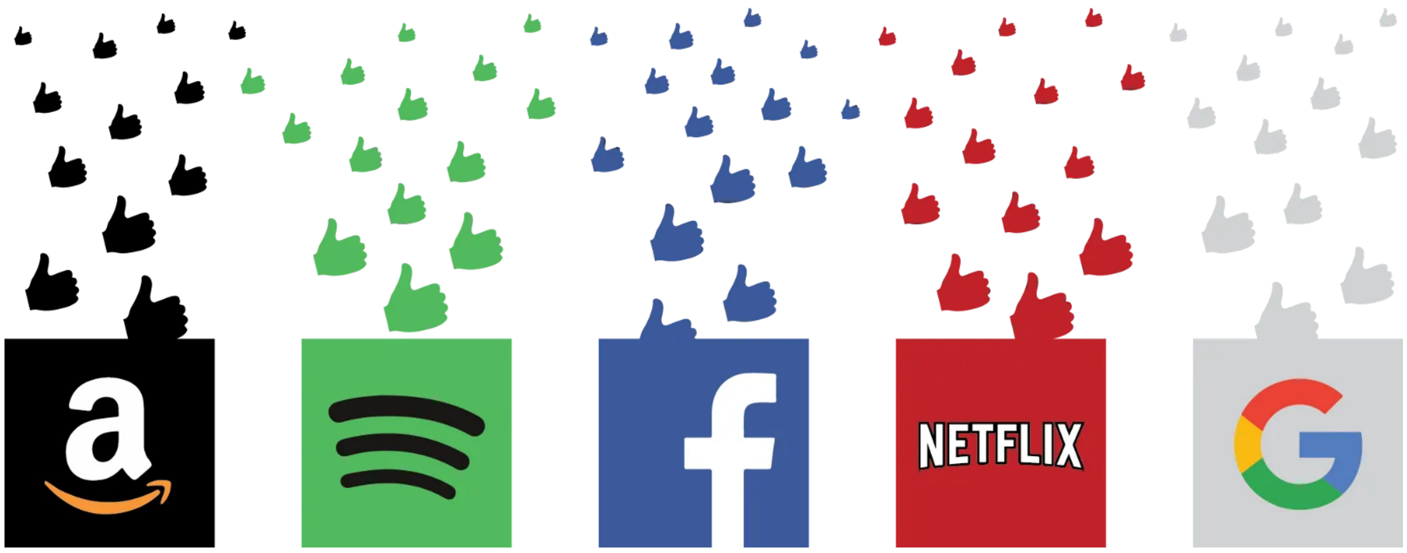
Project T Final: Collaborative Filtering

CS 189/289A: Introduction to Machine Learning, Fall 2020

Team MA: Maxwell Chen and Abinav Routhu



(The Data Scientist: What is the right way to build a recommender system for a startup? May 3, 2018)

Introduction

This notebook will cover various methods to construct a recommender system through the process of collaborative filtering -- algorithms and techniques that are concerned with finding similarities between users and items, and calculating numerical ratings to quantify this similarity. We will ground ourself in a standard and accessible application of recommender systems -- recommending movies to users on a service such as Netflix.

Historical background

The modern age is undergoing rapid and intense changes due to the vast amounts of data being generated in the early 21st century thanks to the Information Age and inventions such as MOSFETs, digital electronics, and the internet. Understanding and leveraging said data has led to the rise of machine learning and data science -- practitioners of these fields are becoming indispensible to virtually every industry. One such "industry" we will focus on in particular is advertising and marketing, which have radically changed through the inception of recommender systems. We see this everywhere -- Amazon products, Spotify songs, Facebook friends, YouTube videos, Google ads -- all these companies are using the concept of learning from data to predict new products to users and customers.

The 2006 Netflix Prize was a $1,000,000 challenge run by Netflix to find the best collaborative filtering algorithm that could improve Netflix movie recommendations. This ran for three years until a team comprised of research scientists bested Netflix's own prediction accuracy by 10.06\%. The contest ignited interest in recommendation and perhaps led to the growth of machine learning competitions through website such as Kaggle.

Learning Objectives

This notebook serves to introduce and explore the topic of Collaborative Filtering through mathematical methods, along with practical application to the task of recommending movies to users.

Collaborative Filtering is a process or algorithm to filter information or patterns through the collaboration of multiple users, agents, or data sources.

We shall approach this through two paradigms:

1. Memory-Based Approaches (Clustering, KNN)
2. Model-Based Approaches (Matrix Factorization)

Table of Contents

In [1]:
```python
# Load Packages
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
import seaborn as sns
```

To begin, we will load the MovieLens Dataset. MovieLens was a research project launched by GroupLens Research at the University of Minnesota, and was one of the earliest modern projects that investigated personalized recommendations via recommender systems. We will be using their Dataset for a similar purpose: recommending a user which movie to watch based on their own interests or preferences.

Question 1: Loading the Dataset

1.1: Import `zipfile` and `from urllib.request import urlretrieve` . Use these libraries to load the MovieLens Dataset (http://files.grouplens.org/datasets/movielens/ml-100k.zip) -- this is the "small" Dataset containing 100,000 ratings. If you are up for it, you can also load the expanded MovieLens Dataset (http://files.grouplens.org/datasets/movielens/ml-latest.zip) -- this contains 27,000,000 ratings. For the purposes of this assignment, loading either Dataset will work.

Reference Material:

- DataCamp Tutorial on zipfile module (https://www.datacamp.com/community/tutorials/zip-file)
- GeeksForGeeks Tutorial on zipfile module (https://www.geeksforgeeks.org/working-zip-files-python/)
- urllib.request Documentation (https://docs.python.org/3/library/urllib.request.html)

In [2]:
```
from urllib.request import urlretrieve
import zipfile
### BEGIN CODE ###
urlretrieve("http://files.grouplens.org/datasets/movielens/ml-100k.zip", "movi
elens.zip")
zip_ref = zipfile.ZipFile('movielens.zip', "r")
zip_ref.extractall()
### END CODE ###
```

1.2: We now have a raw .csv file containing our Dataset. As with many other problems involving machine learning or data mining, we must manipulate our raw data to a form that we can use.

