Movie Recommendation System

Saumya Bhatnagar

Master of Science in Computer Science

Bridgewater State University

May 9, 2019

Acknowledgement

I would like to thank the faculty and students of Computer Science department to provide me the opportunity to work on this project. This project has helped me explore the field of machine learning and predictive analysis, and proved a great learning experience which would help me as I explore the field of data science in future.

I would like to thank Dr. Yiheng Liang for his guidance throughout the project. Professor Liang has always been patient and accommodating while helping me with issues during all the phases of the project. Thank you for your support.

I would also like to thank Dr. Rebecca Metcalf, for being a constant support and source of encouragement, believing in me at times when I needed it the most.

Contents

Abstract....5

Toolkit....6

Dataset....6

Using MovieLens data....7

Movie Recommendation System....9

Recommendation Process:....9

Information Collection Phase....9

Learning Phase....10

Prediction phase....10

Recommendation Techniques....11

Recommendation Metrics....12

MAE....12

RMSE....12

Top-N evaluators....13

i. Hit Rate....13

ii. ARHR....14

Diversity....15

Novelty....17

Recommendation Algorithms....18

Content-based Algorithm 18
Neighborhood-based Collaborative Filtering 26
i. User-based collaborative filter 27
ii. Item-based collaborative filtering 31
Matrix Factorization 33
Hybrid filtering technique 36
Evaluation 37
Comparison and results 39
Recommendations for the test user: 40
Content-based recommendation algorithm 40
User-based collaborative filtering algorithm 41
Item-based collaborative filtering algorithm 41
Matrix Factorization algorithm 42
Using SVD 42
Using SVD++ 42
Hybrid Algorithm 43
ApacheSpark using ALS 44
Future work 45
References 46

Abstract

A Recommender System or a Recommendation System is a subclass of information filtering system that seeks to predict the "rating" or "preference" a user would give to an item. These systems collect user information and choices, and use this information to improve their suggestions in future. There are various approaches that can be used to build such systems. Considerable research and projects have been done on this topic, and scientists have developed various algorithms for this purpose. This project focuses on implementing multiple algorithms and comparing them on various metrics, to determine which approach provides the best recommendation.

Keywords: Recommendation System, Content based filtering, Neighborhood based collaborative filtering, Matrix factorization, Recommendation system evaluation

Toolkit

This project uses Anaconda distribution with Python 3.7. The primary library used is the *scikit-surprise* library. Various other libraries and functionalities are imported throughout the project as and when needed, as well. The command `conda install -c conda-forge scikit-surprise` was used to install the *scikit-surprise* library in the Anaconda environment.

To successfully run the project, below libraries should also be installed:

```
pip install pandas

pip install matplotlib

pip install pillow

pip install wordcloud
```

Dataset

The *MovieLens* dataset (Harper & Konstan, 2015) is used for this project. In particular, the dataset '*ml-latest-small*', which describes 5-star rating and free-text tagging activity from the movie recommendation service, *MovieLens*. This dataset contains 100836 ratings and 3683 tag applications across 9742 movies. The data are contained in the files *links.csv*, *movies.csv*, *ratings.csv* and *tags.csv*. Apart from this, the dataset '*ml-20m*' containing twenty million ratings, are also used for selective analysis.

The Movielens dataset includes many files, of which *ratings.csv* and *movies.csv* files are used in this project. All movie ratings are contained in the file *ratings.csv*. Ratings are made on a 5-star scale, with half-star increments (0.5 stars - 5.0 stars). Each line of this file, after the header row, represents one rating of one movie by one user, and has the following format: *userId,movieId,rating,timestamp*. Movie information is contained in the file *movies.csv*. Each line of this file, after the header row, represents one movie, and has the following format:

movieId,title,genres. Movie titles include the year of release in parentheses. Genres are in a pipe-separated list, and are selected from the following:

1. Action
2. Adventure
3. Animation
4. Children's
5. Comedy
6. Crime
7. Documentary
8. Drama
9. Fantasy
10. Film-Noir
11. Horror
12. Musical
13. Mystery
14. Romance
15. Sci-Fi
16. Thriller
17. War
18. Western

Using MovieLens data

Often half the battle of machine learning is data cleaning and processing. The `MovieLens.py` module is used to load the raw files, that contain ratings and information about the movies. This is done by converting the raw files into datasets that can be used by *surprise-lib*. The class diagram for `MovieLens` class is:

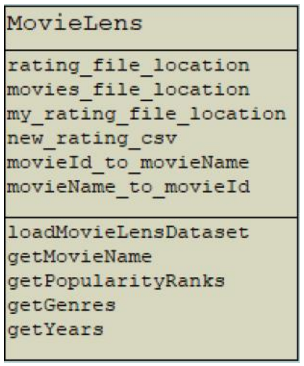
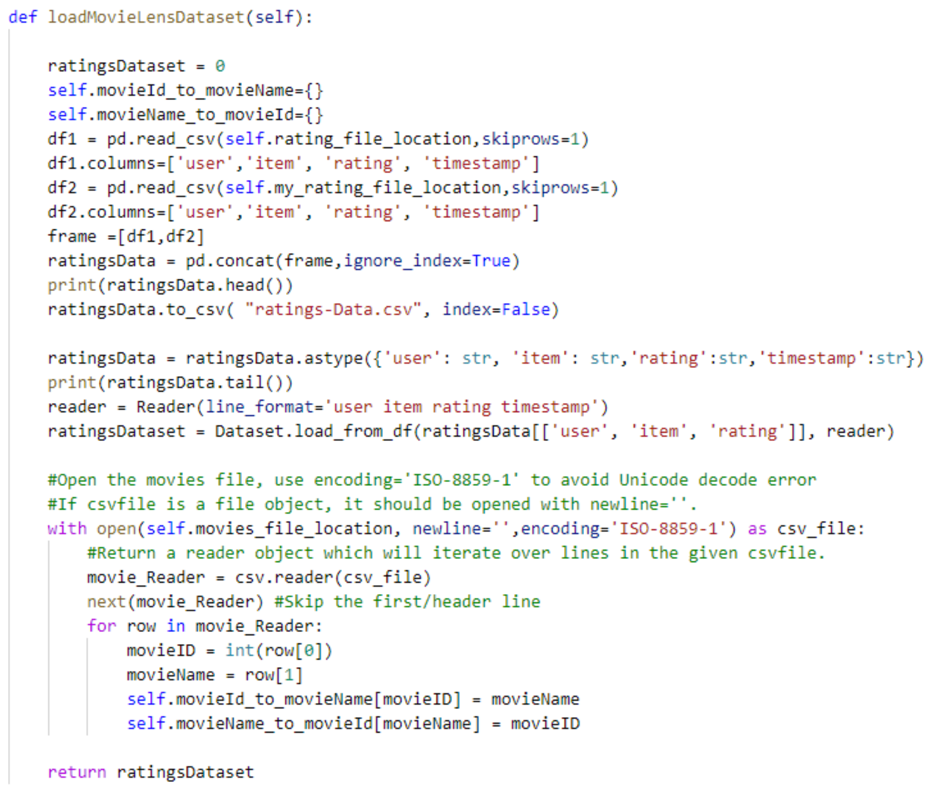


Figure 1: Class diagram for `Movielens`

Below is a snapshot of the data load functionality:



```
def loadMovieLensDataset(self):

    ratingsDataset = 0
    self.movieId_to_movieName={}
    self.movieName_to_movieId={}
    df1 = pd.read_csv(self.rating_file_location,skiprows=1)
    df1.columns=['user','item', 'rating', 'timestamp']
    df2 = pd.read_csv(self.my_rating_file_location,skiprows=1)
    df2.columns=['user','item', 'rating', 'timestamp']
    frame =[df1,df2]
    ratingsData = pd.concat(frame,ignore_index=True)
    print(ratingsData.head())
    ratingsData.to_csv( "ratings-Data.csv", index=False)

    ratingsData = ratingsData.astype({'user': str, 'item': str,'rating':str,'timestamp':str})
    print(ratingsData.tail())
    reader = Reader(line_format='user item rating timestamp')
    ratingsDataset = Dataset.load_from_df(ratingsData[['user', 'item', 'rating']], reader)

    #Open the movies file, use encoding='ISO-8859-1' to avoid Unicode decode error
    #If csvfile is a file object, it should be opened with newline=''.
    with open(self.movies_file_location, newline='',encoding='ISO-8859-1') as csv_file:
        #Return a reader object which will iterate over lines in the given csvfile.
        movie_Reader = csv.reader(csv_file)
        next(movie_Reader) #Skip the first/header line
        for row in movie_Reader:
            movieID = int(row[0])
            movieName = row[1]
            self.movieId_to_movieName[movieID] = movieName
            self.movieName_to_movieId[movieName] = movieID

    return ratingsDataset
```

Figure 2: Snapshot of `LoadMovieLensDataset`

For this task, the program reads the data from csv files as dataframes using *pandas* library. To be able to access the quality of recommendation, rating from a test user, whose movie choices are similar to mine, is added as a dataframe. Next, the three dataframes are merged to create a single dataframe to be used by the program. This final dataframe is then converted to a dataset format the *surprise-lib* can work with. The dataset contains ratings a user has given to a movie, where the movies are represented by unique ids, instead of movie names. The coding above shows that movie names have been extracted separately from the *movies.csv* file. The mappings for movie id to movie name and vice versa are stored in dictionaries which would be used for displaying the recommended movie list.

Movie Recommendation System

Recommendation Systems include software tools and techniques that finds items to be recommended and displays them to a user. These suggestions relate to various decision-making processes. In this project, the suggestion is a recommendation for which movie to watch next. Personalized recommendations are offered as ranked lists of movies. To perform this ranking, the recommendation system predicts what the most suitable movies are, based on the user's preference.

Recommendation Process:

The recommendation process can be divided into 3 phases: Information Collection Phase, Learning Phase, and Recommendation phase.

Information Collection Phase

Collecting relevant information about the user to generate a user profile is the first step. Relevant information includes the user's behavior and the content of the items the user accesses. Recommender systems rely on different types of input such as, explicit feedback, implicit feedback, or hybrid feedback.

Explicit feedback

This feedback is received from the user. It is in the form of ratings for the items the user has consumed. The efficiency of the system relies on the quality of rating provided by the user. This requires efforts from the user, and users are not always ready to provide these ratings. As a result, this is the shortcoming of the method.

