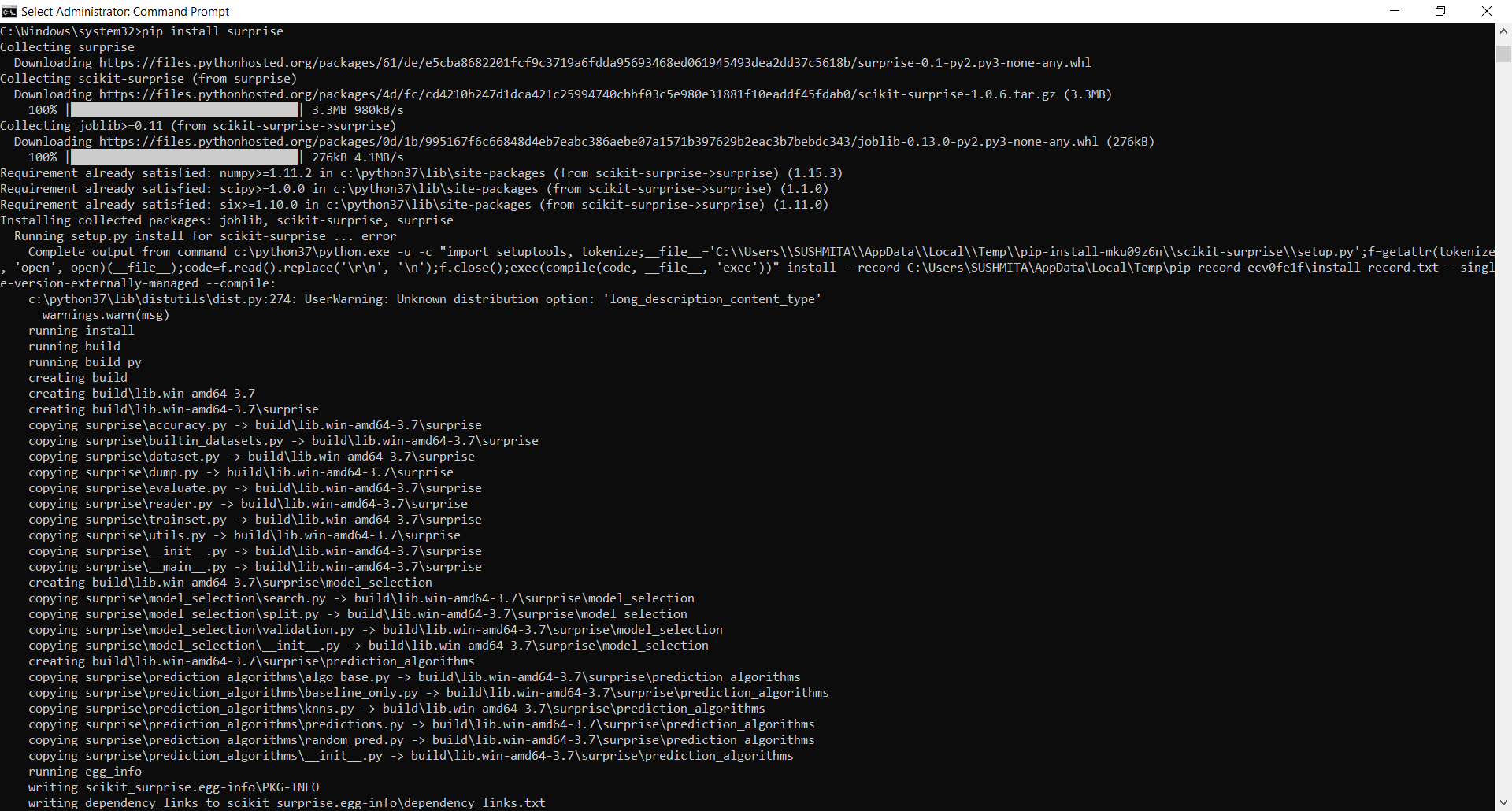
pip install ast // error – did not install

