IMDb Database Exploration

Lei Ding[a], Yi Ding[a], Sonu Mishra[b]

[a]*UCLA Electrical Engineering Department*

[b]*UCLA Computer Science Department*

Objective

In this project, we use social network theory to analyze a dataset that contains movies and actors. Based on the dataset, we generate an actor network and a movie network and use pagerank and community detection to analyze the properties of the network. Each network node has several attributes that contain information like genre, rating or neighbors. After that, we extract features from the networks and try to predict the rating of each movie using machine learning regression algorithms. Various experiments show the efficacy of the approaches taken.

Question 1

Download actors.list.gz, actresses.list.gz (or use the actor_movies.txt and actress_movies.txt files from the cleaned up data), merge those 2 lists into one file, and remove all actors/actresses with less than 5 (so actors who have acted in four or fewer number of movies) movies; Note that you will have to parse the data in these lists as accurately as possible to extract the entities consistently and create the network. So plan on spending some time in cleaning the data set.

Solution

We first combine the two files `actor_movies.txt` and `actress_movies.txt` into one file where we filter the actors with less than five movies. After carefully analyzing the given data, we find that the movie data should be cleaned. Each movie name includes the actual movie name, the year, some actor related information following the year and spaces. For example, "Night of the Demons (2009) (uncredited)". Movies with the same name may be released in different years. So we have to keep the year in the movie record. But we have to remove "(uncredited)" and spaces in the front and the end of this example. Regular expression is a perfect way to do this data processing. So we utilize it in our code.

And we find several kinds of actor related information following the year as below:

Table 1.1: List of actor related information that we removed from movie names

"(uncredited)"	"{{SUSPENDED}}"
"(uncredited voice)"	"(rumored)"
"(voice)"	"(credit only)"
"(VG)"	"(as .*)" where ".*" represents any set of characters

After this we obtained 244K actors and a list of movies they acted in. Actually, we faced two challenges in this question when we do this part with Python.

Challenges

1. In Python, we can use `dict` to realize the hash table. We first used the syntax `k in dict.keys()` to find the (key, value) pair. But it was running extremely slow. Then we found out that `dist.keys()` will return a list not a `dict` and hence `k in` is looked for in a list instead of a hash table. So each `k in dict.keys()` operation runs in time O(n) not O(1) that we were expecting. Therefore we changed it to `k in dict` and achieved 20x increase in speed.
2. In this question, we used regular expression to do data cleaning. But when we tried to remove the (as .*) as shown in Table 1.1**,** we used the **"**`\(+as .*\)`**"** to match it. But Python uses a Greedy approach and finds the longest string that starts with "`\(+as`" and ends with "`\)`". Therefore it removes all the characters between them due to which we lose out many movies.

   For example, if the list of movies for a particular actor is **"**Movie 1 (as pilot), Movie 2, Movie 3 (as cop), Movie 4**"**, eliminating **"**`\(+as .*\)`**"** will yield **"**Movie 1, Movie 4**"**, We are losing out two movies. To circumvent this issue and get a non-greedy behavior we added a **'**?**'** to it as **"**`\(+as .*?\)`**"**. Now it searches for the smallest string matches the pattern.

Question 2

Construct a weighted directed graph G(V; E) from the list and calculated weight as required.

Solution

In this problem, we construct the required weighted graph in following steps:

1. Create a (movie → actors) hash table. To make the hash table more efficient, we map each actor into a distinct number. We then create another hash table (Actor_Number → Actor_Name) which can help us to find out the name of actors. In this process, we will sort the order to facilitate the following steps.
2. Create a (edge → edge count) hash table, where each vertex is a movie. The detail are:
    a. We traverse the (movie → actors) hash table.
    b. For each (key, value) pair, we traverse the value which is a list of actor numbers. We then find out any pair of actors in the value which represents an edge in the actor network.
    c. Judge if the edge already exists in the (edge → edge count) hash table's key list. If it does, then add 1 to its edge count. If not, add the pair (edge, 1) to the hash table. In this way, we can get the edge count for each edge, which can help us calculate the weight for the edge.
3. Based on 2, we use the method shown below to create a weighted edgelist, where $|S_i \bigcap S_j|$ and $|S_i|$ can be obtained directly from edge counts in 2.c. Each line in edgelist files is formatted as follows:

    ```
    (Start vertex, End vertex, edge weight)
    ```

    $$V = \{\text{all actors/actressess in list}\}$$
    $$S_i = \{m | i \in V, \text{m is a movie in which i has acted}\}$$
    $$E = \{(i, j) | i, j \in V, S_i \cap S_j \neq \phi\}$$
    and for each *directed* Edge $i \rightarrow j$, a weight is assigned as $\frac{|S_i \cap S_j|}{|S_i|}$.

4. We create the weighted edgelist in Python, save it in a txt file and then use R `igraph` package to process it, i.e., use the `read.graph` function to create the actor graph.

Challenges

What challenged us most in this part is how to select efficient data structure to construct several intermediate data and how we could refine our code to make it faster under such a huge network data. We finally used hash table to create the edgelist file in Python and map actor names to numbers to save memory and avoid string comparison. Therefore the key data structure in this question was hash tables, which can reduce our running time greatly.

Amazon Web Service Usage

To make our program run faster and store more intermediate data, we created an AMI which is specially designed for R in AWS with 30GB memory. We chose the c4xlarge EC2 machine which is optimized for heavy computing work. The reason why we chose such instance is because we observed that our program often took 100% or even boost to over 100% of the CPU resources when run on other machines or instances. So our work required fast CPUs. We used AWS in the rest questions too.

Question 3

Run pagerank algorithm on the actor/actress network, look into those who are among top 10, do you know their names? List the top 10 famous movie celebrities in your opinion, what are their pagerank scores? Do you see any significant pairings among actors? Any major surprises, in the sense that well-known actors do not show up in the high pagerank list?

Solution

After we constructed the graph using `read.graph` function, we used the `page.rank` function in R to find the page ranks of the nodes (actors) in the graph. We list the top 10 actors as shown in Table 3.1 with their page ranks.

Table 3.1 Top 10 Actor List

Actor Name	Page Rank
Flowers, Bess	0.0001553105
Tatasciore, Fred	0.0001363839
Harris, Sam (II)	0.0001327491
Blum, Steve (IX)	0.0001244093
Jeremy, Ron	0.0001228186
Miller, Harold (I)	0.0001160455
Downes, Robin Atkin	0.0001006273
Sayre, Jeffrey	0.0001002628
Lowenthal, Yuri	0.0000996321
Phelps, Lee (I)	0.0000994408

However, we do not even know these actors. Most of these actors are very old and we do not know about them. We list top 10 of our favorite actors and their ratings in Table 3.2.

Table 3.2 Top 10 Actor List for Us

Actor Name	Page Rank
Cruise, Tom	0.0000029995
Cavill, Henry	0.0000039473
Renner, Jeremy	0.0000017532
Fimmel, Travis	0.0000021040
Johansson, Scarlett	0.0000028016
McAvoy, James	0.0000050201
DiCaprio, Leonardo	0.0000031488
Winslet, Kate	0.0000017223
Eisenberg, Jesse	0.0000031858
Ruffalo, Mark	0.0000016676

Discussion

Comparing these two list of top 10 actors, we find our top 10's ranking scores are quite low. This may due to that what we love are those fashion famous stars who starred films like *Captain America, Xmen*, *Warcraft* and *Now You See Me*. Maybe these actors become famous recently, and not for a long time. We also calculated the average pagerank rating which is 0.0000040985 which is still larger than most of the our top 10 favourite actors. This leads us to think more about the reasonability of the pagerank. There may be some flaws in using pagerank to rate actors: Pagerank ranks actors just base on the number of links. Therefore someone who has acted in more movies will get a higher pagerank. Good actors tend to work in hardly 20-60 movies. Those who work in many movies are mostly non-central characters in the movie. Pagerank approach ranks someone who acted as a non-central character in 100 movies higher than the one who was a lead actor in 30 movies.

