HW1 - Frequent Patterns

Nick Satriano and **Jake Luther**

Class: CSCI 349 - Intro to Data Mining
Semester: Spring 2023 Instructor: Brian King

Part 2

Part 1 is appended to the end of the PDF

In [75]:
```
import numpy as np
import pandas as pd
import matplotlib as mpl
import matplotlib.pyplot as plt
import seaborn as sns
import scipy.stats as stats
from scipy.stats import zscore
from sklearn.metrics import pairwise_distances
from sklearn.decomposition import PCA
from sklearn.preprocessing import MinMaxScaler
from scipy.spatial.distance import pdist, squareform
from mpl_toolkits.mplot3d import Axes3D
from mlxtend.frequent_patterns import apriori, association_rules
from mlxtend.preprocessing import TransactionEncoder
```

Phase 1 - EDA

In [76]:
```
# Reading in the data
links_df = pd.read_csv("../data/ml-latest-small/links.csv")
movies_df = pd.read_csv("../data/ml-latest-small/movies.csv")
ratings_df = pd.read_csv("../data/ml-latest-small/ratings.csv")
tags_df = pd.read_csv("../data/ml-latest-small/tags.csv")
```

Preprocessing the data:

In [77]:
```
# Creating links DataFrame
links_df['movieId'] = links_df['movieId'].astype('category')
links_df['imdbId'] = links_df['imdbId'].astype('category')
links_df['tmdbId'] = links_df['tmdbId'].astype('category')
links_df = links_df.set_index('movieId')
links_df.info()

# Creating movies DataFrame
movies_df['movieId'] = movies_df['movieId'].astype('category')
movies_df['title'] = movies_df['title'].astype('string')
movies_df = movies_df.set_index('movieId')
movies_df.info()
```

```python
# Creating ratings DataFrame
ratings_df['movieId'] = ratings_df['movieId'].astype('category')
ratings_df['userId'] = ratings_df['userId'].astype('category')
ratings_df.info()
ratings_df.head()

# Creating the tags DataFrame
tags_df['movieId'] = tags_df['movieId'].astype('category')
tags_df['userId'] = tags_df['userId'].astype('category')
tags_df['tag'] = tags_df['tag'].astype('string')
tags_df.info()
```

```
<class 'pandas.core.frame.DataFrame'>
CategoricalIndex: 9742 entries, 1 to 193609
Data columns (total 2 columns):
 #   Column  Non-Null Count  Dtype   
---  ------  --------------  -----   
 0   imdbId  9742 non-null   category
 1   tmdbId  9734 non-null   category
dtypes: category(2)
memory usage: 1.0 MB
<class 'pandas.core.frame.DataFrame'>
CategoricalIndex: 9742 entries, 1 to 193609
Data columns (total 2 columns):
 #   Column  Non-Null Count  Dtype 
---  ------  --------------  ----- 
 0   title   9742 non-null   string
 1   genres  9742 non-null   object
dtypes: object(1), string(1)
memory usage: 505.4+ KB
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 100836 entries, 0 to 100835
Data columns (total 4 columns):
 #   Column     Non-Null Count   Dtype   
---  ------     --------------   -----   
 0   userId     100836 non-null  category
 1   movieId    100836 non-null  category
 2   rating     100836 non-null  float64 
 3   timestamp  100836 non-null  int64   
dtypes: category(2), float64(1), int64(1)
memory usage: 2.3 MB
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 3683 entries, 0 to 3682
Data columns (total 4 columns):
 #   Column     Non-Null Count  Dtype   
---  ------     --------------  -----   
 0   userId     3683 non-null   category
 1   movieId    3683 non-null   category
 2   tag        3683 non-null   string  
 3   timestamp  3683 non-null   int64   
dtypes: category(2), int64(1), string(1)
memory usage: 115.5 KB
```

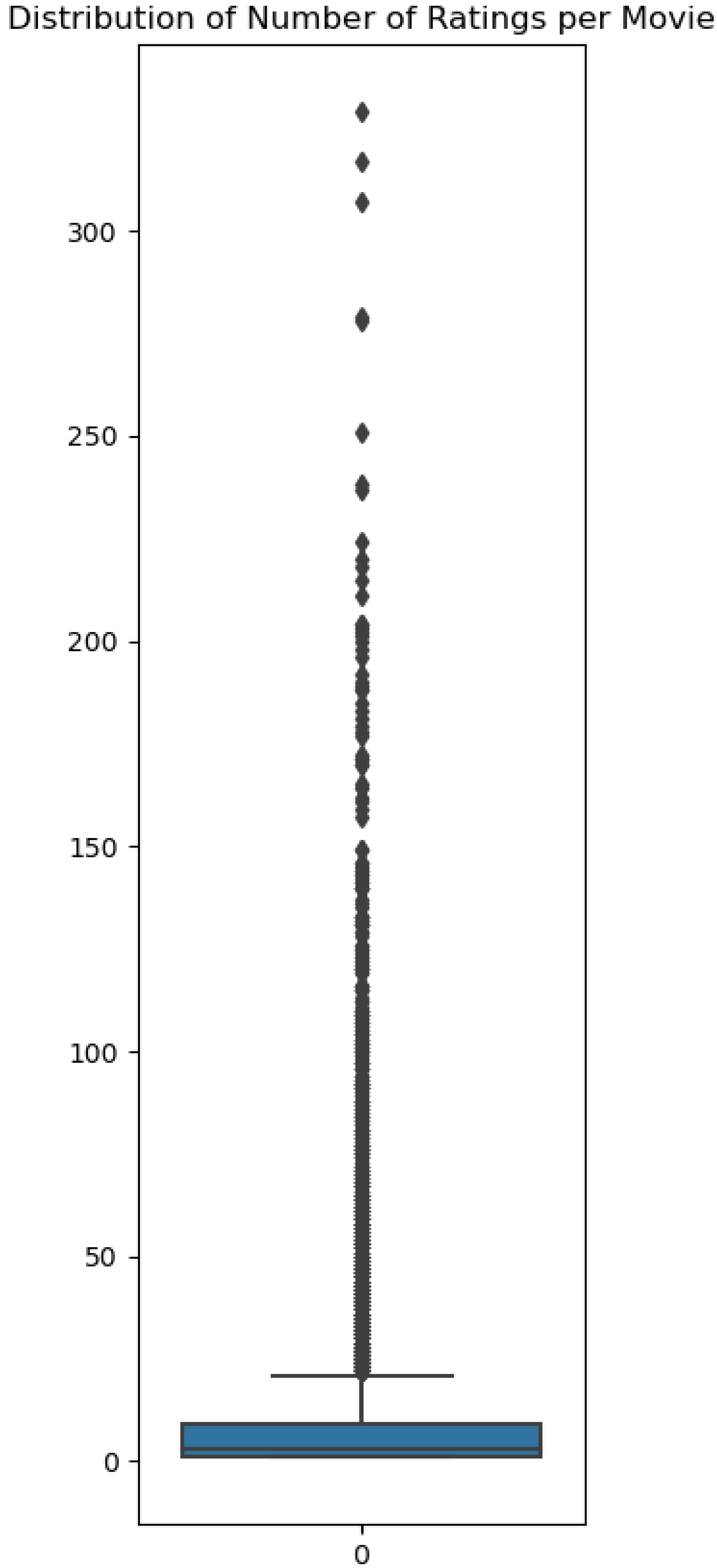
General information about our data:

In [78]:
```python
# General Data
print("Total number of movies: ", len(movies_df))
print("Total number of users: ", ratings_df['userId'].nunique())
print("Total number of reviews: ", len(ratings_df))
```

```
Total number of movies:  9742
Total number of users:  610
Total number of reviews:  100836
```

Distribution of the number of ratings per movie:

In [79]:
```
# Number of Ratings per Movie
movie_ratings = pd.merge(ratings_df, movies_df['title'], on='movieId')
num_rates_by_movie = movie_ratings['title'].value_counts()
fig, ax = plt.subplots(figsize=(3,10))
sns.boxplot(data=num_rates_by_movie, ax=ax)
plt.title("Distribution of Number of Ratings per Movie")
plt.show()
print("Average number of ratings per movie = ", round(num_rates_by_movie.mean(), 2))
print("Median number of ratings per movie = ", num_rates_by_movie.median())
print("\nMovies with the MOST number of ratings:\n", num_rates_by_movie.head())
print("\nMovies with the LEAST number of ratings:\n", num_rates_by_movie.sort_values()
```


Distribution of Number of Ratings per Movie
300
250
200
150
100
50
0
0

```
Average number of ratings per movie =  10.38
Median number of ratings per movie =  3.0

Movies with the MOST number of ratings:
 Forrest Gump (1994)                 329
Shawshank Redemption, The (1994)    317
Pulp Fiction (1994)                 307
Silence of the Lambs, The (1991)    279
Matrix, The (1999)                  278
Name: title, dtype: Int64

Movies with the LEAST number of ratings:
 31 (2016)                        1
Extraordinary Tales (2015)      1
Sex, Drugs & Taxation (2013)    1
How To Change The World (2015)  1
Chasuke's Journey (2015)        1
Name: title, dtype: Int64
```

From our distribution, we can see that "*Forrest Gump (1994)*", "*Shawshank Redemption, The (1994)*", and "*Pulp Fiction (1994)*" were the three most reviewed movies, and there were numerous movies that had only one review. The average number of ratings per movie was 10.38, and the median number of ratings was 3.0. From our boxplot, we can see that our data is very skewed, so it is best to consider the median as the measure of center for the distribution of our data when looking at number of ratings per movie.