First, investigate the structure of the zipped dataset we just downloaded. Open up each of the unzipped files on DataHub or your local machine, and describe the contents of each file:

Answer:

- `links` contains IDs to identify each movie. Each movie is associated with its ID on IMDB (Internet Movie DataBase) or TMDB (The Movie DataBase).

- `movies` contains IDs to identify each movie. Each entry has a 'title' field containing the name of the movie as well as the year it was released, and a 'genres' field consisting of the different genres the movie falls under, separated by vertical bar characters (the | character).

- `ratings` contains ratings from users on different movies. Each row corresponds to a user with a userID, who rates a movie with a movieID, on a scale from 1 to 5. The timestamp of the rating is also given.

- `tags` contains comments from users on the movies they reviewed/rated. Each row corresponds to a user with userID, a movie with movieID, and a short string containing a 'tag' for that movie. There may be multiple rows for the same user and the same movie, implying there are multiple tags that user assigned to that movie.

1.3: Use your knowledge of data cleaning and processing from the first week of 16ML to load the different .csv files into multiple Pandas DataFrames. Use the provided columns stored in `user_features` , `ratings_features` , and `movie_features` . Use appropriate naming conventions for these DataFrames, such as "movies", for example. Then combine the Dataframes into a single DataFrame, using `user_id` as a primary key.

Hint #1: When using `pd.read_csv` , you MUST use the flag `encoding='latin-1'` to properly read from the files.

Hint #2: Use `sep="|"` when reading in the csv file

In [3]:
```
user_features = ["user_id", "age", "sex", "occupation", "zip_code"]
### BEGIN USERS CODE ###
users = pd.read_csv("ml-100k/u.user", sep="|", names=user_features, encoding=
'latin-1')
### END USERS CODE ###


ratings_features = ["user_id", "movie_id", "rating", "unix_timestamp"]
### BEGIN RATINGS CODE ###
ratings = pd.read_csv("ml-100k/u.data", sep="\t", names=ratings_features, enco
ding='latin-1')
### END RATINGS CODE ###


movie_features = ['movie_id', 'title', 'release_date', "video_release_date",
"imdb_url", "genre_unknown", "Action",
                  "Adventure", "Animation", "Children", "Comedy", "Crime", "Do
cumentary", "Drama", "Fantasy",
                  "Film-Noir", "Horror", "Musical", "Mystery", "Romance", "Sci
-Fi", "Thriller", "War", "Western"]
### BEGIN MOVIES CODE ###
movies = pd.read_csv('ml-100k/u.item', sep='|', names=movie_features, encoding
='latin-1')
### END MOVIES CODE ###


### BEGIN MERGE CODE ###
all_data = ratings.merge(movies, on='movie_id').merge(users, on='user_id')
### END MERGE CODE ###
all_data.head()
```

Out[3]:

	user_id	movie_id	rating	unix_timestamp	title	release_date	video_release_date	
0	196	242	3	881250949	Kolya (1996)	24-Jan-1997	NaN	http:/ exa
1	196	257	2	881251577	Men in Black (1997)	04-Jul-1997	NaN	http:/ exa
2	196	111	4	881251793	Truth About Cats & Dogs, The (1996)	26-Apr-1996	NaN	http:/ exa
3	196	25	4	881251955	Birdcage, The (1996)	08-Mar-1996	NaN	http:/ exa
4	196	382	4	881251843	Adventures of Priscilla, Queen of the Desert, ...	01-Jan-1994	NaN	http:/ exa

5 rows × 31 columns

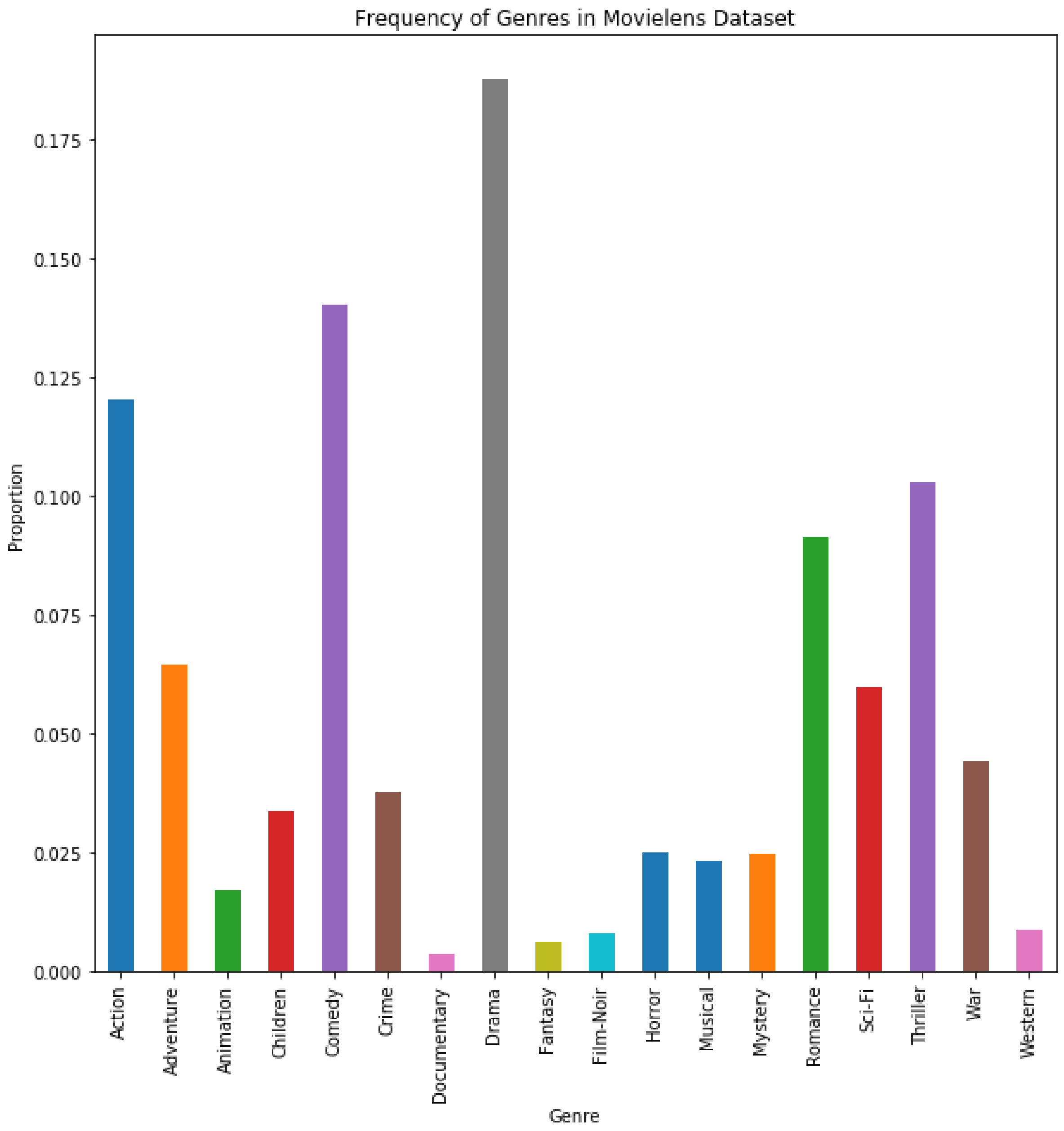
Question 2: Understanding and Visualizing the Dataset

