Pic 16 IMDB Movie Datasets Project

Jue Wang, Xinpei Li, Jiewen Wang

1. Objectives

- To create a network of actors and actresses based on the number of times they work in the same movie.
- Perform PageRank algorithm on the network to get an importance ranking of the actors.
- Create visualization for a subset of the network using Plotly and NetworkX.
- To further analyze the data using K-means Clustering to find some key features of a successful movie.

2. Steps

2.1 Data Cleaning:

Our analysis will rely on the following three datasets:

```
actor_1 = pd.read_csv('name.basics.tsv', '\t')
cast = pd.read_csv('title.principals.tsv', '\t')
rating = pd.read_csv("title.ratings.tsv", '\t')
```

In particular, “title.principals.tsv” provides the information for a movie-cast network, while “title.basics.tsv”, “title.ratings.tsv” offers the detailed data including movies’ ratings, number of votes, and actors and actresses’ ages in a wider scope for us to study for a successful movie formula. All of the data is cleaned once read in.

Clean the actor dataset, turning all birth year from string to integer, and get rid of the ones with missing birth year. This step is crucial for K-means clustering.

```
actor_1 = actor_1[actor_1['birthYear'] != '\N']
actor_1['birthYear'] = actor_1['birthYear'].map(lambda x: int(x))
```

Clean the cast dataset, filtering out only entries whose job category is actor or actress.

```
cast = cast.dropna()
cast_actor = cast[cast['category']=="actor"]
cast_actress = cast[cast['category']=="actress"]
all_cast = pd.concat([cast_actor, cast_actress], axis = 0)
```

In order to analysis the IMDB movie data, we first take two essential variables of a movie project into consideration, movie titles and the corresponding cast. These two are provided in the

“title.principals.tsv” file with “tconst” storing the title ID and arrays of “nconst” representing the top_billed cast. Since there are a lot of repetitions in the cast variable, we use function “.drop_duplicates” to create a cleaner “all_cast” object to hold all of the actor ID. By dropping the duplicates, we remove the cases like a group of actors and actresses perform the same TV show for many episodes and keep only a unique count, which will bias the PageRank calculation. Lastly, applying “groupby” function on “all_cast”, we generate a actor_group and a cast_group by the actorID and movieID accordingly for later use.

```
all_cast.drop_duplicates(IF actorID is duplicated AND character name is duplicated)
```

(By removing duplicates and NA's, we successfully shrunk the cast down to about one thirds of its original size.)

```
len(all_cast)
```

```
3824183
```

To make our result more readable we also need to create a dictionary that gives the actual name of the movie that movieID refers to.

```
movie_name_dict = {}
for index, row in movie_name.iterrows():
    movie_name_dict[row['tconst']] = row['primaryTitle']
```

In order to save time for the algorithm, we need to create another two dictionaries based on rating dataset. The first one is {movieID : average rating}.

```
movie_rating = {}
for index, row in rating.iterrows():
    movie_rating[row['tconst']] = row['averageRating']
```

The second one is {movieID : #votes}.

```
num_votes = {}
for index, row in rating.iterrows():
    num_votes[row['tconst']] = row['numVotes']
```

2.2 Create dictionaries on movie and actor:

After cleaning up the data, we start to create a movie-actor dictionary in preparation for the construction of a network on them.

First, we use a for loop to go through the actor_group and cast_group to create a dictionary with movie ID and actor ID. The pseudo code is as following:

```
for (each group in actor_group) {
            extract actorID and number of movies by each actor(length of each group)
            actor_nummovies= {actorID : number of movies}
}



for (each group in cast_group) {
            movie_actorlist  = {movieID : list of actorID}
            IF (actor already in movie_actorlist):
                        append/extend to the dict.value
            ELSE:
                        movie_actorlist[movie_i] = [ ]
                        then extend the actorID list to it
}
```

actor_nummovies:

```
{'nm6577317': 1,
 'nm5080903': 1,
 'nm3921474': 1,
 'nm3921475': 1,
 'nm2943448': 1,
 'nm4402207': 1,
 'nm3722959': 4,
 'nm4402203': 1,
 'nm3722956': 1,
 'nm7067620': 3,
 'nm5080905': 1,
 'nm3075129': 3,
```

movie_actorlist:

```
'tt7763324': ['nm1752699', 'nm0000502'],
'tt1810373': ['nm0363641', 'nm0755267'],
'tt1810372': ['nm0444786', 'nm0363641', 'nm0217221'],
'tt4847912': ['nm2508789'],
```