Implicit feedback

For implicit feedback, the system automatically infers the user preference by monitoring their activities, such as history of purchase, time spent on web-pages, or types of videos watched on YouTube. Although this method does not require effort from the user, it is less reliable than explicit feedback. Because such information can be susceptible to click baits, it may be erroneous.

Hybrid feedback

This feedback combines the implicit and explicit feedbacks, by taking the user behavior implicitly, while allowing users the option of providing explicit feedback voluntarily.

Learning Phase

The learning phase uses various algorithms to filter, while using the features of information gathered during the first phase.

Prediction phase

The prediction phase predicts and recommends the type of items the user may prefer.

Recommendation Techniques

The use of efficient and accurate recommendation techniques is important. There are various techniques that can be used, as explained in the figure below:

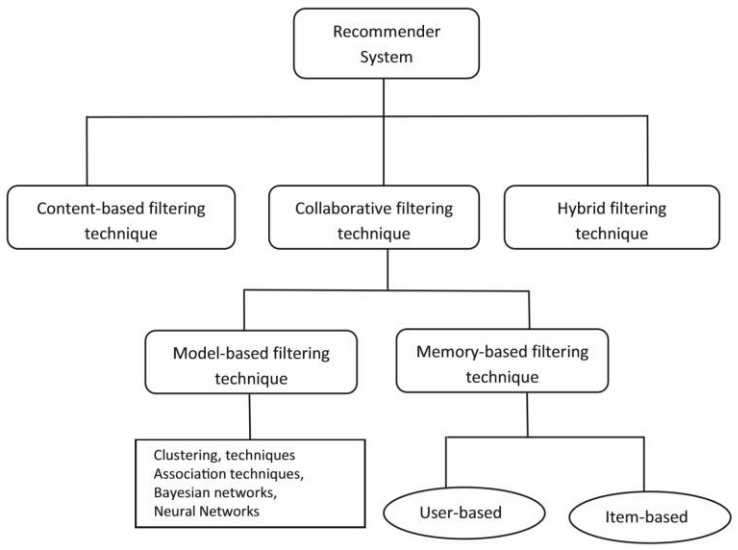

Recommender System
Content-based filtering technique
Collaborative filtering technique
Hybrid filtering technique
Model-based filtering technique
Memory-based filtering technique
Clustering, techniques
Association techniques,
Bayesian networks,
Neural Networks
User-based
Item-based

Figure 3: F.O. Isinkaye et al., Recommendation systems: Principles, methods and evaluation

The recommenders used in movie recommendation system are usually *top-N* recommenders. They produce a finite list (*N*) of best possible movies to recommend to the user. The success of a recommender depends on its ability to find the best of the top recommendations for the user.

Recommendation Metrics

This project uses metrics to evaluate the algorithms that are being implemented. These metrics are explained below. They are implemented in `RecommendationMetrics.py`.

MAE

The most straight-forward metric is mean absolute error, or MAE. Suppose we have *n* ratings in the test set to evaluate. For each rating the predicted value is *y*, and the actual rating the user provided is *x*. The error for that rating prediction is the difference between these two ratings. The sum of these errors across all n-ratings in our test set is divided by *n* to find the average, or mean absolute error. By taking the absolute value of the difference, the MAE metric results in a positive number. The formula to calculate MAE is:

$$MAE = \frac{\sum_{i=1}^{n} |y_i - x_i|}{n}$$

RMSE

The root mean square error metric is more popular than MAE. This is because the RMSE metric penalizes more when the predicted rating is significantly different than the actual rating, and penalize less when the ratings are relatively similar. The difference between the calculations for these metrics is that, instead of summing the absolute values of each rating prediction error, the RMSE metric sums the squares of the rating prediction errors. Taking the square ensures the result is a positive number. This inflates the penalty for larger errors. The final step is to take the square root, which results in a number within the range of predictions. The formula to calculate RMSE is:

$$RMSE = \sqrt{\frac{\sum_{i=1}^{n} (y_i - x_i)^2}{n}}$$

An example of calculation from MAE and RMSE metrics:

x	*y*	Absolute Error	Error square
3	5	2	4
4	1	3	9
5	4	1	1
1	1	0	0

$$MSE = \frac{2+3+1+0}{4} = 1.5$$

$$RMSE = \sqrt{\frac{4+9+1+0}{4}} = 1.87$$

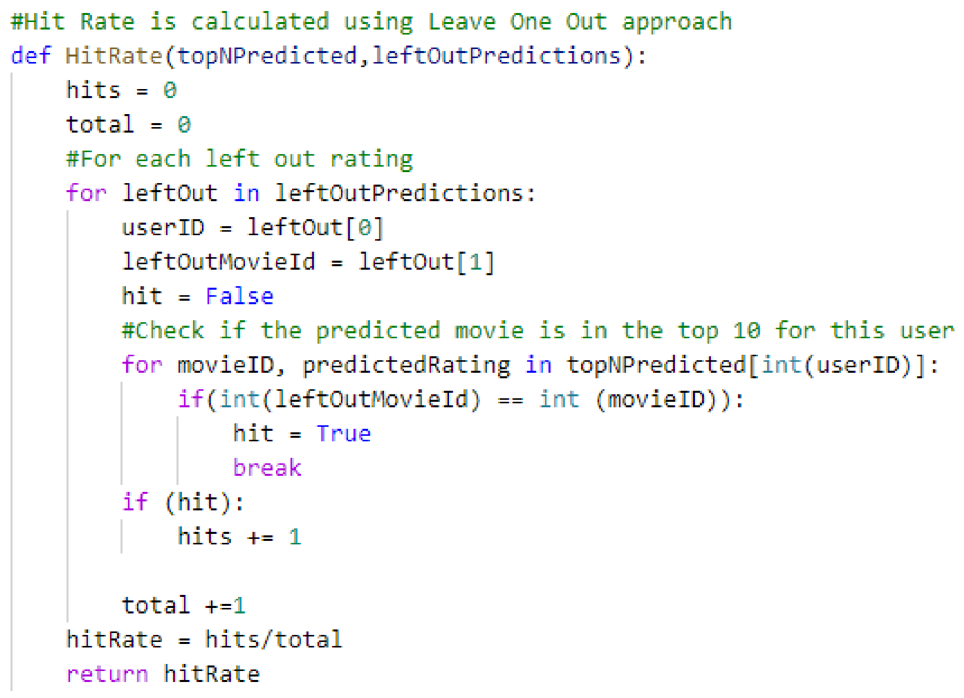
Top-N evaluators

i. Hit Rate

It is considered a “hit” if one of the recommendations is something the user has actually rated. This means, the user is shown something that he found interesting enough to watch on his own already, and thus it is considered a success. *Hit rate* is calculated by adding all the hits in the top-*N* recommendations, for every user in the test set, and dividing by the number of users. Measuring hit rate can be complicated since the same train test or cross validation approach, which was used for measuring accuracy, cannot be used because the accuracy on individual ratings is not being measured. Rather, the accuracy of top-*N* lists for individual users is being measured.

Hit rate can be measured by "Leave-one-out cross validation" approach. As the name implies, this technique computes the top-*N* recommendations for each user in the training data. This is followed by intentionally removing one of the items from that user’s training data. We then test our recommender system's ability to recommend the item that was left out in the top-*N* results it creates for that user is then assessed during the testing phase. This measures the recommender’s ability to recommend an item in a top-*N* list for each user that was left out from the training data.

The process of calculating hit-rate using "Leave-one-out cross validation" is challenging because it is difficult to get one specific movie right while testing than to find one of the recommendations. As a result, the hit-rate tends to be very small and difficult to measure unless there is a large data set available to work with. However, this combination is a much more user focused metric when the recommender system will be producing top-*N* lists in the real world, which is often the case. Below is a snapshot of the code implementing this approach:

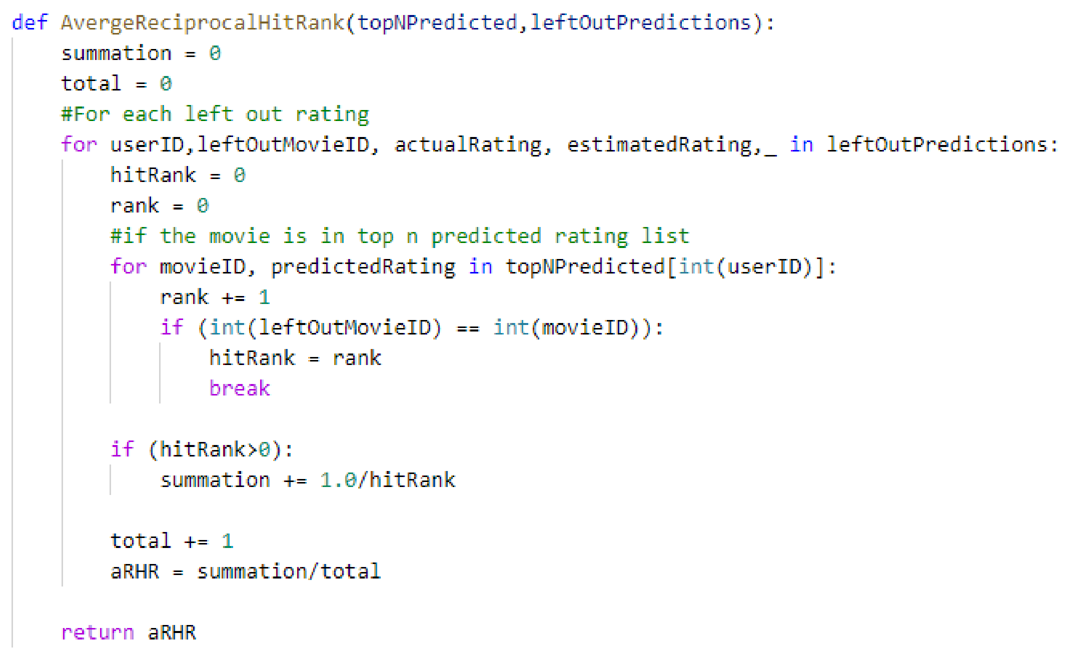


```
#Hit Rate is calculated using Leave One Out approach
def HitRate(topNPredicted,leftOutPredictions):
    hits = 0
    total = 0
    #For each left out rating
    for leftOut in leftOutPredictions:
        userID = leftOut[0]
        leftOutMovieId = leftOut[1]
        hit = False
        #Check if the predicted movie is in the top 10 for this user
        for movieID, predictedRating in topNPredicted[int(userID)]:
            if(int(leftOutMovieId) == int (movieID)):
                hit = True
                break
        if (hit):
            hits += 1

        total +=1
    hitRate = hits/total
    return hitRate
```