2.1: Distribution of Movie Genres

Make a plot of the frequency of each distribution in the dataset. Refer back to material from Week 1 and 2 if you need a refresher on using Pandas and Matplotlib.

Hint: Try a bar plot

In [4]:
```
### BEGIN CODE ###
movie_column_labels = all_data.columns[9:27]

movie_genres = all_data[movie_column_labels]
genre_count = movie_genres.sum()
genre_count_sum = sum(genre_count)
genre_frequency = genre_count / genre_count_sum
### END CODE ###

plt.figure(figsize=(10, 10));
plt.title("Frequency of Genres in Movielens Dataset");
plt.xlabel("Genre");
plt.ylabel("Proportion");
genre_frequency.plot.bar();
```



2.2: It is important to identify biases in our dataset that can skew our results or impact how generalizable our recommendation system is to novel users and novel movies. What might be some issues we run into by using this dataset?

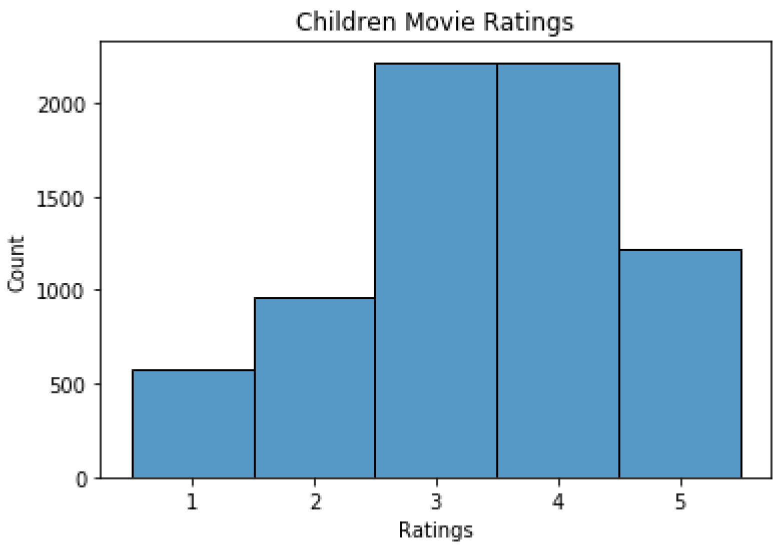
Answer: From the plot, we see that the different genres are not equally represented: We have a high proportion of Drama, Comedy, Action, Thriller, and Romance films, while in contrast, we have very low representation for Documentary, Fantasy, Film-Noir, and Western Movies. This could present a problem if we want to make recommendations for users who like the less-represented films, as that means there is less data to work with.

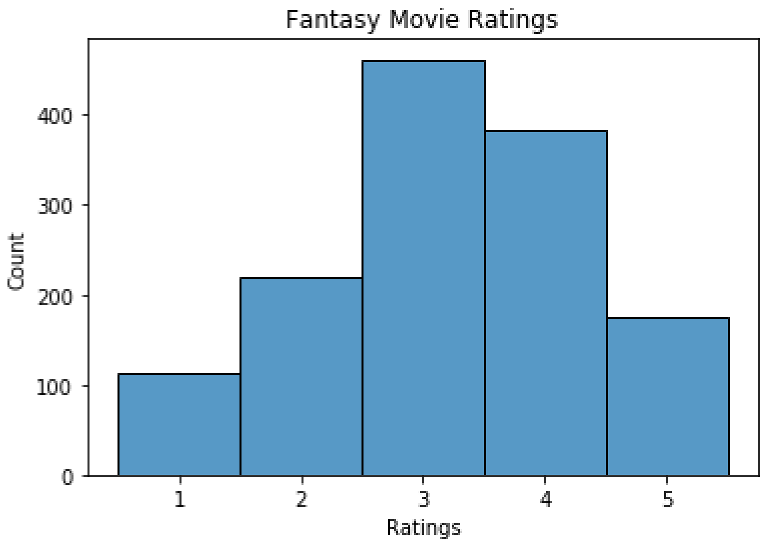
2.2: Distribution of User Ratings

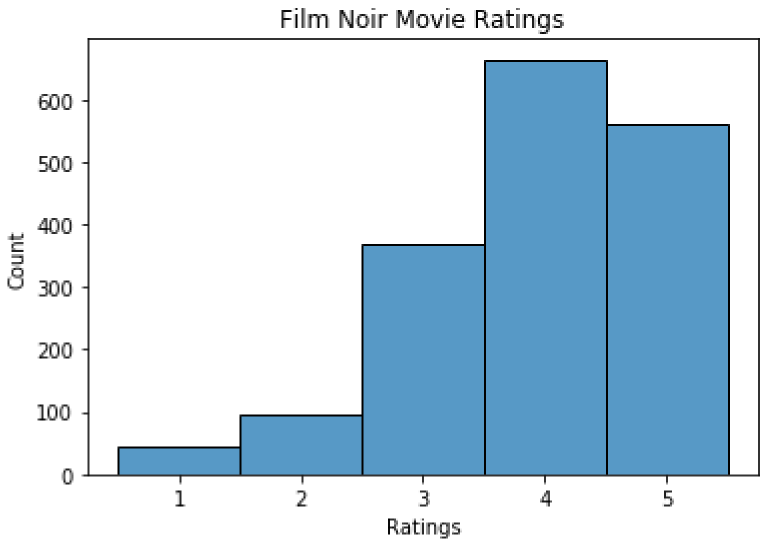
Plot the distribution of user ratings for movies from the Children, Fantasy, and Film-Noir genres -- that is, for each genre, plot a distribution describing the number of ratings from 1 to 5 received by movies belonging to that genre. Note any similarities or differences between your plots -- how does this inform us about biases in the dataset, and how could such bias affect our predictions?

