

RECOMMENDER SYSTEMS USING SIMILARITY ANALYSIS ALGORITHMS AT A LARGE SCALE

January 9, 2018

Project done by -

- Lorraine David
- Adeola Otubusen
- Tanmaiyyi Rao - 140361229
- Blanca Herrero Garcia
- Sebastian Creighton

Abstract

This project is an application of recommendation engines using the movielens data set using clustering and computational intelligence. The aim is to design algorithms for data intensive parallel computing frameworks. The code has been implemented in Spark with Python (locally or via the Jupyter Notebook by adding PySpark).

Recommender systems are information filtering tools that aspire to predict the rating for users and items. Movie recommendation systems provide a mechanism to assist users in classifying users with similar interests. This makes recommender systems essentially a central part of websites and e-commerce applications.

MovieLens recommendation dataset has been used for this project. This dataset was already stored in HDFS **/data/movie-ratings..**

This project implements item-based recommendations. The main goal of the project is to generate a list of additional recommendations for the user beyond what she has already rated. The features recommendations are based on genre information.

The core of the project is the implementation of a similarity metric. Cosine similarity and Pearson similarity have been selected as the metrics that are employed for this type of dataset. Our approach has been explained systematically, and the subsequent results have been discussed.

Introduction

Given the MovieLens data, we first inspect the data and notice we can define each movie available to watch as a discrete, qualitative outcome, then the problem of predicting a movie for a person to watch based on a similar user can be solved with Supervised Learning, as we know the target label. Our outcome is the predicted movie for a given user.

User-User Collaborative Filtering Vs item-Item Collaborative Filtering Item-item collaborative filtering, or item-based, or item-to-item, is a form of collaborative filtering for recommender systems based on the similarity between items calculated using people's ratings of those items.

MovieLens Features

GroupLens Research operates a movie recommender based on collaborative filtering. For this project, of the data provided - two main datasets have been used namely Movies.dat and Ratings.dat.

Ratings Data Structure- All ratings are contained in the file `ratings.dat`. Each line of this file represents one rating of one movie by one user, and has the following format:

UserID::MovieID::Rating::Timestamp

The lines within this file are ordered first by UserID, then, within user, by MovieID. Ratings are made on a 5-star scale, with half-star increments. Timestamp represent seconds since midnight Coordinated Universal Time (UTC) of January 1, 1970.

Movies Data Structure

Movie information is contained in the file `movies.dat`. Each line of this file represents one movie, and has the following format:

MovieID::Title::Genres

MovieID is the real MovieLens id. Genres are a pipe-separated list, and are selected from the following:

- Action
- Adventure
- Animation
- Children's
- Comedy
- Crime
- Documentary
- Drama
- Fantasy
- Film-Noir
- Horror
- Musical
- Mystery
- Romance
- Sci-Fi
- Thriller
- War
- Western

Features

We can consider the following features: movie ratings which can take a value between $0.5 \rightarrow 5$.

For feature selection from the MovieLens dataset, movie ratings from users are chosen to give an accurate recommendation. Since we are comparing item to item similarity, we would compute the similarity between every two movies for each user.

Movie Lens Target Variable/Outcome

- Recommending a set of movies (top 10, etc)

The target is to recommend movies to a user based on how they have rated.

We do not consider Regression as we need discrete rather than continuous (numerical) values. Based on our research we have identified that we can apply a Collaborative Filtering algorithm to recommend further movies to a user.

Recommender System

A recommender system (also referred as a recommendation engine) allows algorithm developers predict what a user may or may not like among a list of items.

Recommender systems uses to compute a similarity index for users and recommend items to them. Item-to-Item CF matches item rated by a target user to similar items and combines those similar items in a recommendation list.

Before implementing a collaborative memory-based recommendation engine, the main idea behind such a system must be understood. There are numerous similarity metrics which can be implemented for this type of dataset. Two of the best known ones are Jaccard's Coefficient similarity and Cosine similarity. In our project, we have used userid and movieid as the features and have given the similarity based on the ratings. The Cosine Similarity and the Pearson (Correlation) - Based Similarity have been used. Similarity based on vectors is highly efficient for item-based collaborative filtering.