Figure 4: Snapshot of code calculating *hit rate*

ii. ARHR

A variation on *hit rate* is *average reciprocal hit rate* (ARHR). This metric is similar to *hit rate* but it accounts for where the hits appear in the top-*N* list. This results in more credit for successfully recommending an item in the top slot, than in the bottom slot. Again, this a more user-focused metric, since users tend to focus on the beginning of lists. The difference is that instead of summing up the number of hits, the reciprocal rank of each hit is summed. If we successfully predict our recommendation in slot three, that counts as 1/3. However, a hit in slot one of our top-*N* recommendations receives the full weight of 1.0. Determining whether or not to use this evaluator depends on how the top-*N* recommendations are displayed. If the user has to scroll to see the lower items in your top-*N* list, then it is reasonable to penalize good recommendations that appear too low in the list, where the user has to work to find them. Below is the snapshot of the code implementing this metric:



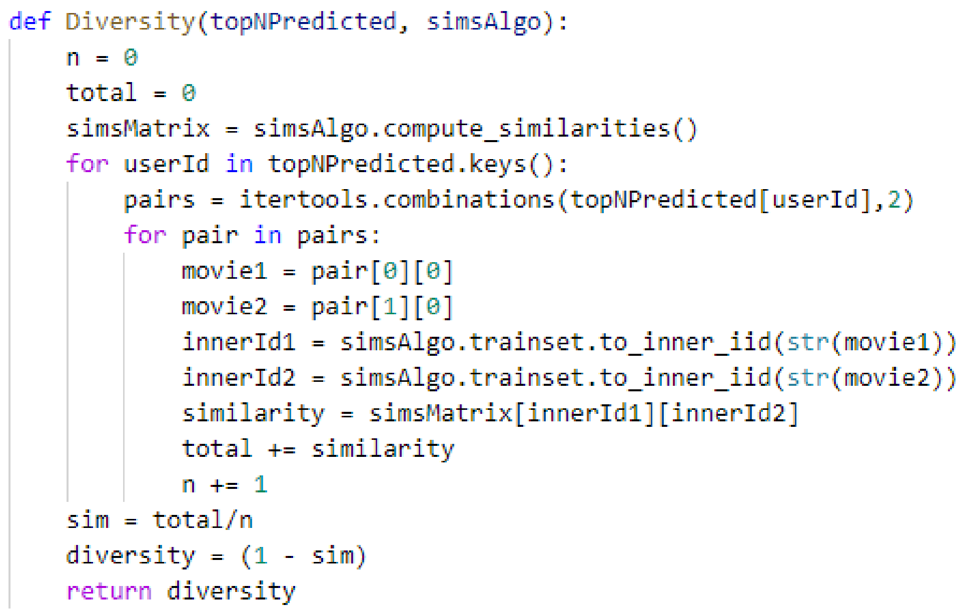
```python
def AvergeReciprocalHitRank(topNPredicted,leftOutPredictions):
    summation = 0
    total = 0
    #For each left out rating
    for userID,leftOutMovieID, actualRating, estimatedRating,_ in leftOutPredictions:
        hitRank = 0
        rank = 0
        #if the movie is in top n predicted rating list
        for movieID, predictedRating in topNPredicted[int(userID)]:
            rank += 1
            if (int(leftOutMovieID) == int(movieID)):
                hitRank = rank
                break

        if (hitRank>0):
            summation += 1.0/hitRank

        total += 1
        aRHR = summation/total

    return aRHR
```

Figure 5: Snapshot of code calculating *ARHR*

Diversity

Although accuracy is important in recommender systems, there are other aspects that can be measured that provide valuable information about the quality of recommendation. For example, a measure of how broad a variety of items the recommender system is putting in front of users can be useful. This is referred to as *Diversity*. An example of low diversity would be a recommender system that only recommends the next movies in a series that the user has started watching, but doesn't recommend other similar movies from different genres or years.

Many recommender systems start by computing similarity metric between items. These similarity scores can be used to measure diversity. To look at the similarity scores of every possible pair in a list of top-*N* recommendations, the scores can be averaged to find a measure of how similar the recommended items in the list are to each other. We can call that measure *sim*. Diversity is considered the opposite of average similarity, which means *1-sim* is the number associated with diversity. It is important to realize that diversity isn't always favorable, in the context of recommender systems. High diversity can be achieved by recommending random items. However, it is generally understood that these would not be considered good recommendations. Unusually high diversity scores are a result of bad recommendations occurring often. This suggests, diversity should be considered alongside other metrics that measure the quality of the recommendations as well. Below is the snapshot of the code implementing this metric:



```
def Diversity(topNPredicted, simsAlgo):
    n = 0
    total = 0
    simsMatrix = simsAlgo.compute_similarities()
    for userId in topNPredicted.keys():
        pairs = itertools.combinations(topNPredicted[userId],2)
        for pair in pairs:
            movie1 = pair[0][0]
            movie2 = pair[1][0]
            innerId1 = simsAlgo.trainset.to_inner_iid(str(movie1))
            innerId2 = simsAlgo.trainset.to_inner_iid(str(movie2))
            similarity = simsMatrix[innerId1][innerId2]
            total += similarity
            n += 1
    sim = total/n
    diversity = (1 - sim)
    return diversity
```

Figure 6: Snapshot of code calculating *diversity*

Novelty

Novelty is a measure of the popularity of the items are that the system is recommending. Novelty is an important factor given the purpose of recommender systems is to put forth items in the “long tail." The figure below represents a plot of movies watched, sorted by the number of ratings for a movie, which is an exponential distribution. The number of times a movie has been rated, or popularity, is on the *y*-axis, and the movie titles are along the *x*-axis. Most rated movies come from a very small portion of the dataset, but taken together, the "long tail" makes up a significant number of movies as well.

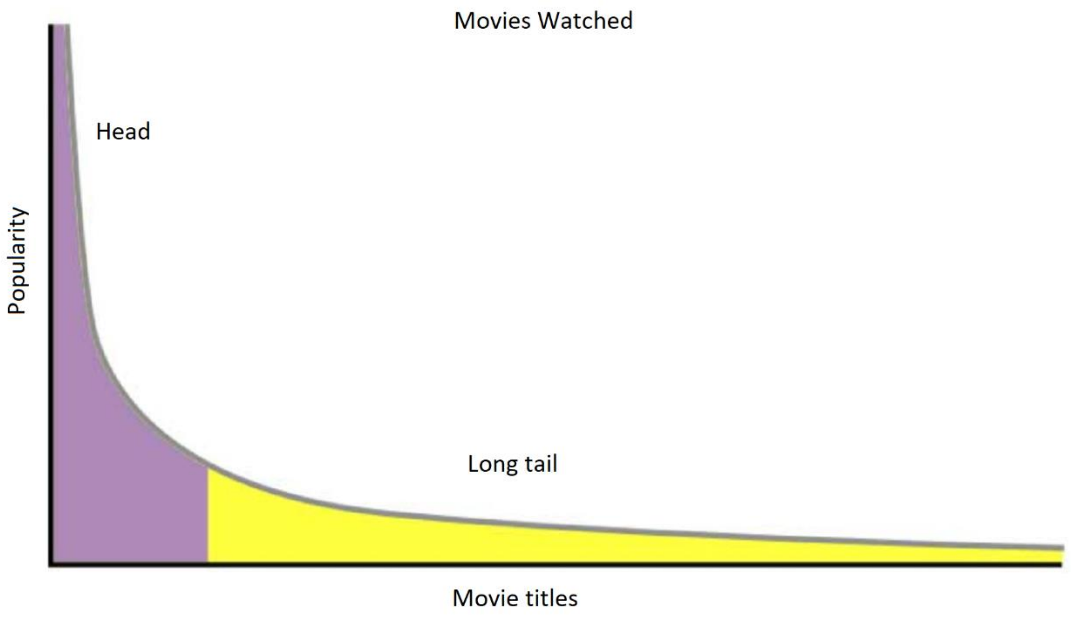
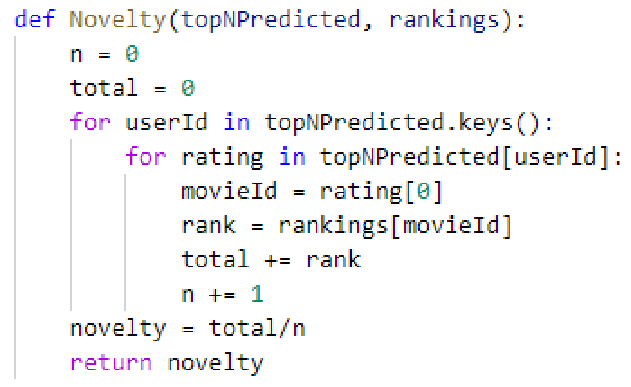


Figure 7: Long tail diagram of movies watched by users

Movies titles in the long tail, represented in yellow above, are the movies that cater to users with unique interests. Recommender systems can help users identify items in the long tail that are relevant to their own interests.

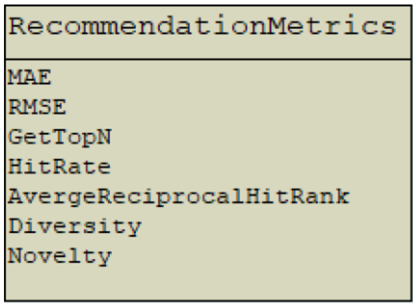
The code snapshot implementing novelty is provided below:



```
def Novelty(topNPredicted, rankings):
    n = 0
    total = 0
    for userId in topNPredicted.keys():
        for rating in topNPredicted[userId]:
            movieId = rating[0]
            rank = rankings[movieId]
            total += rank
            n += 1
    novelty = total/n
    return novelty
```

Figure 8: Snapshot of code calculating *Novelty*

The class diagram for `RecommendationMetrics` is shown below:



RecommendationMetrics
MAE RMSE GetTopN HitRate AvergeReciprocalHitRank Diversity Novelty

Figure 9: Class diagram of `RecommendationMetrics`

Recommendation Algorithms

Content-based Algorithm

The content-based algorithm approach is one of the simplest approaches for building a recommendation system. This algorithm recommends movies based on the attributes of the movies themselves, instead of using collective user behavior data.