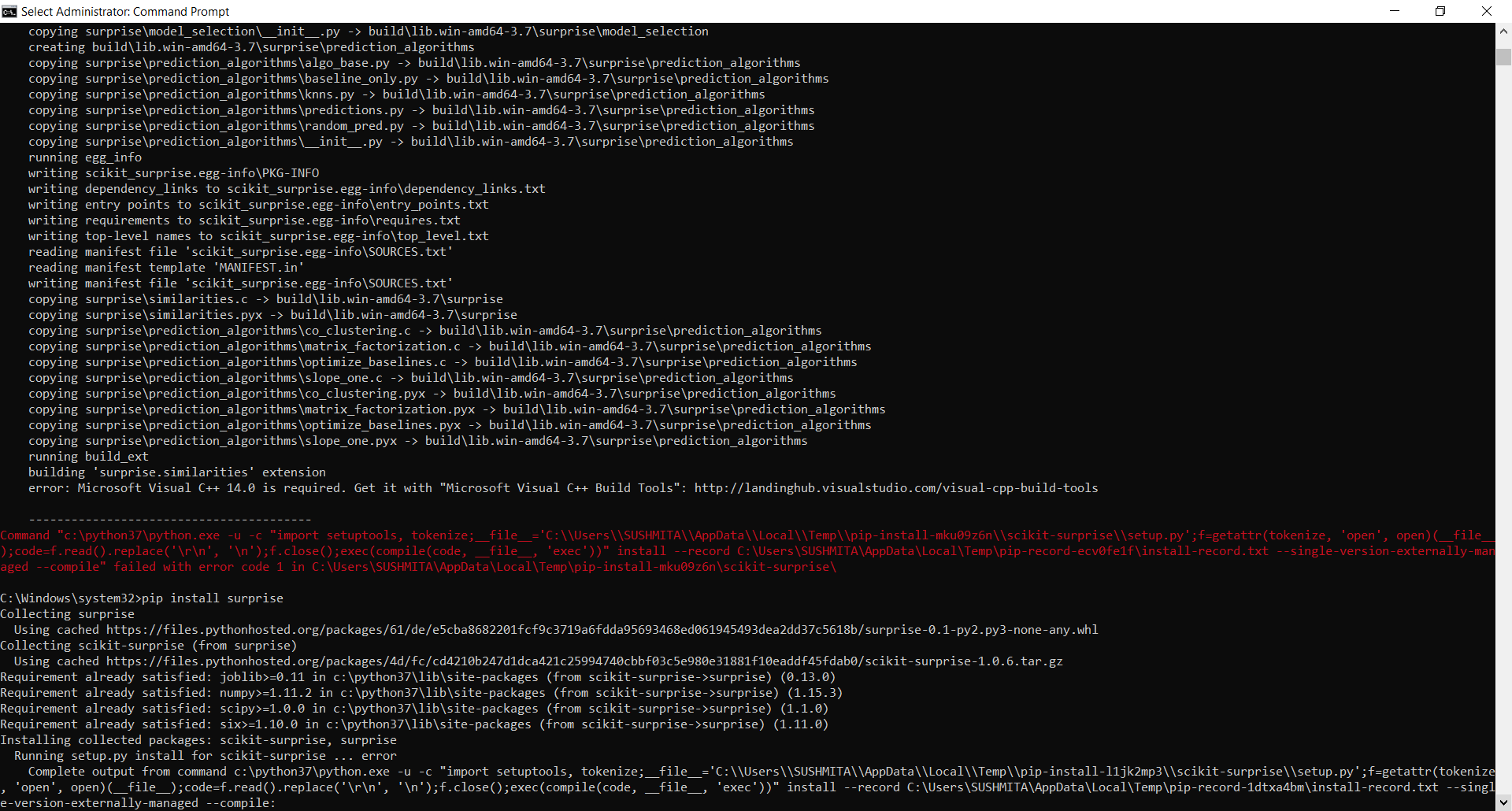


pip install nltk

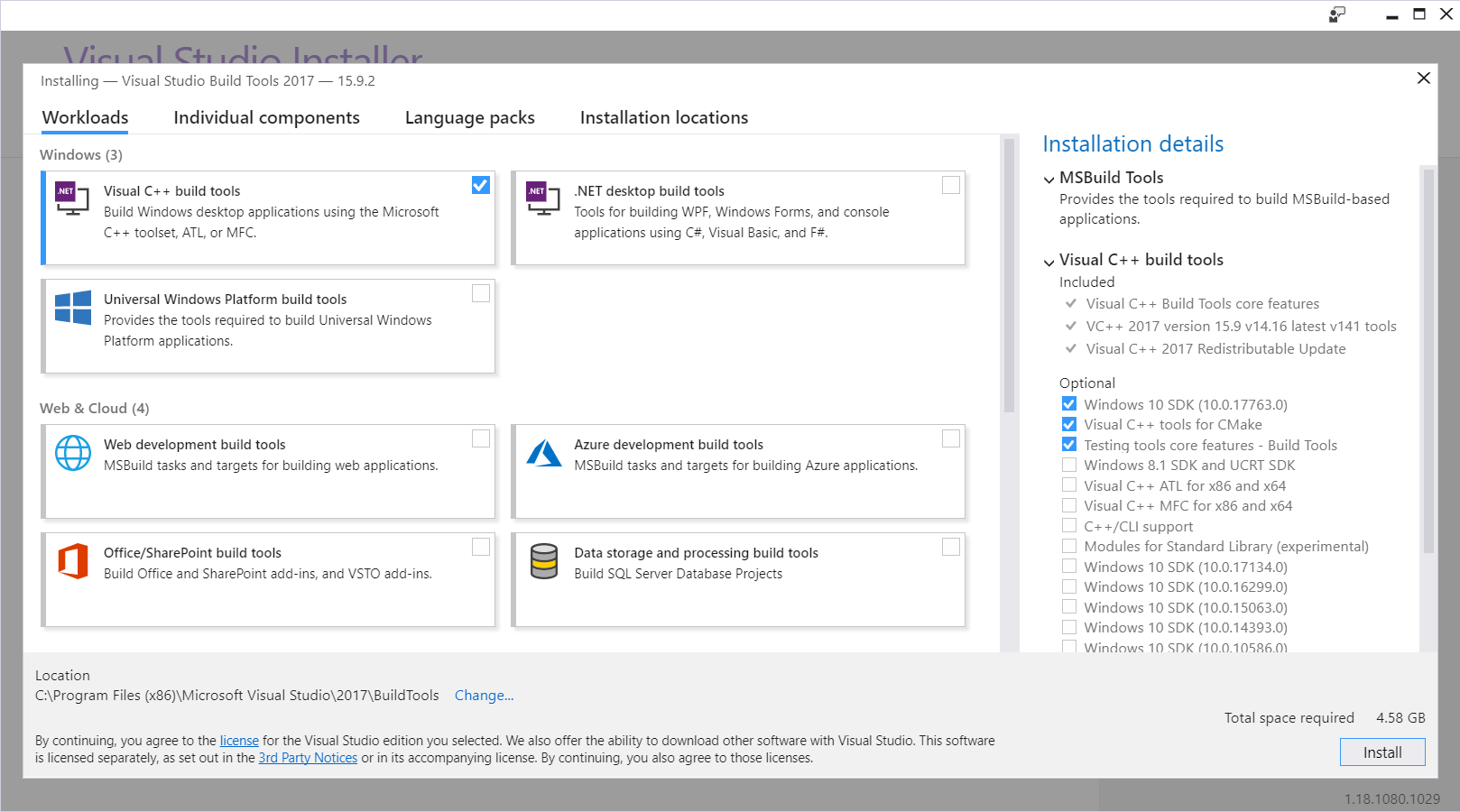


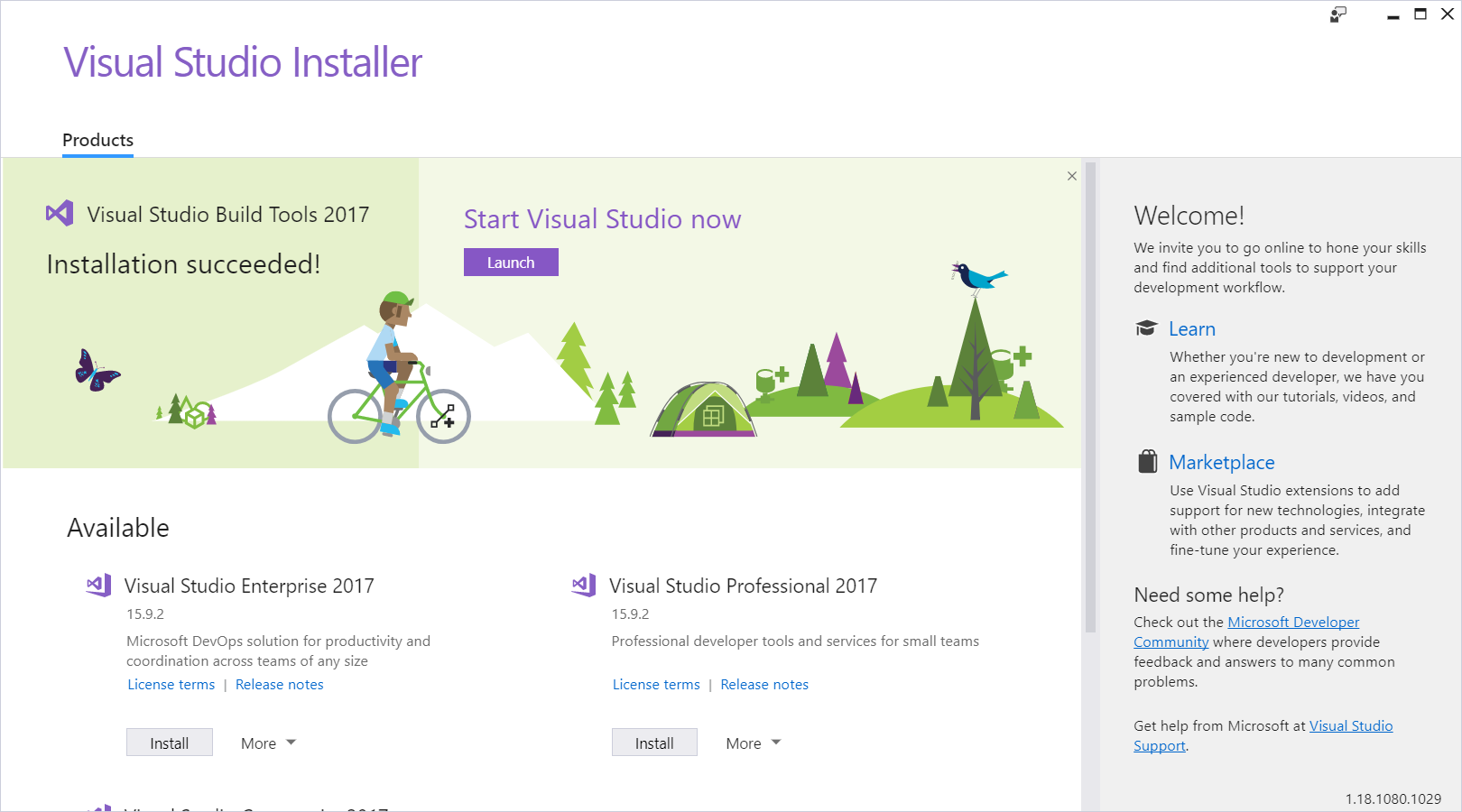
pip install surprise // error – did not install