Cosine Similarity

Finding the cosine similarity between two data objects requires that both objects represent their attributes in a vector. Similarity is then measured as the angle between the two vectors.

$$\cos(\theta) = \frac{AB}{\|A\| \|B\|} \quad (1)$$

This method is useful when finding the similarity between two text documents whose attributes are word frequencies. A perfect correlation will have a score of 1 (or an angle of 0) and no correlation will have a score of 0 (or an angle of 90 degrees).[1]

Pearson (Correlation)-Based Similarity

Just like the cosine similarity, this is also a vector-based similarity. This measures how much the ratings by users for a given pair of item deviates from the average ratings for those items:

$$\text{sim}(i, j) = \frac{\sum_{u \in U} (R_{u,i} - \bar{R}_i)(R_{u,j} - \bar{R}_j)}{\sqrt{\sum_{u \in U} (R_{u,i} - \bar{R}_i)^2} \sqrt{\sum_{u \in U} (R_{u,j} - \bar{R}_j)^2}} \quad (2)$$

In order to design algorithms for the project, Python and PySpark have been selected for the implementation of the code.

Why PySpark over Scala?

Python is easy to learn for Java programmers due to syntax and standard libraries. A strength of Spark is its compatibility with multiple different programming languages. For data scientists and analysts familiar with Python, the PySpark API makes it simple for writing code that takes advantage of Spark to deliver significant improvements in processing speed for large sets of data. Full support for C extensions in Python means that data scientists will typically not need to make major changes to their Python code in order to use Spark. [2]

Out-of-the-box machine learning/statistics packages Python has much more to bring on the table when it comes to out-of-the-box packages implementing most of the standard procedures and models you generally find in the literature and/or broadly adopted in the industry. Scala is still way behind in that yet can benefit from the Java libraries compatibility and the community developing some of the popular machine learning algorithms on

their distributed version directly on top of Spark.

It might make sense in some scenarios to crunch your large dataset into a vector space which can perfectly fit in memory and take advantage of the richness and advanced algorithms available in Python.[3]

RDDs and DATAFRAMES

Resilient Distributed Dataset (RDDs) are the building blocks of Spark. It is the original API that Spark exposed and pretty much all the higher level APIs decompose to RDDs. From a developer's perspective, an RDD is simply a set of Java or Scala objects representing the data. RDDs can be used to perform low-level transformations and actions on the unstructured data. In addition, it is not necessarily need the optimization and performance benefits that DataFrames can offer structured data.

The advantages of RDDs are manifold, but there are also some disadvantages. For example, it is easy to build inefficient transformation chains, they are slow with non-JVM languages such as Python, they can not be optimized by Spark. The transformation chains are not very readable in the sense that the solution won't be immediately shown.

Because of the disadvantages of working with RDDs, the DataFrame API provides a higher level abstraction that allows to use a query language to manipulate the data. This higher level abstraction is a logical plan that represents data and a schema. Spark DataFrames are optimized and therefore also faster than RDDs. Especially when working with structured data, switching your RDD to a DataFrame should be considered.

A Comparison of Collaborative-Filtering Recommendation Algorithms

There are two main categories of collaborative filtering (CF) algorithms: memory-based and model-based methods.

Memory-based methods use the entire or a sample of the user-item database to generate a prediction. Every user is part of a group of people with similar interests. By identifying the neighbors of a new user, a prediction of preferences on new items for him or her can be produced. The most popular two memory-based methods are user-based and item-based collaborative filtering. These methods are example of neighboring-based methods which uses the following steps: firstly, calculate the similarity ω_{ij} , which indicate the distance, correlation, or weight, between two users or two items i and j . Secondly, produce a prediction of the active user by taking the weighted average of all the ratings of the user or item on a certain item or user. Finally, when the task is to generate Top- N recommendation, k most similar items or users have to be found after computing the similarities, then the neighbors have to be aggregated to get the top- N most frequent items as the recommendation.

Model-based methods, on the other hand, build parametrized models and recommend items with the highest rank, returned by model. Model-based CF algorithms, such as Bayesian models, clustering models, and dependency networks, have been investigated to solve the shortcomings of memory-based CF algorithms. Usually, classification algorithms can be used as CF models if the user ratings are categorical, and regression models and SVD methods and be used for numerical ratings. These methods (also called matrix factorization) are more accurate with enough data and got popularity during the Netflix Prize. These methods may differ in: the model they use, learning approach or objective function.