[Hint: Try multiple histograms or bar plots]

In [5]:
```
### BEGIN CODE ###
children_ratings = all_data[all_data["Children"] == 1]['rating'].astype('str')
.sort_values()
fantasy_ratings = all_data[all_data["Fantasy"] == 1]['rating'].astype('str').s
ort_values()
film_noir_ratings = all_data[all_data["Film-Noir"] == 1]['rating'].astype('st
r').sort_values()
### END CODE ###

### BEGIN PLOTTING ###
sns.histplot(data=children_ratings, label="Children")
plt.title("Children Movie Ratings")
plt.xlabel("Ratings")
plt.ylabel("Count")
plt.show()

sns.histplot(data=fantasy_ratings, label="Fantasy")
plt.title("Fantasy Movie Ratings")
plt.xlabel("Ratings")
plt.ylabel("Count")
plt.show()

sns.histplot(data=film_noir_ratings, label="Film Noir")
plt.title("Film Noir Movie Ratings")
plt.xlabel("Ratings")
plt.ylabel("Count")
plt.show()
### END PLOTTING ###
```


Children Movie Ratings
Count
0
500
1000
1500
2000
1
2
3
4
5
Ratings


Fantasy Movie Ratings
Count
0
100
200
300
400
1
2
3
4
5
Ratings


Film Noir Movie Ratings
Count
0
100
200
300
400
500
600
1
2
3
4
5
Ratings

Answer: All distributions have a center/average roughly between 3 and 4. The distributions for the Children and Fantasy genres are fairly normal. However, the Film Noir genre distribution has a higher average and is skewed to the left, which suggests that users who like these movies tend to rate them higher. Perhaps this means we should normalize the ratings across genres based on their individual distributions -- in other words, making a 4 or 5 rating for a Film Noir movie "weigh less" than the same rating for a Fantasy or Children movie.

Question 3: The User-Interaction Matrix

3.1: How many unique users and unique movies are there in our dataset? Assign your answers to `num_users` and `num_movies` , respectively.

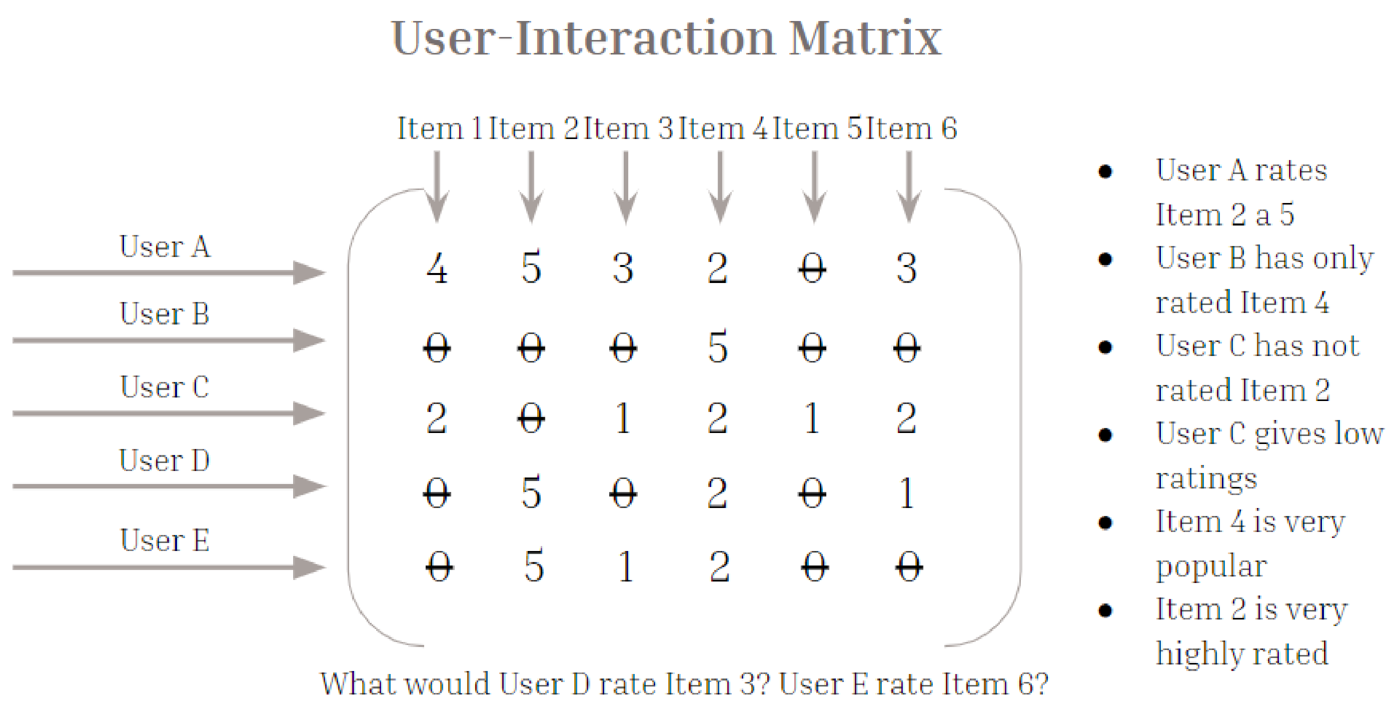
Hint: What data structure did you see this week in CS 61B that could be helpful here?