For example, it can be effective to recommend movies in the same genre as movies known to be enjoyed by the user. The *MovieLens* dataset provides the list of genres for each movie. If it is evident a given user likes science-fiction movies, it's reasonable to recommend other science-fiction movies to that user. *MovieLens* also encodes years of release into movie titles, which can be used for recommendations as well. Therefore, instead of recommending all science fiction movies to a user who likes science-fiction, recommendations can be narrowed further to science fiction movies that were released close to the same year as the movies this user liked.

The sample raw data from *MovieLens* is shown in the table below. For each movie, a piped delimited list of the genres that apply to that movie is provided. In all, there are 18 different possible genres for every movie. This allows for a similarity measure that looks at how many genres any given pair of movies have in common. One approach for this is to use the cosine similarity metric as explained ahead.

movieId	title	genres
1	*Toy Story (1995)*	Adventure|Animation|Children|Comedy|Fantasy
2	*Jumanji (1995)*	Adventure|Children|Fantasy
3	*Grumpier Old Men (1995)*	Comedy|Romance
4	*Waiting to Exhale (1995)*	Comedy|Drama|Romance
5	*Father of the Bride Part II (1995)*	Comedy
6	*Heat (1995)*	Action|Crime|Thriller
7	*Sabrina (1995)*	Comedy|Romance
8	*Tom and Huck (1995)*	Adventure|Children
9	*Sudden Death (1995)*	Action
10	*GoldenEye (1995)*	Action|Adventure|Thriller
188301	*Ant-Man and the Wasp (2018)*	Action|Adventure|Comedy|Fantasy|Sci-Fi

To better visualize a plot of a movie in the 18-dimensional genre plane, consider a 2-dimensional genre plane which would describe every movie such as, comedy and adventure. Every movie could be plotted on a 2D graph where a value of zero is assigned if a movie doesn't belong to a genre, and a value of one if it does belong to the genre.

For example, *Toy Story* is considered both a comedy and an adventure, which results in the coordinates (1,1). According to *MovieLens*, *Grumpier Old Men* is a comedy but not an adventure, resulting in the coordinates (1,0). These positions can be represented as vectors by drawing a line from the origin of the graph to the ordered pair for each movie. A graphical representation is shown on next page:

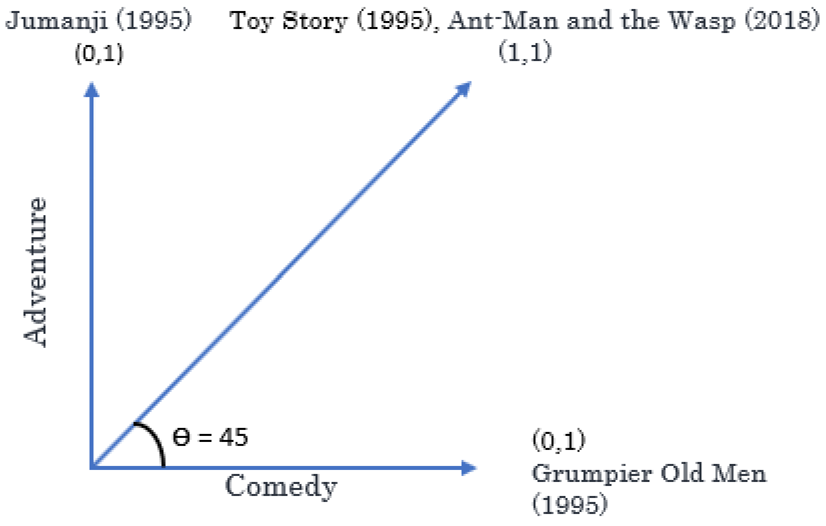


Figure 10: Calculating cosine similarity using 2D genre data

The angle between these vectors, identified as theta (Θ), determines how similar these two movies are in terms of genres. In this example, theta is equal to 45 degrees. When measuring similarity, values range between zero and one, where zero means not at all similar and one means perfectly similar. For this purpose, we take the cosine of angle theta, which is approximately 0.7. It can be said that the cosine similarity score between *Toy Story* and *Grumpy Old Men* is 0.7, based on the movies sharing a genre of comedy.

Next, consider *Ant-Man and the Wasp*, which like *Toy Story*, is both an adventure and a comedy. The angle between *Ant-Man and the Wasp* and *Grumpier Old Men* is 45 degrees, making its similarity score to *Grumpier Old Men* 0.7 (cosine of 45 degrees). Comparing *Ant-*

Man and the Wasp with *Toy Story*, in this case, the theta between these two movies is zero degrees making its similarity score 1.0 (cosine of zero degrees), indicating they are perfectly similar.

Now consider *Jumanji*, which is an adventure but not a comedy. Its coordinates are (0,1). This indicates it has nothing in common with *Grumpier Old Men*, as far as genres are concerned. In this case, the angle theta between *Jumanji* and *Grumpier Old Men* is 90 degrees, resulting in similarity score of zero (cosine of 90 degrees).

Given there are 18 genres in *MovieLens*, this model needs to be scaled to 18 dimensions. This is accomplished by plotting each movie's position in an 18-dimensional space based on its genres and determining the cosine of the angles between each movie, as a measure of their genre-based similarity to each other. This results in each movie being described as a set of 18 coordinates, each of which is zero, if a movie is not of a given genre, and one if it is. The table below depicts the 18-dimensional co-ordinates of the movies being considered, based on their genre:

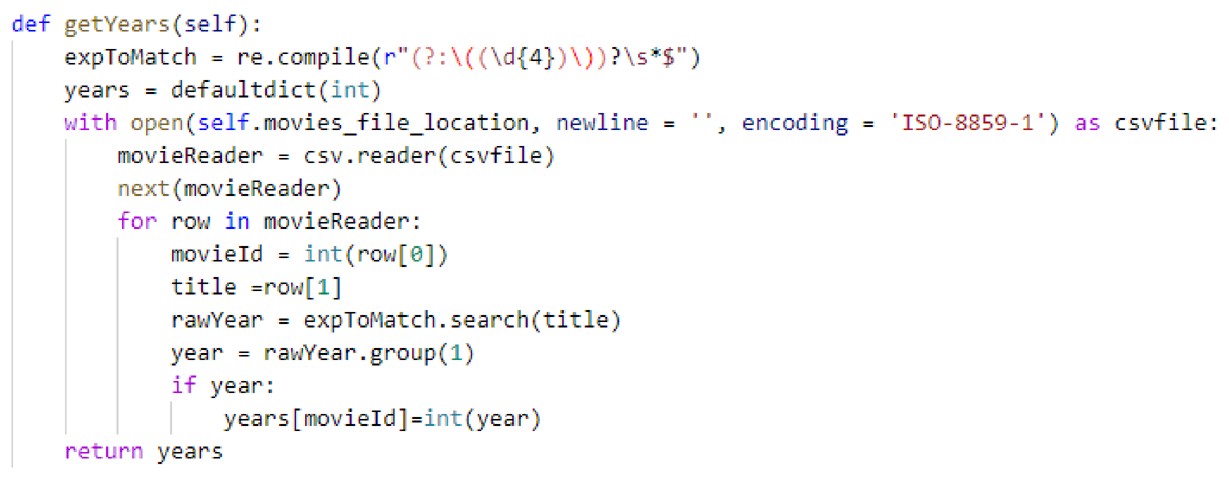
Movie	Action	Adventure	Animation	Children's	Comedy	Crime	Documentary	Drama	Fantasy	Film-Noir	Horror	Musical	Mystery	Romance	Sci-Fi	Thriller	War	Western
Toy Story (1995)	0	1	1	1	1	0	0	0	1	0	0	0	0	0	0	0	0	0
Jumanji (1995)	0	1	0	1	0	0	0	0	1	0	0	0	0	0	0	0	0	0
Grumpier Old Men (1995)	0	0	0	0	1	0	0	0	0	0	0	0	0	1	0	0	0	0
Ant-Man and the Wasp (2018)	1	1	0	0	1	0	0	0	1	0	0	0	0	0	1	0	0	0

Computing the cosine similarity score between any number of dimensions is as below can be achieved by using the mathematical equation below:

$$CosSim(x,y) = \frac{\sum_i x_i\, y_i}{\sqrt{\sum_i x_i^2}\,\sqrt{\sum_i y_i^2}}$$

Note: This equation is simplified in the code a little to avoid taking two square roots, because it would be computationally expensive to use this equation.

The release year of each movie is another attribute that can be considered. This information is included at the end of every title, in parenthesis. String wrangling in our code is used to extract this data. A snapshot of the code to extract the year from movie title is provided below:



```python
def getYears(self):
    expToMatch = re.compile(r"(?:\((\d{4})\))?\s*$")
    years = defaultdict(int)
    with open(self.movies_file_location, newline = '', encoding = 'ISO-8859-1') as csvfile:
        movieReader = csv.reader(csvfile)
        next(movieReader)
        for row in movieReader:
            movieId = int(row[0])
            title =row[1]
            rawYear = expToMatch.search(title)
            year = rawYear.group(1)
            if year:
                years[movieId]=int(year)
    return years
```

Figure 11: Snapshot of code showing string manipulation to get year from movie titles

This code returns a re.match object, which extracts the year from the title, converts it into an integer, and stores it in a dictionary with movieId as the key.