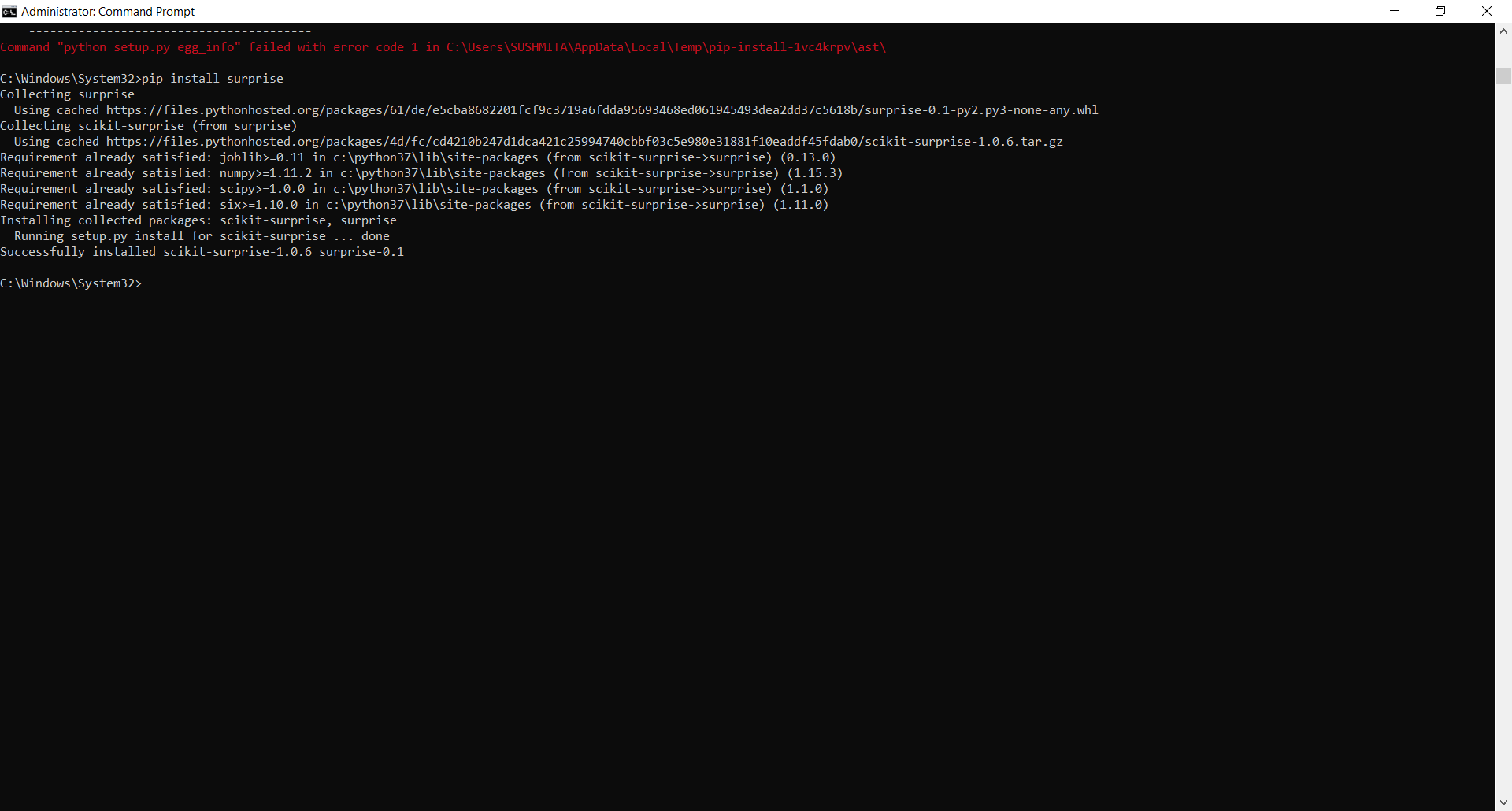


Google - microsoft visual c++ 14.0 -> <https://github.com/benfred/implicit/issues/76> -> <https://visualstudio.microsoft.com/visual-cpp-build-tools/> -> Click “Download Build Tools” -> Build Tools for Visual Studio 2017 (Download)





pip install surprise



pip list



I will build a Simple Recommender using movies from the Full Dataset. As a first step, I will build my simple recommender system.

**Simple Recommender System: -**

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

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

//Loading libraries

**import pandas as pd**

**import numpy as np**

**import matplotlib.pyplot as plt**

**import seaborn as sns**

**from scipy import stats**

**from ast import literal\_eval**

**from sklearn.feature\_extraction.text import TfidfVectorizer, CountVectorizer**

**from sklearn.metrics.pairwise import linear\_kernel, cosine\_similarity**

**from nltk.stem.snowball import SnowballStemmer**

**from nltk.stem.wordnet import WordNetLemmatizer**

**from nltk.corpus import wordnet**

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

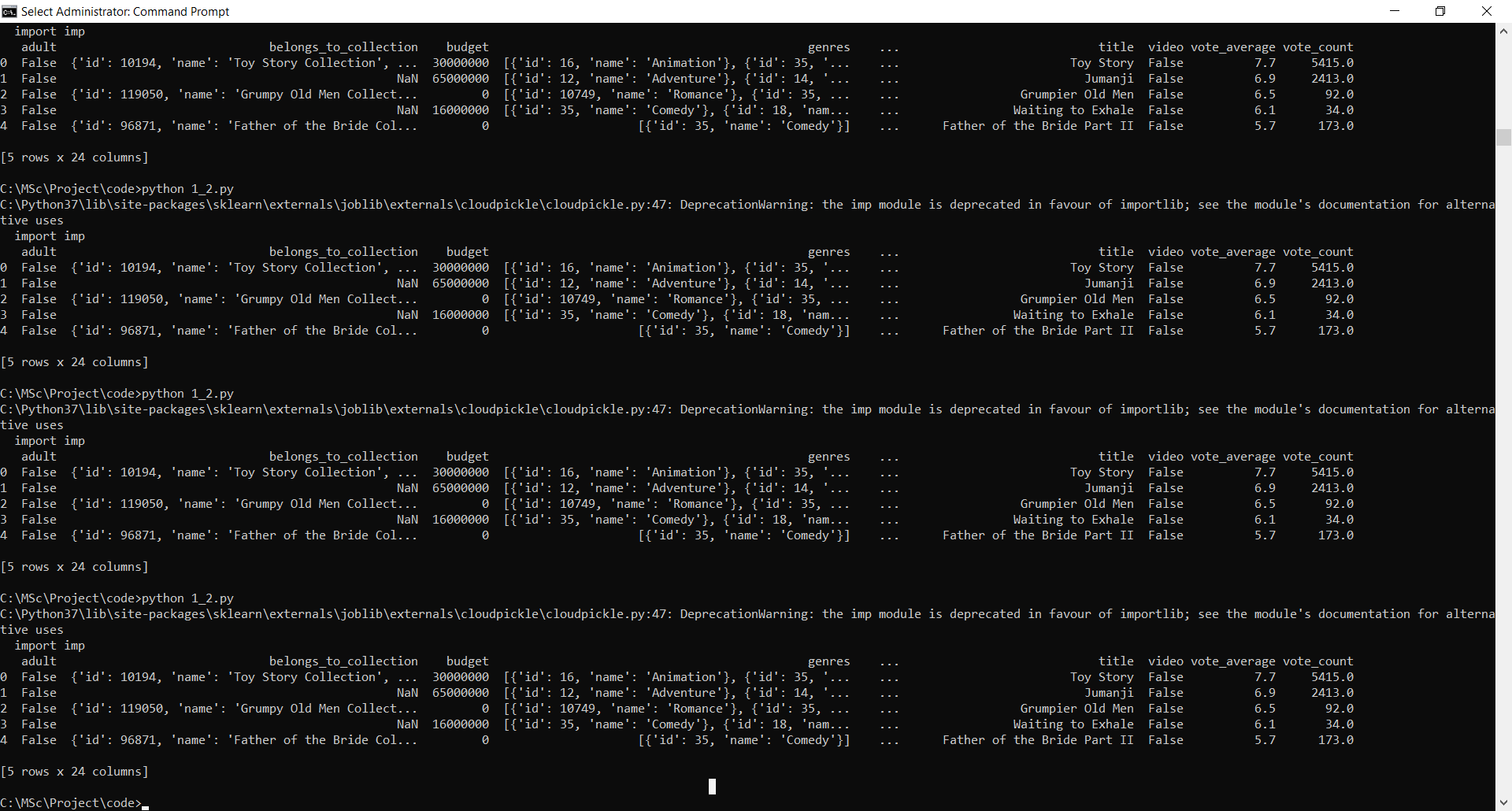
**import warnings; warnings.simplefilter('ignore')**

