FINAL PROJECT #2:

MOVIE RECOMMENDATION BY

COLLABORATIVE FILTERING USING THE NETFLIX DATA

By

Yamini Ranganathan

DSCI 6007

Motivation

Project chosen: Movie Recommendation by Collaborative Filtering using the Netflix Data

Recommendation systems are the main driving force behind most modern business models today as they are driving force for revenue. With the world increasingly becoming digitalized, recommendation systems play a vital role in customer service as it serves the customers with the products they like. With the rising amount of information and with a significant rise in the number of users, it is becoming important for companies to search, map and provide them with the relevant chunk of information according to their preferences and tastes. To show the importance of recommendation engine, it is seen that 35% of Amazon.com's revenue is generated by its recommendation engine. To get an idea of business value of recommendation engines, Netflix thinks its personalized recommendation engine is worth $1 billion per year (Ref: https://artelliq.com/blog/how-netflix-s-ai-recommendation-engine-helps-it-save-1-billion-a-year/). The Netflix recommendation model for can also be used for other industrial application such Cybersecurity where analysts can apply models that automatically deducts the alerts or activities that are truly suspicious and instead of wasting time, teams will know exactly which alerts matter and their order of urgency to address the issue on priority based.

Documentation of approach: Personalization is Key to the Netflix User Experience as to maximize a user's experience, NetFlix needs to improve the user satisfaction by recommending the most relevant and entertaining content. Personalization is extremely challenging as each person is unique. Netflix estimates the likelihood that user will watch a particular title in their catalog based on a number of factors including: user's interactions with their service (such as viewing history and how the user rated other titles) other members with similar tastes and preferences on their service, any information about the titles, such as their genre, categories, actors, release year, etc. In addition to knowing what user have watched on Netflix, to best personalize the recommendations they also look at things like: 1 the time of day the user watch 2) the devices used to watch Netflix on, and 3 how long the user watched something on Nextflix. The recommendation accounts for all these factors when making recommendations. Since mainly the recommendations are based on user preferences, the document of approach for Netflix will be Collaborative Filtering instead of content-based filtering approach.

Problem 1: Collaborative Filtering Approach:

What is collaborative filtering: In simple terms, the process of identifying similar users and recommending what similar users like is called collaborative filtering.

Collaborative Filtering (CF) is a machine technique that is used to identify relationships between pieces of data. The main idea that governs the collaborative methods is that through past user-item interactions when processed through the system, it becomes sufficient to detect similar users or similar items to make predictions based on these estimated facts and insights. Based on the concept of users' historical preference on a set of items with the core assumption that the users who have agreed in the past tend to also agree in the future. As a recommender system it is used to identify similarities between user data and items and recommendation is performed on the

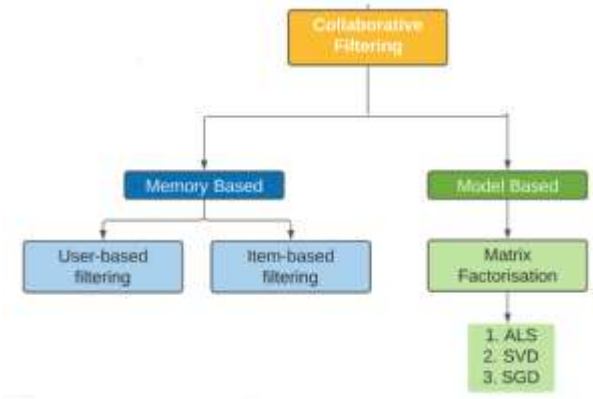
basis of what other ‘similar’ users have found useful. Basically, collaborative filtering is a recommendation system that creates a prediction based on a user’s previous behaviors (Ref: https://analyticsindiamag.com/collaborative-filtering-vs-content-based-filtering-for-recommender-systems/).

Way it works: An embedding or feature vector describing each item and user is built and it sinks both the items and the users in a similar embedding location. Then, it creates enclosures for items and users on its own. Other purchaser’s reactions are taken into consideration while suggesting a specific product to the primary user. It keeps track of the behavior of all users before recommending which item is mostly liked by users. It also relates similar users by similarity in preference and behavior towards a similar product when proposing a product to the primary customer. Two sources are used to record the interaction of a product user. First, through implicit feedback, User likes and dislikes are recorded and it is noticed by their actions like clicks, listening to music tracks, searches, purchase records, page views, etc. On the other hand, explicit feedback is when a customer specifies dislikes or likes by rating or reacting against any specific product on a scale of 1 to 5 stars. This is direct feedback from the users to show like and dislike about the product. It includes both positive and negative feedback. (Ref: https://www.iteratorshq.com/blog/collaborative-filtering-in-recommender-systems/)

Advantages: Serendipitous recommendations: Since CF uses similarities between users and items simultaneously to provide recommendations, it allows for serendipitous recommendations; that is, collaborative filtering models can recommend an item to user A based on the interests of a similar user B. **No domain knowledge necessary**: CF does not require a good amount of product features to work and also embeddings are automatically learned.

Disadvantage: **Cold start problem**: The prediction of the model for a given (user, item) pair is the dot product of the corresponding embeddings. So, if an item is not seen during training, the system can't create an embedding for it and can't query the model with this item.