By doing these, we have created two dictionaries: actor_nummovies and movie_actorlist with actorID and movieID as keys. Then in order to analysis the data on the basis of each movie, we go through the movie_actorlist to create a dictionary called “edge_movienum” of (actor 1, actor2) and the number of times they worked in the same movie.

```
for (m in movie_actorlist.items()):
            m contains [movieID, (actor1, actor2, …)]
            actor_list = m[1]
            if (#actors is less than 2):
                        get rid of this row for convenience
            else:
                        key1 = (actor1,actor2)
                        key2 = (actor2,1)
```

```
# storing the common movies between any two actors
if key1 in edge_movienum:
       increment values of key1] and[key2] by 1
else :
       Initiate the value to be 1
```

edge_movienum:

```
('nm2353149', 'nm1685070'): 2,
('nm7425076', 'nm7425075'): 1,
('nm3139929', 'nm3610334'): 1,
('nm4923841', 'nm1822243'): 1,
('nm1880423', 'nm1884929'): 1,
('nm0716279', 'nm0896035'): 1,
('nm0829558', 'nm0571220'): 1,
('nm9501719', 'nm1089615'): 1,
('nm0999317', 'nm0044481'): 3,
```

Next, we create the weight of each pair in a new dictionary called “edge_movienum_weight”:

```
for keypair in edge_movienum.items():
              keypair[1] = the frequency of pair<actor1, actor2>
              keypair[0] = <actor1, actor2>
              weight = keypair[1] / (total movie number of actor1)
              edge_movienum_weight = {key = keypair[0] : value = weight}
```

edge_movienum_weight:

```
{('nm0739635', 'nm1368516'): 0.16666666666666666,
 ('nm0733051', 'nm0382914'): 0.03571428571428571,
 ('nm1359969', 'nm0501854'): 1.0,
 ('nm2354456', 'nm1507995'): 0.25,
 ('nm0174495', 'nm0482378'): 0.007142857142857143,
 ('nm0002198', 'nm0561158'): 0.011764705882352941,
 ('nm5741835', 'nm0509699'): 1.0,
 ('nm4922627', 'nm6276170'): 0.058823529411764705,
```

(**Note**: a value of one indicates that the two actor in the key does not have connection elsewhere.)

2.3 Create a network using the dictionaries

After we get all the weights, we want to make a subset of edge_movienum_weight to let the weights be more relevant to what we want.

```
new_weight = {}
for pair,weight in edge_movienum_weight.iteritems():
    if (weight is higher than 0.7 and weight is lower than 1.0):
        Append weight to dictionary.value

nummovies = {}
```

```
for pair, weight in new_weight.iteritems():
    if (first actor already in the nummovies dictionary):
        nummovies[pair[0]] += 1
    else:
        nummovies[pair[0]] = 1
    if (second actor already in the nummovies dictionary):
        nummovies[pair[1]] += 1
    else:
        nummovies[pair[1]] = 1
```

We create a dictionary, new_weight, to indicate those weights is larger than 0.7 and lower than 1.0, which is more relevant. After doing this, we create a new dictionary nummovies to count the number of movies for each actor in new_weight has. Thus this is a dictionary which has key as pairs of actor and value as numbers of movies for each of them attend.

```
Then we want to make a subset to of new_weight.
weight_plot = {}
for pair,weight in new_weight.iteritems():
    if (in each pair two actors' amounts of movies are larger than 9):
        weight_plot[pair] = weight
```

By doing this, we can get pairs of actors who attend more than 9 movies in total, so we can make the dataset smaller and more specific.

Then we want to know who are these vectors so we make a list named selected_cast to store them.

```
selected_cast = []
for item in weight_plot.items():
    x, y = item[0]
    if x in selected_cast:
        pass
    else:
        append first actor name to the list
    if y in selected_cast:
        pass
    else:
        append second actor name to the list
```

By doing this we get all actors' number. And then we want to find their names.

```
labels = {}
for i in selected_cast:
    labels[i] = actor_1[actor_1['nconst'] == i]['primaryName'].tolist()[0]
```

We find actor's name in actor_1. By creating a dictionary named labels, we make keys as actorID and values as their name. The dictionary is shown below.

```
{'nm0046307': 'Walter Baele',
 'nm0097635': 'Olga Borys',
 'nm0142044': 'John Cartier',
 'nm0166559': 'Clara Cleymans',
 'nm0256105': 'Tine Embrechts',
 'nm0259301': 'Anders Eriksson',
 'nm0296093': 'Per Fritzell',
 'nm0335175': 'Kerstin Granlund',
 'nm0393443': 'Ann Hood',
 'nm0401992': 'Gillian Humphreys',
 'nm0484822': 'Helen Landis',
 'nm0581896': 'Nathalie Meskens',
 'nm0710129': 'Peter Rangmar',
 'nm0728072': 'Jan Rippe',
 'nm0906822': 'Michael Wakeham',
 'nm0969005': 'Ludmila Diakovska',
 'nm1099293': 'Ivan Pecnik',
 'nm1222640': 'Kostadin Georgiev',
```