In [6]:
```
# Answer to Hint: In Week 6 of CS 61B, Students will be introduced to the idea of Sets. Sets have a special property
# where they do not contain duplicates by construction. This means sets can implicitly remove duplicate entries
# and return just the unique users and movies in our DataFrame.

### BEGIN CODE ###
unique_users = set(all_data["user_id"])
unique_movies = set(all_data["movie_id"])
num_users = len(unique_users)
num_movies = len(unique_movies)
### END CODE ###

print("Number of ratings:", len(all_data))
print("Number of unique users:", num_users)
print("Number of unique movies:", num_movies)
```

```
Number of ratings: 100000
Number of unique users: 943
Number of unique movies: 1682
```

3.2: You should see that the values for the number of unique users and unique movies are much smaller than the dimensions of our raw data matrix. What does that tell us about how many movies each user rated? What would you expect to be the most common number in our raw data matrix?

Answer: We have far fewer unique users and unique movies than the number of ratings in the total dataset. This tells us two things: one, that each user rates many different movies, and two, our matrix is actually very "sparse." Further, since this is a sparse matrix, that means that many of the values are 0.



3.3: Recall the structure of the User-Item Interaction Matrix taken from this week's slides. For `all_data` , construct the corresponding User-Item Interaction Matrix using Pandas. Call it `interaction_matrix.` Print out its dimensions and the first few rows to confirm that the dimensions match with the number of unique users and movies you found in **3.1**.

Hint: Look into the Pandas function `df.pivot` (https://pandas.pydata.org/pandas-docs/stable/reference/api/pandas.DataFrame.pivot.html (https://pandas.pydata.org/pandas-docs/stable/reference/api/pandas.DataFrame.pivot.html))

```
In [7]: ### BEGIN CODE ###
        truncated_data = ratings[["user_id", "movie_id", "rating"]]
        interaction_matrix = truncated_data.pivot(index="user_id", columns="movie_id",
        values="rating").fillna(0);
        ### END CODE ###

        print("Interaction Matrix Dimensions: ", interaction_matrix.shape);
        display(interaction_matrix.head());
```

```
Interaction Matrix Dimensions:  (943, 1682)
```

movie_id	1	2	3	4	5	6	7	8	9	10	...	1673	1674	1675	1676	1677	1678
user_id																	
1	5.0	3.0	4.0	3.0	3.0	5.0	4.0	1.0	5.0	3.0	...	0.0	0.0	0.0	0.0	0.0	0.0
2	4.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	2.0	...	0.0	0.0	0.0	0.0	0.0	0.0
3	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	...	0.0	0.0	0.0	0.0	0.0	0.0
4	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	...	0.0	0.0	0.0	0.0	0.0	0.0
5	4.0	3.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	...	0.0	0.0	0.0	0.0	0.0	0.0

5 rows × 1682 columns

You should see that most of the values in the matrix are 0. This reflects the fact that the original `all_data` matrix was sparse.

Question 4: Memory-Based Approaches and Clustering

Question 4a: Cosine Similarity

To introduce the idea of comparing users and movies, we will look into a commonly-used similarity measure known as "cosine similarity". It boils down to determining the cosine of the angle between every pair of feature vectors, which can be expressed using the following equation:

For vectors i and j of length n,

$$cos(\theta) = \frac{v_i^T v_j}{|v_i||v_j|} = \sum_{k=1}^{n} \frac{v_{i,k} v_{j,k}}{\sqrt{\sum_{k=1}^{n} v_{i,k}^2} \sqrt{\sum_{k=1}^{n} v_{j,k}^2}}$$

4.1: Fill out the following function `cosine_similarity(U, V)` to compute the cosine similarity of the vectors in a matrix:

Note: You can use `np.finfo(float).eps` to add a "fudge" factor and prevent issues with ratings of 0.

```
In [8]: def cosine_similarity(U):
            '''
            Inputs:
                - U: Data of interest represented as vectors in a matrix
            Output:
                - similarity: Matrix where a_{i,j} represents the cosine similarity be
        twee vectors v_i and v_j
            '''
            ### BEGIN CODE ###
            numerator = U @ U.T + np.finfo(float).eps
            root_norm = np.array([np.sqrt(np.diagonal(numerator))])
            similarity = numerator / (root_norm * root_norm.T)
            ### END CODE ###

            return similarity
```

4.2: Use the completed function to find the cosine similarity between users. Assign this to `user_similarity`.

Hint: Treat each row of `interaction_matrix` as a vector.

```
In [9]: ### BEGIN CODE ###
        user_similarity = cosine_similarity(interaction_matrix)
        ### END CODE ###

        user_similarity.head()
```

Out[9]:

user_id	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	
user_id									
1	1.000000	0.166931	0.047460	0.064358	0.378475	0.430239	0.440367	0.319072	0.0781
2	0.166931	1.000000	0.110591	0.178121	0.072979	0.245843	0.107328	0.103344	0.1610
3	0.047460	0.110591	1.000000	0.344151	0.021245	0.072415	0.066137	0.083060	0.0610
4	0.064358	0.178121	0.344151	1.000000	0.031804	0.068044	0.091230	0.188060	0.1012
5	0.378475	0.072979	0.021245	0.031804	1.000000	0.237286	0.373600	0.248930	0.0568

5 rows × 943 columns

4.3: What does each value in the matrix represent? Why is the diagonal 1_m, i.e., why are there 1's along the diagonal?

Answer: Each value in the matrix represents the "similarity" between two users as measured by the cosine similarity of their ratings represented as high-dimensional, sparse vectors. We see 1's along the diagonal, which makes sense, since that diagonal computes the similarity of users with themselves.

4.4: How could we now use this matrix to recommend movies to a user? What drawbacks are there with using this approach?

Answer: We can recommend movies for a particular user in `user_similarity` by looking at the next highest values in each user's row. The highest values represent the most similar users, meaning the movies they rated highly are likely movies this particular user would also rate highly.

One drawback is that this only works for the users in our database. In other words, this `user_similarity` matrix must be recomputed each time a user rates a new movie or each time a new user uses our service. This algorithmic complexity can by shortened by only computing a new vector for the new user.

4.5: How can we change the input to `cosine_similarity` to compute the cosine similarity between movies, rather than users?