Types of Collaborative Filtering: There are two types of the collaborative filtering process: **Memory-based collaborative filtering, and Model-based collaborative filtering (Fig.1.)**



Memory-based collaborative filtering: It calculates the similarity between users or items using the user’s previous data based on ranking. Neighbourhood approaches are most effective at

detecting very localized relationships (neighbours), ignoring other users. But the downsides are that, first, the data gets sparse which hinders scalability, and second, they perform poorly in terms of reducing the RMSE (root-mean-squared-error) compared to other complex methods.

Two methods: User-based Collaborative Filtering: It uses that logic and recommends items by finding similar users to the active user. Fills an user-item matrix and recommending based on the users more similar to the active user and it uses the user-based Nearest Neighbor algorithm. To recommend items to user u1 in the user-user based neighborhood approach first a set of users whose likes and dislikes similar to the user u1 is found using a similarity metrics which captures the intuition that sim(u1, u2) >sim(u1, u3) where user u1 and u2 are similar and user u1 and u3 are dissimilar. similar user is called the neighborhood of user u1.

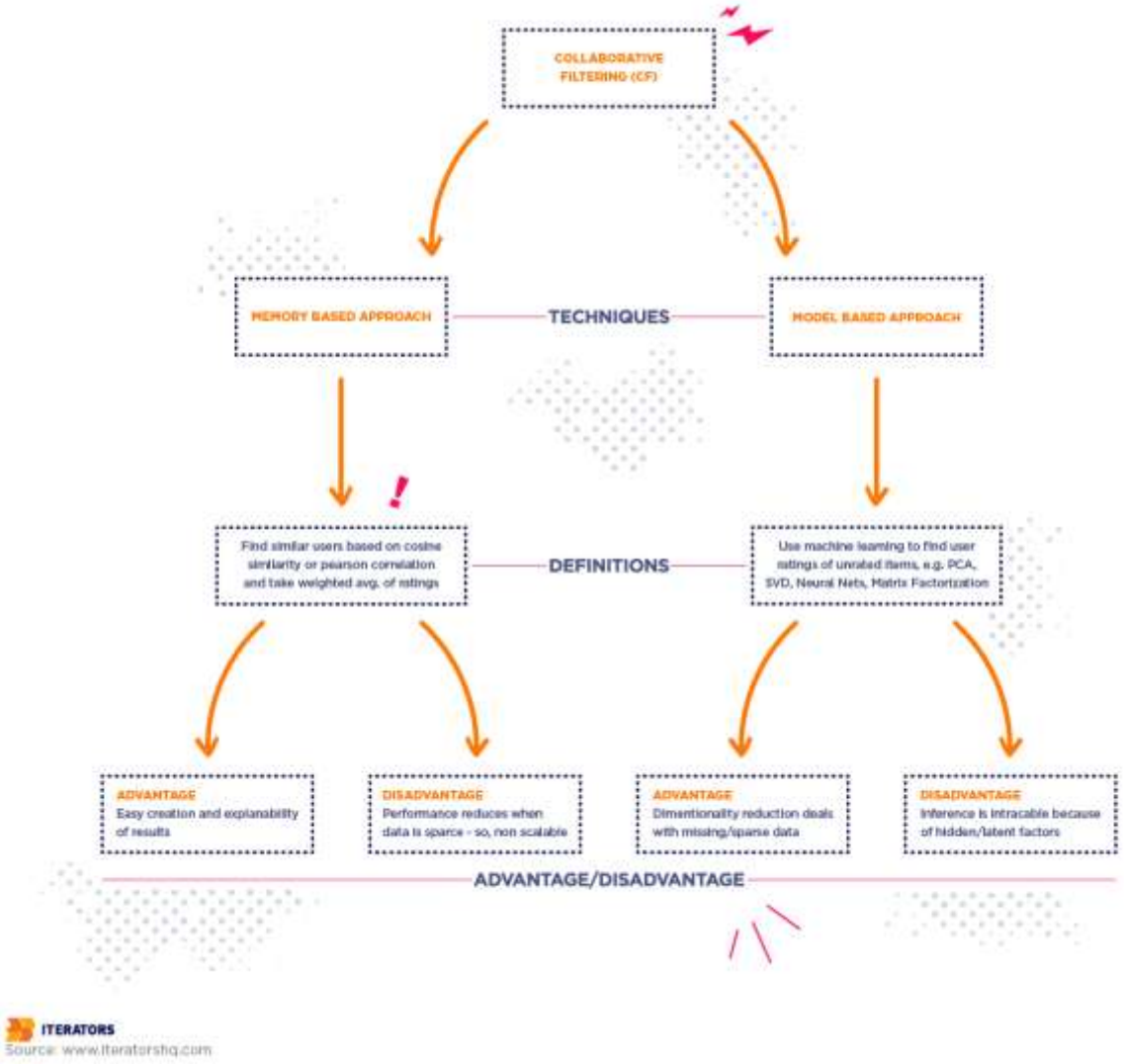
Item-based Collaborative Filtering: Finding the same items that the target user has already viewed. Fills an Item-Item matrix, and recommends based on similar items. which can be calculated using either Cosine-Based Similarity, Correlation-Based Similarity, Adjusted Cosine Similarity, or 1-Jaccard distance, and the prediction is estimated as a Weighted Sum or Regression.

Model-based Collaborative Filtering: Model-based collaborative filtering is not required to remember the based matrix. Instead, the machine models are used to forecast and calculate how a customer gives a rating to each product. These system algorithms are based on machine learning to predict unrated products by customer ratings. These algorithms are further divided into different subsets, i.e., Matrix factorization-based algorithms, deep learning methods, and clustering algorithms like K-Nearest Neighbor.