Then we want to change actorID in weight_plot to their actual names

```
new_weight_plot = {}
for pair, w in weight_plot.iteritems():
    new_weight_plot[(labels[pair[0]], labels[pair[1]])] = w
```

By doing this, we create a new_weight_plot dictionary. And let weight_plot's keys be changed to name. The output of new_weight_plot dictionary is shown below

```
{('Anastasia Hale', 'Evelyn Callifrax'): 0.75,
 ('Anastasia Hale', 'Louis Joe Sherbe'): 0.75,
 ('Andreia Miranda', 'Miguel Trindade'): 0.8,
 ('Ann Hood', 'Gillian Humphreys'): 0.75,
 ('Ann Hood', 'Helen Landis'): 0.75,
 ('Bilguun Ariunbaatar', 'Julia Pietrucha'): 0.8888888888888888,
 ('Chris Van Espen', 'Clara Cleymans'): 0.7746478873239436,
 ('Chris Van Espen', 'Nathalie Meskens'): 0.8309859154929577,
 ('Clara Cleymans', 'Chris Van Espen'): 0.7142857142857143,
 ('Clara Cleymans', 'Nathalie Meskens'): 0.8441558441558441,
 ('Clara Cleymans', 'Walter Baele'): 0.7532467532467533,
 ('Desi Slava Desislava Doneva', 'Vanya Dzhaferovich'):
0.9285714285714286,
```

We make a new list to get the real name of actors.

```
new_selected_cast = []
for i in selected_cast:
    new_selected_cast.append(labels[i])
```

By creating new_selected_cast, we get an empty list. And then we append element in labels to get the real name of each cast.

2.4 Perform pagerank algorithm on the network

In this part we want to calculate pagerank of each actor.

```
G = nx.DiGraph()
G.add_nodes_from(new_selected_cast)
for pair, w in new_weight_plot.items():
    x, y = pair
    G.add_edge(x, y, weight = w)
```

By doing this, we create nodes for each actor in the new_selected cast. And then add edges between actors in new_weight_plot to show they linked together.

```
pr = nx.pagerank(G,alpha = 0.85)
```

