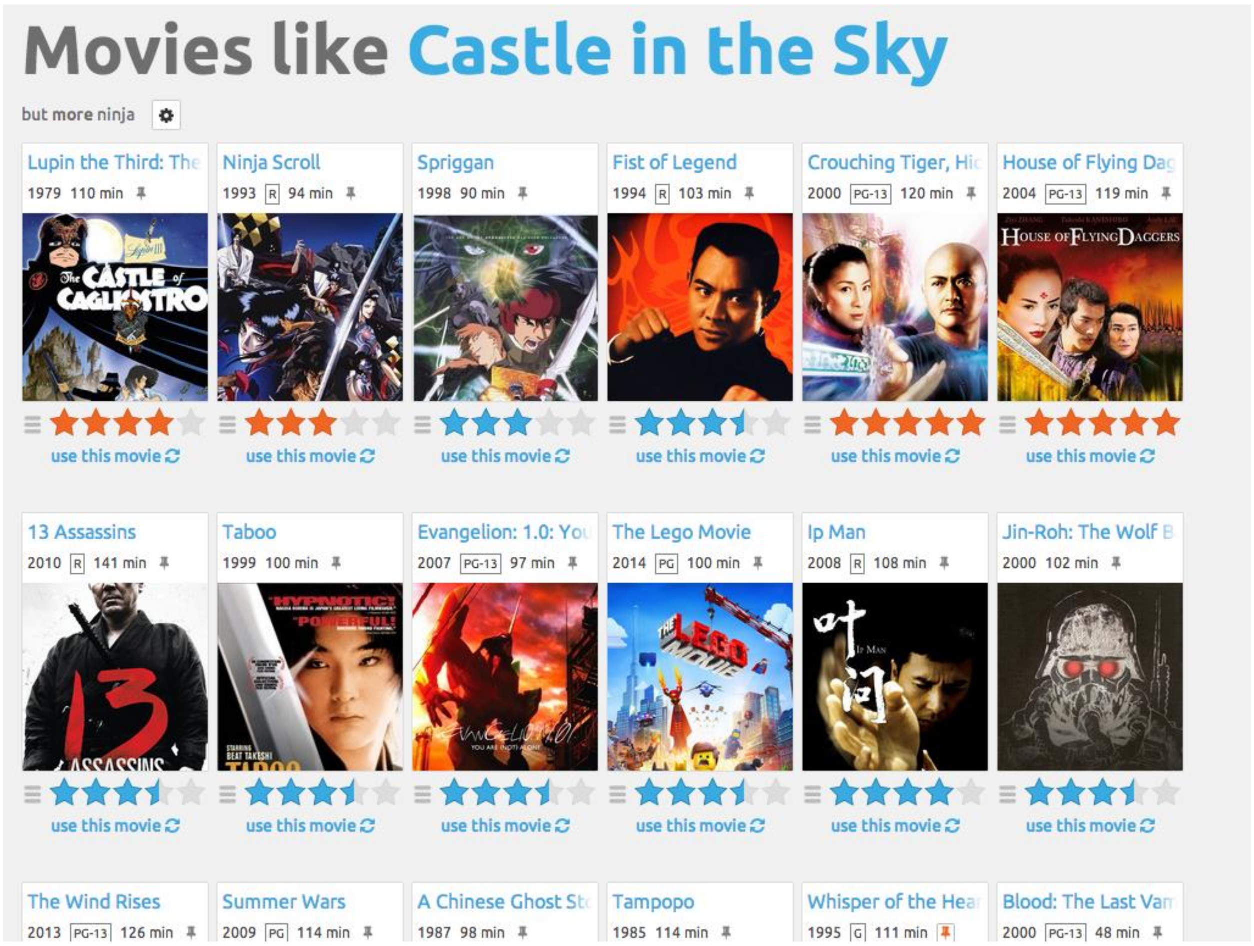
Name: Chitipolu Sri Sudheera
Tool: Python

Recommended New Movies To Users.



The MovieLens DataSet

It has been collected by the GroupLens Research Project at the University of Minnesota. MovieLens 100K dataset. It consists of:

- **100,000 ratings** (1-5) from 943 users on 1682 movies.
- Each user has rated **at least 20 movies.**
- Simple demographic info for the users (age, gender, occupation, zip)
- Genre information of movies