Answer: We can swap the order of matrix multiplication. In other words, if $A = interaction\_matrix$, we use A^TA instead of AA^T, and we instead run this line of code: `cosine_similarity(interaction_matrix.T)` . Now, Each value in the matrix represents the similarity between two movies as measured by the cosine similarity of their ratings represented as high-dimensional, sparse vectors.

Question 4b: K-Nearest Neighbors

When we recommend similar users based on the ratings they give, that is a type of **user-user collaborative filtering**. Alternatively, when we look for similar items based on the ratings given to them by users, that is a type of **item-item collaborative filtering**.

We've just seen that we can use similarity functions to quantify the similarity between users or items. The next step is to make a recommendation by grouping together users who are most similar to one another. In this way, we create groups, or clusters, that represent users who give similar ratings. This boils down to the algorithm known as **K-Nearest Neighbors (KNN)**, which is used to partition data into K clusters of greatest similarity. (This is spiritually similar to the K-Means Clustering you saw in lecture earlier this week -- K-Means Clustering is an unsupervised learning technique that assigns the data into K clusters, whereas KNN looks at the K data points most similar to a certain training point in order to assign it a class or label).

For the purposes of this assignment, we will look at a pre-imlemented KNN Algorithm (https://scikit-learn.org/stable/modules/generated/sklearn.neighbors.NearestNeighbors.html#sklearn.neighbors.NearestNeighbo
from scikit-learn (https://scikit-learn.org/), the machine learning library we have been working with over the last few weeks of 16ML. The KNN Algorithm we use here is a bit simpler than what KNN really is -- here, instead of assigning labels based on the K nearest neighbors, we will use this to simply identify the K nearest neighbors.

[Note: Later on in EECS 16B, you will revisit K-Means Clustering. This will be used in lab to classify voice commands to control your car.]

4.6: Read the linked documentation to understand how to use the KNN Algorithm from Scikit-Learn. What are the relevant functions and return values we can use?

Answer: We can import this Algorithm with the following line of code: `from sklearn.neighbors import NearestNeighbors` . The Algorithm is treated as a class with various methods. Next, we instantiate the Algorithm class with `nn = NearestNeighbors(n_neighbors=x)` . We can then use the `fit(X)` function to run the KNN algorithm on our `interaction_matrix` .

Another useful function is `kneighbors` , which will return a `distance` matrix representing the distance from each vector to every other vector, and an `indices` matrix containing the k nearest neighbors i.e. vectors for each given vector.

4.7: Run KNN on `interaction_matrix` from the earlier part of this question. For now, use k = 10.

In [10]:
```python
### BEGIN CODE ###
from sklearn.neighbors import NearestNeighbors
model = NearestNeighbors(n_neighbors=10)
model.fit(interaction_matrix)

distances, indices = model.kneighbors(interaction_matrix)
indices += 1
### END CODE ###

print("Shape of distances matrix:", distances.shape)
display(distances)
print("Shape of indices matrix:", indices.shape)
display(indices)
```

```
Shape of distances matrix: (943, 10)

array([[ 0.        , 55.06359959, 55.23585792, ..., 57.51521538,
        57.61076288, 57.62811814],
       [ 0.        , 25.63201124, 28.01785145, ..., 28.37252192,
        28.54820485, 28.65309756],
       [ 0.        , 20.95232684, 21.3541565 , ..., 22.09072203,
        22.20360331, 22.22611077],
       ...,
       [ 0.        , 20.71231518, 20.76053949, ..., 21.26029163,
        21.33072901, 21.3541565 ],
       [ 0.        , 37.16180835, 37.52332608, ..., 38.28837944,
        38.37968212, 38.39270764],
       [ 0.        , 43.35896678, 43.8634244 , ..., 45.22167622,
        45.2437841 , 45.40925016]])

Shape of indices matrix: (943, 10)

array([[  1, 738, 521, ..., 773, 215, 774],
       [  2, 701, 266, ..., 520, 651, 111],
       [  3, 317, 656, ..., 685,  61, 809],
       ...,
       [941, 742, 441, ...,  17, 578, 609],
       [942, 163, 856, ..., 205, 202, 656],
       [943, 933, 774, ..., 778, 175, 538]], dtype=int64)
```

4.8: Let's look at the first user in our `interaction_matrix` . What other users is our user most similar to? What are the IDs of the 5 movies these users liked most?

In [11]:
```python
### BEGIN CODE ###
first_user = indices[0]
similar_users = (interaction_matrix.loc[first_user])

print("IDs of top movies:", [i[0] for i in sorted(enumerate(np.sum(similar_users)), key = lambda x: x[1], reverse=True)][:5])
### END CODE ###

print("First user is similar to:", first_user)
display(similar_users)
```

```
IDs of top movies: [49, 175, 227, 173, 182]
First user is similar to: [  1 738 521 933 715  44 868 773 215 774]
```

movie_id	1	2	3	4	5	6	7	8	9	10	...	1673	1674	1675	1676	1677	1678
user_id																	
1	5.0	3.0	4.0	3.0	3.0	5.0	4.0	1.0	5.0	3.0	...	0.0	0.0	0.0	0.0	0.0	0.0
738	5.0	3.0	0.0	4.0	0.0	0.0	4.0	0.0	0.0	0.0	...	0.0	0.0	0.0	0.0	0.0	0.0
521	2.0	3.0	0.0	0.0	0.0	0.0	3.0	3.0	0.0	0.0	...	0.0	0.0	0.0	0.0	0.0	0.0
933	3.0	0.0	0.0	3.0	0.0	0.0	4.0	0.0	3.0	0.0	...	0.0	0.0	0.0	0.0	0.0	0.0
715	5.0	3.0	0.0	4.0	0.0	0.0	3.0	0.0	0.0	0.0	...	0.0	0.0	0.0	0.0	0.0	0.0
44	4.0	0.0	0.0	0.0	4.0	0.0	5.0	0.0	5.0	0.0	...	0.0	0.0	0.0	0.0	0.0	0.0
868	4.0	2.0	0.0	0.0	0.0	0.0	5.0	0.0	0.0	0.0	...	0.0	0.0	0.0	0.0	0.0	0.0
773	3.0	3.0	0.0	0.0	0.0	3.0	2.0	0.0	0.0	0.0	...	0.0	0.0	0.0	0.0	0.0	0.0
215	0.0	0.0	0.0	0.0	0.0	0.0	0.0	2.0	0.0	0.0	...	0.0	0.0	0.0	0.0	0.0	0.0
774	0.0	1.0	0.0	2.0	0.0	0.0	2.0	1.0	0.0	0.0	...	0.0	0.0	0.0	0.0	0.0	0.0

10 rows × 1682 columns

As seen above, this approach with similarity metrics and K-Nearest Neighbors allows us to identify the users any given user is most similar to, and then look at those users' ratings to determine what that given user should watch next.