Data Analysis

Before we started work on the project, the data was inspected.

```
ratings_file = "/data/movie-ratings/ratings.dat"
movies_file = "/data/movie-ratings/movies.dat"
```

```
ratings_raw = sc.textFile(ratings_file)
movies_raw = sc.textFile(movies_file)
```

```
ratings_raw.take(5)
```

Out[12]:

```
[u'1::122::5::838985046 ',
 u'1::185::5::838983525 ',
 u'1::231::5::838983392 ',
 u'1::292::5::838983421 ',
 u'1::316::5::838983392 ']
```

```
movies_raw.take(5)
```

Out[13]:

```
[u'1::Toy Story (1995)::Adventure|Animation|Children|Comedy|Fantasy ',
 u'2::Jumanji (1995)::Adventure|Children|Fantasy ',
 u'3::Grumpier Old Men (1995)::Comedy|Romance ',
 u'4::Waiting to Exhale (1995)::Comedy|Drama|Romance ',
 u'5::Father of the Bride Part II (1995)::Comedy ']
```

There is no header file Notice that the columns are separated by :: Observe also that the field is in the following format

```
movie::titleandyear::genre
```

```
print('There are {0} rows in the {1}'.format(movies_raw.count(), movies_file))
```

There are 10681 rows in the /data/movie-ratings/movies.dat

```
print('There are {0} rows in the {1}'.format(ratings_raw.count(), ratings_file))
```

There are 10000054 rows in the /data/movie-ratings/ratings.dat

Since there are approximately 10M records, it may be faster to set the number of partitions on spark. Since the movie file is relatively small with approximately 10K records we can hold in memory using collect

In order to show the data in a dataframe, the following code has been implemented:

```
ratings_raw = sc.textFile("/data/movie-ratings/ratings.dat")
ratings = ratings_raw.map(lambda line: re.split(r ':: ', line))
rdd_data = sqlContext.createDataFrame(list1, ["userid", "movieid", "rating", "timestamp"])
rdd_data.show(10)
```

Implementation

Firstly, the algorithm has been implemented in PySpark using RDDs and Dataframes. The map function have been used to get the relevant data from the ratings dataset and movies dataset repsectively. Both the datasets (Ratings and Movies) have been converted to dataframes to easily view what we will do on the data to calculate the similarity measures.

userid	movieid	rating	timestamp
1	122	5	838985046
1	185	5	838983525
1	231	5	838983392
1	292	5	838983421
1	316	5	838983392

Only showing top 5 rows.

movieid	title	genre
1	Toy Story (1995)	Adventure Animati...
2	Jumanji (1995)	Adventure Childre...
3	Grumpier Old Men ...	Comedy Romance
4	Waiting to Exhale ...	Comedy Drama Romance
5	Father of the Bri ...	Comedy

Only showing top 5 rows.

A dataframe has been created to check for the different users and their given ratings for each movie. For example -

uaer_id	itemid_rating
36000	[1,2.0]
36000	[32,4.5]
36000	[34,2.0]
36000	[50,4.0]
36000	[107,3.5]

Only showing top 5 rows

This is followed by joining the ratings.dat to itself. The join function combines two datasets (Key,ValueV) and (Key,ValueW) together to get (Key, (ValueV,ValueW)) where the key is the user ID and ValueV and ValueW are tuples which consists of the movie ID and the rating that the user has given that movie.

To view these combinations ina dataframe, the following code is used:

```
user_ratings_data = ratings.join(ratings)
user_ratingsDF2 = sqlContext.createDataFrame(user_ratings_data,['user_id','itemid_rating'])
```


user_id	itemid_rating
48000	[[1,1.0],[1,1.0]]
48000	[[1,1.0],[2,3.0]]
48000	[[1,1.0],[19,0.5]]
48000	[[1,1.0],[34,1.5]]
48000	[[1,1.0],[39,1.5]]