//Load the required dataset

**md = pd. read\_csv('C:/MSc/Project/input/movies\_metadata.csv')**

//By default, will return the first 5 rows of the dataset

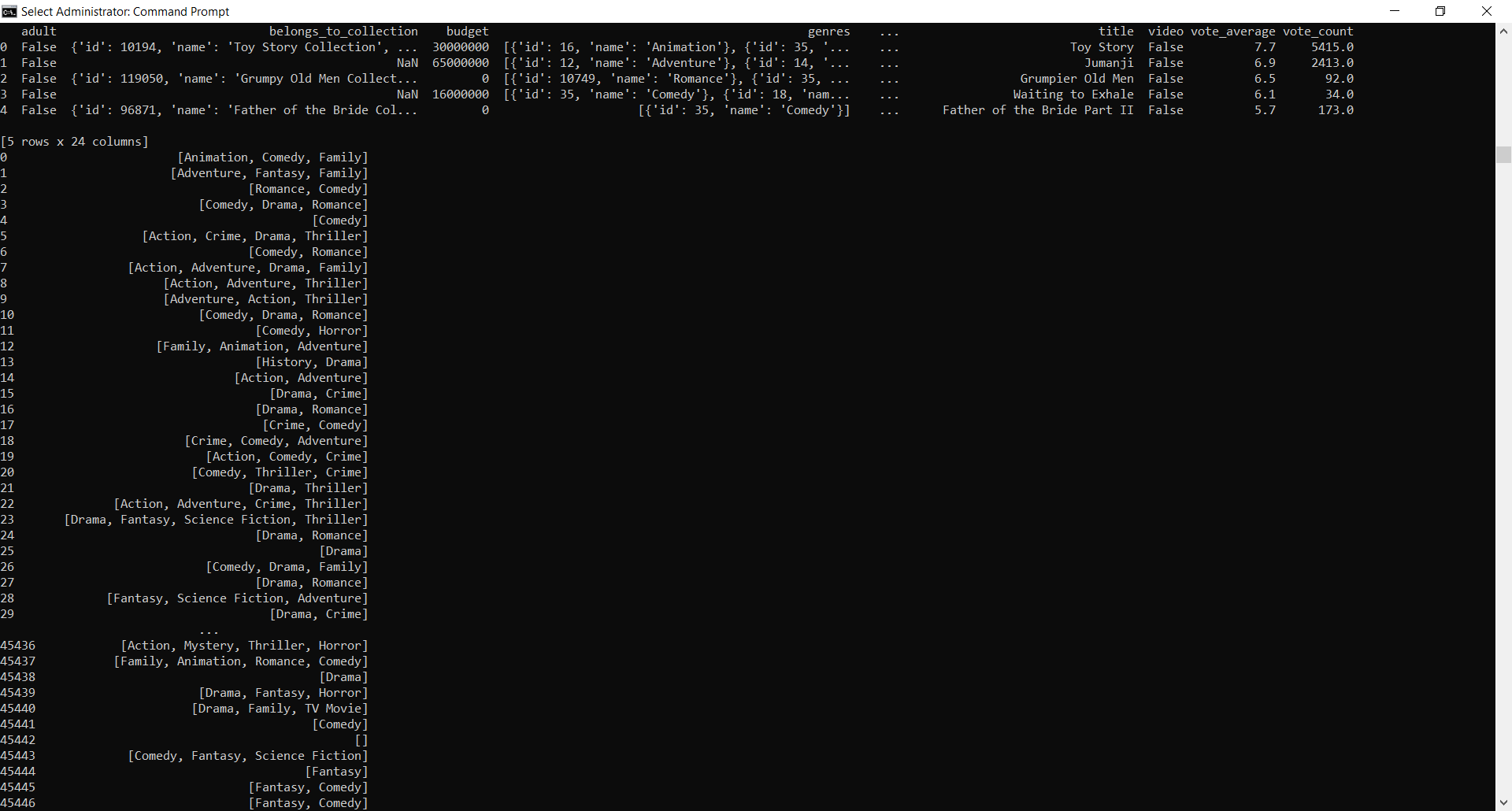
**print(md.head())**



**md['genres'] = md['genres'].fillna('[]').apply(literal\_eval).apply(lambda x: [i['name'] for i in x] if isinstance(x, list) else [])**

//If you print this, the result is as below for all data:-

// print(md['genres'])



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



where,

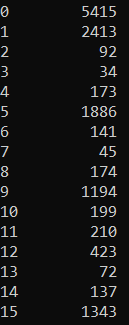
* *v* is the number of votes for the movie
* *m* is the minimum votes required to be listed in the chart
* *R* is the average rating of the movie
* *C* is the mean vote across the whole report

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

I will build our overall Top 250 Chart and will define a function to build charts for a particular genre. Let's begin!

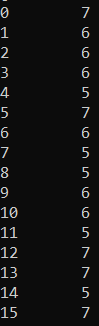
vote\_counts = md[md['vote\_count'].notnull()]['vote\_count'].astype('int')

//Use print(vote\_counts) for output. Gives for all data



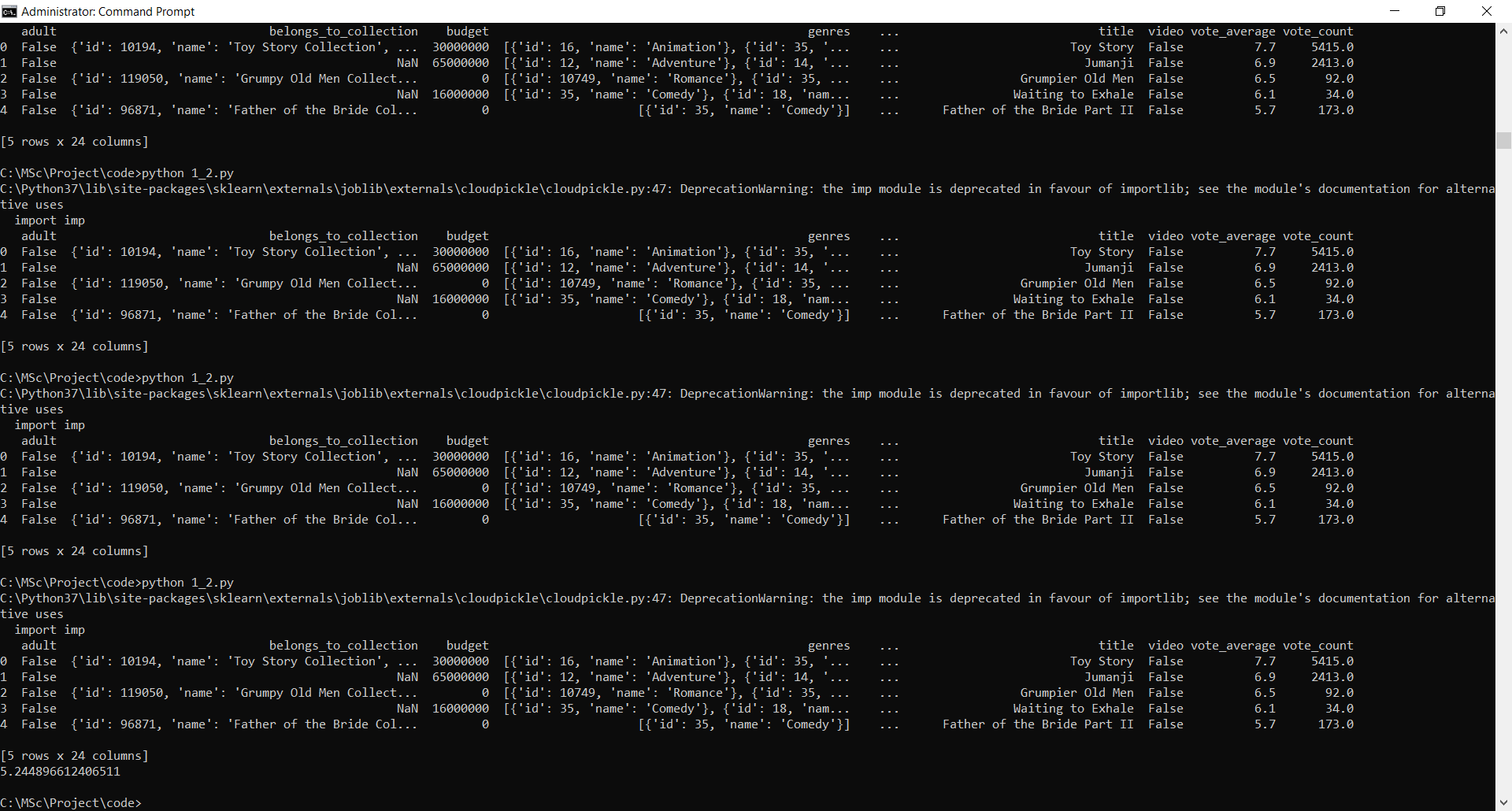
vote\_averages = md[md['vote\_average'].notnull()]['vote\_average'].astype('int')

//Use print(vote\_averages) for output. Gives for all data



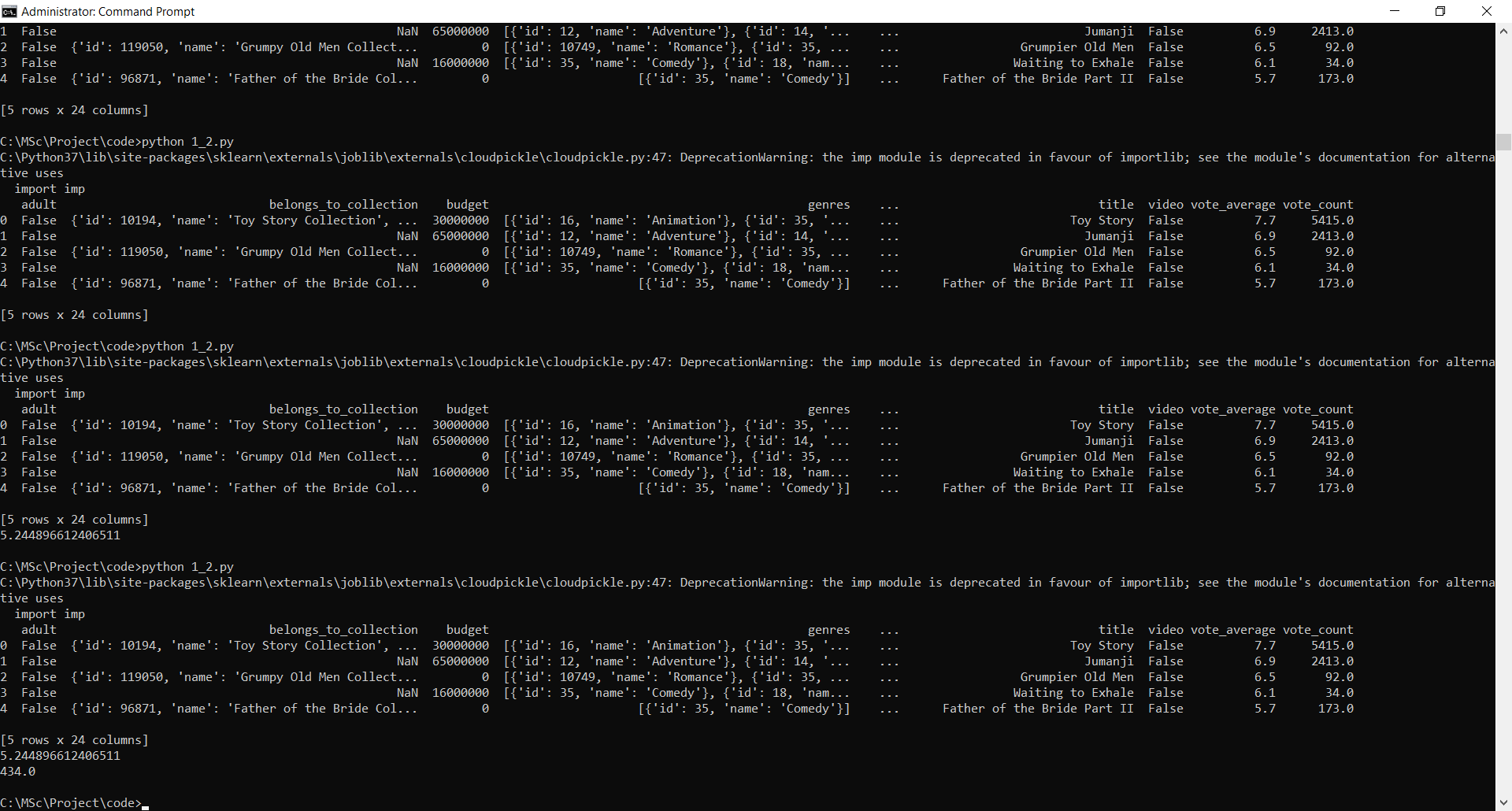
C = vote\_averages.mean() //Remove the decimal part and do floor of each value manually