Question 5: Model-Based Approaches -- Matrix Factorization and Embeddings

Now, we will investigate another paradigm of collaborative filtering: Model-Based Approaches. These methods instead leverage matrix factorization to break apart matrices into other matrices or vectors that have special meanings or interpretations.

5.1: What Matrix Decompositions you have seen in EECS 16A and EECS 16B? What are the tradeoffs or different use cases for one over the other?

Answer: We have seen Diagonalization, aka Eigenvalue Decomposition, in EECS 16A, and have heard of the SVD from earlier weeks of 16ML / references in EECS 16B. Both factorization methods break a matrix apart into three matrices: either $P\Lambda P^T$ for Diagonalization, or $U\Sigma V^T$ the SVD. We should use the SVD because it generalizes Diagonalization to the non-square matrix we are working with. Further, SVD has a natural connection to dimensionality reduction and clustering, making it a great tool for factoring our matrix and understanding its components.

Here is a quick refresher on the SVD:

For a matrix $A \in \mathbb{R}^{mxn}$, the "Full SVD" is the following matrix product:

$$A = U\Sigma V^T$$

where

$$U \in \mathbb{R}$$
$$\Sigma \in \mathbb{R}$$
$$V^T \in \mathbb{R}$$

Alternatively, there is a "Compact SVD" that involves truncating these matrices to remove zero-value singular vectors corresponding to the Nullspace of A:

$$A = U_c \Sigma_c V_c^T$$

where

$$U_c \in \mathbb{R}$$
$$\Sigma_c \in \mathbb{R}$$
$$V_c^T \in \mathbb{R}$$

As popularized during the 2006 Netflix Prize, there is a famed "SVD" Matrix Factorization Algorithm used for creating a recommender system. **NOTE: THIS IS A DIFFERENT SVD!** Even though they are both called "SVD", and are spiritually related in the sense that they are both related to matrix decomposition / factorization, they are in fact, different algorithms entirely. They are related in the sense that, given full information of A, the outputs of this process should converge to the results we would obtain from SVD.

Our goal will be to factorize `interaction_matrix` and **approximate** it as the product $A = UV$ (You may think of the sigma matrix Σ as being "absorbed" into either the U or V matrix). For a matrix $A \in \mathbb{R}^{mxn}$, it will be equal to the product $U \in \mathbb{R}^{mxd}$ and $V \in \mathbb{R}^{dxn}$.

In the context of movie recommendations, U is referred to as the **User Embeddings**, while V is referred to as the **Movie Embeddings**. Each row of U, represented by U_i, denotes the "essence" of user i, while each column of V, represented by V_j, denotes the "essence" of movie j.

5.2: How can we approximate the matrix A using a sum instead of a matrix product?

Answer: We already have `interaction_matrix` A, where the rows represent each user's ratings for the movies in the dataset, while each column represents all of the ratings a single movie received. Each element can therefore be approximated as the vector product of corresponding embeddings:

$$a_{i,j} = \vec{u_i}^T \vec{v_j}$$

5.3: Why can't we use the actual SVD to compute U and V?

Answer: The SVD relies on having full information of A. Unfortunately, we have a very sparse matrix with missing ratings values.

5.4: Calculate A^*, the approximation for `interaction_matrix`, using gradient descent to learn randomly-initialized embedded vectors for U and V. Assume a latent space of dimension $d = 2$, i.e., that our embedded vectors are of length 2.

In [12]:
```
### BEGIN CODE ###
U = np.random.normal(size=[interaction_matrix.shape[0], 2])
V = np.random.normal(size=[interaction_matrix.shape[1], 2])
### END CODE ###
```

In [13]:
```
gamma = 0.04

### BEGIN CODE ###
for i in range(100):
    error_v = [0 for i in range(len(V))]
    for u in range(len(U)):
        error_u = 0
        for v in range(len(V)):
            if (interaction_matrix.iloc[u,v] != 0):
                pred = np.dot(U[u], V[v])
                actual = interaction_matrix.iloc[u,v]
                error_u += (pred - actual) * (-V[v])
                error_v[v] += (pred - actual) * (-U[u])
        U[u] = U[u] + gamma * error_u

    for v in range(len(V)):
        V[v] = V[v] + gamma * error_v[v]
### END CODE ###
```

In [14]:
```
A_star = np.dot(U, V.T)
```

You should see that `A_star - interaction_matrx` returns a matrix where all the values are fairly close to 0. This symbolizes the fact that we have attained appropriate embedding vectors, and thus appropriate user/item embeddings U and V, that best approximate our interaction matrix.

To go further with these embeddings, we can return to the formula in **5.2** and compute the expected rating for a given user for a given movie, by computing $a_{i,j} = \overrightarrow{u_i}^T \overrightarrow{v_j}$.

Summary and Extensions

Congratulations! We've reached the end of the assignment. We hope that this was an eye-opening experience into a very applied topic of interest in machine learning. To summarize what we've covered so far:

- **History of Recommender Systems and the Netflix Prize**
- **Loading Datasets**
- **Exploratory Data Analysis**
- **User-Interaction Matrices**
- **Memory-Based Approaches** - We looked at how we could use similarity measurements such as cosine similarity to determine the similarity between users/items in our user-interaction matrix. Using such measurements allows us to use algorithms such as K-Nearest Neighbors to identify the users or items most similar to a given user or item, respectively.
  - Cosine Similarity
  - KNN
- **Model-Based Approaches** - We looked at another paradigm in collaborative filtering that leverages matrix decompositions. The "SVD" Matrix Factorization Algorithm (which isn't actually the SVD!) was used to decompose our interaction matrix into user and item embeddings. We then generated random embeddings and learned correct values for them by using gradient descent.
  - User and Movie Embeddings
  - Gradient Descent

If you are interested in further investigating collaborative filtering algorithms and building recommender systems, here are some relevant links to sophisticated packages used in production settings, cutting-edge improvements, and current, unsolved issues:

Surpriselib

Surprise is a Package for SciPy. It is a dedicated SciPy package for building and analyzing Recommender Systems. In particular, it has native access to various datasets such as MovieLens, and also has a wide array of prediction algorithms.