Only showing top 5 rows

Cosine and Pearson Similarity will be used to compute the similarity for the ratings. This is done by looking at one movie that user has rated, then seeing what else they have rated and finding all the users who have done the same and finding the similarity between these ratings for the movies in question:

```

from math import sqrt
def cosine_similarity(ratingPairs):

    numPairs = 0
    sum_xx = sum_yy = sum_xy = 0

    for ratingX, ratingY in ratingPairs:

        sum_xx += ratingX * ratingX
        sum_yy += ratingY * ratingY
        sum_xy += ratingX * ratingY
        numPairs += 1

    numerator = sum_xy
    denominator = sqrt(sum_xx) * sqrt(sum_yy)

    score = 0
    if (denominator):
        score = ((float(numerator)) / (float(denominator)))

    return (score, numPairs)

def pearson_similarity(ratingPairs):

    numPairs = 0
    sum_xx = sum_yy = sum_xy = sum_x = sum_y = 0

    for ratingX, ratingY in ratingPairs:
        sum_xx += ratingX * ratingX
        sum_yy += ratingY * ratingY
        sum_xy += ratingX * ratingY
        sum_x += ratingX
        sum_y += ratingY
        numPairs += 1
    numerator = (sum_xy * numPairs) - (sum_x * sum_y)
    denominator = sqrt((sum_xx * numPairs) - (sum_x**2)) * sqrt((sum_yy * numPairs) - (sum_y**2))
    score = 0
    if (denominator):
        score = (numerator / (float(denominator)))
    return (score, numPairs)

```

To ensure that duplicates have been removed, the following code is used:

```
def removeDuplicates((userid , ratings )):
    (item1 , value1) = ratings [0]
    (item2 , value2) = ratings [1]
    return item1 < item2
```

Now, to implement these functions onto the MovieLens data sets (as RDDs), they are first parsed by mapping and splitting the lines as before. An increased number of partitions (written as `partitionBy(numPartitions)` where `numPartitions = 1000`) are also used to run the code more efficiently since we first deal with very large RDDs. Note, the full data set (including approx. 10 million ratings) is parsed through the `if` statement's steps, otherwise if the smaller set is used, it will follow the other:

```
if (ratings_file.find( '.dat' ) != -1):
    ...

else:
    ...
```

Then the rating pairs by a given user are created. This follows from earlier where the tuples (Key, (ValueV, ValueW)) include the pairs of *itemid_rating* tuples by joining the ratings set to itself. But now, duplicates of combinations are removed via the `removeDuplicates` function and the `itemItem` function put into `key(movie ID pair)- value(ratings pair)` pairs. They are then grouped by key:

```
...
movie_pairs_ratings= movie_pairs.groupByKey()
```

Now the similarities are implemented when an the specific argument is called and then sorted by key (the movie ID pair):

```
algorithms=["COSINE" , "PEARSON"]

if algorithm == algorithms[0] :
    item_item_similarities = movie_pairs_ratings.mapValues(cosine_similarity).persist()
elif algorithm == algorithms[1] :
    item_item_similarities = movie_pairs_ratings.mapValues(pearson_similarity).persist()
else:
    item_item_similarities = movie_pairs_ratings.mapValues(cosine_similarity).persist()

item_item_sorted=item_item_similarities.sortByKey()
```

The following sample output is a result corresponding to `algorithm == "COSINE"`:

Sample output:

```
((1 , 2) , (0.9633070604343126 , 71))
((1 , 3) , (0.9269097345177958 , 39))
((1 , 4) , (0.932135764636765 , 7))
```

To filter out the popular movies and the most simialr, certain requirements are set.

Firstly, the movie ID in question needs to be set (e.g. *movie_id* = 1). A threshold is then placed on the similarities (0.97 which is considered as "good") and the co-occurences/min-occurence (set to 1000 users rating the movie in question and the other in the combination):

```
# Filter for movies with this sim that are "good" as defined by
# our quality thresholds above
filteredResults = item_item_sorted.filter(lambda((item_pair , similarity_occurence)) : \
    (item_pair[0] == movie_id or item_pair[1] == movie_id) \
    and similarity_occurence[0] > threshold and similarity_occurence[1] > minOccurence)
```

The top ten recommendations are then given by setting the similarities as the key, sorting them in descending order and extracting the the top 10 movies (setting `topN=10`):

```
if (topN==0):
    topN=10
```

```
results = filteredResults.map(lambda((x,y)): (y,x)).sortByKey(ascending = False)
resultsTopN = sc.parallelize(results.take(topN))
```

Now, to get the titles of these movies, the resulting RDD is rearranged and joined to the movies data set:

```
resultsKey = resultsTopN.map(lambda((x,y)): (y[1],x[0]))
topMoviesJoin = resultsKey.join(movies)
top_N_Movies = topMoviesJoin.map(lambda (x,y): (y[0],(x,y[1][0].encode('ascii', 'ignore'))))
.sortByKey(ascending = False)
```