Movies with the highest and lowest average ratings:

In [80]:
```
avg_mv_ratings = movie_ratings.groupby('movieId')['rating'].mean()
movies_df = pd.merge(movies_df, avg_mv_ratings, on='movieId')
```

In [81]:
```
lowest_avg_mv_ratings = movies_df.sort_values(by='rating')
highest_avg_mv_ratings = avg_mv_ratings.sort_values(ascending=False)
print("Movies with the LOWEST average rating:\n", movies_df[['title', 'rating']].sort_
print("\nMovies with the HIGHEST average rating:\n", movies_df[['title', 'rating']].sc
```

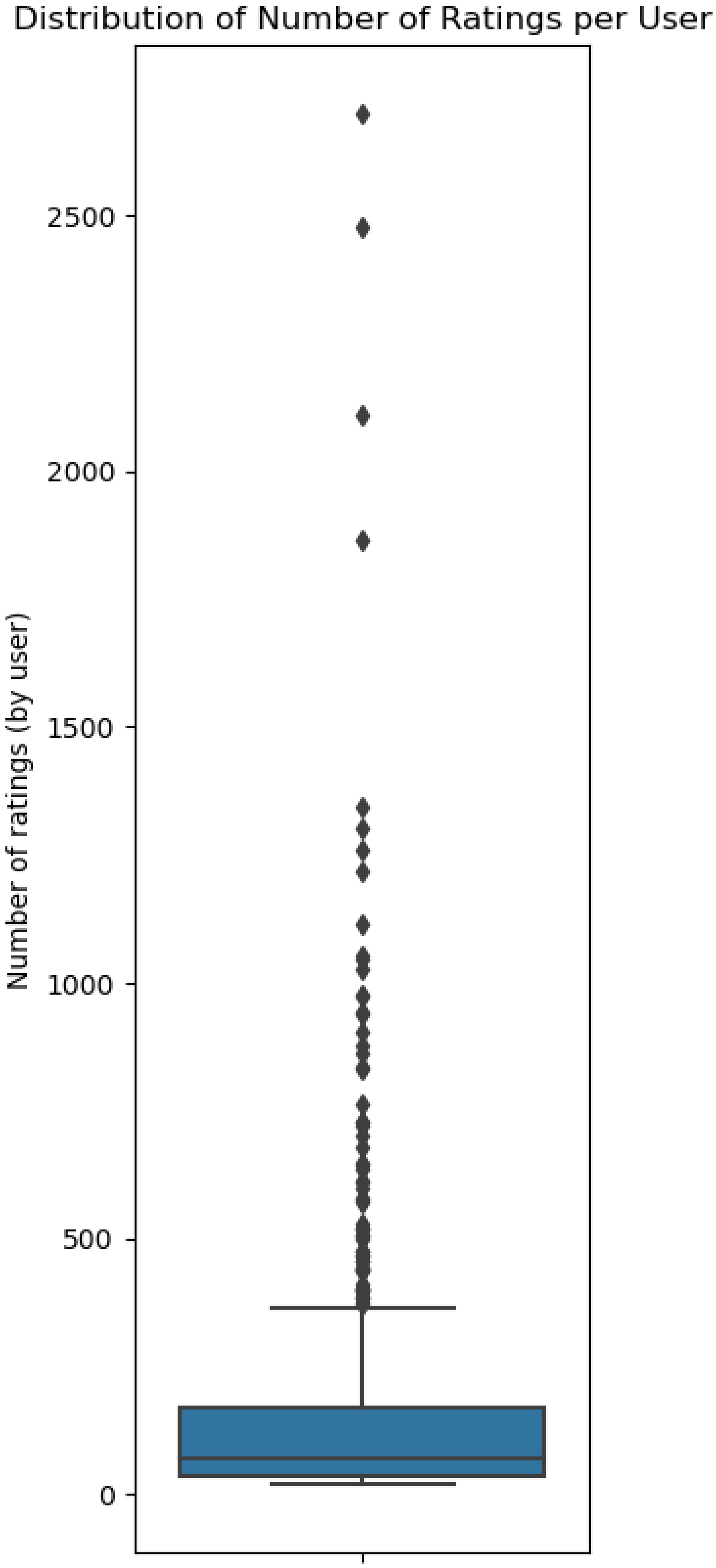
```
Movies with the LOWEST average rating:
                            title  rating
movieId
26696           Lionheart (1990)     0.5
3604                Gypsy (1962)     0.5
7312     Follow Me, Boys! (1966)     0.5
145724    Idaho Transfer (1973)     0.5
76030              Case 39 (2009)     0.5

Movies with the HIGHEST average rating:
                                          title  rating
movieId
88448    Paper Birds (Pájaros de papel) (2010)     5.0
100556               Act of Killing, The (2012)     5.0
143031                          Jump In! (2007)     5.0
143511                             Human (2015)     5.0
143559                      L.A. Slasher (2015)     5.0
```

We can see that "*Lionheart (1990)*", "*Gypsy (1962)*", and "*Follow Me, Boys! (1966)*" are among the lowest overall rated movies. On the other hand, "*Paper Birds (Pájaros de papel) (2010)*", "*Act of*

Killing, The (2012)", and "*Jump In! (2007)*" are among the highest rated movies.

Distribution of number of ratings per user:

In [82]:
```
# Number of Ratings per User
movies_by_user = movie_ratings.groupby('userId').count()
fig, ax = plt.subplots(figsize=(3,10))
sns.boxplot(data=movies_by_user, ax=ax, y='title')
plt.title("Distribution of Number of Ratings per User")
plt.ylabel('Number of ratings (by user)')
plt.show()
```



We can see that the distribution of number of ratings per user is very heavily right skewed. Most of our observations indicate that the number of reviews is heavily concentrated between 1 and 400, though there are many outliers which lay above 400, and even 500 movie reviews.

Distribution of the number of movies watched per user:

In [83]:
```
print("Average number of movie ratings per user = ", round(movies_by_user.mean(), 2))
print("Median number of movie ratings per user = ", movies_by_user.median())
print("Max number of movies rated by a user = ", movies_by_user.max())
print("Min number of movies rated by a user = ", movies_by_user.min())
```

```
Average number of movie ratings per user =  movieId      165.3
rating       165.3
timestamp    165.3
title        165.3
dtype: float64
Median number of movie ratings per user =  movieId      70.5
rating       70.5
timestamp    70.5
title        70.5
dtype: float64
Max number of movies rated by a user =  movieId      2698
rating       2698
timestamp    2698
title        2698
dtype: int64
Min number of movies rated by a user =  movieId      20
rating       20
timestamp    20
title        20
dtype: int64
```

Adding genre dummies to the movies DataFrame for each observation:

In [84]:
```
# Number of Ratings by Genre
genre_dummies = movies_df['genres'].str.get_dummies('|')
movies_df = pd.concat([movies_df, genre_dummies], axis=1)
movies_df.head()
```

Out[84]:

movieId	title	genres	rating	(no genres listed)	Action	Adventure
1	Toy Story (1995)	Adventure|Animation|Children|Comedy|Fantasy	3.920930	0	0	1
2	Jumanji (1995)	Adventure|Children|Fantasy	3.431818	0	0	1
3	Grumpier Old Men (1995)	Comedy|Romance	3.259615	0	0	0
4	Waiting to Exhale (1995)	Comedy|Drama|Romance	2.357143	0	0	0
5	Father of the Bride Part II (1995)	Comedy	3.071429	0	0	0

5 rows × 23 columns

The first 5 observations of our appended movies DataFrame is seen above.

Adding the genre dummies to the movie_ratings DataFrame:

In [85]:
```
movie_ratings = pd.merge(movie_ratings, genre_dummies, on='movieId', how='left')
movie_ratings
```

Out[85]:

	userId	movieId	rating	timestamp	title	(no genres listed)	Action	Adventure	Animation	Chi
0	1	1	4.0	964982703	Toy Story (1995)	0	0	1	1	
1	5	1	4.0	847434962	Toy Story (1995)	0	0	1	1	
2	7	1	4.5	1106635946	Toy Story (1995)	0	0	1	1	
3	15	1	2.5	1510577970	Toy Story (1995)	0	0	1	1	
4	17	1	4.5	1305696483	Toy Story (1995)	0	0	1	1	
...	...	...	...	...	...	...	...	...	...	
100831	610	160341	2.5	1479545749	Bloodmoon (1997)	0	1	0	0	
100832	610	160527	4.5	1479544998	Sympathy for the Underdog (1971)	0	1	0	0	
100833	610	160836	3.0	1493844794	Hazard (2005)	0	1	0	0	
100834	610	163937	3.5	1493848789	Blair Witch (2016)	0	0	0	0	
100835	610	163981	3.5	1493850155	31 (2016)	0	0	0	0	

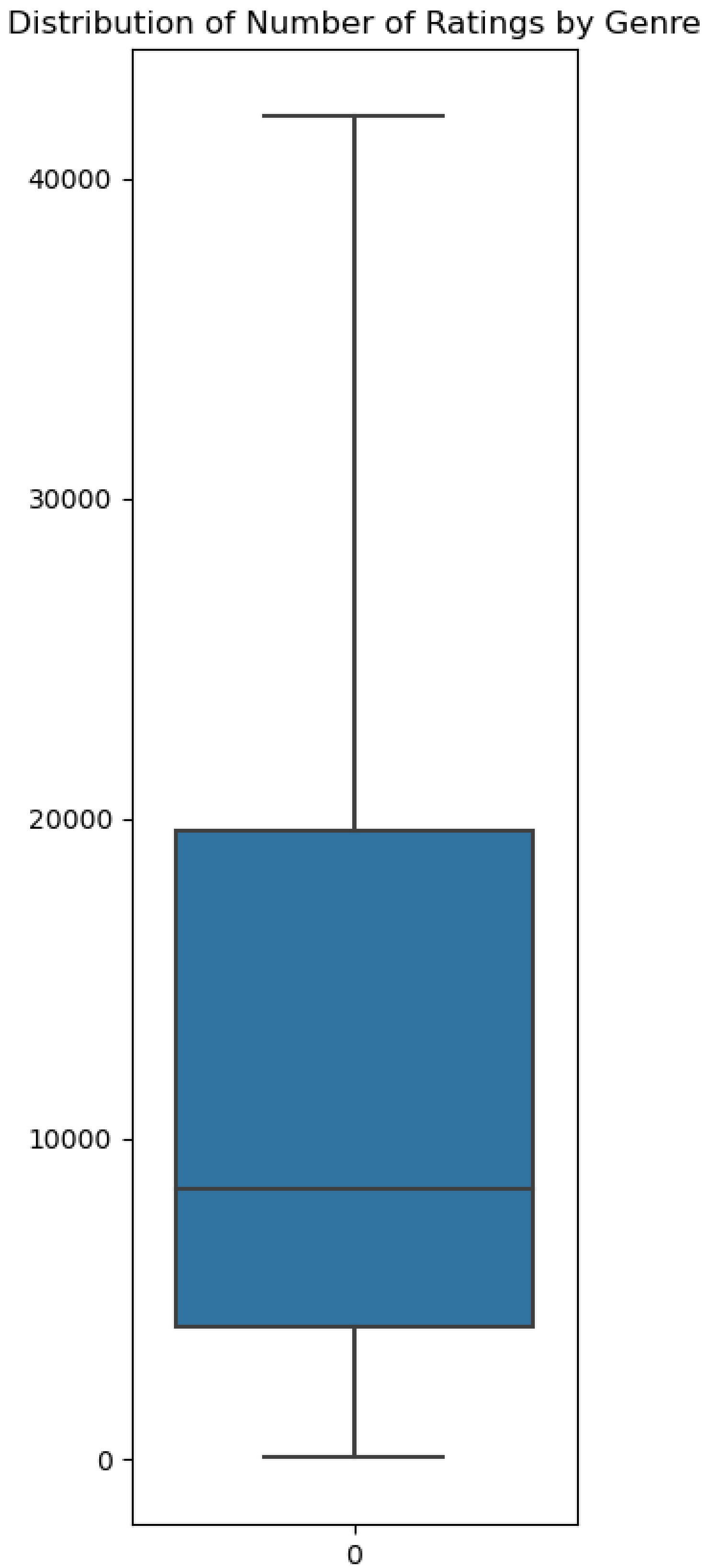
100836 rows × 25 columns

The first 5 observations of our appended movie_ratings DataFrame is seen above.