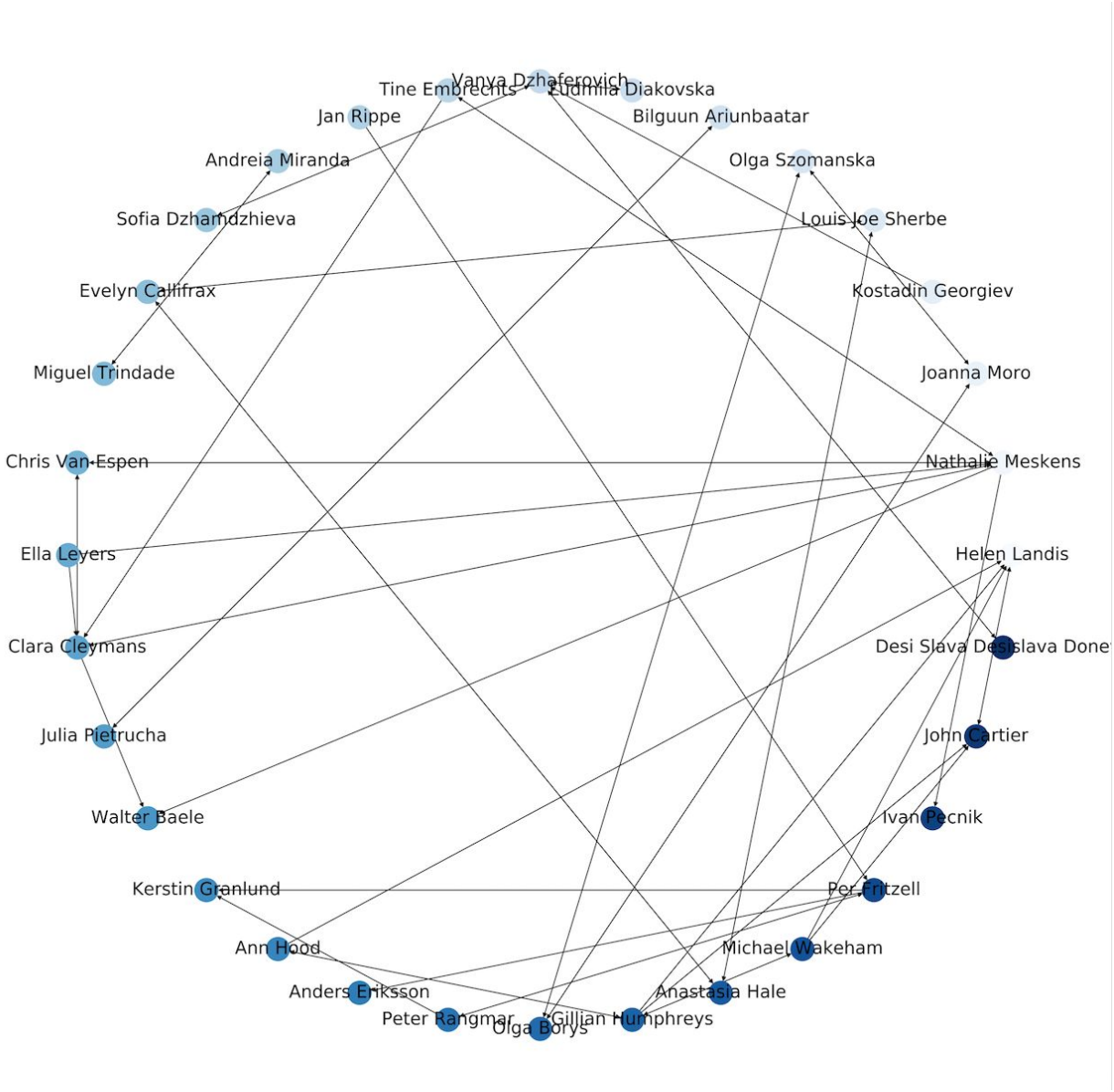
We then calculate pagerank for each actor. We see the descending sorted page rank of actor below.

```
In [47]: sorted_pr
Out[47]:
[('Vanya Dzhaferovich', 0.09771172892412147),
 ('Helen Landis', 0.08873059604892702),
 ('John Cartier', 0.0867116260822305),
 ('Bilguun Ariunbaatar', 0.041085814608721155),
 ('Andreia Miranda', 0.041085814608721155),
 ('Miguel Trindade', 0.041085814608721155),
 ('Julia Pietrucha', 0.041085814608721155),
 ('Joanna Moro', 0.04108581460872115),
 ('Louis Joe Sherbe', 0.04108581460872115),
 ('Olga Szomanska', 0.04108581460872115),
 ('Evelyn Callifrax', 0.04108581460872115),
 ('Olga Borys', 0.04108581460872115),
 ('Anastasia Hale', 0.04108581460872115),
 ('Ludmila Diakovska', 0.03385149063830516),
 ('Sofia Dzhamdzhieva', 0.03385149063830516),
 ('Desi Slava Desislava Doneva', 0.03385149063830516),
 ('Nathalie Meskens', 0.029540897947528726),
 ('Clara Cleymans', 0.026166455809183947),
 ('Walter Baele', 0.018483646533023437),
 ('Chris Van Espen', 0.018108790002283163),
 ('Per Fritzell', 0.01564527365861368),
 ('Kerstin Granlund', 0.015399642214975362),
 ('Gillian Humphreys', 0.012393758946818533),
 ('Ivan Pecnik', 0.01106443559179138),
 ('Tine Embrechts', 0.010634473891157827),
 ('Anders Eriksson', 0.010595699744565763),
 ('Peter Rangmar', 0.010318648023315957),
 ('Ann Hood', 0.008796545982814867),
 ('Michael Wakeham', 0.008796545982814867),
 ('Kostadin Georgiev', 0.00616287220456887),
 ('Jan Rippe', 0.00616287220456887),
 ('Ella Leyers', 0.00616287220456887)]
```