print(C)

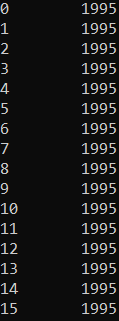


m = vote\_counts.quantile(0.95) // (5 \* 100) / 45467 = 2273.35

print(m)



md['year'] = pd.to\_datetime(md['release\_date'], errors='coerce').apply(lambda x: str(x).split('-')[0] if x != np.nan else np.nan) //Returns all data’s year from released date.



qualified = md[(md['vote\_count'] >= m) & (md['vote\_count'].notnull()) & (md['vote\_average'].notnull())][['title', 'year', 'vote\_count', 'vote\_average', 'popularity', 'genres']]

//Printing the qualified 2274 data with 6 columns using print(qualified)

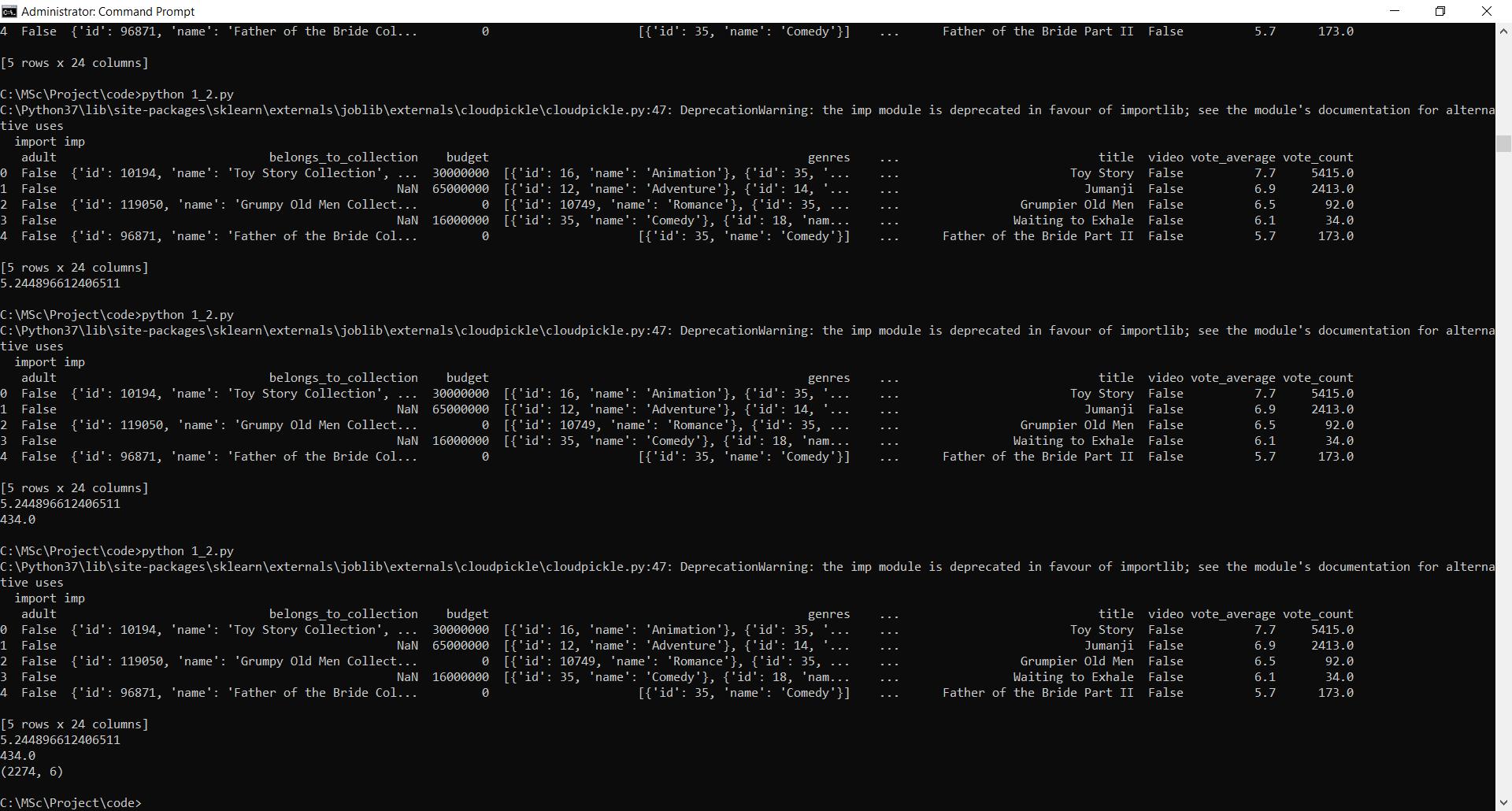
qualified['vote\_count'] = qualified['vote\_count'].astype('int')

// If you do print(qualified['vote\_count']) then it shows those vote\_count which have qualified i.e. of those 2274 data it will show vote\_count column values only.

qualified['vote\_average'] = qualified['vote\_average'].astype('int')

// If you do print(qualified['vote\_average']) then it shows those vote\_average which have qualified i.e. of those 2274 data it will show vote\_average column values only.

print(qualified.shape) //Shows the data row and column count of the qualified data.



Therefore, to qualify to be considered for the chart, a movie has to have at least **434 votes** on TMDB. We also see that the average rating for a movie on TMDB is **5.244** on a scale of 10. **2274** Movies qualify to be on our chart. (95% of )

def weighted\_rating(x): // function to calculated weighted ratio using the formula

v = x['vote\_count']

R = x['vote\_average']