Showing the distribution of the number of ratings per genre:

In [86]:

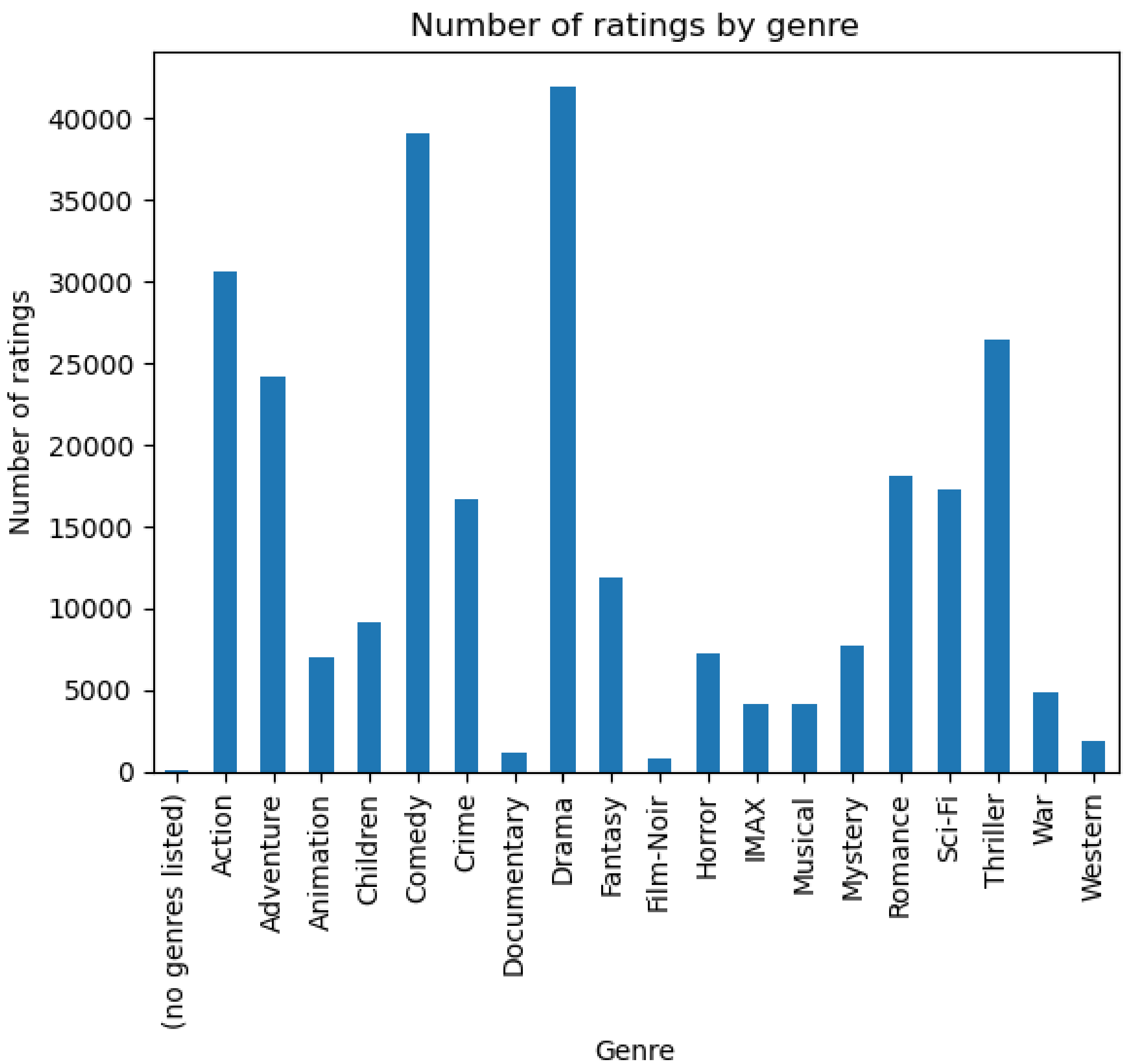
```
num_ratings_by_genre = movie_ratings.loc[:,'(no genres listed)':].sum(axis=0)
fig, ax = plt.subplots(figsize=(3,10))
sns.boxplot(data=num_ratings_by_genre, ax=ax)
plt.title("Distribution of Number of Ratings by Genre")
plt.show()
print("Average number of ratings per genre = ", round(num_ratings_by_genre.mean(), 2))
print("Median number of ratings per genre = ", num_ratings_by_genre.median())
print("Max total number of ratings of a genre = ", num_ratings_by_genre.max())
print("Min total number of ratings of a genre = ", num_ratings_by_genre.min())
```




```
Average number of ratings per genre =  13724.0
Median number of ratings per genre =  8441.0
Max total number of ratings of a genre =  41928
Min total number of ratings of a genre =  47
```

Showing the number of ratings for each genre:

In [87]:
```
num_ratings_by_genre.plot(kind='bar')
plt.title('Number of ratings by genre')
plt.xlabel('Genre')
plt.ylabel('Number of ratings')
plt.show()
```



Drama, Comedy, and Action are the three most reviewed movie genres in order.

Distribution of ratings for each specific genre:

In [88]:
```
# Ratings by Genre
genre_df = pd.DataFrame({'genres':genre_dummies.columns})
genre_df = genre_df.set_index('genres')
for x in np.arange(0.5, 5.5, 0.5):
    genre_df[str(x)] = [0] * len(genre_df)
    for g in genre_df.index:
        rating_df = movie_ratings[movie_ratings['rating'] == x]
        genre_df.loc[g, str(x)] = len(rating_df[rating_df[g]==1])
print("Distribution of each rating by genre:\n", genre_df)
```

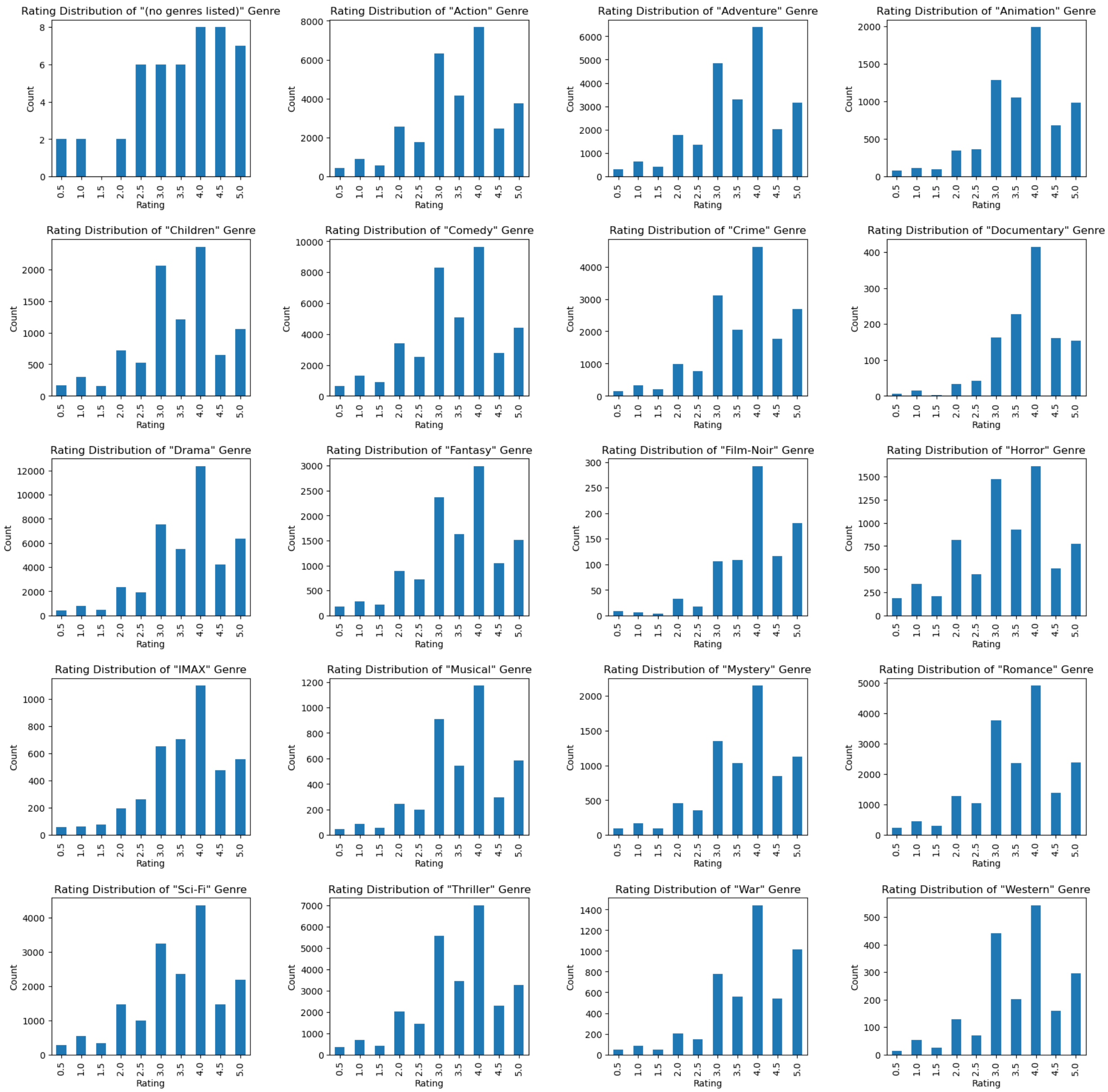
```
Distribution of each rating by genre:
                     0.5   1.0  1.5   2.0   2.5   3.0   3.5    4.0   4.5   5.0
genres
(no genres listed)    2     2    0     2     6     6     6      8     8     7
Action              449   904  577  2548  1777  6331  4153   7678  2468  3750
Adventure           306   627  415  1769  1352  4838  3285   6392  2027  3150
Animation            80   116   96   346   365  1279  1051   1988   682   985
Children            169   301  161   721   530  2054  1205   2358   648  1061
Comedy              632  1317  895  3405  2530  8306  5086   9659  2794  4429
Crime               152   321  204   982   772  3116  2057   4621  1769  2687
Documentary           6    16    2    33    42   163   228    415   161   153
Drama               405   795  485  2339  1922  7541  5514  12360  4217  6350
Fantasy             178   286  214   893   719  2364  1634   2988  1040  1518
Film-Noir             8     6    4    33    17   106   108    292   116   180
Horror              188   338  210   815   448  1473   926   1612   509   772
IMAX                 59    64   76   194   264   653   705   1096   476   558
Musical              46    86   58   247   198   909   545   1171   294   584
Mystery              95   165   93   459   351  1351  1036   2146   850  1128
Romance             231   458  298  1285  1046  3766  2372   4903  1381  2384
Sci-Fi              281   538  342  1464   992  3250  2362   4365  1469  2180
Thriller            347   678  424  2025  1434  5568  3439   6994  2277  3266
War                  46    84   49   202   147   776   560   1441   539  1015
Western              14    54   25   128    70   442   201    543   158   295
```

The above DataFrame shows the distribution of the number of ratings for each movie on a scale of 0.5 to 5.0.