Model based collaborative filtering learns the (latent) user and item profiles (both of dimension K) through matrix factorization by minimizing the RMSE (Root Mean Square Error) between the available ratings yand their predicted values yˆ. Here each item i is associated with a latent (feature) vector xi, each user u is associated with a latent (profile) vector theta(u), and the rating yˆ(ui) is expressed as

$$\hat{y}_{ui} = \mu + b_u + b_i + \theta_u^\top \mathbf{x}_i$$

Fig 1: Overview of the 2 types of collaborative filtering



ITERATORS
Source: www.iteratorshq.com

System Configuration –Setup and configure the EMR cluster:

Hadoop cluster. Go to Amazon Elastic MapReduce (EMR) and follow the below steps to create a cluster. For now, leave the configuration as default:

Launch mode: Cluster
Vender: Amazon
Applications: Spark
Instance type: m4.xlarge
Number of instances: 3
Choose your EC2 key pair and click "Create cluster".

After the EMR Spark cluster is running, follow these steps:

1. Open the Security Groups for the Master node, and add 2 inbound rules for SSH (anywhere) and Custom TCP Port 8888 (anywhere)

2. SSH to the master node (note that the username is hadoop, not ubuntu).

3. run sudo python -m pip install Jupyter

4 After the installation of Jupyter, run these 2 lines:

```
export PYSPARK_DRIVER_PYTHON=/usr/local/bin/jupyter
export PYSPARK_DRIVER_PYTHON_OPTS="notebook --no-browser  --ip=0.0.0.0 --port=8888"
```

5. run source ~/.bashrc followed by pyspark

6. In the default browser copy and paste the EMR SSH connect and the token to start the Jupyter notebook . it should look like this : `http://ec2-XXX-XXX-XXX-XXX.compute-1.amazonaws.com:8888/tree?token=00011122233344455566677788899aaabbbcccdddeeefff#`

Problem 2: Analyzing the Netflix Data

Data exploration: Is performed in Jupyter notebook.

I. Load the datasets & convert into dataframes: Load and convert the 3 datasets to dataframe using Pandas read_csv() method.

1. movie_titles.txt containing the fields: 'movieId', 'year', & 'title',
2. TrainingRatings containing the fields: 'movieId', 'userId', 'rating'
3. TestingRatings containing the fields: 'movieId', 'userId', 'rating'

MovieId, userId are unique and rating are ratings given by each user to all the movieId. Title and year indicate the movie name and year of production.

II. Shape of dataframes: Get the shape of the datasets using Pandas shape property.

Shape of movies_title set: (17770, 3)
Shape of testing_rating set: (100478, 3)
Shape of training_rating df: (3255352, 3)

III, View Basic Statistics: Basic statistical details like percentile, mean, std etc. of a data frame or a series of numeric values is obtained using Pandas describe() method (Table 1, Table 2, Table 3).

Table 1: Describing the movie_titles data frame

	movieId	year
count	17770.000000	17770.000000
mean	8885.500000	1989.479685
std	5129.901477	42.804924
min	1.000000	0.000000
25%	4443.250000	1985.000000
50%	8885.500000	1997.000000
75%	13327.750000	2002.000000
max	17770.000000	2005.000000

Table 2: Describing the TrainingRatings data frame

	movieId	year	userId	rating
count	3.255352e+06	3.255352e+06	3.255352e+06	3.255352e+06
mean	8.724660e+03	1.991215e+03	1.327058e+06	3.481188e+00
std	5.107402e+03	1.289527e+01	7.626887e+05	1.082873e+00
min	8.000000e+00	1.916000e+03	7.000000e+00	1.000000e+00
25%	3.893000e+03	1.986000e+03	6.716970e+05	3.000000e+00
50%	8.825000e+03	1.995000e+03	1.322467e+06	4.000000e+00
75%	1.332600e+04	2.001000e+03	1.988873e+06	4.000000e+00
max	1.774200e+04	2.005000e+03	2.649285e+06	5.000000e+00

Table 3: Describing the TestingRatings data frame

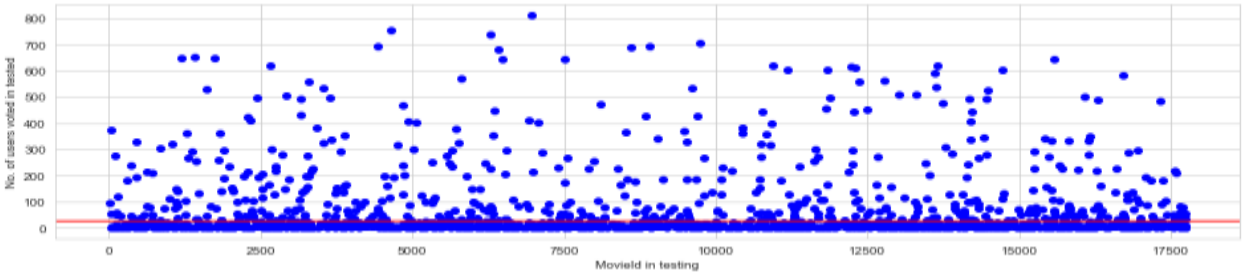
	movieId	year	userId	rating
count	100478.000000	100478.000000	1.004780e+05	100478.000000
mean	8701.547792	1991.242829	1.329956e+06	3.479458
std	5098.075495	12.839269	7.625041e+05	1.085280
min	8.000000	1916.000000	7.000000e+00	1.000000
25%	3893.000000	1986.000000	6.774300e+05	3.000000
50%	8699.000000	1995.000000	1.325031e+06	4.000000
75%	13298.000000	2001.000000	1.995052e+06	4.000000
max	17742.000000	2005.000000	2.649285e+06	5.000000

IV. Check for duplicates: Pandas duplicated() method helps in analyzing if there are any duplicate values and it is observed that there are no duplicate values in any of the 3 datasets.