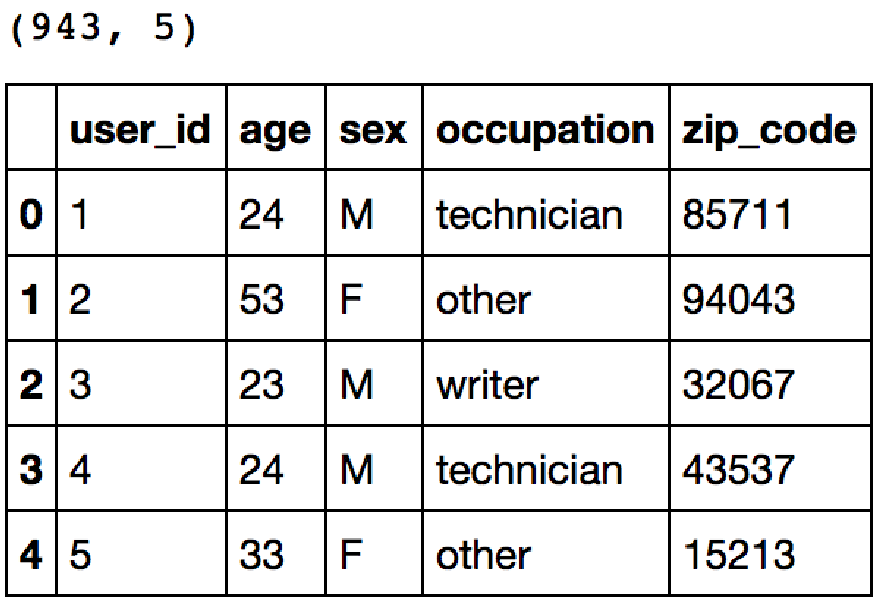
```
import pandas as pd
```

```
# pass in column names for each CSV and read them using pandas.

# Column names available in the readme file



#Reading users file:

u_cols = ['user_id', 'age', 'sex', 'occupation', 'zip_code']

users = pd.read_csv('ml-100k/u.user', sep='|', names=u_cols,

 encoding='latin-1')



#Reading ratings file:

r_cols = ['user_id', 'movie_id', 'rating', 'unix_timestamp']

ratings = pd.read_csv('ml-100k/u.data', sep='\t', names=r_cols,

 encoding='latin-1')



#Reading items file:
```

```
i_cols = ['movie id', 'movie title' ,'release date','video release date', 'IMDb URL',
'unknown', 'Action', 'Adventure',

 'Animation', 'Children\'s', 'Comedy', 'Crime', 'Documentary', 'Drama', 'Fantasy',

 'Film-Noir', 'Horror', 'Musical', 'Mystery', 'Romance', 'Sci-Fi', 'Thriller', 'War',
'Western']

items = pd.read_csv('ml-100k/u.item', sep='|', names=i_cols,

 encoding='latin-1')
```

- **Users**

```
print users.shape

users.head()
```



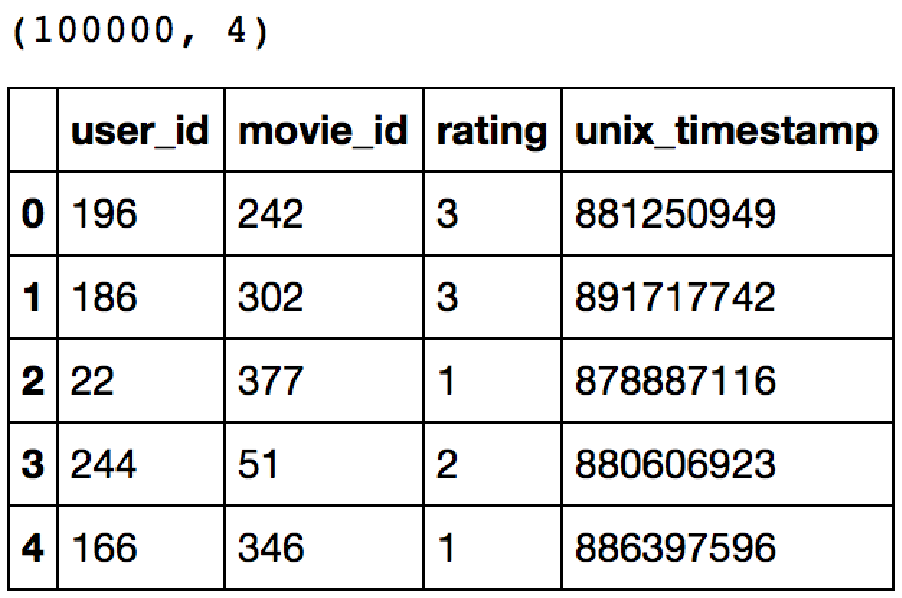
```
(943, 5)
```

	user_id	age	sex	occupation	zip_code
0	1	24	M	technician	85711
1	2	53	F	other	94043
2	3	23	M	writer	32067
3	4	24	M	technician	43537
4	5	33	F	other	15213

This reconfirms that there are 943 users and we have 5 features for each namely their unique ID, age, gender, occupation and the zip code they are living in.