2.5 Visualize the network using Networkx and Plotly

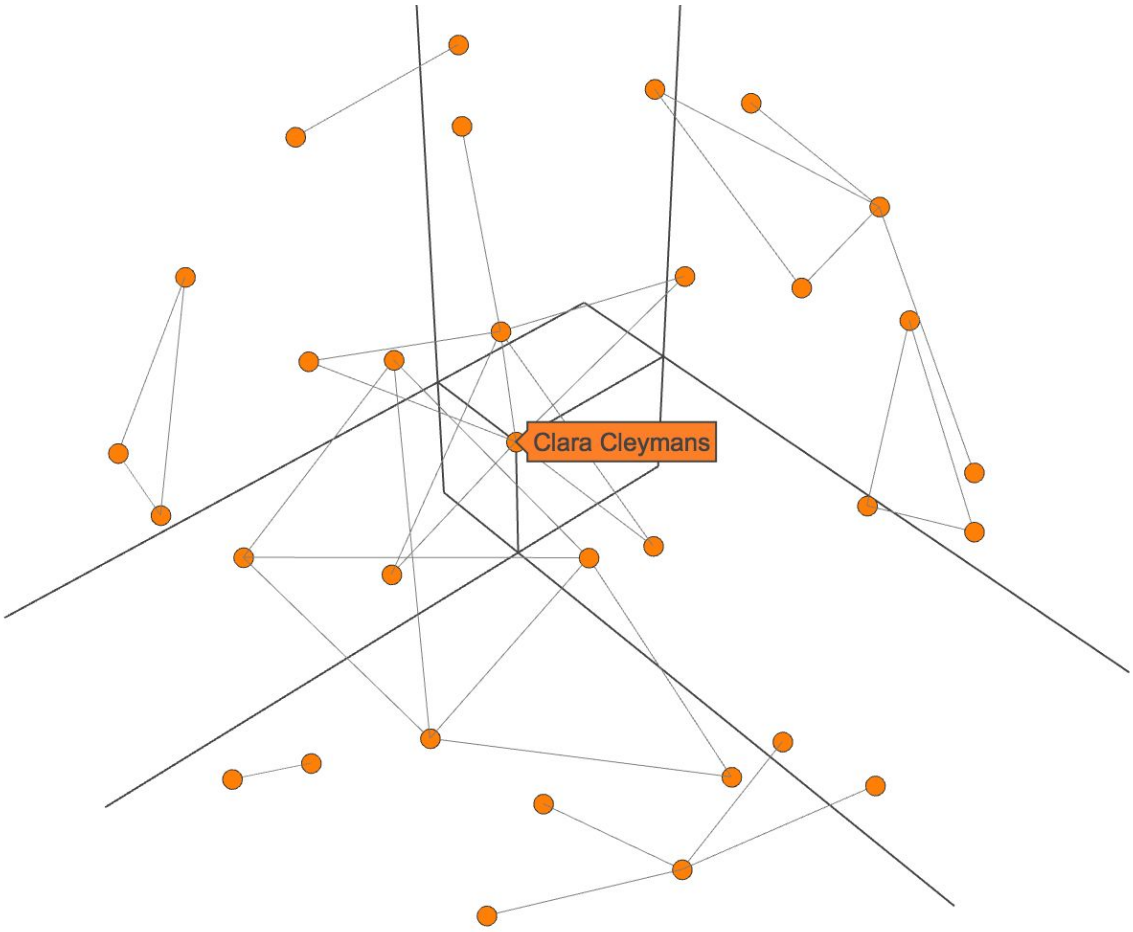
2.5.1 Using NetworkX

```
nx.draw_circular(G,font_size = 20, with_labels = True,
node_color=range(len(new_selected_cast)), node_size = 700, cmap=plt.cm.Blues)
```