Here is the Distribution of Every Different Genre:

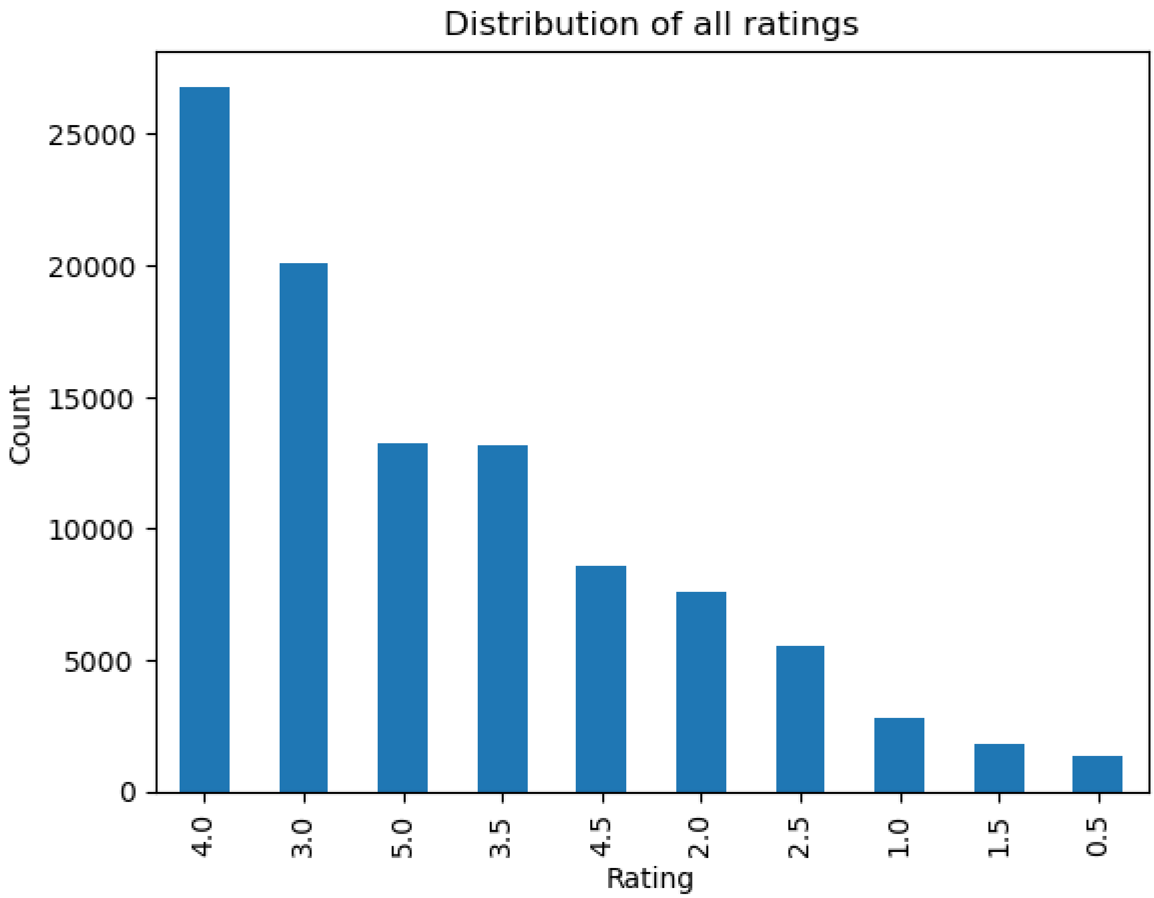
In [110…

```
fig, axs = plt.subplots(nrows=5, ncols=4, figsize=(20, 20))
for i, ax in enumerate(axs.flatten()):#genre_df.index:
    g = genre_df.index[i]
    genre_df.loc[g, :].plot(kind='bar', ax=ax)
    ax.set_title('Rating Distribution of "{}" Genre'.format(g))
    ax.set_xlabel('Rating')
    ax.set_ylabel('Count')
plt.subplots_adjust(left=0.1, right=0.9, bottom=0.1, top=0.9, wspace=0.4, hspace=0.4)
plt.show()
```



Showing the distribution of all movie ratings in the data:

In [89]:
```
rating_counts = movie_ratings['rating'].value_counts()
rating_counts.plot(kind='bar')
plt.title('Distribution of all ratings')
plt.xlabel('Rating')
plt.ylabel('Count')
plt.show()
```



Phase 2

Convert the ratings file into a set of transactions, with each transaction representing one customer, and where the universe of all possible items are movies:

In [90]:
```
user_ratings_df = ratings_df.groupby('userId')['movieId'].apply(list).reset_index()

user_df = user_ratings_df.rename(columns={'movieId': 'movies'})

user_df = user_df.set_index('userId')

print("Here are the first 5 observations of our new DataFrame:\n", user_df.head())
```

```
Here are the first 5 observations of our new DataFrame:
                                                     movies
userId
1       [1, 3, 6, 47, 50, 70, 101, 110, 151, 157, 163,...
2       [318, 333, 1704, 3578, 6874, 8798, 46970, 4851...
3       [31, 527, 647, 688, 720, 849, 914, 1093, 1124,...
4       [21, 32, 45, 47, 52, 58, 106, 125, 126, 162, 1...
5       [1, 21, 34, 36, 39, 50, 58, 110, 150, 153, 232...
```

The new DataFrame "user_df" contains the userId as the index for each observation and a single element, "movies", which is a column of lists containing of every movieId that the user reviewed for each userId.

Generate the top 20 most frequent patterns:

In [91]:
```
te = TransactionEncoder()
te_ary = te.fit(user_df['movies']).transform(user_df['movies'])
df = pd.DataFrame(te_ary, columns=te.columns_)
df

df_freq = apriori(df, min_support=0.3, use_colnames=True)

df_freq = df_freq.sort_values(by = 'support', ascending = False)

df_freq.head(20)
```

Out[91]:

	support	itemsets
8	0.539344	(356)
7	0.519672	(318)
6	0.503279	(296)
15	0.457377	(593)
22	0.455738	(2571)
5	0.411475	(260)
10	0.390164	(480)
3	0.388525	(110)
34	0.378689	(356, 318)
32	0.377049	(296, 356)
13	0.367213	(589)
31	0.363934	(296, 318)
11	0.360656	(527)
24	0.357377	(2959)
0	0.352459	(1)
18	0.345902	(1196)
33	0.339344	(296, 593)
2	0.334426	(50)
23	0.334426	(2858)
1	0.332787	(47)

Output the strongest association rules:

In [92]:
```
rules = association_rules(df_freq, metric="confidence", min_threshold=0.7)

rules
```

Out[92]:

	antecedents	consequents	antecedent support	consequent support	support	confidence	lift	leverage	cor
0	(356)	(318)	0.539344	0.519672	0.378689	0.702128	1.351097	0.098406	1
1	(318)	(356)	0.519672	0.539344	0.378689	0.728707	1.351097	0.098406	1
2	(296)	(356)	0.503279	0.539344	0.377049	0.749186	1.389068	0.105609	1
3	(296)	(318)	0.503279	0.519672	0.363934	0.723127	1.391506	0.102395	1
4	(318)	(296)	0.519672	0.503279	0.363934	0.700315	1.391506	0.102395	1
5	(593)	(296)	0.457377	0.503279	0.339344	0.741935	1.474204	0.109156	1
6	(593)	(318)	0.457377	0.519672	0.326230	0.713262	1.372522	0.088543	1
7	(593)	(356)	0.457377	0.539344	0.326230	0.713262	1.322461	0.079546	1
8	(480)	(356)	0.390164	0.539344	0.324590	0.831933	1.542489	0.114157	2
9	(1196)	(260)	0.345902	0.411475	0.311475	0.900474	2.188403	0.169145	5
10	(260)	(1196)	0.411475	0.345902	0.311475	0.756972	2.188403	0.169145	2
11	(110)	(356)	0.388525	0.539344	0.300000	0.772152	1.431649	0.090451	2
12	(260)	(2571)	0.411475	0.455738	0.300000	0.729084	1.599788	0.112475	2

The above DataFrame contains the output of the association_rules function on our data for rules with a minimum confidence of 0.7.

Adding new columns to the rules DataFrame for print formatting:

In [93]:

```
# Adding new columns to our DataFrame
for col in ['antecedents', 'consequents']:
    rules['movieId'] = rules[col].apply(lambda x: list(x)[0])
    rules = pd.merge(rules, movies_df['title'], on='movieId', how='left', suffixes=['_
    rules[col+str(2)] = rules['movieId']
rules = rules.drop(columns=['movieId', 'antecedents2', 'consequents2'])
for idx in rules.index:
    ants = [rules.loc[idx, 'title_antecedents']]
    cons = [rules.loc[idx, 'title_consequents']]
    print("Rule #", idx, ": ", ants," -> ",cons, "\n\t  confidence = ", rules.loc[idx,
```

```
Rule # 0 :  ['Forrest Gump (1994)']  ->  ['Shawshank Redemption, The (1994)']
          confidence =  0.7021276595744682      lift =  1.351097389086516

Rule # 1 :  ['Shawshank Redemption, The (1994)']  ->  ['Forrest Gump (1994)']
          confidence =  0.7287066246056783      lift =  1.351097389086516

Rule # 2 :  ['Pulp Fiction (1994)']  ->  ['Forrest Gump (1994)']
          confidence =  0.749185667752443       lift =  1.3890676514558975

Rule # 3 :  ['Pulp Fiction (1994)']  ->  ['Shawshank Redemption, The (1994)']
          confidence =  0.723127035830619       lift =  1.3915062834595509

Rule # 4 :  ['Shawshank Redemption, The (1994)']  ->  ['Pulp Fiction (1994)']
          confidence =  0.7003154574132493      lift =  1.3915062834595509

Rule # 5 :  ['Silence of the Lambs, The (1991)']  ->  ['Pulp Fiction (1994)']
          confidence =  0.7419354838709677      lift =  1.4742040558999685

Rule # 6 :  ['Silence of the Lambs, The (1991)']  ->  ['Shawshank Redemption, The (19
94)']
          confidence =  0.7132616487455198      lift =  1.3725224155670885

Rule # 7 :  ['Silence of the Lambs, The (1991)']  ->  ['Forrest Gump (1994)']
          confidence =  0.7132616487455198      lift =  1.3224608077044593

Rule # 8 :  ['Jurassic Park (1993)']  ->  ['Forrest Gump (1994)']
          confidence =  0.8319327731092437      lift =  1.542489336159996

Rule # 9 :  ['Star Wars: Episode V - The Empire Strikes Back (1980)']  ->  ['Star War
s: Episode IV - A New Hope (1977)']
          confidence =  0.9004739336492892      lift =  2.18840278695644

Rule # 10 :  ['Star Wars: Episode IV - A New Hope (1977)']  ->  ['Star Wars: Episode
V - The Empire Strikes Back (1980)']
          confidence =  0.7569721115537849      lift =  2.18840278695644

Rule # 11 :  ['Braveheart (1995)']  ->  ['Forrest Gump (1994)']
          confidence =  0.7721518987341772      lift =  1.4316494171059213

Rule # 12 :  ['Star Wars: Episode IV - A New Hope (1977)']  ->  ['Matrix, The (199
9)']
          confidence =  0.7290836653386454      lift =  1.5997878987646537
```