- **Ratings**

```
print ratings.shape


ratings.head()
```



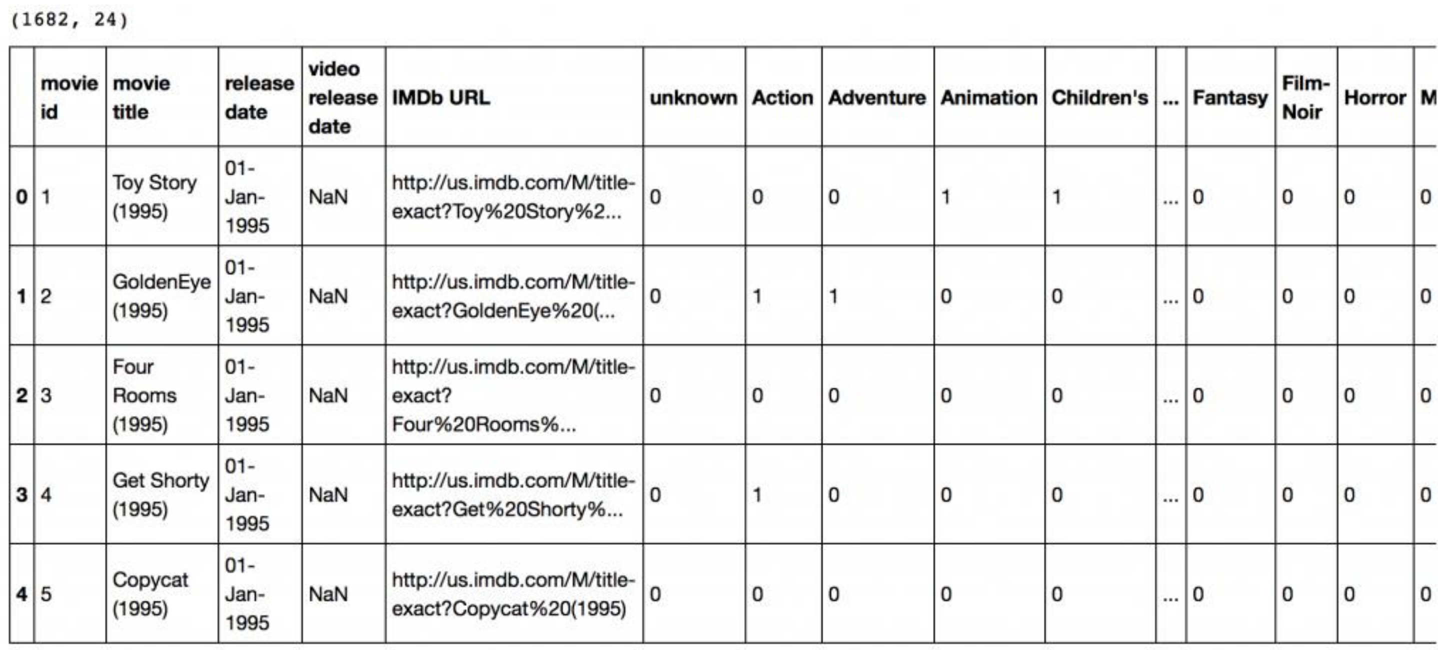
```
(100000, 4)
```

	user_id	movie_id	rating	unix_timestamp
0	196	242	3	881250949
1	186	302	3	891717742
2	22	377	1	878887116
3	244	51	2	880606923
4	166	346	1	886397596

This confirms that there are 100K ratings for different user and movie combinations. Also notice that each rating has a timestamp associated with it.

- **Items**

```
print items.shape


items.head()
```



```
(1682, 24)
```

	movie id	movie title	release date	video release date	IMDb URL	unknown	Action	Adventure	Animation	Children's	...	Fantasy	Film-Noir	Horror	M
0	1	Toy Story (1995)	01-Jan-1995	NaN	http://us.imdb.com/M/title-exact?Toy%20Story%2...	0	0	0	1	1	...	0	0	0	0
1	2	GoldenEye (1995)	01-Jan-1995	NaN	http://us.imdb.com/M/title-exact?GoldenEye%20(...	0	1	1	0	0	...	0	0	0	0
2	3	Four Rooms (1995)	01-Jan-1995	NaN	http://us.imdb.com/M/title-exact?Four%20Rooms%...	0	0	0	0	0	...	0	0	0	0
3	4	Get Shorty (1995)	01-Jan-1995	NaN	http://us.imdb.com/M/title-exact?Get%20Shorty%...	0	1	0	0	0	...	0	0	0	0
4	5	Copycat (1995)	01-Jan-1995	NaN	http://us.imdb.com/M/title-exact?Copycat%20(1995)	0	0	0	0	0	...	0	0	0	0

This dataset contains attributes of the 1682 movies. There are 24 columns out of which 19 specify the genre of a particular movie. The last 19 columns are for each genre and a value of 1 denotes movie belongs to that genre and 0 otherwise.