V. Data Visualization: Capturing stories behind the data: For this part only the testing dataset is used for visualization and all graphs are plotted using matplotlib.

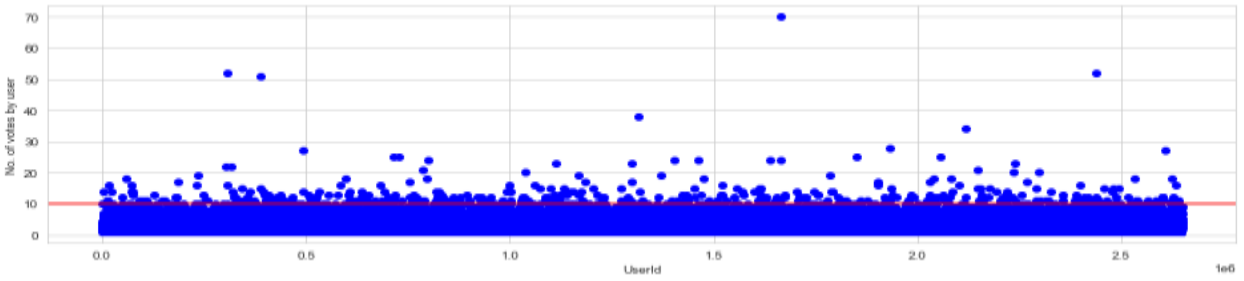
Plotting the number of users who voted and the number of movies that were voted from Fig. 2 it is observed that some movies are rated by many users, meanwhile, many of movies are rated only by a few users. The red line indicates a minimum of 10 users who rated a movie.

Fig 2: The number of users in testing set rating per movie.



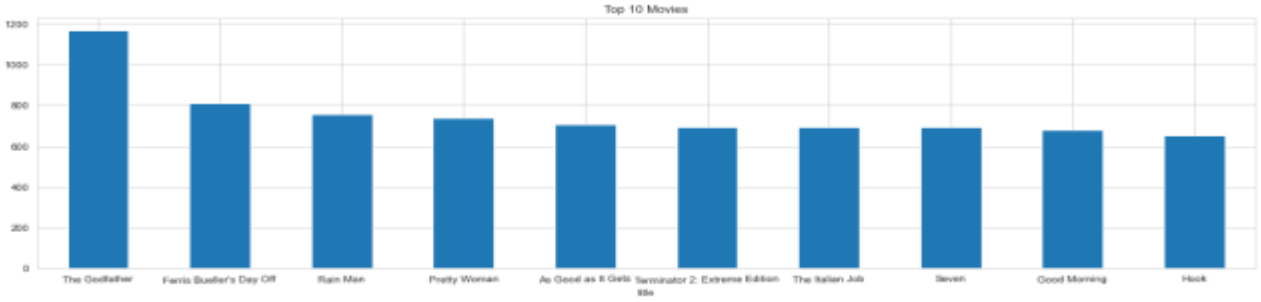
Plotting the number of movies by each user rated for all the users in the testing set, it is observed from as Fig. 3 informs that many users rated only around 10 movies.

Fig. 3: Number of movie ratings by each user in the testing set



Plotting the number of ratings for each movie, Fig, 4, shows top 10 movies based on rating received from the users in the testing set and it is seen that observed that popular movies like The Godfather received more votes from users in the testing data set than other movies.

Fig 4: Top 10 movie titles based on number of users rating the movies in the testing set



Plotting the number of movies in each rating category from Fig 5 it is observed that more movies get rated in the 4-rating category and less than 10 in the rating 1 category and plotting the number of ratings for the first 25 movies in the testing data set indicates that popular movies receive more ratings than other movies.

STORY CAPTURED: Based on the above data visualizations it is observed that there is a trend to rate more popular movies leading to the issue of **POPULARITY BIAS.**