Question 4

Similarly, remove all movies with less than 5 actors/actresses on list, construct a movie network according to the set of actors/actresses, with weight assigned as the jaccard index of the actor sets of 2 movies. Now we have an undirected network instead.

Solution

In this question, we first get the actor - movie data from Question 1 and then we do the following steps:

1. Create a hash table for Movie → Actor similar to Question 1.
2. Filter the movies with less than 15 actors in it.
3. Create a hash table for (Actor → Movie) based on (Movie → Actor). In detail, we will traverse the (Movie → Actor) hash table. For each key - value pair, we will traverse the value which is a list of actors and try to find if each actor is already in the (Actor → Movie) table. If it does, we add the movie to the actor's value. If not, we add a (actor, movie) pair to the (Actor → Movie) hash table.
4. Based on the (Actor → Movie), we create a hash table for (edge → edge weight). To be specific, we traverse the (Actor → Movie) hash table and for each key-value pair, we traverse the value which is a list of movies and choose every two of the movies as a edge. Then we let the edge be a key of the (edge → edge weight) hash table and if the key originally exists, we add increase the value by 1. If the key does not exist, we will create a key-value pair and set the value as 1. The value represents the number of actors common in two movies. We can use it to calculate the weight of each edge.
5. Based on the (edge → edge weight) hash table, we use jaccard index to calculate the weight of each edge. During the 1-4 process, we have saved the number of actors a movie has. The jaccard index can be calculated as below, where A and B can be seen as the actors in movie A and movie B:

$$J(A,B) = \frac{A \cap B}{A + B - A \cap B} \tag{4.1}$$

6. Create the edgelist using (edge → edge weight) for generating the movie graph just as Question 2 does. Use `read.graph` in R to generate the movie graph.

Challenges

This part relates to Question 5 closely. So how we filter data will affect Question 5 a lot. Question 5 is about community detection, which is a heavy computing work. And we tried as the spec says to filter the movies with less than 5 actors but it took over one and a half day but did not finish. We tried several ways to fast the computing like using a faster CPU but it did not improve the performance much. So after enquiring the TA, we changed the filter threshold to 15, which means we filtered out movies with less than 15 movies. In this way, we could finish the community detection in 6-7 hours.

Question 5

Do a community finding on the movie network; use the Fast Greedy Newman algorithm. Tag each community with the genres that appear in 20% or more of the movies in the community. Are these tags meaningful?

Solution

Based on the movie graph generated from Question 4, we use the function `fastgreedy.community` to do the community detection and analyze the community structure of this movie network. However, what the Question 4 says is not very clear. I first try one way as the Question 4 exactly says. That is we combine actor_movies and actress_movies as in Question 1 but without filtering out the actors with less than 5 movies. And then we follow the steps in Question 4 and generate a movie graph.

In this way, we get about 140K movies and over 3350K actors or actresses. After the community detection algorithm, we get 172 communities. About 70 of these communities have less than 4 members and about 50 of these movies have less than 3 members. It means that there are too many communities containing very few members and most members gather in few communities. It is not reasonable. It is also easy to understand. If we do not remove these actors who have less than 5 movies, there will be a lot of movies connecting to just few other movies. In this way, these isolated movies will be considered as a very small community. So we quit this way of cleaning data.

In another way, we use the data got from Question 1 where actors with less than 5 movies are filtered out. And we take such data and follow steps in Question 4 and generate a movie graph. Now we get about 79K movies and 244K actors. We then do a community detection using `fastgreedy.community` algorithm and get 26 communities. The modularity of these community structure is 0.75. Using the movie_genres data, we get the genre for each movie. And for each community, we calculate the distribution of genres and find out those taking over 20%. The result is shown in Table 5.1.