Now we have to divide the ratings data set into test and train data for making models. Luckily GroupLens provides pre-divided data wherein the test data has 10 ratings for each user, i.e. 9430 rows in total. Lets load that:

```
r_cols = ['user_id', 'movie_id', 'rating', 'unix_timestamp']

ratings_base = pd.read_csv('ml-100k/ua.base', sep='\t', names=r_cols, encoding='latin
-1')

ratings_test = pd.read_csv('ml-100k/ua.test', sep='\t', names=r_cols, encoding='latin
-1')

ratings_base.shape, ratings_test.shape

Output: ((90570, 4), (9430, 4))
```

Since we'll be using GraphLab, lets convert these in SFrames.

```
import graphlab


train_data = graphlab.SFrame(ratings_base)


test_data = graphlab.SFrame(ratings_test)
```

We can use this data for training and testing. Now that we have gathered all the data available. Note that here we have user behaviour as well as attributes of the users and movies. So we can make content based as well as collaborative filtering algorithms.

A Simple Popularity Model

Lets start with making a popularity based model, i.e. the one where **all the users have same recommendation** based on the most popular choices. We'll use the graphlab recommender functions popularity_recommender for this.

We can train a recommendation as:

```
popularity_model = graphlab.popularity_recommender.create(train_data, user_id='user_i

d', item_id='movie_id', target='rating')
```

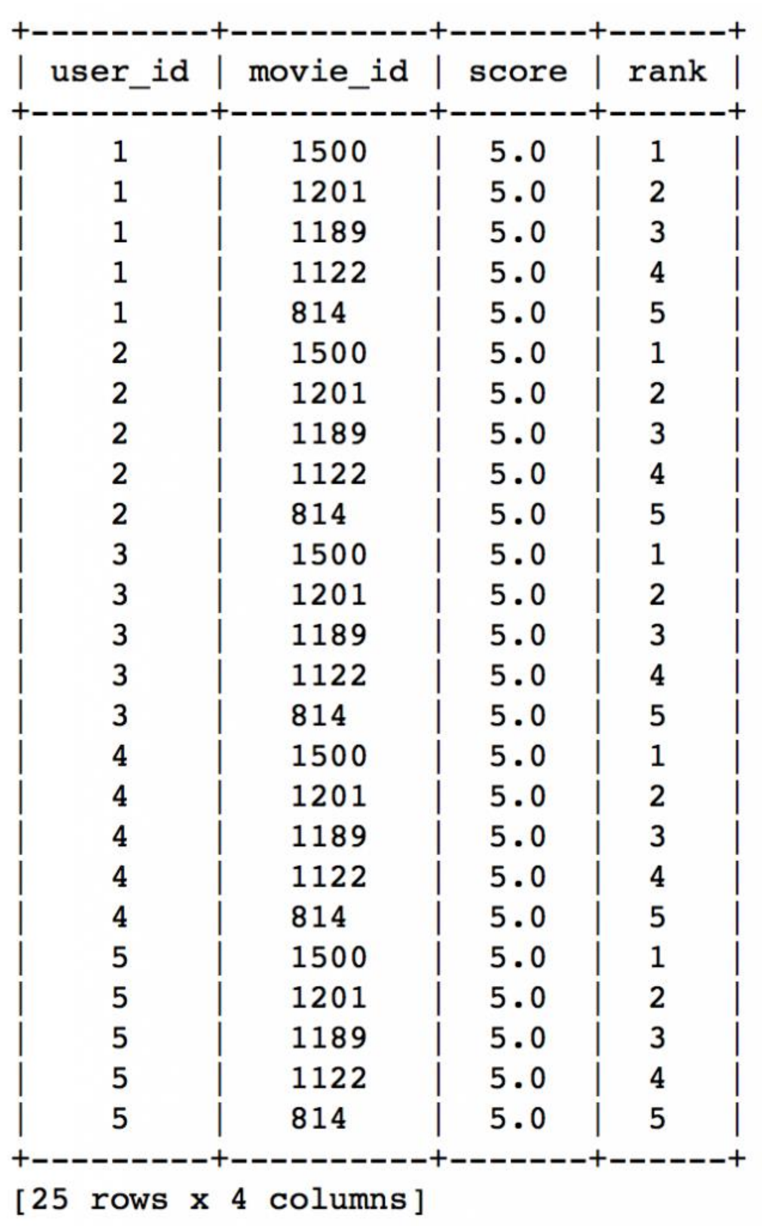
Arguments:

- **train_data**: the SFrame which contains the required data
- **user_id**: the column name which represents each user ID
- **item_id**: the column name which represents each item to be recommended
- **target:** the column name representing scores/ratings given by the user