Fig. 5: Number of movies in each rating category

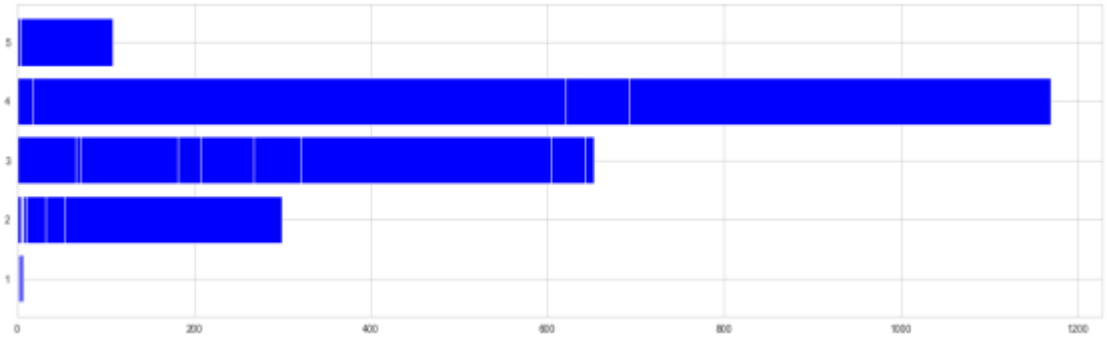
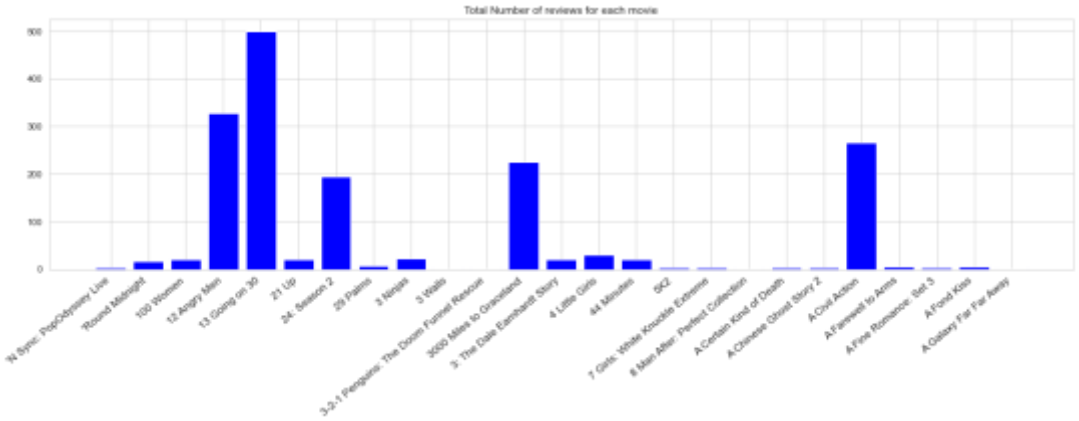
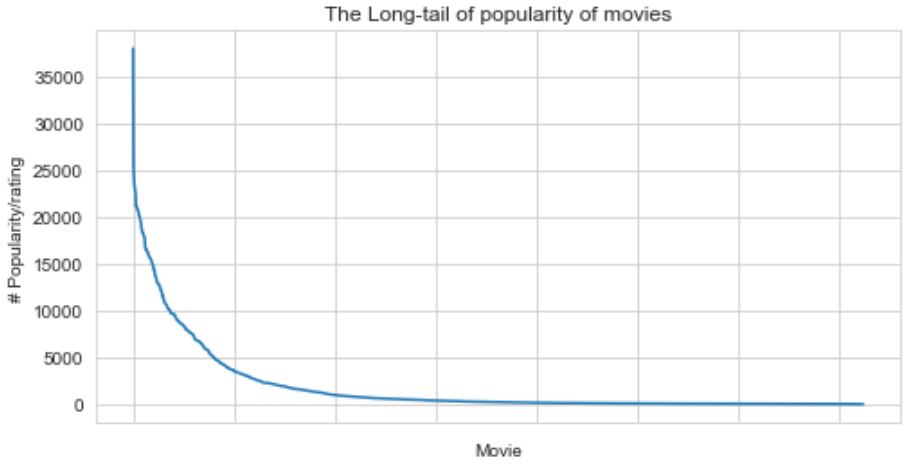


Fig 6: Number of reviews for first 25 movies



Plotting the Long-tail graph shows the distribution of ratings or popularity among movies (Fig.7) it is observed that movies on left side are called as popular because their popularity is higher than those in the long-tail area. The items in the right side of graph are less rated than the those in left side. This means that there are much more sparsity or unobserved areas for those items in ratings matrix.

Fig, 7. Long -Tail of ratings per movie in the training data



Measuring user similarities are measured by overlapping items & item similarities are measured by overlapping users: All below analysis were implemented in SPARK in Jupyter Notebooks running on AWS EMR. For details on how to run Spark in Jupyter Notebooks running on AWS EMR refer to Sparkify lab 6.

a. How many distinct items & how many distinct users are there in the test set (TestingRatings.txt)?

Using the countDistinct() method of Pandas, it is observed that there are 1701 distinct items and 27555 users in the testing set.

```
+------------------------+
|count(DISTINCT movie_Id)|
+------------------------+
|                    1701|
+------------------------+

+----------------------+
|count(DISTINCT userId)|
+----------------------+
|                 27555|
+----------------------+
```

b. Which is higher average overlap of items rated by the users in the training set for users in the test set or average overlap of users that rated items in the training set for items appearing in the test set?

Use pyspark.sql.functions import col, count, mean, and import pyspark.sql.functions as F to call the pyspark functions for counting, for calculating mean, and for filtering by specific column value.

The average overlap of items rated by the users in the training set for a user in the test set is calculated by picking a user (userId=79) and count the number of movies the user rated in the training set using the groupby() and count() methods. It is seen that User 79 from testing set has rated 84 movies in the training set. Then, get the movies User 79 has rated in the training set as a movies list using the flatmap() method as shown below.

```
user_data = (training_df[training_df.userId == '79'])
distinct_movies =user_data.select("title").rdd.flatMap(lambda x: x).collect()
print(distinct_movies)
```

```
['The Phantom of the Opera: Special Edition', 'Stir Crazy', 'Back to the Future Part III', 'Uptown Girls', "National Lampoo
n's Vacation", 'Hook', 'No Way Out', 'Belly of the Beast', 'The American President', 'Beverly Hills Cop', 'Undertow', 'MVP:
```