Table 5.1 Community Genres

Community ID	Genre	Percentage (%)
1	Drama	21.9
	Romance	21.3
2	Drama	24.1
3	Comedy	40.5
	Drama	26.0
4	None	19.96 for largest one
5	Drama	30.5
6	Drama	30.8
7	Comedy	23.1
	Drama	24.7
8	Drama	20.8
9	Drama	21.4
10	Western	22.2
11	Drama	22.2
12	Drama	26.7
	Romance	27.4
13	Comedy	25.0
	Drama	50.0
	Romance	20.0
14	Comedy	21.7
	Drama	23.9
15	Drama	23.4
16	Drama	25.8

17	Comedy	26.5
	Drama	28.3
18	Drama	37.1
	War	20.1
19	Short	85.3
20	Drama	24.9
21	Drama	33.3
	Sport	33.3
22	Drama	25.0
	Short	25.0
	Thriller	25.0
23	Comedy	100.0
24	Romance	75.0
25	Short	87.5
26	Comedy	20.0
	Short	60.0
	Thriller	20.0

Discussion

About whether these tags are meaningful, we think it is. We can first look at the community structure. The modularity is 0.75 which shows the efficiency of the community detection. Then we can see that `Drama` is the most popular movie genre. About 76.9% of the community are marked as `Drama`. And one community usually can be marked as the combination of `Drama` and other genres. This is easy to understand. `Drama` is a general genre, which contains other genres. Also, we can see `Short` is also similar to `Drama` in this way because `Short` movies can usually be marked as other genres. And we do see some weird community like community with genre `Comedy` and `Thriller`. `Comedy` and `Thriller` seem to have little in common.

But in our mind, the graph is constructed based on movies actors. Sometimes, actors will act in movies across genres. In this case, `Comedy` and `Thriller` may exist in one community. From this point, we can see that this graph has some limitations. But totally, we think the tags are meaningful.

Question 6

Add the following nodes into the network, For each of them, return the top 5 nearest neighbors. Which communities does each of them belong to?

- ➢ *Batman v Superman: Dawn of Justice (2016)*
- ➢ *Mission: Impossible - Rogue Nation (2015)*
- ➢ *Minions (2015)*

Solution

Based on the graph created in Question 4 and Question 5, we can use the function `incident()` and `neighbor()` in R to get the neighbors and induced edges for each of the three movies above. We use the edge weight as the metric to determine whether a neighbor is nearest. After finding the top 5 nearest neighbors for each of the three movies, we use the fastgreedy community object to determine which community the neighbors belong to. The result is shown in Table 6.1.

Table 6.1 Nearest neighbor information

New Movie	Top 5 neighbor	Community ID
Batman v Superman: Dawn of Justice (2016) Community ID: 4	Eloise (2015)	4
	Into the Storm (2014)	4
	Grain (2015)	4
	Man of Steel (2013)	4
	Love and Honor (2013)	4
Mission: Impossible - Rogue Nation (2015) Community ID: 4	Fan (2015)	4
	Phantom (2015)	4
	Breaking the Bank (2014)	4
	The Program (2015/II)	4
	The Rise of the Krays (2015)	4
Minions (2015) Community ID: 4	The Lorax (2012)	4
	Inside Out (2015)	4

	Despicable Me 2 (2013)	4
	Gake no ue no Ponyo (2008)	4
	Up (2009)	4

Discussion

We can see from **Table 6.1** all the three movies and their neighbors belong to `Community 4`. From the point that these three movies are in the same community with their top neighbors, we can see that this community structure is reasonable. Movies with similar attributes will connect to each other and stay in the same community. And `Community 4` doesn't have a genre that takes 20% of the movies. The top genres in `Community 4` are `Comedy` (11.2%), `Drama` (19.8%), `Romance` (10.9%) and `Thriller` (19.9%). We can see that `Community 4` is complex and multiplex. The genres for the three movies are `Sci-Fi`, `Thriller` and `Family` respectively. And these genres take 4.6%, 19.9% and 1.95% respectively. From this we can see that although the community structure can represent some information about movies, it cannot cover a lot. This may be due to the limitation of the network which is constructed by actors in movies.