Sample output:

```
(3114, (0.9870973879980668, (u'ToyStory2(1999)', u'Adventure|Animation|Children|Comedy|Fantasy')))
```

To get rid of the unwanted fields, the following is executed:

```
top_N_Movies_Sorted = top_N_Movies.map(lambda (x,y): (y[1],y[0],x))
```

Sample output:

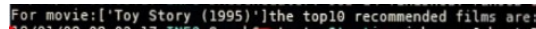
```
(0.9870973879980668, (3114, 'ToyStory2(1999)'))
```

Results

Following the code, the aim is to find the top 10 films based on the similarity measures. To make it clear which movie the recommendations are based off, the following can be executed and shown:

```
film_in_q = resultsTopN.map(lambda((x,y)): (y[0],x[0]))
focus_join= film_in_q.join(movies)
one_movie = focus_join.map(lambda (x,y): (y[1][0].encode('ascii', 'ignore'))).take(1)
print("For movie" + str(one_movie) + ", the Top 10 recommended films are:")
```

Example: Toy Story(1995) (movie ID = 1). The film that is in focus can be shown in the terminal with the output and by using the code below: (screenshot of the string)

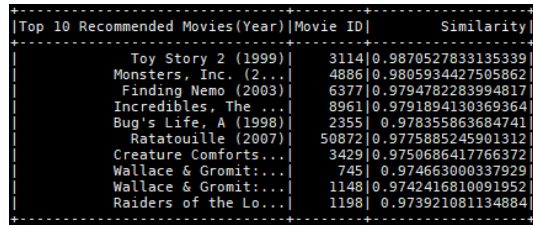


```
For movie:['Toy Story (1995)']the top10 recommended films are:
```

Figure 1:

Also, by converting the *top_N_Movies_Sorted* to a data frame, it is easy to see the output in-terminal: (screen shots of both cosine and pearson)

Cosine:



Top 10 Recommended Movies(Year)	Movie ID	Similarity
Toy Story 2 (1999)	3114	0.9870527833135339
Monsters, Inc. (2001)	4886	0.9805934427505862
Finding Nemo (2003)	6377	0.9794782283994817
Incredibles, The (2003)	8961	0.9791894130369364
Bug's Life, A (1998)	2355	0.978355863684741
Ratatouille (2007)	50872	0.9775885245901312
Creature Comforts (1990)	3429	0.9750686417766372
Wallace & Gromit: The Dog of the Week (1989)	745	0.974663000337929
Wallace & Gromit: The Big Dog on Farm (1989)	1148	0.9742416810091952
Raiders of the Lost Ark (1959)	1198	0.973921081134884

Figure 2:

Pearson:

Top 10 Recommended Movies(Year)	Movie ID	Similarity
Toy Story 2 (1999)	3114	0.7299088450991106
Bug's Life, A (1998)	2355	0.5371062285901946
Monsters, Inc. (2001)	4886	0.513778533618603
Finding Nemo (2003)	6377	0.4949078491878249
Incredibles, The (2004)	8961	0.4337938711705013
Cars (2006)	45517	0.4272029092702506
Aladdin (1992)	588	0.41957723648319767
One Hundred and One Dalmatians (a.k.a. 101 Dalmatians) (1961)	2085	0.4137735898783171
Lion King, The (1994)	364	0.40955247128224576
Little Mermaid, The (1989)	2081	0.4009381558896973

Figure 3:

Where the corresponding RDD output is as follows:

Cosine:

```
('Toy Story 2 (1999)', 3114, 0.9870527833135339)
('Monsters, Inc. (2001)', 4886, 0.9805934427505862)
('Finding Nemo (2003)', 6377, 0.9794782283994817)
('Incredibles, The (2004)', 8961, 0.9791894130369364)
('Bug's Life, A (1998)', 2355, 0.978355863684741)
('Ratatouille (2007)', 50872, 0.9775885245901312)
('Creature Comforts (1989)', 3429, 0.9750686417766372)
('Wallace & Gromit: A Close Shave (1995)', 745, 0.974663000337929)
('Wallace & Gromit: The Wrong Trousers (1993)', 1148, 0.9742416810091952)
('Raiders of the Lost Ark (Indiana Jones and the Raiders of the Lost Ark) (1981)', 1198, 0.973921081134884)
```