Lets use this model to make top 5 recommendations for first 5 users and see what comes out:

```
#Get recommendations for first 5 users and print them
```

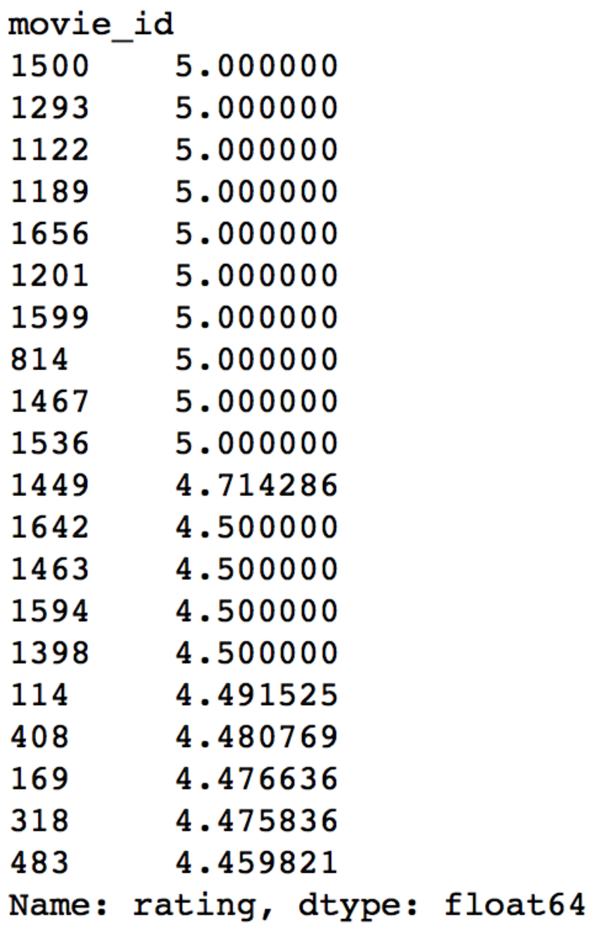
```
#users = range(1,6) specifies user ID of first 5 users

#k=5 specifies top 5 recommendations to be given

popularity_recomm = popularity_model.recommend(users=range(1,6),k=5)

popularity_recomm.print_rows(num_rows=25)
```



```
+---------+----------+-------+------+
| user_id | movie_id | score | rank |
+---------+----------+-------+------+
|    1    |   1500   |  5.0  |  1   |
|    1    |   1201   |  5.0  |  2   |
|    1    |   1189   |  5.0  |  3   |
|    1    |   1122   |  5.0  |  4   |
|    1    |   814    |  5.0  |  5   |
|    2    |   1500   |  5.0  |  1   |
|    2    |   1201   |  5.0  |  2   |
|    2    |   1189   |  5.0  |  3   |
|    2    |   1122   |  5.0  |  4   |
|    2    |   814    |  5.0  |  5   |
|    3    |   1500   |  5.0  |  1   |
|    3    |   1201   |  5.0  |  2   |
|    3    |   1189   |  5.0  |  3   |
|    3    |   1122   |  5.0  |  4   |
|    3    |   814    |  5.0  |  5   |
|    4    |   1500   |  5.0  |  1   |
|    4    |   1201   |  5.0  |  2   |
|    4    |   1189   |  5.0  |  3   |
|    4    |   1122   |  5.0  |  4   |
|    4    |   814    |  5.0  |  5   |
|    5    |   1500   |  5.0  |  1   |
|    5    |   1201   |  5.0  |  2   |
|    5    |   1189   |  5.0  |  3   |
|    5    |   1122   |  5.0  |  4   |
|    5    |   814    |  5.0  |  5   |
+---------+----------+-------+------+
[25 rows x 4 columns]
```

Did you notice something? The recommendations for all users are same – 1500,1201,1189,1122,814 in the same order. This can be verified by checking the movies with highest mean recommendations in our ratings_base data set:

```
ratings_base.groupby(by='movie_id')['rating'].mean().sort_values(ascending=False).hea

d(20)
```



```
movie_id
1500    5.000000
1293    5.000000
1122    5.000000
1189    5.000000
1656    5.000000
1201    5.000000
1599    5.000000
814     5.000000
1467    5.000000
1536    5.000000
1449    4.714286
1642    4.500000
1463    4.500000
1594    4.500000
1398    4.500000
114     4.491525
408     4.480769
169     4.476636
318     4.475836
483     4.459821
Name: rating, dtype: float64
```

This confirms that all the recommended movies have an average rating of 5, i.e. all the users who watched the movie gave a top rating. Thus we can see that our popularity system works as expected. But it is good enough? We'll analyze it in detail later.