After getting the distinct movie list to compute how many other users in the training set have rated the same movies list use the filter() method to get all the users who rated the movies on the distinct movie list.

```
user_training=training_df.filter(F.col("title").isin(distinct_movies))
print(user_training.show(3))
```

```
+--------+----+--------------------+-------+------+
|movie_Id|year|               title| userId|rating|
+--------+----+--------------------+-------+------+
|     361|2004|The Phantom of th...|2569320|   4.0|
|     361|2004|The Phantom of th...|1987434|   2.0|
|     361|2004|The Phantom of th...| 946314|   2.0|
```

Count the number of users and calculate the mean to get the average overlap of items rated by the users in the training set for a user in the test set, which was 35.11.

```
count_movie=user_training.groupBy("userId").agg(F.count('title').alias("cntMovie"))
count_movie.show(3)
user_training.groupBy("userId").count().select(avg("count")).show()
```

```
+-------+--------+
| userId|cntMovie|
+-------+--------+
| 953170|      45|
| 279120|      32|
|1553158|      34|
+-------+--------+
only showing top 3 rows

[Stage 168:====================================>                    (5 + 3) / 8]

+----------------+
|      avg(count)|
+----------------+
|35.1158465042446|
+----------------+
```

To calculate the average overlap of users that rated items in the training set for items appearing in the test set use the same above process but now instead of a user, pick a movieId. For this analysis, the movieID chosen was 8 from test set and extract all the users that has rated this movieId in the training set and it is observed that 2381 users has rated the movieId 8 in the training set. Get all the users as distinct user list and then see how many movies these distinct users rated in the training set. Get the average user count for movies that were rated by the users who also rated movieId: 8 (What the #$*! Do ...) by calling the mean() method and it is seen that Estimated average overlap of user for items 127.9

```
count_item=item_training.groupBy("title").agg(F.count('userId').alias("cntUser"))
count_item.show(3)
item_training.groupBy("title").count().select(avg("count")).show()
```

```
+--------------------+-------+
|               title|cntUser|
+--------------------+-------+
|Spike & Mike's Cu...|     11|
|But I'm a Cheerle...|    787|
|Horrors of Spider...|     13|
+--------------------+-------+
only showing top 3 rows

+------------------+
|       avg(count)|
+------------------+
|200.05414364640885|
```

From the calculations above it is answered that estimated average overlap of user for items is higher than the estimated average overlap of items rated by the users.

The Model Selection: Explanation and Justification

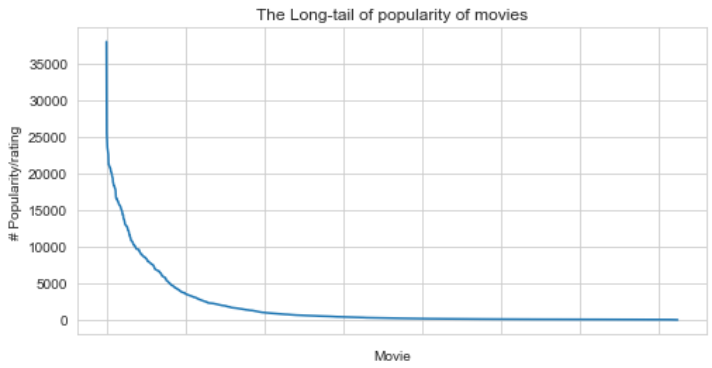
Choices: Memory based User or Item Based OR Model CF.

Paraments: Efficiency and prediction quality

Memory based User or Item Based: In the case where there are more users than items in the dataset which is generally the case in the real world, and in the current Netflix dataset as seen from the distinct count analysis (1701 distinct movies and 27555 users in the testing set), it would be effective to use item-item collaborative filtering. Additionally, the user preferences and ratings changes over time so it is difficult to tackle this problem in user-user collaborative filtering but with item generally it is seen the rating of item doesn't change much over a course of time. Item-item similarities remain more static over time, so based on this the obvious choice will be Item-item based collaborative filtering.

Short-comings in choosing Item based CF model: Three major limitations for efficiency and prediction quality will be 1. Popularity Bias, 2. Cold-Sart, and 3. Sparsity:

Popularity Bias: The issue of Popularity Bias arises when only a small number of items receive an extremely high number of interactions and due to the presence of unpopular items or items that only receive a fraction of interactions when compared to the highly popular ones as seen from Fig. 7 as shown again below:



Sparsity: Movies in the right side of graph are less rated than the those in left side. This means that there are much more sparsity or unobserved areas for those movies in ratings matrix. This can cause a recommender system which relies on neighborhood algorithms produce bad results. The more we move the threshold to right side, The worse item-based recommendation system results.

Also, the for Sparsity for Training & Testing datasets for Ratings is calculated using the formula:

$$sparsity = 1 - \frac{count\_nonzero(A)}{total\_elements\_of\_A}$$

SPARSITY CALULATION IN TRAINING & TESTING SETS: It is observed that the ratings in training set is 93.83% empty, and the ratings in testing set is 99.79% empty.

Cold Start: refers to when movies added to the catalogue have either none or very little interactions while recommender rely on the movie's interactions to make recommendations. A item based collaborative algorithm cannot recommend the item if there are no interactions are available, and the quality of recommendations will be very poor in the case of availability of few interactions.

Scalability Issue: Another additionally issue faced while using item based KNN model is scalability due to its time complexity is O(nd + kn), where n is the cardinality of the training set and d the dimension of each sample. KNN takes more time in making inference than training, thus increasing the prediction latency.