For the purpose of this project, the movies' release years that are a decade or more apart are considered not similar. After taking the absolute value of the difference in release years for two movies, the similarity will be one when the movies' release dates are from the same year, and zero when there is a difference of ten or more years in their releases. For this purpose, we need a mathematical function that smoothly scales this difference into the range [0,1]. An exponential decay function as shown below can be used to implement this:

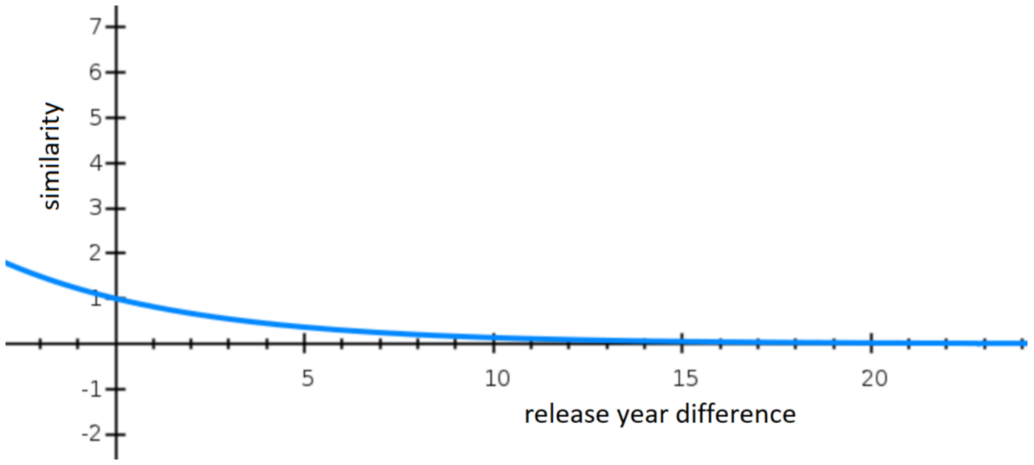
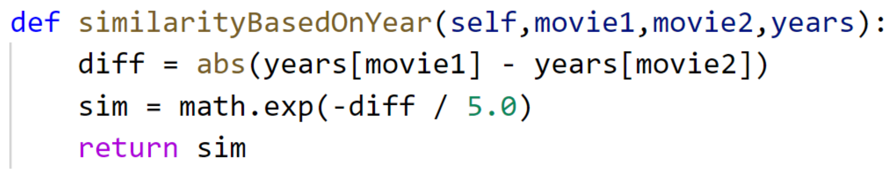


Figure 12: relationship between difference in release years and similarity between movies, generated from graphsketch.com f(x)=$e^{-x/5}$

The snapshot of code below is used to implement this:



```
def similarityBasedOnYear(self,movie1,movie2,years):
    diff = abs(years[movie1] - years[movie2])
    sim = math.exp(-diff / 5.0)
    return sim
```

Figure 13: Snapshot of code computing similarity based on difference in release years

At a year difference of zero, the similarity score on the *y*-axis is equal to one. This is the desired result. This similarity score decays exponentially, becoming small at approximately a difference of 10 years and closely approaching zero at 20.

The recommendation algorithms in this project are implemented using *surprise-lib*, which predicts a rating for a given user for a given movie. One way to provide this rating is through a technique called k-nearest-neighbors. The process is started by measuring the content-based similarity between every movie a user has rated, and the movie for which the users' rating has to be predicted. Next, a number *K* (by default 40) is selected, as the nearest-neighbors of the movie whose rating is being predicted. Consider the nearest-neighbors as the movies with the highest content-based similarity scores to the movie rating being predicted. Next, 40 movies are selected, whose genres and release dates most closely match the movie being evaluated for this user. In order to turn these top 40 closest movies into a rating prediction, a weighted average of their similarity scores to the movie whose rating being predicted can be calculated, by weighing the movie by rating the user gave. Below is the representation of the process:

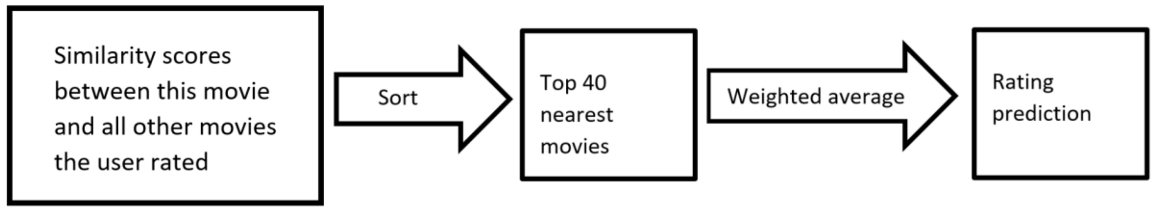

Similarity scores between this movie and all other movies the user rated
Sort
Top 40 nearest movies
Weighted average
Rating prediction

Figure 14: Process to implement content-based algorithm

The snapshot of code for implementing this algorithm is below:

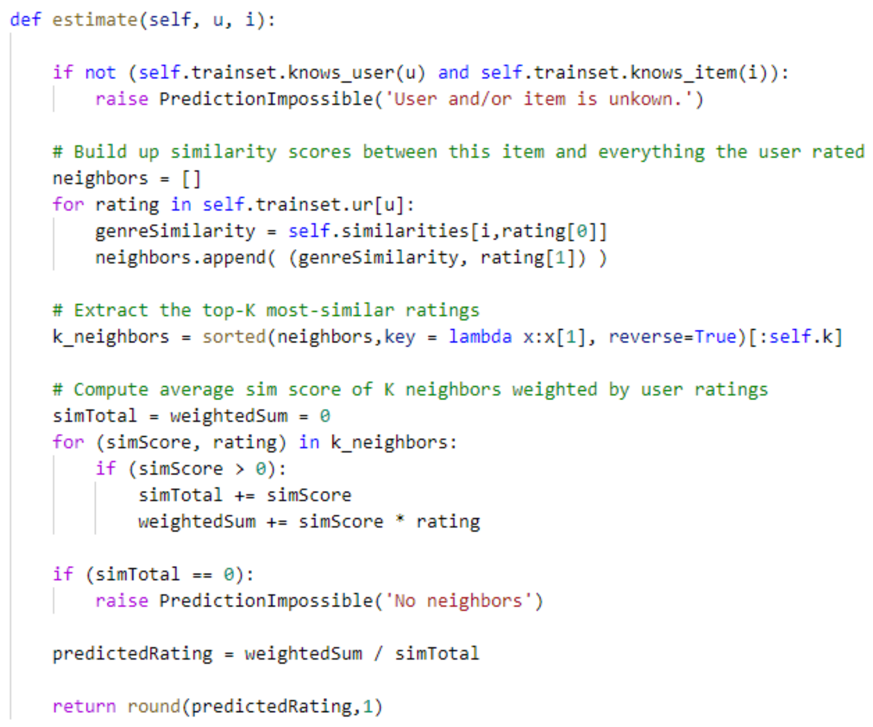

```python
def estimate(self, u, i):

    if not (self.trainset.knows_user(u) and self.trainset.knows_item(i)):
        raise PredictionImpossible('User and/or item is unkown.')

    # Build up similarity scores between this item and everything the user rated
    neighbors = []
    for rating in self.trainset.ur[u]:
        genreSimilarity = self.similarities[i,rating[0]]
        neighbors.append( (genreSimilarity, rating[1]) )

    # Extract the top-K most-similar ratings
    k_neighbors = sorted(neighbors,key = lambda x:x[1], reverse=True)[:self.k]

    # Compute average sim score of K neighbors weighted by user ratings
    simTotal = weightedSum = 0
    for (simScore, rating) in k_neighbors:
        if (simScore > 0):
            simTotal += simScore
            weightedSum += simScore * rating

    if (simTotal == 0):
        raise PredictionImpossible('No neighbors')

    predictedRating = weightedSum / simTotal

    return round(predictedRating,1)
```

Figure 15: Snapshot of the estimate function calculating prediction ratings for content-based algorithm

This part of prediction function receives as arguments user, *u*, and an item, *i*, that we want to predict a rating for. Starting with a list called neighbors, iterate over every movie the user has rated, populate the list with a content-based similarity score between each movie and the movie for which the rating has to be predicted. These similarity scores are pre-computed in the `self.similarities` array. Next, the list is sorted to provide the top-*K* movies with the highest similarity scores to the movie in question. Finally, the weighted average of the top-K similar movies, weighted by the ratings the user gave them, is computed. Assuming there is data supplied by the user, the result is the rating prediction for this user and item.

The class diagram of `ContentBasedAlgorithm` is shown below:

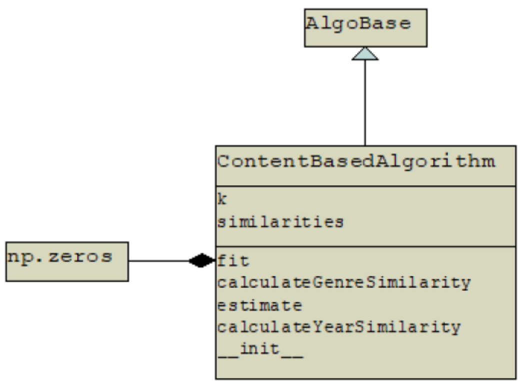


Figure 16: Class diagram of `ContentBasedAlgorithm`

Neighborhood-based Collaborative Filtering

Neighborhood based collaborative filtering leverages the behavior of others to inform what a user might enjoy. It recommends items based on based on other users' (neighborhood) collaborative behavior. The heart of this approach is in the database of item similarities. The first step in collaborative filtering is measuring the similarity between items, or the similarity between users, so like-users or like-items can be found. Although there are various ways of measuring similarity, this project uses cosine similarity. The difference between this and a content-based algorithm is, in a collaborative filtering algorithm, the dimensions are based on user behavior, instead of content attributes. For the most part, the data used in this technique is very sparse. Majority of movies haven't been watched by a given user. Conversely, majority of users haven't watched a given movie. This makes it difficult for collaborative filtering to work well, unless there is an abundance of user behavior data to work with. It isn't possible to compute a meaningful cosine similarity between two users, when they have nothing in common, nor between two items when they have no users in common.