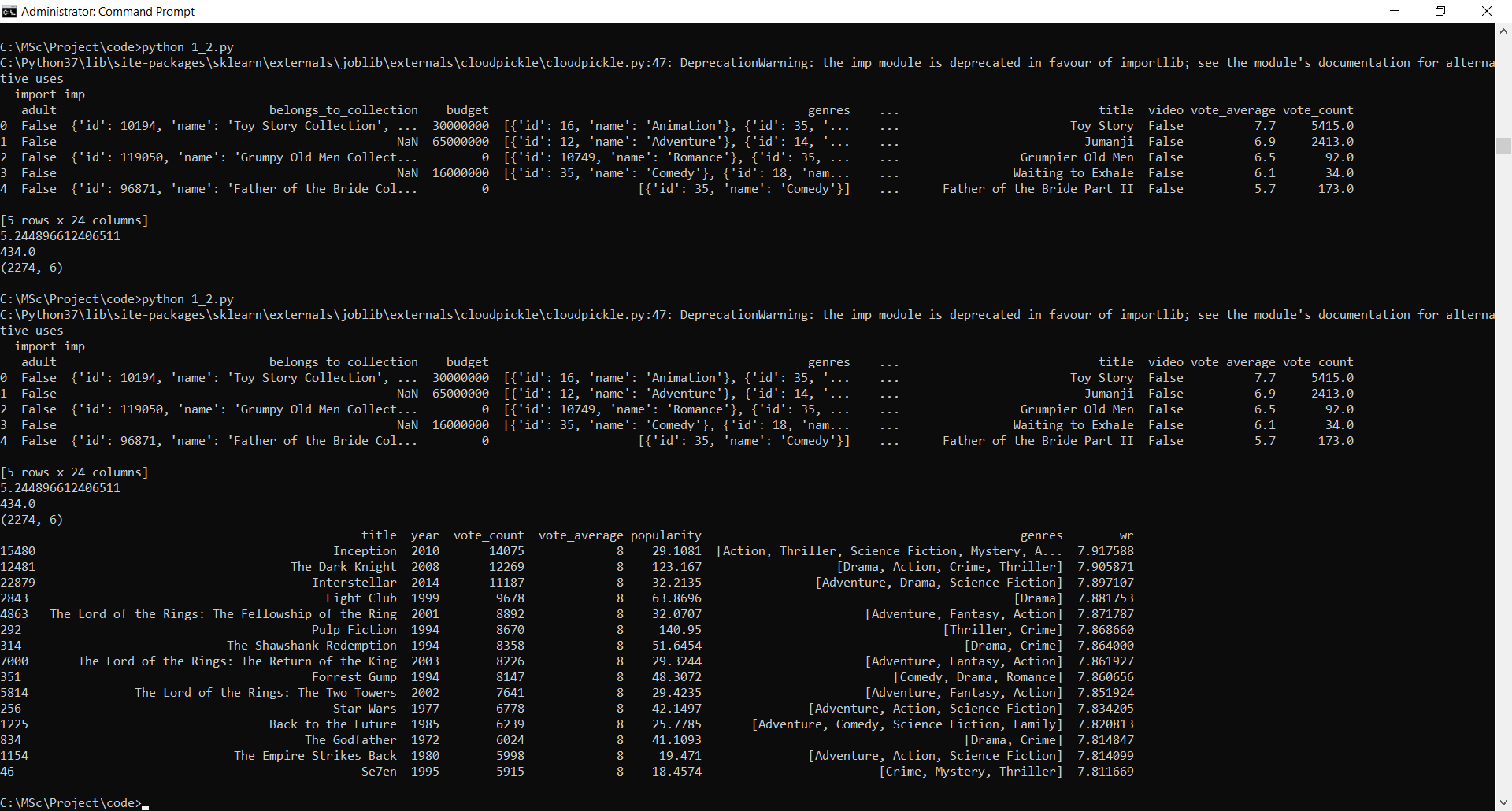
return (v/(v+m) \* R) + (m/(m+v) \* C)

qualified['wr'] = qualified.apply(weighted\_rating, axis=1) //all qualified movie’s weighted ratio

qualified = qualified.sort\_values('wr', ascending=False).head(250) //give me top 250 movies sorted in descending order according to weighted ratio

### **Top Movies:-**

print(qualified.head(15)) //Of the above 250, give me top 15



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

Let us now construct our function that builds charts for particular genres. For this, we will use relax our default conditions to the **85th** percentile instead of 95.

s = md.apply(lambda x: pd.Series(x['genres']),axis=1).stack().reset\_index(level=1, drop=True)

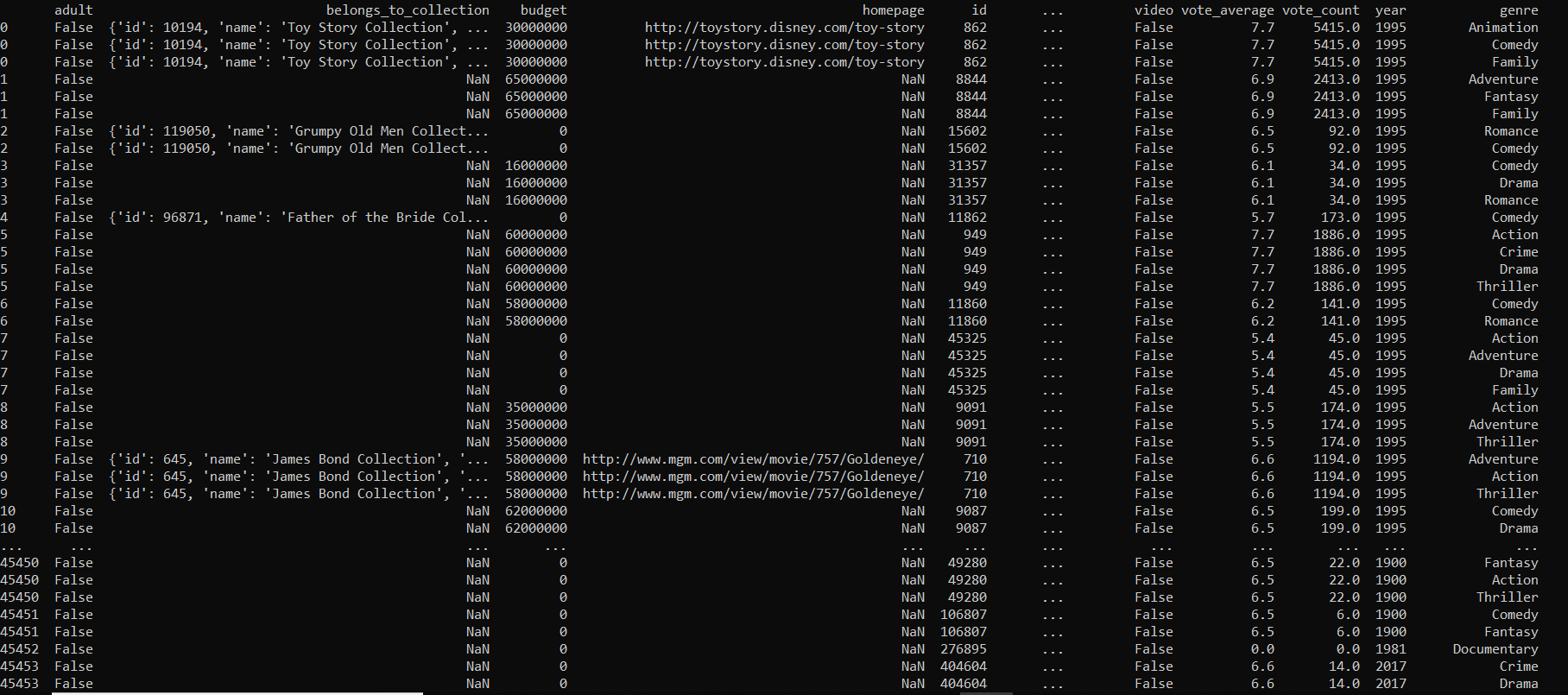
//If you do print(s), then it returns the genre of each movie (for all the movies) as:-



s.name = 'genre' // variable set to genre

gen\_md = md.drop('genres', axis=1).join(s)

//If you do print(gen\_md), then the following is the output for all the data:-



def build\_chart(genre, percentile=0.85):

df = gen\_md[gen\_md['genre'] == genre]

vote\_counts = df[df['vote\_count'].notnull()]['vote\_count'].astype('int')

vote\_averages = df[df['vote\_average'].notnull()]['vote\_average'].astype('int')

C = vote\_averages.mean()

m = vote\_counts.quantile(percentile)

qualified = df[(df['vote\_count'] >= m) & (df['vote\_count'].notnull()) & (df['vote\_average'].notnull())][['title', 'year', 'vote\_count', 'vote\_average', 'popularity']]

qualified['vote\_count'] = qualified['vote\_count'].astype('int')

qualified['vote\_average'] = qualified['vote\_average'].astype('int')

qualified['wr'] = qualified.apply(lambda x: (x['vote\_count']/(x['vote\_count']+m) \* x['vote\_average']) + (m/(m+x['vote\_count']) \* C), axis=1)