Solution: With such a sparse matrix, it becomes essential to choose an CF that can effectively solving a data sparsity problem. In collaborative filtering, matrix factorization is the state-of-the-art solution for sparse data problem.

Matrix factorization: A matrix factorization is a factorization of a matrix into a product of matrices. In the case of CF, matrix factorization algorithms work by decomposing the user-item interaction matrix into the product of two lower dimensionality rectangular matrices. One matrix can be seen as the user matrix where rows represent users and columns are latent factors. The other matrix is the item matrix where rows are latent factors and columns represent items.

How does matrix factorization solve sparsity problem? Model learns to factorize rating matrix into user and movie representations, which allows model to predict better personalized movie ratings for users. With matrix factorization, less-known movies can have rich latent representations as much as popular movies have, which improves recommender's ability to recommend less-known movies. In the sparse user-item interaction matrix, the predicted rating user u will give item i is computed as:

$$\tilde{r}_{ui} = \sum_{f=0}^{nfactors} H_{u,f} W_{f,i}$$

Rating of item i given by user u can be expressed as a dot product of the user latent vector and the item latent vector. Notice in above formula, the number of latent factors can be tuned via cross-validation. Latent factors are the features in the lower dimension latent space projected from user-item interaction matrix. The idea behind matrix factorization is to use latent factors to represent user preferences or movie topics in a much lower dimension space. Matrix factorization is one of very effective dimension reduction techniques in machine learning. A matrix factorization with one latent factor is equivalent to a most popular or top popular recommender (e.g. recommends the items with the most interactions without any personalization). Increasing the number of latent factors will improve personalization, until the number of factors becomes too high, at which point the model starts to overfit. A common strategy to avoid overfitting is to add regularization terms to the objective function (Ref: https://towardsdatascience.com/prototyping-a-recommender-system-step-by-step-part-2-alternating-least-square-als-matrix-4a76c58714a1).

The objective of matrix factorization is to minimize the error between true rating and predicted rating:

$$\arg\min_{H,W} \|R - \tilde{R}\|_{\mathrm{F}} + \alpha\|H\| + \beta\|W\|$$

So, to solve the issues, choose a matrix factorization algorithm.

CHOICE CF MODEL: Alternating Least Square (ALS) with Spark ML

Alternating Least Square (ALS) is a matrix factorization algorithm and it runs itself in a parallel fashion. ALS is implemented in Apache Spark ML and built for a large-scale collaborative filtering problem. ALS is doing a pretty good job at solving scalability and sparseness of the Ratings data, and it's simple and scales well to very large datasets.

Why ALS with Spark ML: ALS uses L2 regularization and minimizes two loss functions alternatively; It first holds user matrix fixed and runs gradient descent with item matrix; then it holds item matrix fixed and runs gradient descent with user matrix. Its scalability: ALS runs its gradient descent in parallel across multiple partitions of the underlying training data from a cluster of machines

Problem 3: Collaborative Filtering Implementation

Step 1: Implementation: The instantiation of an ALS model, running hyperparameter tuning, cross validation and fitting the model is all implementation in Jupyter Notebook using PySPARK on an AWS EMR cluster.

Step 2: Execution and Evaluation: **ALS model in Spark**: spark.ml supports model-based collaborative filtering, in which users and products are described by a small set of latent factors that can be used to predict missing entries. spark.ml uses the alternating least squares (ALS) algorithm to learn these latent factors (more info: https://spark.apache.org/docs/2.2.0/ml-collaborative-filtering.html & https://s3.amazonaws.com/assets.datacamp.com/production/course_7049/slides/chapter3.pdf)

1. Build the ALS model:

To build the model first specify the columns. Set nonnegative as 'True', since the rating is greater than 0. The model also gives an option to select implicit ratings. Since the current work is with explicit ratings, set it to 'False'. By default, Spark assigns NaN predictions during ALSModel.transform when a user and/or item factor is not present in the model, so set cold start strategy to 'drop' to ensure that not to get NaN evaluation metrics.

```
# Import the required functions
from pyspark.ml.evaluation import RegressionEvaluator
from pyspark.ml.recommendation import ALS
from pyspark.ml.tuning import ParamGridBuilder, CrossValidator
```

```
# Create ALS model
als = ALS(userCol="userId", itemCol="movie_Id",
          ratingCol="rating", nonnegative = True, implicitPrefs = False, coldStartStrategy="drop")
type(als)
```

```
pyspark.ml.recommendation.ALS
```

2. Hyperparameter tuning and cross validation: Most important hyper-params in Alternating Least Square (ALS) are:

maxIter: the maximum number of iterations to run (defaults to 10)
rank: the number of latent factors in the model (defaults to 10)
regParam: the regularization parameter in ALS (defaults to 1.0)

ParamGridBuilder: Define the tuning parameter using param_grid function. For this analysis only 3 parameters for each grid were chosen. This will result in 9 models for training.

```
from pyspark.ml.evaluation import RegressionEvaluator
from pyspark.ml.tuning import ParamGridBuilder, CrossValidator

# Add hyperparameters and their respective values to param_grid
param_grid = ParamGridBuilder() \
            .addGrid(als.rank, [5,10,15]) \
            .addGrid(als.regParam, [.01, .05, 0.1]) \
            .build()
```