Question 7

Download the ratings list, derive a function to predict the ratings of the above 3 movies using the movie network. (hint: try to use the ratings of neighbor movies and movies in the same community.)

Solution

For this problem, we are trying to predict a movie's rating based on other related movies' ratings. In this way, we have to find those movies that have close relationship to the required movie. So based on the network structure, the neighbors of the movie and the movies in the same community may contain the potential rating information. We design two steps to do the prediction:

1. Find out the common movies which are the neighbors of the required movie and in the same community as well.
2. Select the top n of the common movies and calculate the mean value of them as the predicting value for the required movie, where we set n as 3, 4, 5 and calculate the predicting value respectively.

Table 7.1 shows the results for the prediction. We also list the real ratings of these three movies which we find in **IMDb**.

Table 7.1 Prediction of Rating using Communities

Value of n	Batman v Superman: Dawn of Justice (2016)	Mission: Impossible - Rogue Nation (2015)	Minions (2015)
Real Rating	7.1	7.5	6.4
3	5.9	8.6	7.6
4	6.6	8.6	7.6
5	6.3	8.4	7.7

Discussion

From the table, we can see that the average error of each movie is around 1. And we can see that setting the value of n to 4 is a good choice. But we still think this prediction is not good enough as the error is relatively large. In our mind, it is still due to the limitations of the network. If we

use this way to predict a movie's rating, we have to make sure that the network can represent the rating information well. However, the network is built based on actors common among movies. There are some cases where actors act in a wide range of movies, which may lead to two movies connecting to each other without much common features. Although we choose top movies to make this method more reasonable, this limitation still cannot be removed.

Question 8

Using a set of features that include the following:

- *top 5 pageranks of the actors (five floating point values) in each movie.*
- *if the director is one of the top 100 directors or not (101 boolean values). These are directors of the top 100 movies from the "IMDb top 250". You can also find a list of these movies in the ratings.list.gz file.*

train a regression model and predict the ratings of the 3 movies mentioned above. Specify the exact feature set you use and how you compute the numerical values for these features. Compute and state the goodness of fit for your regression model.

Solution

The questions involves cleaning and reading the director data. In this data, we have a list of movies corresponding to each director. The names of the movies often have extra information like “co-director”, “uncredited”, “rumored only”, etc. For cleaning we took an approach similar to Question 1 by using regular expression substitutions in Python. We eliminated all such extra information other than the movie name and year of release. As explained before, many movies are released with the same name in different years, year of release is a critical information to distinguish between them. We also obtained the page ranks of the actors, similar to Question 3, and stored them. This was done using `page.rank` function in igraph library in R.

After the cleaning and initial pre-processing, we obtained the required features mentioned in the question above.

- For each movie, we sorted the page-ranks of the actors and selected top 5 of them. The idea is that the movie rating will have decent correlation with the top 5 actors.
- For each movie we have 100 boolean features, each representing a director of top 100 movies on IMDb. If the movie has a director among top 100, we indicate this by putting 1 in the corresponding feature. We have an additional boolean variable that indicates the movie has no director in top 100.

Apart from these, we also included an additional feature representing the average rating of the genre of the movie. This is calculated by finding the average of ratings of all the movies falling in the same genre.

Feature analysis:

We now have 107 features for each movie. In order to understand how effective they are in estimating the overall rating of the movie, we found the correlation of all these features with the rating of the movie. We found that most features have very small correlation with the overall rating. I am including below a subset of 10 features and their correlations with the rating.