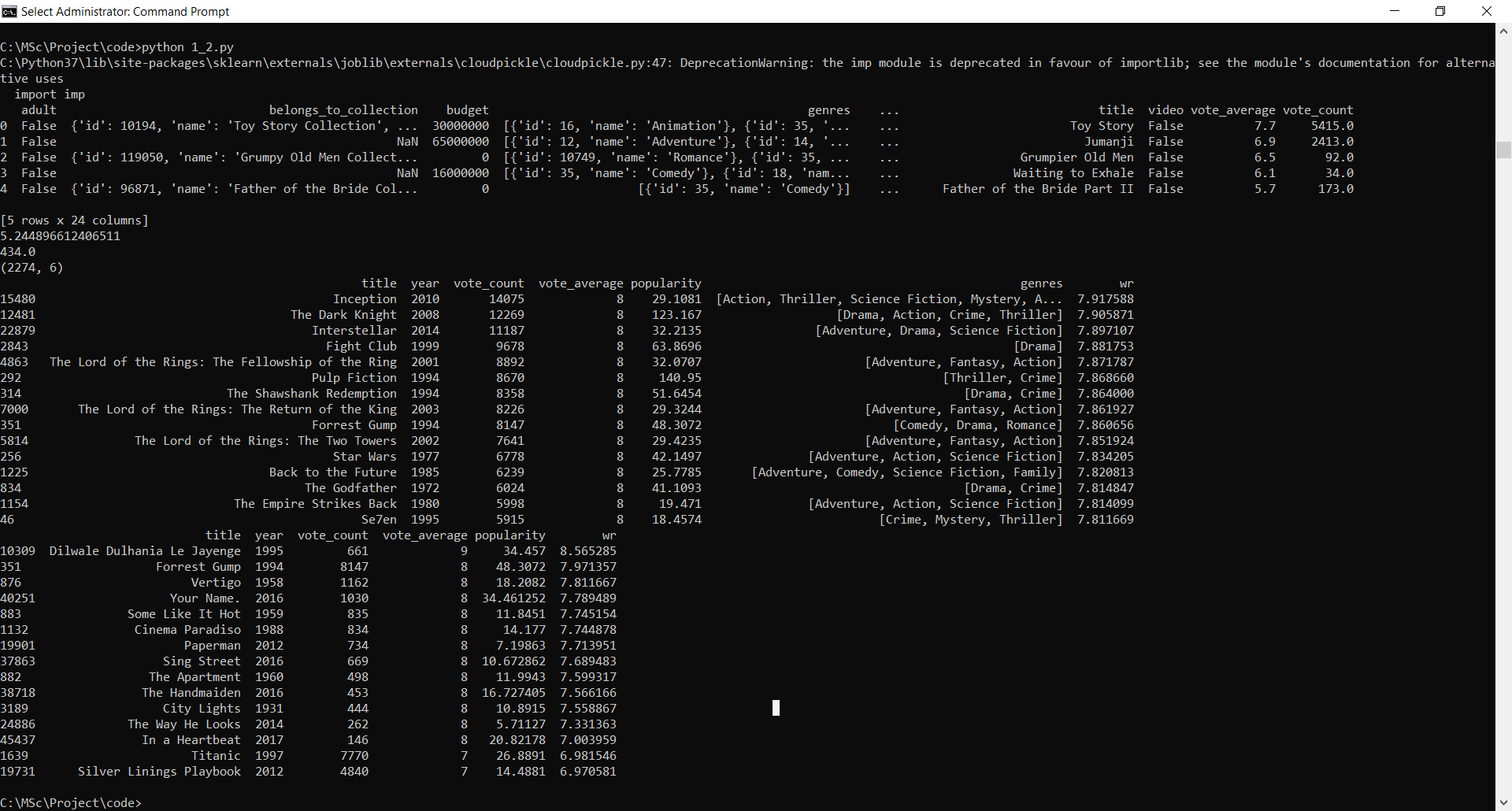
qualified = qualified.sort\_values('wr', ascending=False).head(250)

return qualified

Let us see our method in action by displaying the Top 15 Romance Movies (Romance almost didn't feature at all in our Generic Top Chart despite being one of the most popular movie genres).

### **Top 10 Romantic Movies:-**

print(build\_chart('Romance').head(15))



The top romance movie according to our metrics is Bollywood's **Dilwale Dulhania Le Jayenge**. This Shahrukh Khan starrer also happens to be one of my personal favourites.

**Content Based Recommender: -**

The recommender we built in the previous section suffers some severe limitations. For one, it gives the same recommendation to everyone, regardless of the user's personal taste. If a person who loves romantic movies (and hates action) were to look at our Top 15 Chart, s/he wouldn't probably like most of the movies. If s/he were to go one step further and look at our charts by genre, s/he wouldn't still be getting the best recommendations.

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

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

Also, as mentioned in the introduction, I will be using a subset of all the movies available to us due to limiting computing power available to me.

I will build two Content Based Recommenders based on:

1. Movie Overviews (description) and Taglines
2. Movie Cast, Crew, Keywords and Genre

import pandas as pd

import numpy as np

import matplotlib.pyplot as plt

import seaborn as sns

from scipy import stats

from ast import literal\_eval

from sklearn.feature\_extraction.text import TfidfVectorizer, CountVectorizer

from sklearn.metrics.pairwise import linear\_kernel, cosine\_similarity

from nltk.stem.snowball import SnowballStemmer

from nltk.stem.wordnet import WordNetLemmatizer

from nltk.corpus import wordnet

from surprise import Reader, Dataset, SVD, evaluate

import warnings; warnings.simplefilter('ignore')

md = pd. read\_csv('C:/MSc/Project/input/movies\_metadata.csv')

md['genres'] = md['genres'].fillna('[]').apply(literal\_eval).apply(lambda x: [i['name'] for i in x] if isinstance(x, list) else [])

vote\_counts = md[md['vote\_count'].notnull()]['vote\_count'].astype('int')

vote\_averages = md[md['vote\_average'].notnull()]['vote\_average'].astype('int')

C = vote\_averages.mean()

m = vote\_counts.quantile(0.95)

md['year'] = pd.to\_datetime(md['release\_date'], errors='coerce').apply(lambda x: str(x).split('-')[0] if x != np.nan else np.nan)

qualified = md[(md['vote\_count'] >= m) & (md['vote\_count'].notnull()) & (md['vote\_average'].notnull())][['title', 'year', 'vote\_count', 'vote\_average', 'popularity', 'genres']]

qualified['vote\_count'] = qualified['vote\_count'].astype('int')

qualified['vote\_average'] = qualified['vote\_average'].astype('int')

def weighted\_rating(x):

v = x['vote\_count']

R = x['vote\_average']

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

qualified['wr'] = qualified.apply(weighted\_rating, axis=1)

qualified = qualified.sort\_values('wr', ascending=False).head(250)

print("Content-based filtering:-")

links\_small = pd.read\_csv('C:/MSc/Project/input/links\_small.csv')

links\_small = links\_small[links\_small['tmdbId'].notnull()]['tmdbId'].astype('int') //Do print(links\_small), 9126 total rows – 13 (since 13 blank values in tmdbId) – 1 (since heading row), 9126-13-1 = 9112)

md = md.drop([19730, 29503, 35587]) //These are the row numbers after which there’s 2 blank in vote\_count and vote\_average

#Check EDA Notebook for how and why I got these indices.

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

//print(md['id']) - Output is 45463 whereas excel is showing 45466

smd = md[md['id'].isin(links\_small)] //In an excel sheet, In column A ‘id’ from movies\_metadata.csv and in column C ‘tmdbid’ from links\_small.csv. In column B2 - =IF(ISERROR(VLOOKUP(A2,$C$2:$C$45467, 1, FALSE)),FALSE,TRUE ) Drag this column values.. True values 9099.

print(smd.shape)



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

### **Movie Description Based Recommender**

Let us first try to build a recommender using movie overviews (description) and taglines. We do not have a quantitative metric to judge our machine's performance so this will have to be done qualitatively.

smd['tagline'] = smd['tagline'].fillna('') //If you do print(smd['tagline']), show the tagline of these 9099 movies

smd['description'] = smd['overview'] + smd['tagline']

//print(smd['description'][1]) shows that we are doing contatenation of overview and tagline column for the id.

// print(smd['description']) does this for all 9099 movies

smd['description'] = smd['description'].fillna('')

tf = TfidfVectorizer(analyzer='word',ngram\_range=(1, 2),min\_df=0, stop\_words='english')

//print(tf)

TfidfVectorizer(analyzer='word', binary=False, decode\_error='strict',

dtype=<class 'numpy.float64'>, encoding='utf-8', input='content',

lowercase=True, max\_df=1.0, max\_features=None, min\_df=0,

ngram\_range=(1, 2), norm='l2', preprocessor=None, smooth\_idf=True,

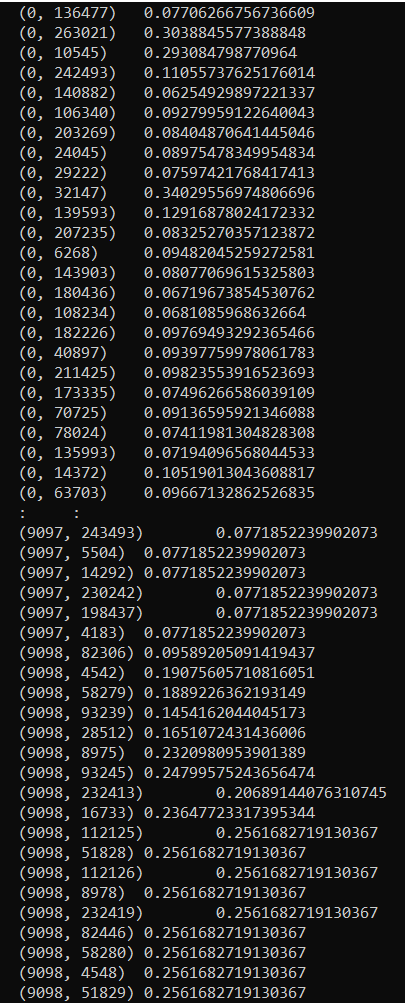
stop\_words='english', strip\_accents=None, sublinear\_tf=False,

token\_pattern='(?u)\\b\\w\\w+\\b', tokenizer=None, use\_idf=True,

vocabulary=None)

tfidf\_matrix = tf.fit\_transform(smd['description'])

// print(tfidf\_matrix) – returns for all 9099 movies.