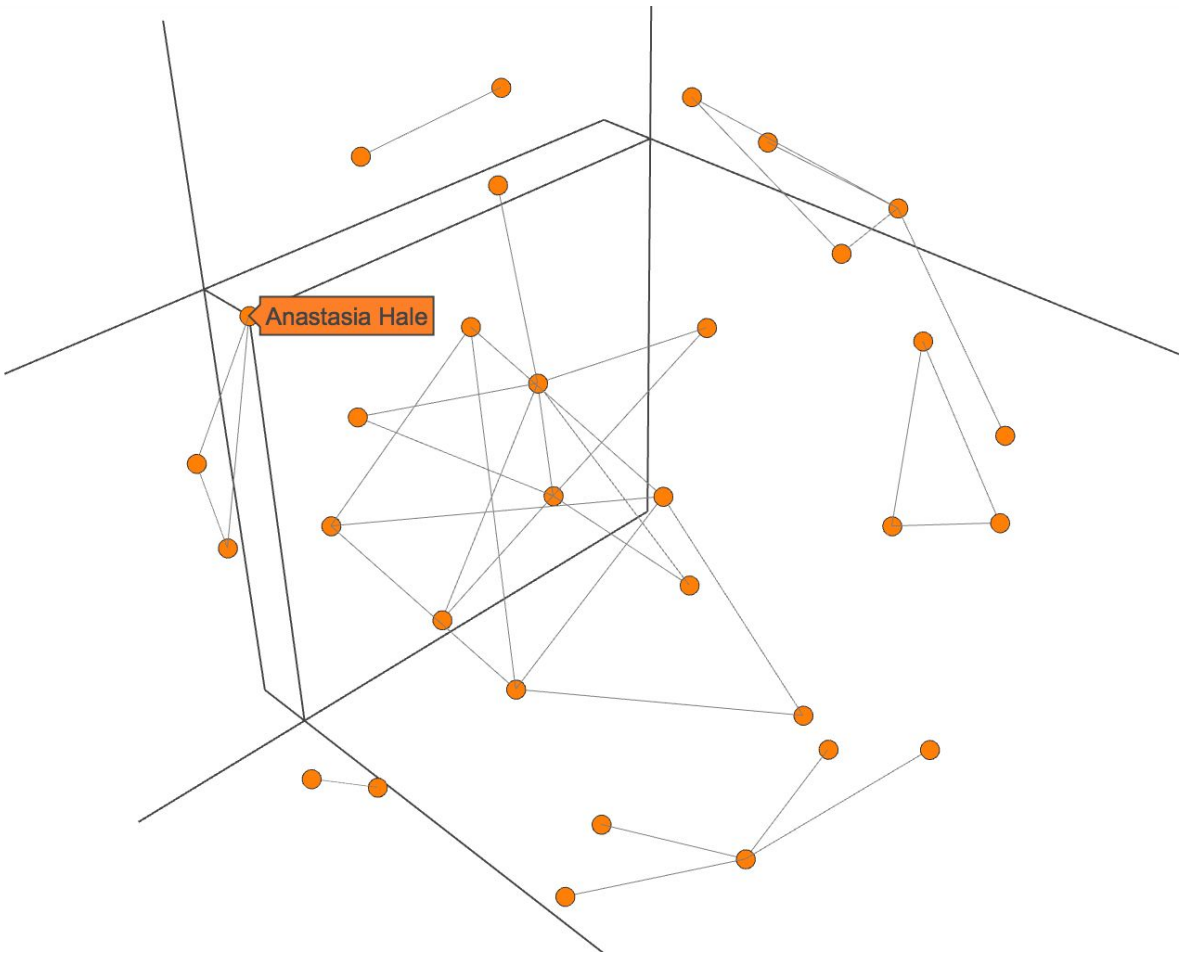


(The darker the color is, the higher the pagerank is.)

2.5.2 Using Plotly



We can see which actor each node represents by moving the mouse on the node. It is a 3D graph so it is explained in a more detailed way in the video. Using a 3D graph instead of a 2D one, we can make the graph more visually pleasing and informative, since user can move the cursor to rotate, zoom in and out the graph to get a view of a particular subgraph.




```
import plotly.plotly as py
import plotly.graph_objs as go
trace1=go.Scatter3d(x=Xe,
                y=Ye,
                z=Ze,
                mode='lines',
                line=dict(color='rgb(125,125,125)', width=1),
                hoverinfo='none'
                )
trace2=go.Scatter3d(x=Xn,
                y=Yn,
                z=Zn,
                mode='markers',
                name='actors',
                marker=dict(symbol='dot',
                              size=6,
                              colorscale='Viridis',
                              line=dict(color='rgb(50,50,50)', width=0.5)
                              ),
                text = labels,
                hoverinfo='text'
                )
axis=dict(showbackground=False,
          showline=False,
          zeroline=False,
          showgrid=False,
          showticklabels=False,
          title=''
          )
```

We only provide a small part of the code chunk here since it is a bit long. For the detailed code please see the attached .py file.

2.6 Classification by K-means Clustering

The several datasets given us contain more categorical than numerical features of the movies. In order for K-means clustering to work well, we selected 4 features that we believe may affect the success of a particular movie: #Votes, #Ratings received, Average age of the cast at the time of production, and #Movies the leading actor has been in. We narrowed down our analysis on the movies of rating > 7.0 and #Votes > 10000 so the algorithm runs faster. This subsetting allow us to see what traits does one of the cluster have that make them more successful.

We subset the rating dataset using the criterion rating > 7.0 and #Votes > 10000.

```
movie_subset = rating[rating > 7.0 and #Votes > 10000]
```

2.6.1 Creating a dictionary of movies and average age of the cast at the time of production

We already have the dictionary of actorID to birth year in step 1 when we cleaned the data.

```
'nm0295581': 1986,
'nm1797636': 1970,
'nm7356899': 1980,
'nm0295585': 1964,
'nm1126790': 1978,
'nm1797630': 1973,
'nm0351656': 1961,
'nm5270363': 1929,
'nm0351655': 1944,
'nm0824897': 1967,
'nm1499052': 1966,
```

We need a dictionary **“movie_year”** of movies and their start year via similar steps.

```
for index, row in movie_name.iterrows():
    movie_year[row['actorID']] = row['start year']
```

With these two dictionaries and the help of the cast_group we generated in previous steps, we can easily create another dictionary **“movie_avg_age”** of {movie: average cast age}

```
for i, j in movie_actorlist.iteritems():
    if i in movie_year:
        total_age = 0
        n = 0
        for k in j:
            if k in actor_birth:
                total_age += (movie_year[i] - actor_birth[k])
            else:
                n += 1
        if len(j) is Not n: # prevent dividing by 0
            movie_avg_age[i] = total_age / (len(j) - n)
```

With similar procedure two more dictionaries “num_votes” and “rating” are created using the movie dataset.