10 Best Features									
Genre	Actor 1	Actor 2	Actor 3	Actor 4	Actor 5	DirNon	Dir 1	Dir 2	Dir 3
0.25	0.12	0.12	0.11	0.10	0.09	-0.10	0.03	0.03	0.03

Regression Models

We utilized `caret` and `AppliedPredictiveModeling` packages in R to run various regression algorithms on our data. We did a 10-fold cross validation and tuned the parameters.

- Cubist Regression with 20 committees and neighbours
- Linear Regression. This is unreliable because our feature space is too sparse and therefore can potentially be rank deficient.
- Random Forest on reduced feature set with Actor 1-5, Genre ratings and DirNon features.
- Gradient Boosting on reduced feature set with n.trees = 150, interaction.depth = 3, shrinkage = 0.1 and n.minobsinnode = 10

The estimates of ratings and the RMSE values of the models are given in Table 8.1

Table 8.1 Prediction of Rating using Actor page-rank and top100 director

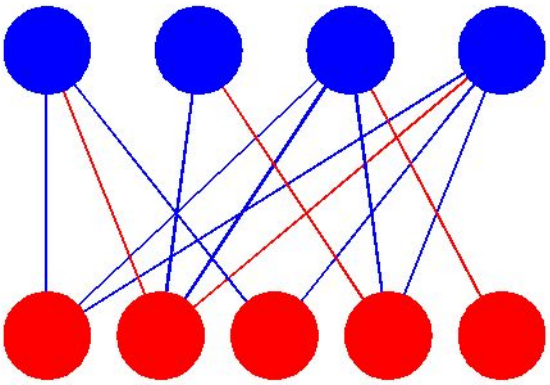
Approach	**RMSE**	**Batman v Superman: Dawn of Justice (2016)**	**Mission: Impossible - Rogue Nation (2015)**	**Minions (2015)**
Real Rating	NA	7.1	7.5	6.4
Linear Regr	1.12	6.6	6.8	6.9
Cubist	1.10	7.1	7.3	7.3
Random Forest	0.85	6.0	6.3	6.9
Gradient Boost	1.10	5.8	6.0	6.1

Question 9

(Bonus) Try predicting the ratings of these movies with a different approach. Construct a bipartite graph, with actors/actresses representing the vertices of one part and movies representing the vertices of the other part. An actor/actress is connected to all the movies he has played in. Assign a score to each actor based on the ratings of the movies he has played in (it's on you to define an appropriate metric here; natural choices are the average among all ratings, the highest, or average among the top ones). Then predict the ratings of the new movies based on the scores you have assigned to the actors.

Solution

A bipartite graph is one in which the vertices are divided into two sets such that there are no links between the edges of same sets. For example in the following network, vertices/nodes are divided into two sets. Each set is shown in different color. Vertices in red set have edges only to the vertices in the blue set, and vice versa.



In this problem we construct a bipartite graph, with actors/actresses representing the vertices of one part and movies representing the vertices of the other part. An actor/actress is connected to all the movies he has played in. The high level idea is that we will assign to each actor a score based on the ratings of the movies they have acted in. This will be our model. When we get a new movie, we can find its rating based on the scores of the actors who have acted in this movie. There can be many possible ways to assign scores to actors. We primarily followed two approaches, explained shortly. Using each of these approaches, we find the ratings of the 3 new movies given in Question 6, and compare those to the actual IMDb ratings of these movies.

Hubs and Authority:

In web graph, an authority page is defined as one which has many incoming links from hub pages, and a hub page is defined as one which has many outgoing links to authority pages. Here we assign to each vertex two scores: authority score and hub score. The authority score of a page is the sum of the hub scores of all the pages pointing to it. The hub score of a page is the sum of the authority scores of the pages it points to. This is an iterative approach similar to the page-rank algorithm. Score update equations are:

$$A(p) = \sum_{(q,p) \in E} H(q) \quad \text{and } H(p) = \sum_{(p,r) \in E} A(r) \tag{9.1}$$

As our graph is bipartite, we do not assign two scores to each vertex. We consider movies as authorities and their ratings as authority scores. We consider actors as hubs and assign to them hub scores. The challenge however is to fit our problem in this approach. We made following modifications to this hubs and authority (HITS) algorithm.