To implement this algorithm, we are using six steps:

Step 1: Create a rating matrix (user-item or item-user)

Step 2: Create a similarity matrix (user-user or item-item)

Step 3: Identify similar users/items

Step 4: Generate possible candidates

Step 5: Generate possible scoring

Step 6: Filter the recommendations

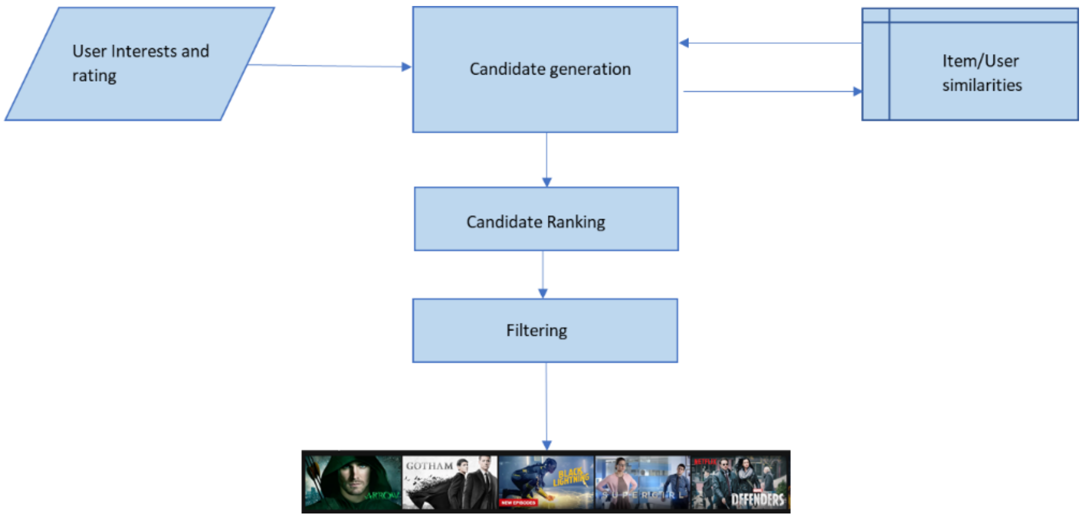


Figure 17: Implementation of collaborative filtering algorithm

i. User-based collaborative filter

User based collaborative filtering, starts by finding other users similar to the test user, based on all users' ratings history, and then make recommendations for other movies they liked which the test user hasn't seen.

Consider the example below of a limited, sparse dataset. The goal is to find recommendations for Alice:

Step 1: Create a user-item rating matrix

User/Movie	*E.T.*	*Ant-Man and the Wasp*	*Jumanji*	*Pulp fiction*	*Toy story*
Alice	5		4	4	
Bob		1			3
Charlie		3		3	5
David	3	3		5	
Elena	1		5		1

From this matrix, each user can be represented as a five-dimensional vector, as shown below:

Alice = [5,0,4,4,0]

Bob = [0,1,0,0,3]

Charlie = [0,3,0,3,5]

David = [3,3,0,5,0]

Elena = [1,0,5,0,1]

Step 2: Create a user-user similarity matrix

$$CosSim(x,y) = \frac{\sum_i x_i \, y_i}{\sqrt{\sum_i x_i^2} \, \sqrt{\sum_i y_i^2}}$$

	Alice	Bob	Charlie	David	Elena
Alice	1	0	1	0.94	0.76
Bob	0	1	0.98	1	1
Charlie	1	0.98	1	0.98	1
David	0.94	1	0.98	1	1
Elena	0.76	1	1	1	1

Step 3: Identify similar users to Alice

Users similar to Alice in order of similarity are:

i. Charlie

ii. David

iii. Elena

iv. Bob

Step 4: Generate possible candidates

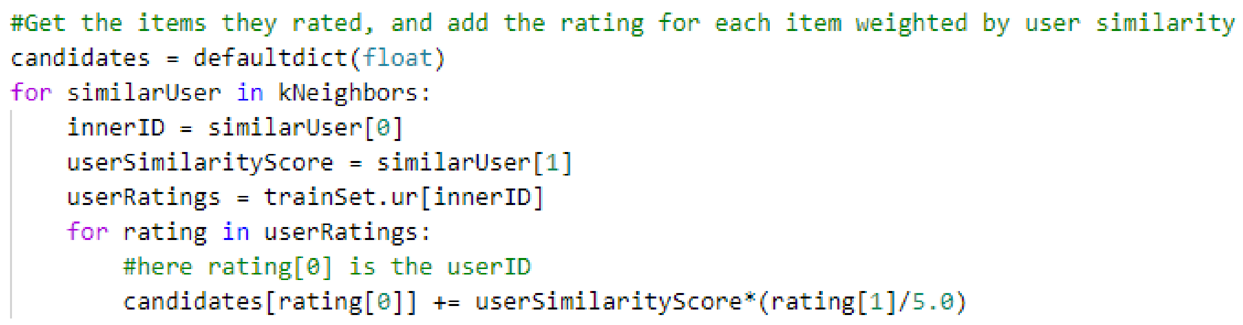
Based on a threshold (for example, minimum similarity score 80%), pick the top two similar users in this case. Our possible candidates are:

i. Charlie

ii. David

Step 5: Generate possible scoring

The goal is to recommend movies that the similar users liked. For this purpose, the rating score needs to be normalized. The snapshot from the code which normalized the rating score is on the next page:



```
#Get the items they rated, and add the rating for each item weighted by user similarity
candidates = defaultdict(float)
for similarUser in kNeighbors:
    innerID = similarUser[0]
    userSimilarityScore = similarUser[1]
    userRatings = trainSet.ur[innerID]
    for rating in userRatings:
        #here rating[0] is the userID
        candidates[rating[0]] += userSimilarityScore*(rating[1]/5.0)
```

Figure 17: Snapshot of the code implementing rating normalization for similar movies using user-based CF

Using this logic, we determine the following scores for movie suggestions:

Movie	Score
E.T.	0.56
Ant-Man and the Wasp	1.16
Pulp-fiction	1.55
Toy story	1.0

Step 6: Filter the recommendations

Begin by checking if the movies selected in the step above have already been viewed by Alice. If Alice has already provided a rating for a movie in dataset, that means Alice has already viewed the movie, and it should not be recommended. After removing the movies Alice has already viewed, the remaining movies in the list are the recommendations.

Thus, in this example, we recommend Alice to watch (recommendation in order):

1. *Ant-Man and the Wasp*
2. *Toy story*

ii. Item-based collaborative filtering

In item-based collaborative filtering, the table created in user-based collaborative filtering is reversed. Rather than looking for other users similar to our user and recommending movies they liked, the objective is to look at the movies the user liked and recommend movies that are similar to these movies. The table below defers from user-based collaborative filtering in that rows list movies and columns list users.

Step 1: Create an item-user rating matrix

	Alice	Bob	Charlie	David	Elena
E.T.	5			3	1
Ant-Man and the Wasp		1	3	3	
Jumanji	4				5
Pulp fiction	4		3	5	
Toy story		3	5		1

Step 2: Create an item-item similarity matrix

$$CosSim(x,y) = \frac{\sum_i x_i\, y_i}{\sqrt{\sum_i x_i^2}\ \sqrt{\sum_i y_i^2}}$$

	E.T.	*Ant-Man and the Wasp*	*Jumanji*	*Pulp fiction*	*Toy story*
E.T.	1	1	0.76	0.94	1
Ant-Man and the Wasp	1	1	0	0.98	0.96
Jumanji	0.76	0	1	1	1
Pulp fiction	0.94	0.980	1	1	1
Toy story	1	0.96	1	1	1

Step 3: Identify movies similar to movies Alice likes

Alice likes E.T. the most, so based on this preference, we identify the movies similar to *E.T.* as per our item-item similarity matrix. They are:

i. Ant-Man and the Wasp

ii. *Toy story*

iii. *Pulp fiction*

iv. *Jumanji*

Step 4: Generate possible candidates

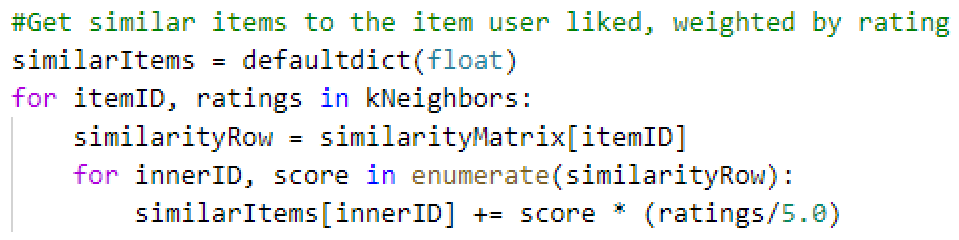
Using a threshold of 80%, below are the similar movies:

i. *Ant-Man and the Wasp*

ii. *Toy story*

iii. *Pulp fiction*

Step 5: Generate possible scoring



```
#Get similar items to the item user liked, weighted by rating
similarItems = defaultdict(float)
for itemID, ratings in kNeighbors:
    similarityRow = similarityMatrix[itemID]
    for innerID, score in enumerate(similarityRow):
        similarItems[innerID] += score * (ratings/5.0)
```

Figure 18: Snapshot of the code implementing rating normalization for similar movies using item-based CF

Using the above logic, we determine the following scores for movie suggestions:

Movie	Score
Ant-Man and the Wasp	1
Toy story	1
Pulp fiction	0.94

Step 6: Filter the recommendations

Begin by checking if the movies selected in the step above have already been viewed by Alice. If Alice has already provided a rating for a movie in dataset, that means Alice has already viewed the movie, and it should not be recommended. After removing the movies Alice has already viewed, the remaining movies in the list are the recommendations.

Thus, in this example, we recommend Alice to watch (recommendation in order):

1. *Ant-Man and the Wasp*
2. *Toy story*

Note: The recommendations in this example are same for user-based and item-based collaborative filtering. However, when given a larger dataset, which is often the case with real world data, these recommendations would be different.

Matrix Factorization

Matrix factorization is a model-based recommendation technique, as instead of finding items and users similar to each other, it uses data science and machine learning techniques to extract predictions from the rating data. The model is trained by the user rating data, which is then used to predict the rating new items by the users.

There are a wide variety of techniques that fall under the category of matrix factorization. These techniques find broader features of users and items on their own. They describe user and items as combinations of different amounts of each feature. The ratings exist in a 2-dimensional matrix, where rows represent users and column represent movies. Most of the cells in this matrix are unknown, and the aim of this approach is to predict these unknown values.

Principal Component Analysis (PCA) is a machine learning technique described as a dimensionality reduction problem, or a feature extraction tool.

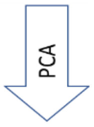
The ratings matrix, with users as rows and movies as columns, is called *R*. Using *PCA*, the dimensions in *R* matrix can be reduced, giving a new matrix called *U*. The columns of this matrix *U* describe the users for each latent feature produced.

PCA is also run on the item matrix, with items as rows and movies as columns. This item matrix is a transpose of user matrix *R* called R^T. *PCA* identifies the latent features and describe each movie as a combination of these latent features in a matrix called *M*.