From the association rules that we generated, we can see that **['Star Wars: Episode V - The Empire Strikes Back (1980)'] -> ['Star Wars: Episode IV - A New Hope (1977)']** is clearly the strongest association rule, as it has a **confidence of 0.9** and a **lift of 2.19.** Generally, we consider association rules with **confidence > 0.7** and **lift > 1.25** to be very strong association rules. From this, we can deduce that the above rules that we have listed are all quite strong.

Phase 3 - Genre

Selecting the "Action", "Drama", and "Crime" genres and finding association rules for each:

In [94]:
```
for g in ['Action', 'Drama', 'Crime']:
    print("Strongest association rules for ", g," movies:\n")
    g_ratings_df = movie_ratings[movie_ratings[g]==1].groupby('userId')['movieId'].app

    # Rename the movieId column to movies_watched
    g_df = g_ratings_df.rename(columns={'movieId': 'movies'})

    g_df = g_df.set_index('userId')

    te2 = TransactionEncoder()
    te_ary2 = te2.fit(g_df['movies']).transform(g_df['movies'])
    df2 = pd.DataFrame(te_ary2, columns=te2.columns_)

    df_freq2 = apriori(df2, min_support=0.3, use_colnames=True)

    df_freq2 = df_freq2.sort_values(by = 'support', ascending = False)

    # Getting Strongest Association Rules
    rules2 = association_rules(df_freq2, metric="confidence", min_threshold=0.7)
    for col in ['antecedents', 'consequents']:
        rules2['movieId'] = rules2[col].apply(lambda x: list(x)[0])
        rules2 = pd.merge(rules2, movies_df['title'], on='movieId', how='left', suffix
        rules2[col+str(2)] = rules2['movieId']
    rules2 = rules2.drop(columns=['movieId', 'antecedents2', 'consequents2'])
    for idx in rules2.index:
        ants = [rules2.loc[idx, 'title_antecedents']]
        cons = [rules2.loc[idx, 'title_consequents']]
        print("Rule #", idx, ": ", ants," -> ",cons, "\n\t  confidence = ", rules2.loc
    print("\n")
```

```
Strongest association rules for  Action  movies:

Rule # 0 :  ['Star Wars: Episode V - The Empire Strikes Back (1980)']  ->  ['Star War
s: Episode IV - A New Hope (1977)']
          confidence =  0.9004739336492892      lift =  2.18840278695644

Rule # 1 :  ['Star Wars: Episode IV - A New Hope (1977)']  ->  ['Star Wars: Episode V
- The Empire Strikes Back (1980)']
          confidence =  0.7569721115537849      lift =  2.18840278695644

Rule # 2 :  ['Star Wars: Episode IV - A New Hope (1977)']  ->  ['Matrix, The (1999)']
          confidence =  0.7290836653386454      lift =  1.5997878987646537



Strongest association rules for  Drama  movies:

Rule # 0 :  ['Forrest Gump (1994)']  ->  ['Shawshank Redemption, The (1994)']
          confidence =  0.7021276595744682      lift =  1.351097389086516

Rule # 1 :  ['Shawshank Redemption, The (1994)']  ->  ['Forrest Gump (1994)']
          confidence =  0.7287066246056783      lift =  1.351097389086516

Rule # 2 :  ['Pulp Fiction (1994)']  ->  ['Forrest Gump (1994)']
          confidence =  0.749185667752443       lift =  1.3890676514558975

Rule # 3 :  ['Pulp Fiction (1994)']  ->  ['Shawshank Redemption, The (1994)']
          confidence =  0.723127035830619       lift =  1.3915062834595509

Rule # 4 :  ['Shawshank Redemption, The (1994)']  ->  ['Pulp Fiction (1994)']
          confidence =  0.7003154574132493      lift =  1.3915062834595509

Rule # 5 :  ['Braveheart (1995)']  ->  ['Forrest Gump (1994)']
          confidence =  0.7721518987341772      lift =  1.4316494171059213



Strongest association rules for  Crime  movies:

Rule # 0 :  ['Pulp Fiction (1994)']  ->  ['Shawshank Redemption, The (1994)']
          confidence =  0.723127035830619       lift =  1.3915062834595509

Rule # 1 :  ['Shawshank Redemption, The (1994)']  ->  ['Pulp Fiction (1994)']
          confidence =  0.7003154574132493      lift =  1.3915062834595509

Rule # 2 :  ['Silence of the Lambs, The (1991)']  ->  ['Pulp Fiction (1994)']
          confidence =  0.7419354838709677      lift =  1.4742040558999685

Rule # 3 :  ['Silence of the Lambs, The (1991)']  ->  ['Shawshank Redemption, The (19
94)']
          confidence =  0.7132616487455198      lift =  1.3725224155670885
```

This method for finding association rules within the movie review dataset might paint us a better picture of the association rules between the movies because most of the movies have multiple genres. To add to this, find some more meaning in our data analysis by adding the grouping by genre to better understand how similar-genre movies' viewership might be associated. However, this method may also filter out some strong association rules that exist between movies of different genres.

Phase 4 - Genre Rules

Creating a new DataFrame that contains the list of genres reviewed by each userId:

In [95]:
```
# Creating a new DataFrame containing the list of unique genres reviewed by each userI
movie_dat_df = pd.merge(movies_df, ratings_df, on='movieId')

movie_dat_df = movie_dat_df.groupby('userId').apply(lambda x: list(set(x['genres'].str

movie_dat_df
```

Out[95]:
```
userId
1      [War, Film-Noir, Adventure, Animation, Romance...
2      [War, Adventure, Western, Romance, Thriller, I...
3      [War, Adventure, Animation, Sci-Fi, Thriller, ...
4      [Film-Noir, Western, Sci-Fi, Romance, Comedy, ...
5      [War, Adventure, Animation, Romance, Thriller,...
                             ...
606    [Film-Noir, Western, Romance, Sci-Fi, Comedy, ...
607    [War, Adventure, Animation, Romance, IMAX, Chi...
608    [Film-Noir, Western, Romance, Sci-Fi, Comedy, ...
609    [War, Adventure, Animation, Documentary, Thril...
610    [Film-Noir, Western, Sci-Fi, Romance, Comedy, ...
Length: 610, dtype: object
```

Finding frequent patterns among the genres reviewed by each userId:

In [96]:
```
tencoder = TransactionEncoder()
te_arry = tencoder.fit(movie_dat_df).transform(movie_dat_df)
gdat_df = pd.DataFrame(te_arry, columns=tencoder.columns_)

df_gfreq = apriori(gdat_df, min_support=0.8, use_colnames=True)

df_gfreq = df_gfreq.sort_values(by = 'support', ascending = False)

df_gfreq.head(10)
```

Out[96]:

	support	itemsets
6	1.000000	(Drama)
4	0.998361	(Comedy)
80	0.998361	(Thriller, Drama)
59	0.998361	(Comedy, Drama)
12	0.998361	(Thriller)
0	0.996721	(Action)
65	0.996721	(Thriller, Comedy)
332	0.996721	(Thriller, Comedy, Drama)
19	0.996721	(Action, Drama)
156	0.995082	(Thriller, Action, Drama)

Generating the association rules for genre's frequent patterns:

In [97]:

```
grules = association_rules(df_gfreq, metric="confidence", min_threshold=0.7)

grules
```

Out[97]:

	antecedents	consequents	antecedent support	consequent support	support	confidence	lift	leverage
0	(Thriller)	(Drama)	0.998361	1.000000	0.998361	1.000000	1.000000	0.000000
1	(Drama)	(Thriller)	1.000000	0.998361	0.998361	0.998361	1.000000	0.000000
2	(Comedy)	(Drama)	0.998361	1.000000	0.998361	1.000000	1.000000	0.000000
3	(Drama)	(Comedy)	1.000000	0.998361	0.998361	0.998361	1.000000	0.000000
4	(Thriller)	(Comedy)	0.998361	0.998361	0.996721	0.998358	0.999997	-0.000003
...	...	...	...	...	...	...	...	...
686933	(Children)	(War, Drama, Crime, Comedy, Fantasy)	0.916393	0.855738	0.800000	0.872987	1.020158	0.015808
686934	(Drama)	(War, Children, Crime, Comedy, Fantasy)	1.000000	0.800000	0.800000	0.800000	1.000000	0.000000
686935	(Crime)	(War, Children, Drama, Comedy, Fantasy)	0.988525	0.803279	0.800000	0.809287	1.007480	0.005939
686936	(Comedy)	(War, Children, Drama, Crime, Fantasy)	0.998361	0.801639	0.800000	0.801314	0.999594	-0.000325
686937	(Fantasy)	(War, Children, Drama, Crime, Comedy)	0.955738	0.824590	0.800000	0.837050	1.015110	0.011908

686938 rows × 9 columns

Finding the strongest association rules and formatting the output:

In [98]:
```
# Getting Strongest Association Rules
print(grules.index)
for idx in grules.index[:10]:
    ants = [list(grules.loc[idx, 'antecedents'])[0]]
    cons = [list(grules.loc[idx, 'consequents'])[0]]
    print("Rule #", idx, ": ", ants," -> ",cons, "\n\t  confidence = ", grules.loc[idx
```

```
RangeIndex(start=0, stop=686938, step=1)
Rule # 0 :  ['Thriller']  ->  ['Drama']
          confidence =  1.0     lift =  1.0

Rule # 1 :  ['Drama']  ->  ['Thriller']
          confidence =  0.9983606557377049      lift =  1.0

Rule # 2 :  ['Comedy']  ->  ['Drama']
          confidence =  1.0     lift =  1.0

Rule # 3 :  ['Drama']  ->  ['Comedy']
          confidence =  0.9983606557377049      lift =  1.0

Rule # 4 :  ['Thriller']  ->  ['Comedy']
          confidence =  0.9983579638752053      lift =  0.9999973037173648

Rule # 5 :  ['Comedy']  ->  ['Thriller']
          confidence =  0.9983579638752053      lift =  0.9999973037173648

Rule # 6 :  ['Thriller']  ->  ['Drama']
          confidence =  1.0     lift =  1.0

Rule # 7 :  ['Thriller']  ->  ['Comedy']
          confidence =  0.9983579638752053      lift =  0.9999973037173648

Rule # 8 :  ['Comedy']  ->  ['Thriller']
          confidence =  0.9983579638752053      lift =  0.9999973037173648

Rule # 9 :  ['Thriller']  ->  ['Comedy']
          confidence =  0.9983579638752053      lift =  0.9999973037173648
```

From the association rules printed above, we can see that the strongest rules are **['Drama'] -> ['Thriller']**, **['Thriller'] -> ['Drama']**, and **['Comedy'] -> ['Drama']**. The association rules that we are able to make about genres aren't quite optimal just yet, so we can't definitively say that these are the strongest association rules for genres.