Pearson:

```
('Toy Story 2 (1999)', 3114, 0.7299088450991106)
('Bug's Life, A (1998)', 2355, 0.5371062285901946)
('Monsters, Inc. (2001)', 4886, 0.513778533618603)
('Finding Nemo (2003)', 6377, 0.4949078491878249)
('Incredibles, The (2004)', 8961, 0.4337938711705013)
('Cars (2006)', 45517, 0.4272029092702506)
('Aladdin (1992)', 588, 0.41957723648319767)
('One Hundred and One Dalmatians (a.k.a. 101 Dalmatians) (1961)', 2085, 0.4137735898783171)
('Lion King, The (1994)', 364, 0.40955247128224576)
('Little Mermaid, The (1989)', 2081, 0.4009381558896973)
```

Movie ID	Top 10 Recommended Movies (Year)	Cosine Similarity
3114	Toy Story 2(1999)	0.987052783
4886	Monsters Inc. (2001)	0.980593443
6377	Finding Nemo (2003)	0.979478228
8961	Incredibles, The (2004)	0.979189411
2355	Bug's Life, A (1998)	0.978355864
50872	Ratatouille (2007)	0.977588525
3429	Creature Comforts (1989)	0.975068642
745	Wallace & Gromit: A Close Shave (1995)	0.974663
1148	Wallace & Gromit: The Wrong Trousers (1993)	0.974241681
1198	Indiana Jones and the Raiders of the Lost Ark) (1981)	0.973921081

Figure 4:

Movie ID	Top 10 Recommended Movies (Year)	Pearson Similarity
3114	Toy Story 2 (1999)	0.729908845
2355	Bug's Life, A (1998)	0.537106229
4886	Monsters, Inc. (2001)	0.513778534
6377	Finding Nemo (2003)	0.494907849
8961	Incredibles, The (2004)	0.433793871
45517	Cars (2006)	0.427202909
588	Aladdin (1992)	0.419577236
2085	One Hundred and One Dalmatians (a.k.a. 101 Dalmatians) (1961)	0.41377359
364	Lion King, The (1994)	0.409552471
2081	Little Mermaid, The (1989)	0.400938156

Figure 5:

Visualisation

For the first graph below, obtained the number of counts per rating plotting a bar graph for outcome. This assumes a user has awarded multiple ratings.

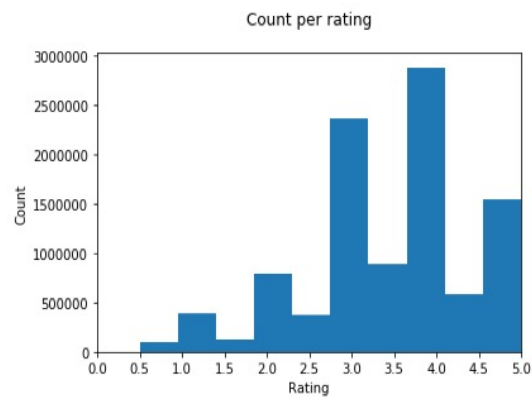


Figure 6: For results set, the top3 ratings counts came from 4.0 with 2875850, 3.0 with 2356676 and 5.0 with 1544812. Meanwhile, the lowest coming from 0.5 with 94988.

The graph below, obtained the top 10 Cosine Similarity values for Toys Story to another plotting a bar graph for outcome.

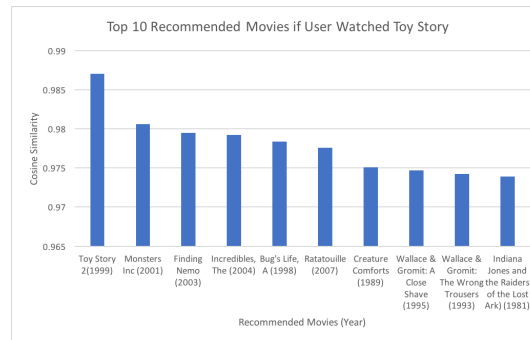


Figure 7: For results set, the top 3 cosine similarity films from $movie_id = 1$ came from $movie_id2 = 3114$ (Toy Story 2) with 0.987052783313534, $movie_id2 = 4886$ (Monsters, Inc (2001)) with 0.9805934428 and $movie_id2 = 6377$ (Finding Nemo (2003)) with 0.9794782284. One assumption that can be made is that top6 films are closest due to all being Disney films.