Below is a representation of the *PCA* applied on *R* and *M* matrixes.

R

User/Movie	*E.T.*	*Ant-Man and the Wasp*	*Jumanji*	*Pulp fiction*	*Toy story*
Alice	5		4	4	
Bob		1			3
Charlie		3		3	5
David	3	3		5	
Elena	1		5		1

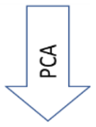


U

User/Feature	Action	Comedy	Science-Fiction
Alice	.3	.3	.4
Bob	.3	.4	.3
Charlie	.4	.5	.1
David	.2	.6	.2
Elena	.8	.1	.1

$\boxed{R^T}$

	Alice	Bob	Charlie	David	Elena
E.T.	5			3	1
Ant-Man and the Wasp		1	3	3	
Jumanji	4				5
Pulp fiction	4		3	5	
Toy story		3	5		1



$\boxed{\mathbf{M}}$

Movie/Feature	Action	Comedy	Science-Fiction
E.T.	.1	.2	.7
Ant-Man and the Wasp	.3	.3	.4
Jumanji	.7	.2	.1
Pulp fiction	.4	.5	.1
Toy story	.3	.6	.1

The matrices *U* and *M* can be used to fill the blanks in the sparse *R* matrix.

$$R = U\Sigma M^T$$

Where *Σ* is an identity matrix. A way to compute *U*, *Σ* and M^T matrix is by using *Singular Value Decomposition* (SVD). *SVD* runs *PCA* on user and item matrix and returns the matrices *U* and M^T. However, since *PCA* can't be used on a sparse matrix, the missing values of *R* need to be updated. Assuming there are at least a few known ratings for any row and column in *U* and M^T, finding the values of these complete rows and columns that best minimize the error in the known ratings in *R* becomes a minimization problem. One such machine learning technique is *Stochastic Gradient Descend* (*SGD*), where is the algorithm loops through all the ratings in a trainset, and computes a predicted rating and the associated prediction error. Then it modifies the parameters and computed the prediction error again until a minimum error value is achieved.

Hybrid filtering technique

Each algorithm has its own strengths and weaknesses. Multiple algorithms can be combined, which might create a better recommendation system. Such an approach of combining multiple algorithms is called Hybrid filtering technique. The hybrid approach can be useful in solving the cold start problem – when there is not enough information on a new user to produce good recommendations for the user.

The class diagram implementing Hybrid algorithm is below:

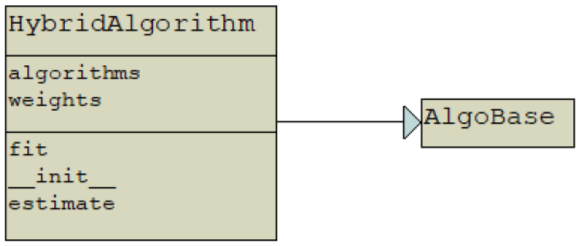
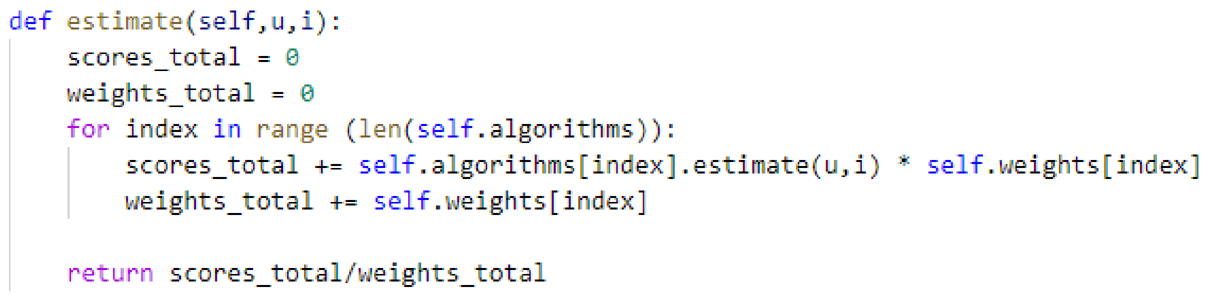


Figure 19: Class diagram of `HybridAlgorithm`

One way to implement hybrid approach is to generate rating predictions using multiple algorithms, and taking the weighted average of result of these algorithms. Below is the snapshot of the code implementing the same:



```
def estimate(self,u,i):
    scores_total = 0
    weights_total = 0
    for index in range (len(self.algorithms)):
        scores_total += self.algorithms[index].estimate(u,i) * self.weights[index]
        weights_total += self.weights[index]

    return scores_total/weights_total
```

Figure 20: Snapshot of Hybrid Algorithm implementation with weights

In this project, content-based algorithm and a random predictor from *surprise-lib*, which produced random recommendations, have been used to create a hybrid algorithm system.

Evaluation

Once the algorithms are implemented, the next step is to evaluate them based on the metrices defined earlier. Depending upon the metric to be checked, different datasets are prepared by `EvaluationData.py`. Top-*N* recommenders are evaluated using Leave-One-Out approach. The `EvaluatedAlgorithm.py` class is created to ease process of applying the metrics on algorithms. This class contains a function called `Evaluate` that computes all the metrics for a given algorithm. There are different ways to evaluate a recommender system, for which we need to create different training and test sets. `EvaluationData.py` class takes in a Dataset and creates all the train/test splits needed by our `EvaluatedAlgorithm` class. The class diagrams for these are shown below:

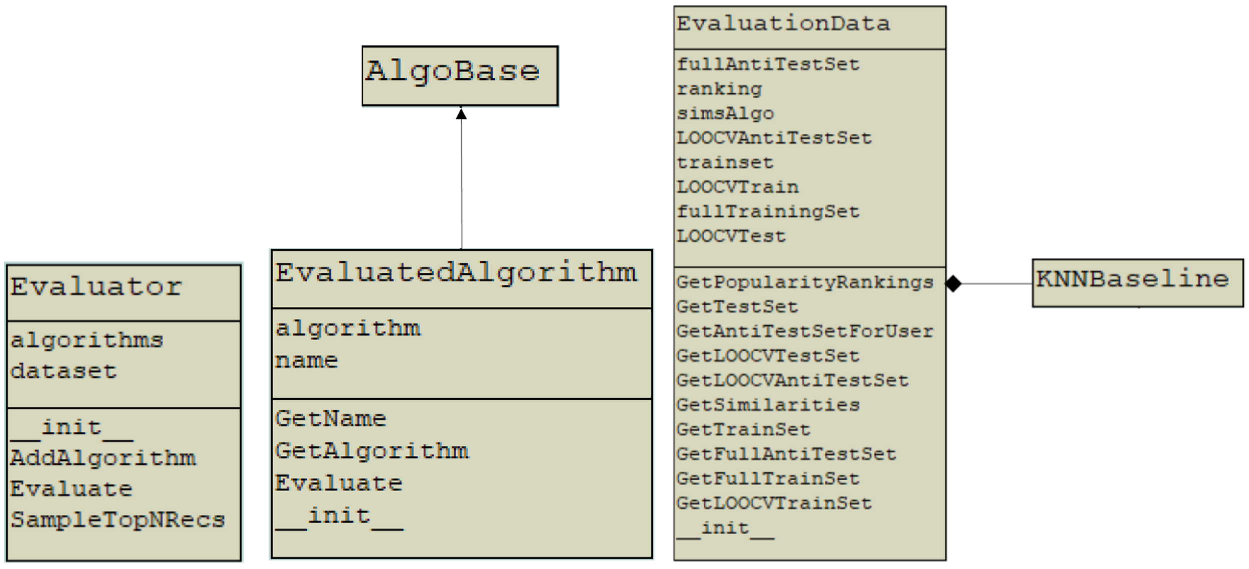


Figure 21: Class diagram for evaluator, EvaluatedAlgorithm, and EvaluationData

At first, an instance of `EvaluationData` for a given dataset is created, which transforms the data as needed, to test the algorithms by wrapping those algorithms with an `EvaluatedAlgoritm` instance. Next, the `evaluate` function is called on `EvaluatedAlgorithm` passing the evaluated data to measure the performance of that algorithm. `EvaluatedAlgorithm` uses all the functions defined in `RecommendationMetrics` to measure *MAE*, *RMSE*, *hit rate*, *ARHR*, *diversity*, and *novelty*.

The `Evaluator.py` class takes in the raw dataset from `MovieLens.py` and creates an `EvaluatedDataset` object. Then, `addAlgorithm` is called for each algorithm we want to compare. Finally, upon calling evaluate, for each EvaluatedAlgorithm, the metrics are evaluated, and the results of the metrics are printed on screen in a tabular form. The `doTopN` parameter allows to bypass the computation of *hit rank* metrics because they can be computationally expensive to produce. `SampleTopNRecs` function prints the actual movies recommended by each algorithm.

Comparison and results

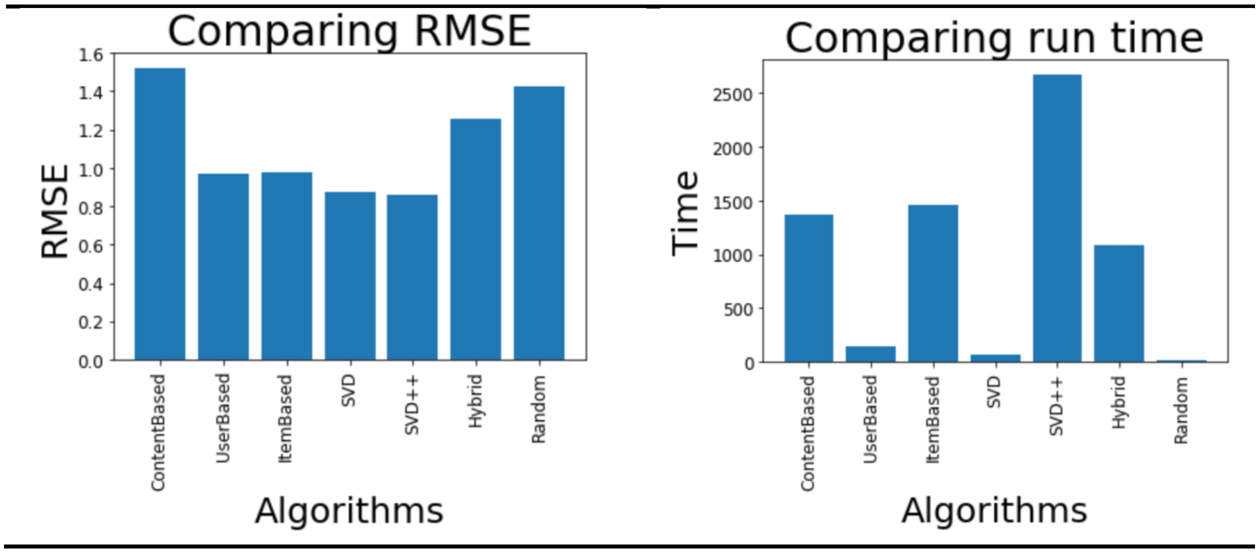


Figure 22: Result

The algorithms are compared based on their accuracy and speed. Other than these parameters, below mentioned metrices are also calculated for these algorithms.