Phase 5 - Incorporating Additional Variables

Defining a new function, get_decade() to find the decades that each userId reviewed:

In [99]:

```python
import re
# Define function to extract decade from movie title
def get_decade(title):
    year = re.findall(r"\(\d{4}\)", title)
    if len(year) > 0:
        year = int(year[0][1:5])
        decade = (year // 10) * 10
        return decade
    else:
        return None
```

Adding the decade column to the movies_df:

In [100…

```
# Add decade column to movies DataFrame
movies_df["decade"] = movies_df["title"].apply(get_decade)

# Merge movies and ratings DataFrames
merged_df = ratings_df.merge(movies_df, on="movieId", how="left")

merged_df.head()
```

Out[100]:

	userId	movieId	rating_x	timestamp	title	genres	rating
0	1	1	4.0	964982703	Toy Story (1995)	Adventure|Animation|Children|Comedy|Fantasy	3.9209
1	1	3	4.0	964981247	Grumpier Old Men (1995)	Comedy|Romance	3.2596
2	1	6	4.0	964982224	Heat (1995)	Action|Crime|Thriller	3.9460
3	1	47	5.0	964983815	Seven (a.k.a. Se7en) (1995)	Mystery|Thriller	3.9753
4	1	50	5.0	964982931	Usual Suspects, The (1995)	Crime|Mystery|Thriller	4.2377

5 rows × 28 columns

Making a new DataFrame, decade_df, which contains the list of decades for each movie that was reviewed per userId:

In [101…

```
decade_df = merged_df.groupby('userId')['decade'].apply(list).reset_index()

decade_df = decade_df.set_index('userId')

decade_df
```

Out[101]:

	decade
userId	
1	[1990.0, 1990.0, 1990.0, 1990.0, 1990.0, 1990....
2	[1990.0, 1990.0, 1990.0, 2000.0, 2000.0, 2000....
3	[1990.0, 1990.0, 1990.0, 1990.0, 1990.0, 1990....
4	[1990.0, 1990.0, 1990.0, 1990.0, 1990.0, 1990....
5	[1990.0, 1990.0, 1990.0, 1990.0, 1990.0, 1990....
...	...
606	[1990.0, 1990.0, 1990.0, 1990.0, 1990.0, 1990....
607	[1990.0, 1990.0, 1990.0, 1990.0, 1990.0, 1990....
608	[1990.0, 1990.0, 1990.0, 1990.0, 1990.0, 1990....
609	[1990.0, 1990.0, 1990.0, 1990.0, 1990.0, 1990....
610	[1990.0, 1990.0, 1990.0, 1990.0, 1990.0, 1990....

610 rows × 1 columns

Generating the transaction encoder and rules for the decade variable:

In [102…

```
te = TransactionEncoder()
te_ary = te.fit(decade_df['decade']).transform(decade_df['decade'])
df = pd.DataFrame(te_ary, columns=te.columns_)
df


decade_freq = apriori(df, min_support=0.7, use_colnames=True)

decade_freq = decade_freq.sort_values(by = 'support', ascending = False)


decade_rules = association_rules(decade_freq, metric="confidence", min_threshold=0.7)

decade_rules
```

Out[102]:

	antecedents	consequents	antecedent support	consequent support	support	confidence	lift	leverage	cor
0	(1980.0)	(1990.0)	0.908197	0.998361	0.908197	1.000000	1.001642	0.001489	
1	(1990.0)	(1980.0)	0.998361	0.908197	0.908197	0.909688	1.001642	0.001489	1
2	(1970.0)	(1990.0)	0.775410	0.998361	0.775410	1.000000	1.001642	0.001271	
3	(1990.0)	(1970.0)	0.998361	0.775410	0.775410	0.776683	1.001642	0.001271	1
4	(1970.0)	(1980.0)	0.775410	0.908197	0.726230	0.936575	1.031247	0.022005	1
5	(1980.0)	(1970.0)	0.908197	0.775410	0.726230	0.799639	1.031247	0.022005	1
6	(2000.0)	(1990.0)	0.726230	0.998361	0.726230	1.000000	1.001642	0.001191	
7	(1990.0)	(2000.0)	0.998361	0.726230	0.726230	0.727422	1.001642	0.001191	1
8	(1970.0, 1980.0)	(1990.0)	0.726230	0.998361	0.726230	1.000000	1.001642	0.001191	
9	(1970.0, 1990.0)	(1980.0)	0.775410	0.908197	0.726230	0.936575	1.031247	0.022005	1
10	(1980.0, 1990.0)	(1970.0)	0.908197	0.775410	0.726230	0.799639	1.031247	0.022005	1
11	(1970.0)	(1980.0, 1990.0)	0.775410	0.908197	0.726230	0.936575	1.031247	0.022005	1
12	(1980.0)	(1970.0, 1990.0)	0.908197	0.775410	0.726230	0.799639	1.031247	0.022005	1
13	(1990.0)	(1970.0, 1980.0)	0.998361	0.726230	0.726230	0.727422	1.001642	0.001191	1

Shown above are the association rules generated for the decade variable with a minimum support of 0.7 and a minimum confidence of 0.7. We only have 13 association rules that meet a minimum support of 0.7 and confidence of 0.7, so we know that these are the strongest rules for the decade variable.

In [103…

```
decade_rules = decade_rules.sort_values(by = 'confidence', ascending = False)

for idx in decade_rules.index[:8]:
    ants = list(decade_rules.loc[idx, 'antecedents'])
    cons = list(decade_rules.loc[idx, 'consequents'])
    print("Rule #", idx, ": ", ants," -> ",cons, "\n\t  confidence = ", decade_rules.l
```

```
Rule # 0 :  [1980.0]  ->  [1990.0]
          confidence =  1.0     lift =  1.0016420361247946

Rule # 2 :  [1970.0]  ->  [1990.0]
          confidence =  1.0     lift =  1.0016420361247946

Rule # 6 :  [2000.0]  ->  [1990.0]
          confidence =  1.0     lift =  1.0016420361247946

Rule # 8 :  [1970.0, 1980.0]  ->  [1990.0]
          confidence =  1.0     lift =  1.0016420361247946

Rule # 4 :  [1970.0]  ->  [1980.0]
          confidence =  0.9365750528541226      lift =  1.031246899352012

Rule # 9 :  [1970.0, 1990.0]  ->  [1980.0]
          confidence =  0.9365750528541226      lift =  1.031246899352012

Rule # 11 :  [1970.0]  ->  [1980.0, 1990.0]
          confidence =  0.9365750528541226      lift =  1.031246899352012

Rule # 1 :  [1990.0]  ->  [1980.0]
          confidence =  0.909688013136289       lift =  1.0016420361247946
```

Out of all the association rules generated, we found that the above 7 rules have the highest confidence by a large margin, all of which have above a 0.9 confidence level. Though the lift is not very high, we can consider these 7 rules as the strongest rules when it comes to association rules between decades for movie titles reviewed.

For the second part of this phase, we will analyze the movie review tags. Here, we find the most/least common tags:

In [104…

```
# Use tags_df to get most/least commonly used tags
tot_num_tags = len(tags_df['tag'].value_counts())
print("Number of unqiue movie tags: ",tot_num_tags,"\n")

top_rated_df = pd.merge(tags_df, ratings_df, on='movieId')
#top_rated_df = top_rated_df[top_rated_df['rating'] == 5.0]
top_rated_df['userId'] = top_rated_df['userId_x']
print(top_rated_df)

#tag_counts = tags_df['tag'].value_counts()
tag_counts = top_rated_df['tag'].value_counts()

z = 0
print("20 Most Commonly Used Tags:")
for n in tag_counts.index[:20]:
    z += 1
    print("{}. {}  | count = {}".format(z, n, tag_counts[z-1]))

z = 0
print("\n20 Least Commonly Used Tags:")
for n in tag_counts.index[-20:]:
    z += 1
    print("{}. {}  | count = {}".format(z, n, tag_counts[-21+z]))
```

```
Number of unqiue movie tags:  1589

       userId_x  movieId              tag  timestamp_x userId_y  rating  \
0             2    60756            funny   1445714994        2     5.0
1             2    60756            funny   1445714994       18     3.0
2             2    60756            funny   1445714994       62     3.5
3             2    60756            funny   1445714994       68     2.5
4             2    60756            funny   1445714994       73     4.5
...         ...      ...              ...          ...      ...     ...
233208      610     3265  heroic bloodshed   1493843978      380     4.0
233209      610     3265  heroic bloodshed   1493843978      469     3.0
233210      610     3265  heroic bloodshed   1493843978      599     4.0
233211      610     3265  heroic bloodshed   1493843978      603     5.0
233212      610     3265  heroic bloodshed   1493843978      610     5.0

        timestamp_y userId
0        1445714980      2
1        1455749449      2
2        1528934376      2
3        1269123243      2
4        1464196221      2
...             ...    ...
233208   1494036091    610
233209    965661994    610
233210   1498498587    610
233211    963177579    610
233212   1479542010    610

[233213 rows x 8 columns]
20 Most Commonly Used Tags:
1. sci-fi  | count = 2527
2. thought-provoking  | count = 2487
3. twist ending  | count = 2434
4. atmospheric  | count = 2227
5. dark comedy  | count = 2056
6. superhero  | count = 1787
7. psychology  | count = 1750
8. Disney  | count = 1748
9. time travel  | count = 1730
10. suspense  | count = 1716
11. classic  | count = 1625
12. imdb top 250  | count = 1506
13. quirky  | count = 1414
14. space  | count = 1413
15. mindfuck  | count = 1401
16. disturbing  | count = 1378
17. psychological  | count = 1339
18. surreal  | count = 1336
19. action  | count = 1322
20. great soundtrack  | count = 1299

20 Least Commonly Used Tags:
1. austere  | count = 1
2. italy  | count = 1
3. representation of children  | count = 1
4. lions  | count = 1
5. remix culture  | count = 1
6. animal movie  | count = 1
7. music industry  | count = 1
8. human rights  | count = 1
```

```
9. Suspenseful  | count = 1
10. rap  | count = 1
11. Narrative pisstake  | count = 1
12. Van Gogh  | count = 1
13. Not available from Netflix  | count = 1
14. Anne Boleyn  | count = 1
15. convent  | count = 1
16. deafness  | count = 1
17. Tolstoy  | count = 1
18. Cole Porter  | count = 1
19. parenthood  | count = 1
20. Titanic  | count = 1
```

The tags sci-fi (count = 2527), thought-provoking (count = 2487), twist ending (count = 2434), atmospheric (count = 2227), and dark comedy (count = 2056) are found most often the review data.