2.6.2 Building the dataset to cluster from

The dataset should be in the form of a NumPy array. Using “movie_subset” that we generated before, we provide its entries as follow:

```
train_data = np.zeros([len(movie_subset), 4])
movie_subset = movie_subset.reset_index(drop=True)
for i in range(4156):
    a = movie_subset.loc[i]['averageRating']
    b = movie_subset.loc[i]['numVotes']
    c = movie_avg_age[movie_subset.loc[i]['movieID']]
    d = actor_nummovies[movie_actorlist[movie_subset.loc[i]['movieID']][0]]
    train_data[i,:] = [a, b, c, d]
```

2.6.3 Using K-means clustering to cluster the dataset

We decided to cluster the dataset into 3 groups using these lines of code:

```
from sklearn.cluster import KMeans
kmeans = KMeans(3)
kmeans.fit(train_data)
```

We then subset the dataset into 3 groups according to the result of K-means.

```
a = [0, 1, 1] # the labels of the 3 groups
seq_0 = []
seq_1 = []
seq_2 = []
for i in range(len(a)):
    if a[i] == 0:
```

```
        seq_0.append(i)
    if a[i] == 1:
        seq_1.append(i)
    if a[i] == 2:
        seq_2.append(i)

group_1 = movie_subset.loc[seq_0]
group_2 = movie_subset.loc[seq_1]
group_3 = movie_subset.loc[seq_2]
```

3. Analysis on the K-means Clustering Result

We need to check the set of movie names, average rating and number of votes in the 3 clusters created in the last step. We hope to see one group being more successful in general than the other two, and we aim to explore the common features of this particular group, so that the formula for making a successful movie is made clear.

We first inspect the list of movies in each cluster, to see if there is any common features that each cluster have.

Part of Group_1

```
'The Martian',
'Bridge of Spies',
'Rogue One: A Star Wars Story',
'La La Land',
'The Nice Guys',
'Baby Driver',
'Guardians of the Galaxy Vol. 2',
'Manchester by the Sea',
'Avengers: Infinity War',
'Mr. Robot',
'John Wick: Chapter 2',
'Stranger Things',
'Split',
'Moonlight',
'Dunkirk',
'Three Billboards Outside Ebbing, Missouri',
'Get Out',
'The Shape of Water']
```

Part of Group_2

```
'The Hangover',
'Shutter Island',
'Harry Potter and the Deathly Hallows: Part 2',
'Gran Torino',
'Iron Man 3',
'The Dark Knight Rises',
'Inception',
'The Hunger Games',
'Mad Max: Fury Road',
'Deadpool',
'Gravity',
'Sherlock',
'The Walking Dead',
'Django Unchained',
'Guardians of the Galaxy',
'Gone Girl',
'Star Wars: The Force Awakens']
```

Part of Group_3

```
'Tales from the Crypt',
'The Abyss',
'The Adventures of Baron Munchausen',
'Born on the Fourth of July',
'Casualties of War',
'The Cook, the Thief, His Wife & Her Lover',
'Crimes and Misdemeanors',
'The Killer',
'Do the Right Thing',
'Time of the Gypsies',
'Driving Miss Daisy',
'Drugstore Cowboy',
'Field of Dreams',
'Glory',
'Heathers',
'Henry V',
'Knick Knack',
'Lean on Me',
...]
```

From the above list, we can see that movies in group_1 and group_2 are more well-known than group_3, with high grossing movies such as Iron Man and Dunkirk.

Among the three clusters, group_1 has the smallest amount of movies inside, which are mostly blockbusters. We predict that this will be the group that represents successful movies. We need to further verify this idea by looking at its other features such as average rating among all the movies in the group.

```
ratings_1 = 0.0
for index, row in group_1.iterrows():
    rate = movie_rating[row['tconst']]
```

```
    ratings_1 = ratings_1 + rate
avg_rating_1 = ratings_1 / len(group_1)
avg_rating_1
```

Group	Average Rating
1	7.783
2	8.370
3	7.789

This table further confirms our prediction that group_2 is our target group with the most successful movies.

We then inspect the features in group_2 to see what the formula for making a successful movie is. We first compute the average cast age using one of the dictionaries we constructed in the first step.

```
age_2 = 0
n = 0
for index, row in group_2.iterrows():
    if row['tconst'] in movie_avg_age:
        age_2 += movie_avg_age[row['tconst']]
    else:
        n += 1
avg_age_2 = age_2 / (len(group_2) - 1)
```

The average age for the cast in group_2 is 33 at the time of the production of each movie. Using similar process we compute the average age for group_1 is 37. The average age for group_3 is extremely low because we set average age to be 0 if the birth year is missing. Hence, age do matters when making a movie, and a relatively young cast will tend to attract more audience and make the movie more successful. A more successful movie tends to have a average cast age around 33.