A Collaborative Filtering Model

Lets start by understanding the basics of a collaborative filtering algorithm. The core idea works in 2 steps:

1. Find similar items by using a similarity metric
2. For a user, recommend the items most similar to the items (s)he already likes

To give you a high level overview, this is done by making an **item-item matrix** in which we keep a record of the pair of items which were rated together.

In this case, an item is a movie. Once we have the matrix, we use it to determine the best recommendations for a user based on the movies he has already rated. Note that there a few more things to take care in actual implementation which would require deeper mathematical introspection, which I'll skip for now.

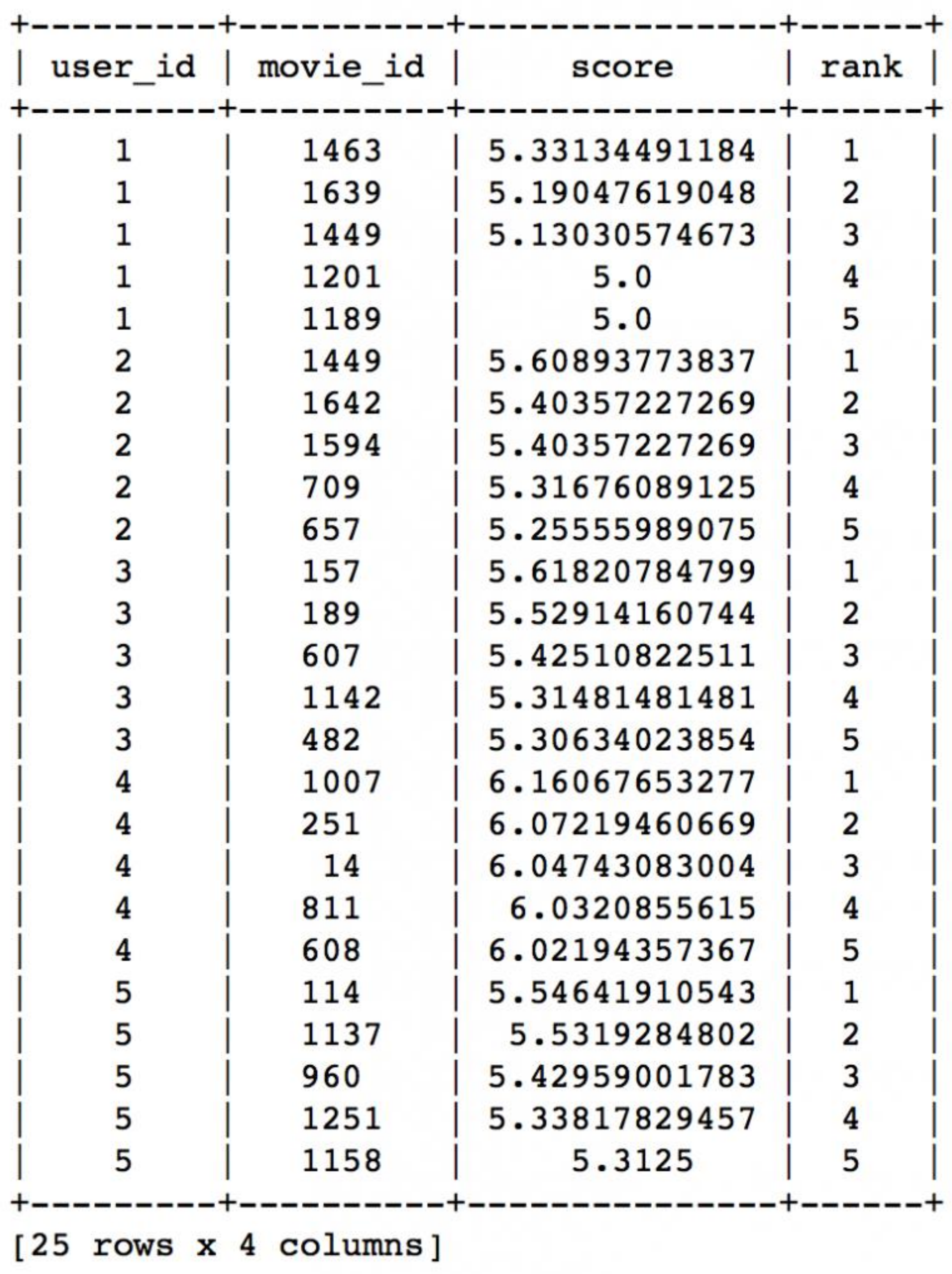
I would just like to mention that there are 3 types of item similarity metrics supported by graphlab. These are:

1. **Jaccard Similarity:**
   - Similarity is based on the number of users which have rated item A and B divided by the number of users who have rated either A or B
   - It is typically used where we don't have a numeric rating but just a boolean value like a product being bought or an add being clicked
2. **Cosine Similarity:**
   - Similarity is the cosine of the angle between the 2 vectors of the item vectors of A and B
   - Closer the vectors, smaller will be the angle and larger the cosine
3. **Pearson Similarity**
   - Similarity is the pearson coefficient between the two vectors.