For the graph below, obtained the top 10 Pearson Similarity values from Toy Story to another plotting a bar graph for outcome.

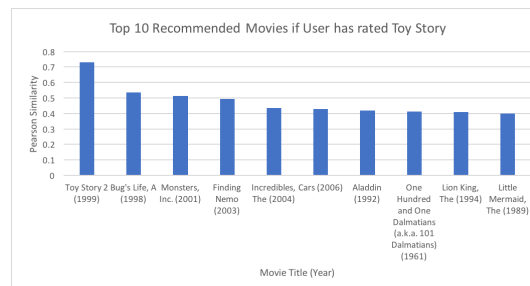


Figure 8: For results set, the top 3 Pearson similarity films came from Toy Story 2 with 0.729908845, A Bug's Life with 0.537106229 and Monster's Inc with 0.513778534. One assumption that can be made is that all of top 10 films are from Disney.

It is clear that the Pearson Similarity decreases more rapidly than the Cosine Similarity as we go from the most similar. This shows how the adjusting the similarity around the mean rating a film can affect the measure. This is reflected in the positions of the recommended movies.

Conclusion

To conclude, an item-based collaborative filtering algorithm has been used to build this movie recommender. Where, Cosine and Pearson based similarity (both of which are based on vectors) have been used to compute the similarity. Users have been given the top ten recommendations based on the movies they have rated. The problem has been addressed and an appropriate solution has been discussed in this report. The results and the relevant graphs have been presented along with a detailed explanation of the method. The corresponding code has been attached in the source files.

Appendix

1. To run the source file in terminal through PySpark, run:

```
spark-submit MovieLenSim.py <ratings file> <movies file> < movie ID> <threshold> <top N><minOccurence> <algorithm>
```

Example: `spark-submit MovieLenSim.py /data/movie-ratings/ratings.dat /data/movie-ratings/movies.dat 1 0.97 10 1000 COSINE`

Bibliography

- [1] mines.humanoriented.com (n.d.) Mining Similarity Using Eucidean Distance, Person Correlation, and Filtering [online] Available at:[http : //mines.humanoriented.com/classes/2010/fall/csci568/portfolio_exports/mvoget/similarity/similarity.html](http://mines.humanoriented.com/classes/2010/fall/csci568/portfolio_exports/mvoget/similarity/similarity.html) [Accessed 8 Jan. 2018]
- [2] dezyre.com (n.d.). Scala vs Python for Apache Spark. [online] Available at:[https : //www.dezyre.com/article/scala-vs-python-for-apache-spark/213](https://www.dezyre.com/article/scala-vs-python-for-apache-spark/213) [Accessed 8 Jan. 2018].
- [3] datasciencevademecum.wordpress.com (n.d.) 6 Points to Compare Python and Scala for Data Science using Apache Spark [online] Available at: [https : //datasciencevademecum.wordpress.com/2016/01/28/6-points-to-compare-python-and-scala-for-data-science-using-apache-spark/](https://datasciencevademecum.wordpress.com/2016/01/28/6-points-to-compare-python-and-scala-for-data-science-using-apache-spark/) [Accessed 8 Jan. 2018].
- [4] www.cs.carleton.edu/cs_comps/0607/recommend/recommender/itembased.html [Accessed 8 Jan. 2018].
- [5] Kane, F. (n.d.). Frank Kane's Taming big data with Apache Spark and Python. [Accessed 8 Jan. 2018]
- [6] Matplotlib.org. (2018).matplotlib.pyplot.bar Matplotlib2.1.1.post967+g3c1f423 documentation. [online] Available at: [https : //matplotlib.org/devdocs/api/_as_gen/matplotlib.pyplot.bar.html](https://matplotlib.org/devdocs/api/_as_gen/matplotlib.pyplot.bar.html) [Accessed 8 Jan. 2018].
- [7] Stackoverflow.com, (2018). Finding all combinations by groups PySpark. [online] Available at: [https : //stackoverflow.com/questions/44485057/finding-all-combinations-by-groups-pyspark](https://stackoverflow.com/questions/44485057/finding-all-combinations-by-groups-pyspark) [Accessed 8 Jan. 2018].
- [8] Su, X. and Khoshgoftaar, T. (2018). A Survey of Collaborative Filtering Techniques.[Accessed 8 Jan. 2018]