1. *No iteration or normalization*: Without normalization, HITS algorithm often diverges. But we do not have any fixed sum of all the ratings and scores of movies and actors, respectively. The only way to circumvent this issue was to assign hub scores to actors in just one iteration based on the ratings (authority scores) of movies.
2. *Weighted average of ratings*: Rather than taking the sum of ratings (authority scores) of the movies the actor has acted in, take a weighted average. The movies with more actors will get more weights and those with just handful actors will get less weight.
3. *Weighted average of scores*: When we get a new movie whose rating we have to estimate, we find the weighted average of the scores of the actors who have acted in this movie. Actors who have acted in more movies get more weight. We understand that great actors often are very selective about the movies they do and hence do just 30-50 movies in their lifetime. On the other hand, common actors do lots of movies. In order to circumvent a scenario in which we give more weight to an ordinary actor who has acted in 100 movies more than a great actor who acted in just 50 movies, we put an upper

bound on the weights. All actors who have acted in 50 or more movies get the same weight.

The ratings of the 3 movies estimated using this approach is shown in Table 9.1.

Learning the scores:

In another approach, rather than assigning scores to actors heuristically, we consider learning it by formulating it as a linear regression problem. If we consider this bipartite graph in the form of an adjacency matrix where each row represents a movie and each column represents an actor, we observe that we can directly assume this matrix as a feature matrix.

Let us consider this toy example in which we have only 3 movies and 4 actors. The required adjacency matrix representation of the bipartite graph is the matrix X below. $X[i][j] = 1$ if movie i had actor j. Our Y matrix for regression will be the ratings of the movies.

$$X = \begin{bmatrix} 1 & 0 & 0 & 1 \\ 0 & 1 & 0 & 0 \\ 1 & 1 & 1 & 0 \end{bmatrix} \qquad Y = \begin{bmatrix} 7.5 \\ 6.8 \\ 7.1 \end{bmatrix}$$

Linear regression is the most suitable approach for our problem, because it assigns a coefficient (score) to each feature (boolean corresponding to each actor). When we get a new movie we can estimate its ratings just by taking the sum of the weights assigned to its actor.

Challenges:

1. *Huge Amount of data*: We are dealing with huge amount of data. We have over 100K movies and over 100K actors. Storing it will be extremely difficult. This would require huge amount of memory. We observed that although the matrix is huge, it is sparse. In Python, we have `scipy.sparse` library that has efficient ways to store matrices, e.g. `sparse.lil_matrix` and `sparse.csr_matrix`.
2. *Curse of Dimensionality*: Our data is very high-dimensional. It has over 100K features. We cannot apply dimensionality reduction method like PCA because we want to obtain weights (scores) corresponding to each actor. Apart from not very accurate, the regression model would run too slow with these many features. To circumvent this issue

we used the stochastic gradient descent version of linear regression in `SkLearn` library in Python. If the regression equation is $y = Ax$ each SGD step is of the form

$$x_{t+1} = x_t + \eta\,(y_t - a_t^T x_t)\,a_t x_t + 1 = x_t + \eta\,(y_t - a_t^T x_t)a_t \qquad (9.2)$$

Now if each row a_t^T has just z entries on average, each update step will take around $2z$ multiplication and z additions. Estimated ratings are shown in Table 9.1.

Table 9.1 Prediction of Rating using Bipartite graph

Approach	**Batman v Superman: Dawn of Justice (2016)**	**Mission: Impossible - Rogue Nation (2015)**	**Minions (2015)**
Real Rating	7.1	7.5	6.4
HITS	6.6	6.6	6.9
SGD	7.9	7.4	6.8

We can see that linear regression with SGD outperforms approximated HITS algorithm. This estimates the ratings close to the actual ratings with small positive bias. We believe, actual ratings depend on many factors other than just cast; given that our model performs a good job.