Lets create a model based on item similarity as follow:

```
#Train Model


item_sim_model = graphlab.item_similarity_recommender.create(train_data, user_id='use

r_id', item_id='movie_id', target='rating', similarity_type='pearson')





#Make Recommendations:


item_sim_recomm = item_sim_model.recommend(users=range(1,6),k=5)
```

```
item_sim_recomm.print_rows(num_rows=25)
```



```
+---------+----------+---------------+------+
| user_id | movie_id |     score     | rank |
+---------+----------+---------------+------+
|    1    |   1463   | 5.33134491184 |  1   |
|    1    |   1639   | 5.19047619048 |  2   |
|    1    |   1449   | 5.13030574673 |  3   |
|    1    |   1201   |      5.0      |  4   |
|    1    |   1189   |      5.0      |  5   |
|    2    |   1449   | 5.60893773837 |  1   |
|    2    |   1642   | 5.40357227269 |  2   |
|    2    |   1594   | 5.40357227269 |  3   |
|    2    |   709    | 5.31676089125 |  4   |
|    2    |   657    | 5.25555989075 |  5   |
|    3    |   157    | 5.61820784799 |  1   |
|    3    |   189    | 5.52914160744 |  2   |
|    3    |   607    | 5.42510822511 |  3   |
|    3    |   1142   | 5.31481481481 |  4   |
|    3    |   482    | 5.30634023854 |  5   |
|    4    |   1007   | 6.16067653277 |  1   |
|    4    |   251    | 6.07219460669 |  2   |
|    4    |    14    | 6.04743083004 |  3   |
|    4    |   811    |  6.0320855615 |  4   |
|    4    |   608    | 6.02194357367 |  5   |
|    5    |   114    | 5.54641910543 |  1   |
|    5    |   1137   |  5.5319284802 |  2   |
|    5    |   960    | 5.42959001783 |  3   |
|    5    |   1251   | 5.33817829457 |  4   |
|    5    |   1158   |     5.3125    |  5   |
+---------+----------+---------------+------+
[25 rows x 4 columns]
```

Here we can see that the recommendations are different for each user. So, personalization exists. But how good is this model? We need some means of evaluating a recommendation engine. Lets focus on that in the next section.

Evaluating Recommendation Engines

For evaluating recommendation engines, we can use the concept of precision-recall. You must be familiar with this in terms of classification and the idea is very similar. Let me define them in terms of recommendations.

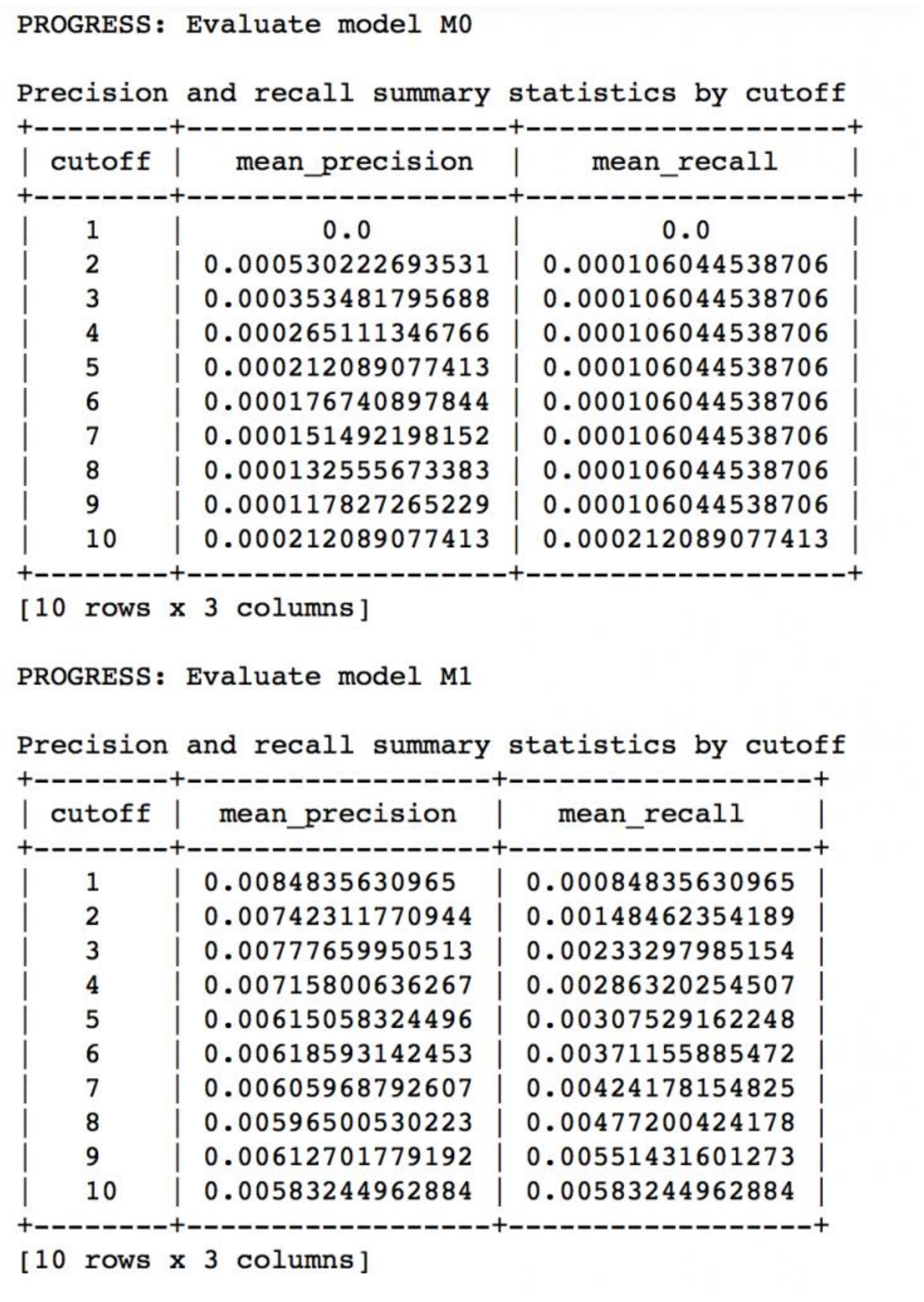
- **Recall:**
    - What ratio of items that a user likes were actually recommended.
    - If a user likes say 5 items and the recommendation decided to show 3 of them, then the recall is 0.6
- **Precision**
    - Out of all the recommended items, how many the user actually liked?
    - If 5 items were recommended to the user out of which he liked say 4 of them, then precision is 0.8