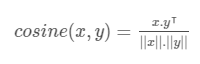


print(tfidf\_matrix.shape)



#### **Cosine Similarity**

I will be using the Cosine Similarity to calculate a numeric quantity that denotes the similarity between two movies. Mathematically, it is defined as follows: -



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

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

print(cosine\_sim[0])



We now have a pairwise cosine similarity matrix for all the movies in our dataset. The next step is to write a function that returns the 30 most similar movies based on the cosine similarity score.

smd = smd.reset\_index()

titles = smd['title']

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

def get\_recommendations(title):

idx = indices[title]

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

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

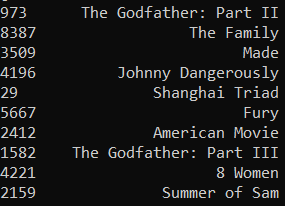
sim\_scores = sim\_scores[1:31]

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

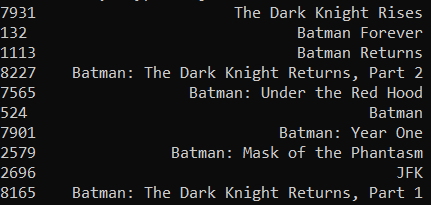
return titles.iloc[movie\_indices]

We're all set. Let us now try and get the top recommendations for a few movies and see how good the recommendations are.

print(get\_recommendations('The Godfather').head(10))



print(get\_recommendations('The Dark Knight').head(10))



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

Therefore, we are going to use much more suggestive metadata than **Overview** and **Tagline**. In the next subsection, we will build a more sophisticated recommender that takes **genre**, **keywords**, **cast** and **crew** into consideration.

### **Metadata Based Recommender**

To build our standard metadata based content recommender, we will need to merge our current dataset with the crew and the keyword datasets. Let us prepare this data as our first step.

credits = pd.read\_csv('C:/MSc/Project/input/credits.csv')

keywords = pd.read\_csv('C:/MSc/Project/input/keywords.csv')

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

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

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

print(md.shape)



md = md.merge(credits, on='id')

md = md.merge(keywords, on='id')

smd = md[md['id'].isin(links\_small)]

print(smd.shape)



We now have our cast, crew, genres and credits, all in one dataframe. Let us wrangle this a little more using the following intuitions:

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

smd['cast'] = smd['cast'].apply(literal\_eval)

smd['crew'] = smd['crew'].apply(literal\_eval)

smd['keywords'] = smd['keywords'].apply(literal\_eval)

smd['cast\_size'] = smd['cast'].apply(lambda x: len(x))

smd['crew\_size'] = smd['crew'].apply(lambda x: len(x))

def get\_director(x):

for i in x:

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

return i['name']

return np.nan

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

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

smd['cast'] = smd['cast'].apply(lambda x: x[:3] if len(x) >=3 else x)

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

My approach to building the recommender is going to be extremely *hacky*. What I plan on doing is creating a metadata dump for every movie which consists of **genres, director, main actors and keywords.** I then use a **Count Vectorizer** to create our count matrix as we did in the Description Recommender. The remaining steps are similar to what we did earlier: we calculate the cosine similarities and return movies that are most similar.

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

1. **Strip Spaces and Convert to Lowercase** from all our features. This way, our engine will not confuse between **Johnny Depp** and **Johnny Galecki.**
2. **Mention Director 3 times** to give it more weight relative to the entire cast.

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

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

smd['director'] = smd['director'].apply(lambda x: [x,x, x])

#### **Keywords**

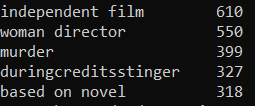
We will do a small amount of pre-processing of our keywords before putting them to any use. As a first step, we calculate the frequent counts of every keyword that appears in the dataset.

s = smd.apply(lambda x: pd.Series(x['keywords']),axis=1).stack().reset\_index(level=1, drop=True)

s.name = 'keyword'

s = s.value\_counts()

print(s[:5])



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

s = s[s > 1]

stemmer = SnowballStemmer('english')

print(stemmer.stem('dogs'))



def filter\_keywords(x):

words = []

for i in x:

if i in s:

words.append(i)

return words

smd['keywords'] = smd['keywords'].apply(filter\_keywords)

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

smd['keywords'] = smd['keywords'].apply(lambda x: [str.lower(i.replace(" ", "")) for i in x])

smd['soup'] = smd['keywords'] + smd['cast'] + smd['director'] + smd['genres']

smd['soup'] = smd['soup'].apply(lambda x: ' '.join(x))

count = CountVectorizer(analyzer='word',ngram\_range=(1, 2),min\_df=0, stop\_words='english')

count\_matrix = count.fit\_transform(smd['soup'])

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

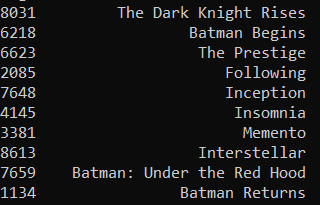
smd = smd.reset\_index()

titles = smd['title']

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

We will reuse the get\_recommendations function that we had written earlier. Since our cosine similarity scores have changed, we expect it to give us different (and probably better) results. Let us check for **The Dark Knight** again and see what recommendations I get this time around.

print(get\_recommendations('The Dark Knight').head(10))

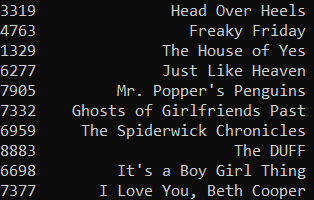


I am much more satisfied with the results I get this time around. The recommendations seem to have recognized other Christopher Nolan movies (due to the high weightage given to director) and put them as top recommendations. I enjoyed watching **The Dark Knight** as well as some of the other ones in the list including **Batman Begins**, **The Prestige** and **The Dark Knight Rises**.

We can of course experiment on this engine by trying out different weights for our features (directors, actors, genres), limiting the number of keywords that can be used in the soup, weighing genres based on their frequency, only showing movies with the same languages, etc.

Let me also get recommendations for another movie, **Mean Girls** which happens to be my girlfriend's favorite movie.

print(get\_recommendations('Mean Girls').head(10))



#### **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 mm, we will calculate the weighted rating of each movie using IMDB's formula like we did in the Simple Recommender section.

def improved\_recommendations(title):

idx = indices[title]

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

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

sim\_scores = sim\_scores[1:26]

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

movies = smd.iloc[movie\_indices][['title', 'vote\_count', 'vote\_average', 'year']]

vote\_counts = movies[movies['vote\_count'].notnull()]['vote\_count'].astype('int')

vote\_averages = movies[movies['vote\_average'].notnull()]['vote\_average'].astype('int')

C = vote\_averages.mean()

m = vote\_counts.quantile(0.60)

qualified = movies[(movies['vote\_count'] >= m) & (movies['vote\_count'].notnull()) & (movies['vote\_average'].notnull())]

qualified['vote\_count'] = qualified['vote\_count'].astype('int')

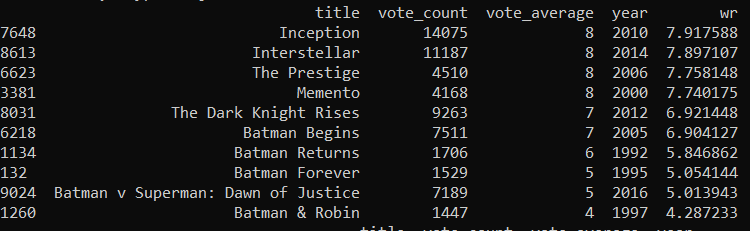
qualified['vote\_average'] = qualified['vote\_average'].astype('int')

qualified['wr'] = qualified.apply(weighted\_rating, axis=1)

qualified = qualified.sort\_values('wr', ascending=False).head(10)

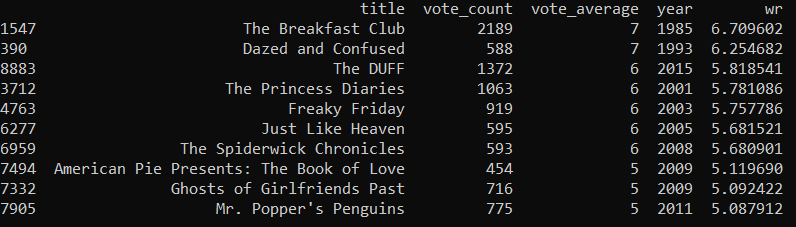
return qualified

print(improved\_recommendations('The Dark Knight'))



Let me also get the recommendations for **Mean Girls**, my girlfriend's favorite movie.

print(improved\_recommendations('Mean Girls'))



Unfortunately, **Batman and Robin** does not disappear from our recommendation list. This is probably due to the fact that it is rated a 4, which is only slightly below average on TMDB. It certainly doesn't deserve a 4 when amazing movies like **The Dark Knight Rises** has only a 7. However, there is nothing much we can do about this. Therefore, we will conclude our Content Based Recommender section here and come back to it when we build a hybrid engine.

**Collaborative Filtering:-**

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.