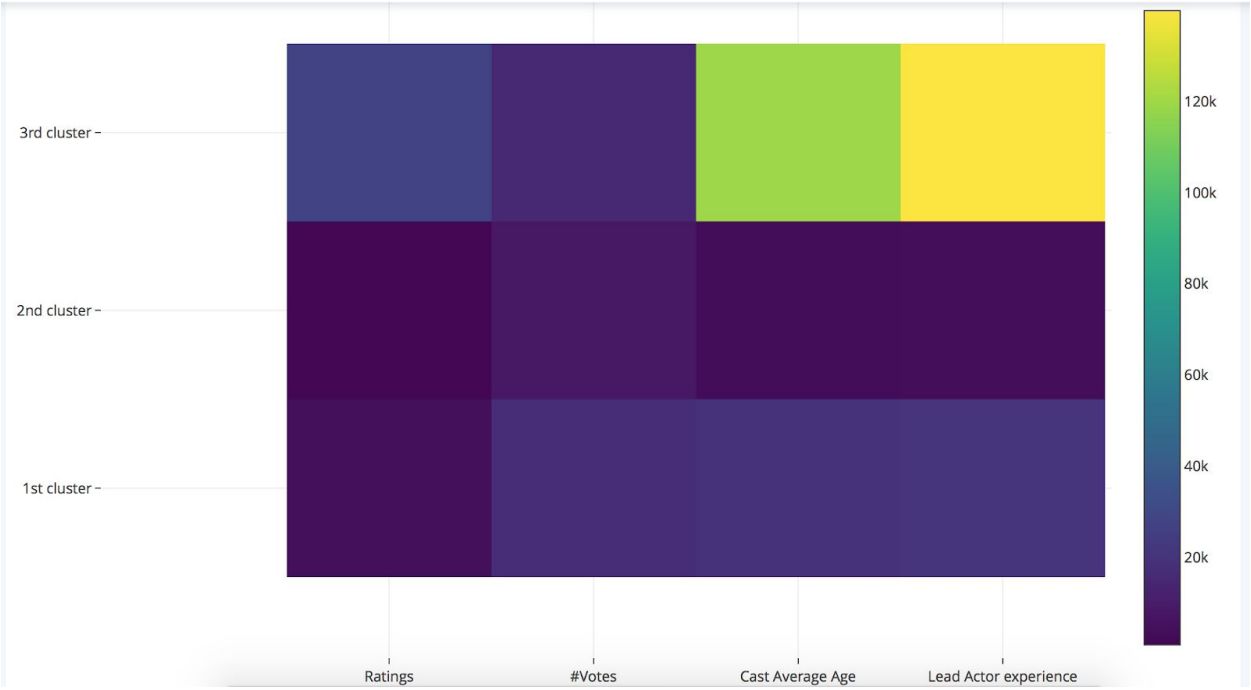
We then compute the average number of movies that the lead actor has been in.

```
num_movies = 0
n = 0
for index, row in group_2.iterrows():
    if row['tconst'] in movie_actorlist:
        num_movies += actor_nummovies[movie_actorlist[row['tconst']][0]]
    else:
        n += 1

avg_num_movies_2 = num_movies / (len(group_2) - n)
```

We compute the number of movies the lead actor has been in for group_2 to be 43, meaning it is crucial for a movie to have a leading actor who is very experienced.

We further create a plot using plotly to see the effectiveness of the clustering. The graph shows that lead actor experience and cast average age are two more important features when determining whether a movie will be successful.



4. Further Improvements

1. We need a better way of subsetting the network. Instead of filtering the connections whose weight in the network are larger than a certain value, we need to use a smarter algorithm to get the subset of the network that has the most number of connections, so the graph would be more clustered.
2. We can incorporate more features in the K-means clustering algorithm so the classification is more robust.
3. Instead of unsupervised learning, we can pre-process the data by first giving them some labels based on some kind of criterion. In this way we may get a clearer picture of the classification of the movies.

5. Conclusion

- We can use PageRank algorithm to get an importance ranking of actors and actresses. 3D plot is a more efficient and visually pleasing way of displaying the information.
- Using K-means clustering technique, we found that average cast age, lead actor experience, number of votes of the movie and average ratings are four important features of making a successful movie. We would suggest that directors cast actors and actresses who have appeared in more than 40 movies, and to make the cast's average age to be about 33.