Now if we think about recall, how can we maximize it? If we simply recommend all the items, they will definitely cover the items which the user likes. So we have 100% recall! But think about precision for a second. If we recommend say 1000 items and user like only say 10 of them then precision is 0.1%. This is really low. Our aim is to maximize both precision and recall.

An idea recommender system is the one which only recommends the items which user likes. So in this case precision=recall=1. This is an optimal recommender and we should try and get as close as possible.

Lets compare both the models we have built till now based on precision-recall characteristics:

```
model_performance = graphlab.compare(test_data, [popularity_model, item_sim_model])


graphlab.show_comparison(model_performance,[popularity_model, item_sim_model])
```



```
PROGRESS: Evaluate model M0

Precision and recall summary statistics by cutoff
+--------+-------------------+-------------------+
| cutoff |   mean_precision  |    mean_recall    |
+--------+-------------------+-------------------+
|   1    |        0.0        |        0.0        |
|   2    | 0.000530222693531 | 0.000106044538706 |
|   3    | 0.000353481795688 | 0.000106044538706 |
|   4    | 0.000265111346766 | 0.000106044538706 |
|   5    | 0.000212089077413 | 0.000106044538706 |
|   6    | 0.000176740897844 | 0.000106044538706 |
|   7    | 0.000151492198152 | 0.000106044538706 |
|   8    | 0.000132555673383 | 0.000106044538706 |
|   9    | 0.000117827265229 | 0.000106044538706 |
|   10   | 0.000212089077413 | 0.000212089077413 |
+--------+-------------------+-------------------+
[10 rows x 3 columns]

PROGRESS: Evaluate model M1

Precision and recall summary statistics by cutoff
+--------+------------------+-------------------+
| cutoff |  mean_precision  |    mean_recall    |
+--------+------------------+-------------------+
|   1    | 0.0084835630965  | 0.00084835630965  |
|   2    | 0.00742311770944 | 0.00148462354189  |
|   3    | 0.00777659950513 | 0.00233297985154  |
|   4    | 0.00715800636267 | 0.00286320254507  |
|   5    | 0.00615058324496 | 0.00307529162248  |
|   6    | 0.00618593142453 | 0.00371155885472  |
|   7    | 0.00605968792607 | 0.00424178154825  |
|   8    | 0.00596500530223 | 0.00477200424178  |
|   9    | 0.00612701779192 | 0.00551431601273  |
|   10   | 0.00583244962884 | 0.00583244962884  |
+--------+------------------+-------------------+
[10 rows x 3 columns]
```

Here we can make 2 very quick observations:

1. The item similarity model is definitely better than the popularity model (by atleast 10x)
2. On an absolute level, even the item similarity model appears to have a poor performance. It is far from being a useful recommendation system.