Algorithm	RMSE	MAE	HR	ARHR	Diversity	Novelty	Time (secs)
ContentBased	1.522	1.1964	0.0065	0.0028	0.2984	2399.366	1363.891
UserBased	0.9722	0.7523	0	0	0.8885	6131.331	140.258
ItemBased	0.9769	0.7619	0	0	0.7172	6908.707	1454.978
SVD	0.8752	0.6755	0.0262	0.0092	0.032	525.8528	68.948
SVD++	0.8627	0.664	0.0196	0.0086	0.0772	932.2735	2675.819
Hybrid	1.2578	0.9726	0.0033	0.001	0.2076	1557.98	1088.37
Random	1.4224	1.1366	0.0131	0.0051	0.0493	858.8121	11.973

Ultimate Algorithm bake-off in 8673.424 seconds

The legend explaining these metrices is on the next page:

RMSE:	Root Mean Squared Error. Lower values mean better accuracy.
MAE:	Mean Absolute Error. Lower values mean better accuracy.
HR:	Hit Rate; how often we can recommend a left-out rating. Higher is better.
ARHR:	Average Reciprocal Hit Rank - Hit rate that takes the ranking into account. Higher is better.
Diversity:	1-S, where S is the average similarity score between every possible pair of recommendations for a given user. Higher means more diverse.
Novelty:	Average popularity rank of recommended items. Higher means more novel.

Recommendations for the test user:

Content-based recommendation algorithm

1 - Citizen Kane (1941) 4.7

2 - Quiet Man, The (1952) 4.7

3 - American Beauty (1999) 4.7

4 - Good Will Hunting (1997) 4.7

5 - Before Sunrise (1995) 4.7

6 - Circle of Friends (1995) 4.7

7 - Piano, The (1993) 4.7

8 - Lone Star (1996) 4.7

9 - Breakfast at Tiffany's (1961) 4.7

10 - Casablanca (1942) 4.7

11 - Beautiful Thing (1996) 4.7

12 - Afterglow (1997) 4.7

13 - Buffalo '66 (a.k.a. Buffalo 66) (1998) 4.7

14 - West Side Story (1961) 4.7

15 - Sixth Sense, The (1999) 4.7

User-based collaborative filtering algorithm

1 - The Jinx: The Life and Deaths of Robert Durst (2015) 5

2 - Afterglow (1997) 5

3 - I'm the One That I Want (2000) 5

4 - Far From Home: The Adventures of Yellow Dog (1995) 5

5 - Lassie (1994) 5

6 - Lesson Faust (1994) 5

7 - Black Mirror 5

8 - Dylan Moran: Monster (2004) 5

9 - Bill Hicks: Revelations (1993) 5

10 - My Sassy Girl (Yeopgijeogin geunyeo) (2001) 5

11 - Strictly Sexual (2008) 5

12 - Thief (1981) 5

13 - Career Girls (1997) 5

14 - Siam Sunset (1999) 5

15 - Little Murders (1971) 5

Item-based collaborative filtering algorithm

1 - Reefer Madness: The Movie Musical (2005) 5

2 - Ring, The (1927) 5

3 - Carabineers, The (Carabiniers, Les) (1963) 5

4 - General Died at Dawn, The (1936) 5

5 - Hard Ticket to Hawaii (1987) 5

6 - Roommate, The (2011) 5

7 - Pearl Jam Twenty (2011) 5

8 - FairyTale: A True Story (1997) 5

9 - Merlin (1998) 5

10 - Animal Farm (1954) 5

11 - Zoom (2006) 5

12 - Last Legion, The (2007) 5

13 - Earthsea (Legend of Earthsea) (2004) 5

14 - Dead Like Me: Life After Death (2009) 5

15 - Hunt For Gollum, The (2009) 5

Matrix Factorization algorithm

Using SVD

1 - Lawrence of Arabia (1962) 4.3

2 - Seven Samurai (Shichinin no samurai) (1954) 4.3

3 - Shawshank Redemption, The (1994) 4.3

4 - Goodfellas (1990) 4.2

5 - Godfather: Part II, The (1974) 4.2

6 - Wallace & Gromit: The Best of Aardman Animation (1996) 4.2

7 - Great Escape, The (1963) 4.2

8 - Happiness (1998) 4.2

9 - Back to the Future (1985) 4.2

10 - Dial M for Murder (1954) 4.2

11 - Forrest Gump (1994) 4.2

12 - City of God (Cidade de Deus) (2002) 4.2

13 - Traffic (2000) 4.2

14 - His Girl Friday (1940) 4.2

15 - Young Frankenstein (1974) 4.2

Using SVD++

For user 0 we recommend:

1 - Braveheart (1995) 4.5

2 - Forrest Gump (1994) 4.5

3 - Matrix, The (1999) 4.2

4 - Postman, The (Postino, Il) (1994) 4.2

5 - Hoop Dreams (1994) 4.1

6 - His Girl Friday (1940) 4.1

7 - Indiana Jones and the Last Crusade (1989) 4.1

8 - Independence Day (a.k.a. ID4) (1996) 4.1

9 - Paths of Glory (1957) 4.1

10 - Green Mile, The (1999) 4.1

11 - Road Warrior, The (Mad Max 2) (1981) 4.1

12 - Remember the Titans (2000) 4.1

13 - Fantastic Mr. Fox (2009) 4.0

14 - Seventh Seal, The (Sjunde inseglet, Det) (1957) 4.0

15 - Guess Who's Coming to Dinner (1967) 4.0

Hybrid Algorithm

1 - Rocketeer, The (1991) 5

2 - Legend (1985) 5

3 - NeverEnding Story, The (1984) 5

4 - Office Space (1999) 5

5 - Frankenstein (1931) 5

6 - Iron Giant, The (1999) 5

7 - Love in the Afternoon (1957) 5

8 - Hairspray (1988) 5

9 - Clueless (1995) 5

10 - Pretty Woman (1990) 5

11 - Cutthroat Island (1995) 5

12 - Vampire in Brooklyn (1995) 5

13 - Before and After (1996) 5

14 - Secret of Roan Inish, The (1994) 5

15 - Malice (1993) 5

ApacheSpark using ALS

```
Root-mean-square error = 1.0773268245117327

Model trained in 0.399 seconds

Just Go with It (2011)
Flushed Away (2006)
Julie & Julia (2009)
Pitch Perfect 2 (2015)
Grown Ups 2 (2013)
Happy Feet (2006)
3 Idiots (2009)
Sunshine (2007)
Win Win (2011)
Avengers: Infinity War - Part I (2018)
Vacation (2015)
The Lego Batman Movie (2017)
King of Kong, The (2007)
Grave of the Fireflies (Hotaru no haka) (1988)
```

Future work

The other files provided by *MovieLens* like *tags.csv* can be used in conjunction with genres and year to produce recommendations.

The *MovieLens* dataset can be tied to other data set sources, like *IMDB* or *Netflix*, to get information about the directors and actors in each movie, or critic review scores.

Neural Networks can be used to implement a recommendation system. Significant research has been done in this area with implementation of *Restricted Boltzman Machines* most commonly used.

There are various other algorithms that can be used together, creating a hybrid recommendation algorithm with superior accuracy.

References

Learn. (n.d.). Retrieved from https://www.kaggle.com/learn/overview

Francesco Ricci, Lior Rokach, Bracha Shapira and Paul B. Kantor. Recommender Systems Handbook: A Complete Guide for Research Scientists and Practitioners. Springer.

Sebastian Raschka and Vahid Mirjalili. Machine Learning and Deep Learning with Python, scikit-learn, and TensorFlow. Second Edition. Packt Publishing.

Welcome to Surprise' documentation!¶. (n.d.). Retrieved from https://surprise.readthedocs.io/en/stable/index.html

Kane, F. [Sundog education]. (2017, August 26). Making Movie Recommendations to People. Retrieved from https://www.youtube.com/watch?v=PA1XIDSHldc&t=227s

Kane, F. [Sundog education]. (2017, August 24). User-Based Collaborative Filtering. Retrieved from https://www.youtube.com/watch?v=6mGMBipt7kU

F. Maxwell Harper and Joseph A. Konstan. 2015. The MovieLens Datasets: History and Context. ACM Transactions on Interactive Intelligent Systems (TiiS) 5, 4, Article 19 (December 2015), 19 pages. DOI=http://dx.doi.org/10.1145/2827872

Yehuda Koren, Robet Bell, and Chris Volinsky. 2009.Matrix Factorization Techniques for Recommender Systems.

F.O. Isinkaye, Y.O. Folajimi, B.A. Ojokoh, Recommendation systems: Principles, methods and evaluation, Egyptian Informatics Journal, Volume 16, Issue 3, 2015, Pages 261-273

Dheeraj Bokde, Sheetal Girase, Debajyoti Mukhopadhyay, Matrix Factorization Model in Collaborative Filtering Algorithms: A Survey, Procedia Computer Science, Volume 49, 2015, pp. 136-146

Ruslan Salakhutdinov and Andriy Mnih. Probabilistic matrix factorization. 2008. URL: http://papers.nips.cc/paper/3208-probabilistic-matrix-factorization.pdf.

Li, S.,(2019, January 16). Evaluating A Real-Life Recommender System, Error-Based and Ranking-Based. Retrieved from https://towardsdatascience.com/evaluating-a-real-life-recommender-system-error-based-and-ranking-based-84708e3285b