- SurpriseLib Website (http://surpriselib.com/)
- GitHub Link (https://github.com/NicolasHug/Surprise)

Here is a demo for using Surprise to perform prediction using cosine similarity and KNN on a small example training set:

In [ ]:
```
!pip install scikit-surprise
```

```
In [ ]: from surprise import Dataset
from surprise import Reader
from surprise import KNNWithMeans

'''
Surprise_Ratings contains mappings of (item, user, rating) pairs
    - item: Name/ID for item
    - user: Name of user
    - rating: Rating user assigns to corresponding item
'''

surprise_ratings = {
    "item": ["A", "B",
             "A", "B",
             "A", "B",
             "A", "B",
             "A"],
    "user": ["Allen", "Allen",
             "Bill", "Bill",
             "Cathy", "Cathy",
             "Devin", "Devin",
             "Evan"],
    "rating": [1, 2,
               2, 4,
               2.5, 4,
               4.5, 5,
               3],
}

reader = Reader(rating_scale=(1, 5))
df = pd.DataFrame(surprise_ratings)

# Load elements from surprise_ratings into form that Surpriselib can use
surprise_data = Dataset.load_from_df(df[["user", "item", "rating"]], reader)

sim_options = {
    "name": "cosine", # Use item-based cosine similarity
    "user_based": False, # Flag to determine whether to do user-user or item-i
tem similarity -- here we do item-item similarity
}

# Generate KNN Algorithm and train on data to create predictor
surprise_algorithm = KNNWithMeans(sim_options=sim_options)
surprise_training_data = surprise_data.build_full_trainset()
surprise_algorithm.fit(surprise_training_data)

# Predict Evan's rating for item "B"
surprise_prediction = algo.predict("Evan", "B")
print(surprise_prediction.est)
```

Alternative and Advanced Methods

Deep Learning and Deep Neural Networks

Later on in 16ML, we will introduce the idea of Deep Learning, which relies heavily on using Neural Networks with many, many layers to perform complex computations. These are considered state-of-the-art models that are at the forefront of recent advances in many different fields of Machine Learning, and are a valid option for developing a recommender system.

Regularization

When performing Model-Based Collaborative Filtering, we can run into the issue of underregularization -- in other words, the waya we perform such similarity measurements and clustering techniques make it so that we are very likely to overfit. This is a concept that was discussed earlier in 16ML during Week 4. In essence, regularization allows us to "smooth" out our data by "lifting" up small, near-zero values in our data that can cause numerical instability and/or unexpected magnification of small values. Regularization can help with Collaborative Filtering by dealing with movies that were not rated by many users, or users who did not rate many movies. In this way, it can help reduce error in predicting or recommending new movies to users, or conversely, identifying new users who might like a given movie.

Issues

Cold Start

A fundamental flaw with recommender systems as they currently exist is a concept called "Cold Start." Consider the following: we've been working with a massive matrix containing ratings given by various users to various movies. What happens when we add a new movie to the database? Any new movies will have no ratings from any user, meaning that our collaborative filtering algorithms have no information whatsoever on how a user will like it. That means that these movies will never be recommended with an naive algorithm. This means that we must carefully consider how we add new movies to our database. Some considerations include assigning it the same user ratings as a movie we believe to be very similar, or perhaps using some metric such as median or average to get a representative value of the new movie's genre. We could also apply content-based filtering when first looking a a new movie by using certain factors, such as genre or length, to establish initial ratings.

As we saw earlier, the matrices we are working with are very sparse, which will also inherently contribute to the Cold Start problem.

References

- The Netflix Prize. https://en.wikipedia.org/wiki/Netflix_Prize (https://en.wikipedia.org/wiki/Netflix_Prize)
- Baptiste Rocca. Introduction to recommender systems. https://towardsdatascience.com/introduction-to-recommender-systems-6c66cf15ada,2019 (https://towardsdatascience.com/introduction-to-recommender-systems-6c66cf15ada,2019).
- Simon Funk. Netflix update:Try this at home. https://sifter.org/simon/journal/20061211.html (https://sifter.org/simon/journal/20061211.html), 2006.
- Yehuda Koren. The bellkor solution to the netflix grand prize. Published on Netflix PrizeForums, 2009.
- Build a Recommendation Engine With Collaborative Filtering. https://realpython.com/build-recommendation-engine-collaborative-filtering/ (https://realpython.com/build-recommendation-engine-collaborative-filtering/)
- Prince Grover. Various Implementations of Collaborative Filtering https://towardsdatascience.com/various-implementations-of-collaborative-filtering-100385c6dfe0 (https://towardsdatascience.com/various-implementations-of-collaborative-filtering-100385c6dfe0)
- Intro to Recommender Systems: Collaborative Filtering https://www.ethanrosenthal.com/2015/11/02/intro-to-collaborative-filtering/ (https://www.ethanrosenthal.com/2015/11/02/intro-to-collaborative-filtering/)
- Alesha Tony. All You Need to Know About Collaborative Filtering https://www.digitalvidya.com/blog/collaborative-filtering/ (https://www.digitalvidya.com/blog/collaborative-filtering/)
- MovieLens Dataset https://grouplens.org/datasets/movielens/ (https://grouplens.org/datasets/movielens/)
- Surprise: A Python scikit for recommender systems. http://surpriselib.com/ (http://surpriselib.com/)
- Movies Recommender System with Surpriselib. https://medium.com/analytics-vidhya/movies-recommender-system-with-surpriselib-33580ae9b47c (https://medium.com/analytics-vidhya/movies-recommender-system-with-surpriselib-33580ae9b47c)