Grouping each tag into a list for each userId:

In [105…

```
movie_tags = top_rated_df[top_rated_df['rating']==5].groupby('userId')['tag'].apply(li
movie_tags.head(10)
```

Out[105]:

```
userId
2      [funny, funny, funny, Highly quotable, Highly ...
7      [way too long, way too long, way too long, way...
18     [Al Pacino, Al Pacino, Al Pacino, Al Pacino, A...
21     [romantic comedy, romantic comedy, wedding, we...
49     [black hole, black hole, black hole, black hol...
62     [comedy, comedy, comedy, funny, funny, funny, ...
63     [classic, classic, classic, classic, classic, ...
76     [action, action, action, action, action, actio...
103    [EPIC, EPIC, EPIC, EPIC, EPIC, EPIC, EPIC, EPI...
106    [Everything you want is here, Everything you w...
Name: tag, dtype: object
```

Creating the transaction encoder and running the Apriori algorithm:

In [106…

```
tencode = TransactionEncoder()
te_array = tencode.fit(movie_tags).transform(movie_tags)
tdat_df = pd.DataFrame(te_array, columns=tencode.columns_)

df_tfreq = apriori(tdat_df, min_support=0.1, use_colnames=True)

df_tfreq = df_tfreq.sort_values(by = 'support', ascending = False)

df_tfreq.head(10)
```

Out[106]:

	support	itemsets
12	0.172414	(sci-fi)
1	0.155172	(atmospheric)
4	0.155172	(dark comedy)
15	0.137931	(suspense)
6	0.137931	(funny)
11	0.120690	(psychology)
17	0.120690	(twist ending)
16	0.120690	(thought-provoking)
22	0.120690	(suspense, mindfuck)
8	0.120690	(music)

Our tag itemsets all have very low support.

Creating the association rules for the tag itemsets:

In [107…

```
trules = association_rules(df_tfreq, metric="lift", min_threshold=1.2)
trules
```

Out[107]:

	antecedents	consequents	antecedent support	consequent support	support	confidence	lift	leverage	cor
0	(suspense)	(mindfuck)	0.137931	0.120690	0.120690	0.875000	7.250000	0.104043	7
1	(mindfuck)	(suspense)	0.120690	0.137931	0.120690	1.000000	7.250000	0.104043	
2	(atmospheric)	(dark comedy)	0.155172	0.155172	0.103448	0.666667	4.296296	0.079370	2
3	(dark comedy)	(atmospheric)	0.155172	0.155172	0.103448	0.666667	4.296296	0.079370	2
4	(surreal)	(atmospheric)	0.103448	0.155172	0.103448	1.000000	6.444444	0.087396	
5	(atmospheric)	(surreal)	0.155172	0.103448	0.103448	0.666667	6.444444	0.087396	2
6	(thought-provoking)	(atmospheric)	0.120690	0.155172	0.103448	0.857143	5.523810	0.084721	5
7	(atmospheric)	(thought-provoking)	0.155172	0.120690	0.103448	0.666667	5.523810	0.084721	2

In [108…

```
for idx in trules.index:
    ants = [list(trules.loc[idx, 'antecedents'])[0]]
    cons = [list(trules.loc[idx, 'consequents'])[0]]
    print("Rule #", idx+1, ": ", ants," -> ",cons, "\n\t  confidence = ", round(trules
```

```
Rule # 1 :  ['suspense']  ->  ['mindfuck']
          confidence =  0.875   lift =  7.25

Rule # 2 :  ['mindfuck']  ->  ['suspense']
          confidence =  1.0     lift =  7.25

Rule # 3 :  ['atmospheric']  ->  ['dark comedy']
          confidence =  0.667   lift =  4.296

Rule # 4 :  ['dark comedy']  ->  ['atmospheric']
          confidence =  0.667   lift =  4.296

Rule # 5 :  ['surreal']  ->  ['atmospheric']
          confidence =  1.0     lift =  6.444

Rule # 6 :  ['atmospheric']  ->  ['surreal']
          confidence =  0.667   lift =  6.444

Rule # 7 :  ['thought-provoking']  ->  ['atmospheric']
          confidence =  0.857   lift =  5.524

Rule # 8 :  ['atmospheric']  ->  ['thought-provoking']
          confidence =  0.667   lift =  5.524
```

Of the above rules, we can say that **['suspense'] -> ['mindfuck']**, **['mindfuck'] -> ['suspense']**, **['surreal'] -> ['atmospheric']**, and **['thought-provoking'] -> ['atmospheric']** are the strongest association rules for tags in the data, since they all have **confidence > 0.85** and **lift > 4**.

Part 1

Exercise 1 - The Apriori Algorithm:

TID	*items_bought*
T100	{M, O, N, K, E, Y}
T200	{D, O, N, K, E, Y }
T300	{M, A, K, E}
T400	{M, U, C, K, Y}
T500	{C, O, O, K, I, E}

a)

1-itemsets:
{M}: 0.6
{O}: 0.6
{N}: Does not meet min_sup
{K}: 1.0
{E}: 0.8
{Y}: 0.6
{D}: Does not meet min_sup
{A}: Does not meet min_sup
{U}: Does not meet min_sup
{C}: Does not meet min_sup
{I}: Does not meet min_sup

2-itemsets:
Any 2-itemsets containing N, D, A, U, C, and I are eliminated by Apriori Property.

{M, O}: Does not meet min_sup
{M, K}: 0.6
{M, E}: Does not meet min_sup
{M, Y}: Does not meet min_sup
{O, K}: 0.6
{O, E}: 0.6
{O, Y}: Does not meet min_sup
{K, E}: 0.8
{K, Y}: 0.6
{E, Y}: Does not meet min_sup

3-itemsets:
Any 3-itemsets containing {M,O}, {M, E}, {M, Y}, {O, Y}, {E, Y}, {M, K} are eliminated by Apriori Property.

{O, K, E}: 0.6

b)

A closed frequent itemset is a set of items that appears frequently in a dataset and is not a subset of any other frequent itemset with the same frequency count. In other words, a closed frequent itemset is a set of items that has the maximum support among all the itemsets with the same set of items.

From the above list, {O, K, E}, {K, Y}, {M, K} are all closed frequent itemsets.

c)

A max frequent itemset is a set of items that appears frequently in a dataset and is not a subset of any other frequent itemset with a higher support count. In other words, a max frequent itemset is a set of items that has the maximum support count among all the itemsets that have the same items but with different cardinalities.

From the above list, {K} and {E} are the max frequent itemsets

What is absolute support?
Use number of items
Like 3!!!

The maximum frequent itemset is the itemset with the highest support in a dataset, where support refers to the proportion of transactions in the dataset that contain that itemset. In other words, it is the itemset that appears most frequently in the transactions in the dataset.
For example, consider a dataset of grocery store transactions, where each transaction contains a set of items purchased by a customer. If the itemset {bread, milk} appears in 40% of all transactions in the dataset, and no other itemset appears in a higher percentage of transactions, then {bread, milk} is the maximum frequent itemset in the dataset.
Finding the maximum frequent itemset is an important task in data mining and machine learning, as it can provide insights into the most popular combinations of items in a dataset, which can be useful for various applications such as product recommendation, market basket analysis, and customer segmentation.

d)

How to generate association rules:

1. Start with each frequent itemset of size 2 or more.
2. For each frequent itemset, generate all possible non-empty subsets of the itemset.
3. For each subset, compute the confidence and lift measures of the association rule that has the subset as the antecedent and the complement of the subset as the consequent.

4. If the confidence and lift measures exceed certain threshold values (e.g., 0.7 for confidence and 1.2 for lift), then the association rule is considered strong.

{M, K}:
    {K → M}:
        Confidence = 0.6 / 0.8 = 0.75
        Lift = 0.6 / (0.6 * 1.0) = 1.0
    {M → K}:
        Confidence = 0.6 / 0.6 = 1.0
        Lift = 0.6 / (0.6 * 1.0) = 1.0

{O, K}:
    O: 0.6, K: 1.0
    {O → K}:
        Confidence = 0.6 / 0.6 = 1.0
        Lift: 0.6 / (0.6 * 1.0) = 1.0
    {K → O}:
        Confidence = 0.6 / 1.0 = 0.6
        Lift: 0.6 / (1.0 * 0.6) = 1.0
{O, E}:
    O: 0.6, E: 0.8
    {O → E}:
        Confidence: 0.6 / 0.6 = 1.0
        Lift: 0.6 / (0.6 * 0.8) = 1.25
    {E → O}:
        Confidence: 0.6 / 0.8 = 0.75
        Lift: 0.6 / (0.8 * 0.6) = 1.25
{K, E}:
    K: 1.0, E: 0.8
    {K → E}:
        Confidence: 0.8 / 1.0 = 0.8
        Lift: 0.8 / (1.0 * 0.8) = 1
    {E → K}:
        Confidence: 0.8 / 0.8 = 1.0
        Lift: 0.8 / (0.8 * 1.0) = 1
{K, Y}:
    K: 1.0, Y: 0.6
    {K → Y}:
        Confidence: 0.6 / 1.0 = 0.6
        Lift: 0.6 / (1.0 * 0.6) = 1.0
    {Y → K}:
        Confidence: 0.6 / 0.6 = 1.0
        Lift = 0.6 / (0.6 * 1.0) = 1.0

{O, K, E}:

{O,K,E}: 0.6
{O,K}: 0.6, {K,E}: 0.8, {O,E}: 0.6
O: 0.6, K: 1.0, E: 0.8

{{O,K} → {E}}:
Confidence: 0.6 / 0.6 = 1.0
Lift: 0.6 / (0.6 * 0.8) = 1.25
{{E, K} → {O}}:
Confidence: 0.6 / 0.8 = 0.75
Lift: 0.6 / (0.8 * 0.6) = 1.25
{{O, E} → {K}}:
Confidence: 0.6 / 0.6 = 1.0
Lift: 0.6 / (0.6 * 1.0) = 1.0

```
confidence(A → B) = support(A ∪ B) / support(A)
lift(A → B) = support(A ∪ B) / (support(A) × support(B))

confidence({bread, milk} → {cheese}) = support({bread, milk, cheese}) /
support({bread, milk}) = 2 / 3 = 0.67

lift({bread, milk} → {cheese}) = support({bread, milk, cheese}) /
(support({bread, milk}) × support({cheese})) = 2 / (3 × 3/4) = 1.33
```

e)

{{O,K} → {E}}:
Confidence: 0.6 / 0.6 = 1.0
Lift: 0.6 / (0.6 * 0.8) = 1.25
{O → E}:
Confidence: 0.6 / 0.6 = 1.0
Lift: 0.6 / (0.6 * 0.8) = 1.25

This is the strongest rule output as it has the highest confidence and lift and contains the most number of items for all frequent itemsets that have the same measures. Additionally, we can say that the rule {O → E} is also a strong rule because it has the same level of confidence and lift as {{O,K} → {E}.

Exercise 2 - The FP-Growth Algorithm:

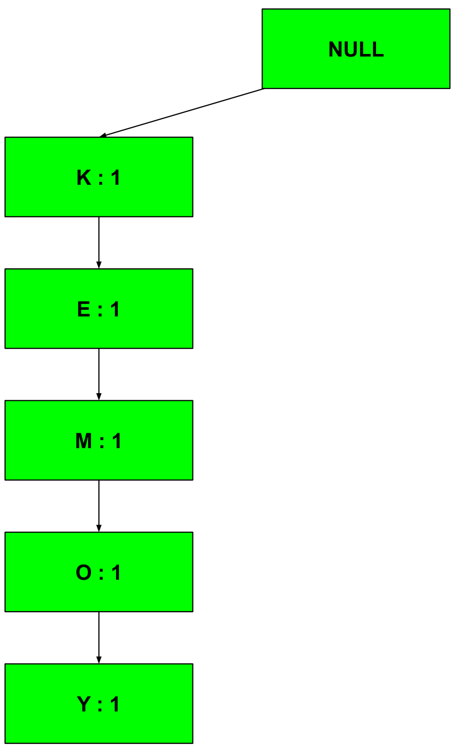
a) **F-List**
{K}: 1.0, 5
{E}: 0.8, 4
{M, O, Y}: 0.6, 3

{C, N}: 0.4, 2
{A, D, I, U}: 0.2, 1

Frequent Pattern set = {K : 5, E : 4, M : 3, O : 3, Y : 3}

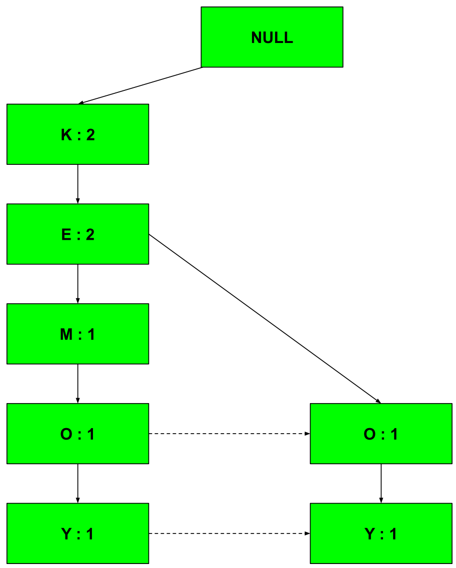
Transaction ID	Items	Ordered-Item Set
100	{M, O, N, K, E, Y}	{K, E, M, O, Y}
200	{D, O, N, K, E, Y}	{K, E, O, Y}
300	{M, A, K, E}	{K, E, M}
400	{M, U, C, K, Y}	{K, M, Y}
500	{C, O, O, K, I, E}	{K, E, O}

b) Initial FP-Tree

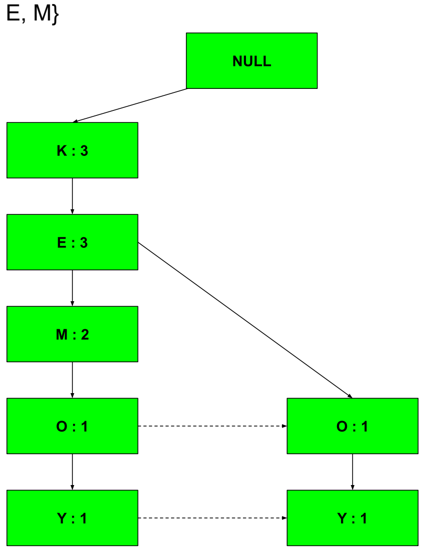


c) Executing FP_growth()

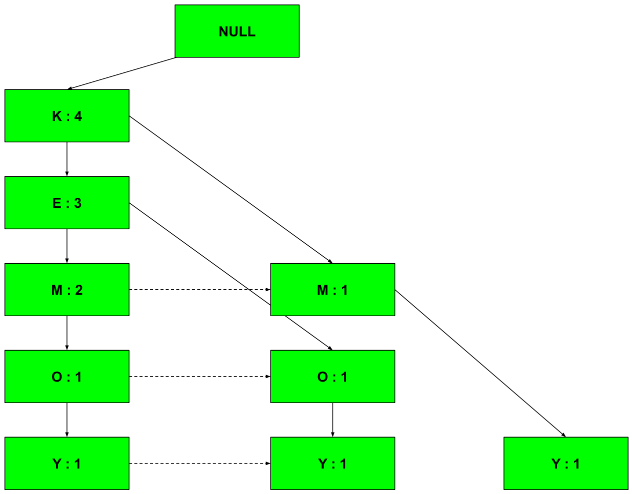
tree$_{\{K, E, O, Y\}}$



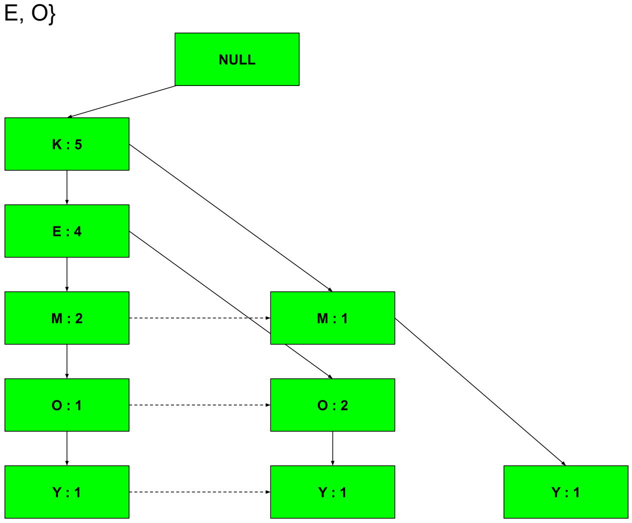
tree$_{\{K, E, M\}}$



tree$_{\{K, M, Y\}}$



tree$_{\{K, E, O\}}$



Item	Conditional Pattern Base	Conditional Frequent Pattern Tree	Frequent Pattern Generated
Y	{K, E, M, O : 1}, {K, E, O : 1}, {K, M : 1}	{K : 3}	{ K, Y : 3 }
O	{K, E, M : 1}, {K, E : 2}	{K, E : 3}	{ (K, O : 3), (E, O : 3), (E, K, O : 3) }
M	{K, E : 2}, {K : 1}	{K : 3}	{ K, M : 3 }
E	{K : 4}	{K : 4}	{ E , K : 3 }
K			

d) Comparing Apirori and FP-growth

It definitely takes less space to use the FP-growth algorithm to find frequent patterns, but it certainly took us a lot more time to compute manually compared to the Apriori algorithm.

Exercise 3 - The Eclat Algorithm:

a) **Vertical Data Format**

Transaction ID	Item	Frequency
100	M	1
100	O	1
100	N	1
100	K	1
100	E	1
100	Y	1
200	D	1
200	O	1
200	N	1
200	K	1
200	E	1
200	Y	1
300	M	1
300	A	1
300	K	1
300	E	1
400	M	1
400	U	1
400	C	1
400	K	1
400	Y	1
500	C	1
500	O	2
500	K	1

500	I	1
500	E	1

b) ECLAT algorithm with minimum support threshold set to 2:

Item	Support Count
K	5
E	4
O	4
M	3
Y	3
C	2
N	2
A	1
D	1
I	1
U	1
K, E	4
K, M	3
K, O	3
K, Y	3
O, E	3
C, K	2
K, N	2
N, Y	2
N, E	2
O, N	2
O, Y	2

M, E	2
M, Y	2
E, Y	2

Exercise 4:

	A	NOT A
B	65	40
NOT B	35	10

a) Compute the support and confidence for {A → B}

```
confidence(A → B) = support(A ∪ B) / support(A)
```

{A → B}: Support val for: A = 0.35, B = 0.40
Support: 0.65
Confidence: 0.65 / 0.35 = 1.86

Answer: Yes, this is a moderately strong rule.

b) Compute the lift for {A → B}

```
lift(A → B) = support(A ∪ B) / (support(A) × support(B))
```

{A → B}:
Lift: 0.65 / (0.35 * 0.40) = 4.64

Answer: This lift level shows how much the occurrence of A is dependent on B.

c) Compute the expected values:

To compute the expected values for each observed value in the contingency table, we can use the following formula:
Eij = (Ai * Bj) / N
where Eij is the expected count for cell (i,j), Ai is the total count for row i, Bj is the total count for column j, and N is the total count of all observations.

For (1,1):
Observed: 65
Expected: E_{ij} = (100*105) / 150 = 70

For (1, 2):
Observed: 40
Expected: E_{ij} = (50 * 105) / 150 = 35

For (2, 1):
Observed: 35
Expected: E_{ij} = (100 * 45) / 150 = 30

For (2, 2):
Observed: 10
Expected: E_{ij} = (50 * 45) / 150 = 15

d)

X2 = Σ((O-E)2 / E)

Where Σ is the sum over all cells in the contingency table, O is the observed count in a cell, and E is the expected count in that cell.

X2 = (((65-70)*2/70) + ((40-35)*2/35) + ((35-30)*2/30) + ((10-15)*2/15)) = -0.19

This does not imply dependency among A because of how low the value is

e) Rule {A → NOT B}, what is the confidence

f) Kulczynski(A, B) = |A ∩ B| / (|A| + |B| - |A ∩ B|)