CrossValidator: Feed both param_grid and evaluator into the build in crossvalidator included in the ALS model. For this analysis the number of folds chosen was 5.

```
# Cross validation using CrossValidator
cv = CrossValidator(estimator=als, estimatorParamMaps=param_grid, evaluator=evaluator, numFolds=5)
print(cv)
```

```
CrossValidator_ccea8fa97ab4
```

RegressionEvaluator: Define the evaluator, select rmse as metricName in evaluator since it allows to set only metricName.

```
# Define evaluator as RMSE and print length of evaluator
evaluator = RegressionEvaluator(metricName="rmse", labelCol="rating", predictionCol="prediction")
print ("Num models to be tested: ", len(param_grid))
```

```
Num models to be tested:  9
```

3. Check the best model parameters: Checkout which parameters out of the 9 parameters fed into the crossvalidator resulted in the best model. For this analysis rank 15, maxIter 10, and regParam 0.05 was the ALS parameters for the best model.

```
# Print best_model &  ALS model parameters
print(type(best_model))
print("**Best Model**")
print("  Rank:", best_model._java_obj.parent().getRank())
print("  MaxIter:", best_model._java_obj.parent().getMaxIter())
print("  RegParam:", best_model._java_obj.parent().getRegParam())
```

```
<class 'pyspark.ml.recommendation.ALSModel'>
**Best Model**
  Rank: 15
  MaxIter: 10
  RegParam: 0.05
```

4. Fit the best model and evaluate predictions: Fit the model and make predictions on both train and test dataset. Based on the range of parameters chosen 9 models will be tested.

```
# View the predictions
test_predictions = best_model.transform(testing_df)
RMSE = evaluator.evaluate(test_predictions)
print("RMSE for testing data is:", RMSE)

mae = evaluator.evaluate(test_predictions, {evaluator.metricName: "mae"})
print("MAE for testing data is:: %.3f" % mae)
```

```
RMSE for testing data is: 0.8371780505884548
[Stage 8366:=========================================>          (160 + 8) / 200]
MAE for testing data is:: 0.660
```

The RMSE for the best model is 0.837 and MAE is 0.660 for test set which means that on average the model predicts 0.837 above or below values of the original ratings matrix.

5. Make Recommendations: Generate recommendations based on the best model for all users using the recommendForAllUsers(n) function in als takes n recommendations. For this analysis 5 recommendations for all users listed.

```
#  Recommendations for all users and list top 10
reco = best_model.recommendForAllUsers(10)
reco.limit(10).show()
```

```
[Stage 4223:=================================================> (98 + 2) / 100]
+------+--------------------+
|userId|     recommendations|
+------+--------------------+
|   481|[[6991, 5.7360873...|
|  2678|[[12125, 4.468851...|
|  3595|[[12293, 4.680737...|
|  6460|[[12232, 4.913296...|
|  7284|[[12544, 5.472037...|
|  7576|[[15557, 5.109460...|
|  9597|[[12952, 4.429915...|
| 15191|[[14283, 4.873109...|
| 15846|[[11284, 5.180236...|
| 20461|[[2939, 5.0058455...|
+------+--------------------+
```

6. Convert recommendations into interpretable format: The recommendations are generated in a format that easy to use in pyspark. As seen in the above the output, the recommendations are saved in an array format with movie id and ratings. To make these recommendations easy to read and compare to check if recommendations make sense, add more information like movie name and then explode array to get rows with single recommendations.

```
from pyspark.sql.functions import UserDefinedFunction, explode, desc
from pyspark.sql.functions import col, avg, when, count

reco1 = reco\
    .withColumn("rec_exp", explode("recommendations"))\
    .select('userId', col("rec_exp.movie_Id"), col("rec_exp.rating"))

reco1.limit(10).show()
```

```
[Stage 8527:>                                                       (91 + 8) / 100]
+------+--------+---------+
|userId|movie_Id|   rating|
+------+--------+---------+
|   481|    6991| 5.736088|
|   481|   14648| 5.140416|
|   481|   10743| 5.124085|
|   481|   14361| 5.055481|
|   481|    7569|5.0513654|
|   481|   12952|5.0289598|
|   481|     634| 4.975964|
|   481|    4238|4.9639053|
|   481|   15567|  4.94801|
|   481|   10947| 4.940893|
+------+--------+---------+
```

```
#view the recommendation with movie titles
reco1.join(movies_titles_df, on='movie_Id').show(10)
```

```
+--------+------+---------+----+--------------------+
|movie_Id|userId|   rating|year|               title|
+--------+------+---------+----+--------------------+
|    6991|   481|5.7360873|2001|    A History of God|
|   14648|   481| 5.140416|2003|Finding Nemo (Ful...|
|   10743|   481| 5.124085|2001|Pearl Jam: Tourin...|
|   14361|   481| 5.055481|1999|SpongeBob SquareP...|
|    7569|   481| 5.051366|2004|Dead Like Me: Sea...|
|   12952|   481|5.0289598|2005|The God Who Wasn'...|
|     634|   481| 4.975964|1989|Christmas with Th...|
|    4238|   481|4.9639053|2000|            Inu-Yasha|
|   15567|   481|  4.94801|1987|Grateful Dead: Ti...|
|   10947|   481| 4.940893|2004|     The Incredibles|
+--------+------+---------+----+--------------------+
only showing top 10 rows
```

6. Do the recommendations make sense? To check if the recommendations make sense, join movie name to the above table and randomly pick a user to check if the recommendations make sense. It is seen that the movies actually preferred by the userId 79 match very closely with the one's predicted by the ALS model.

UserId: 79 ALS Recommendations:

```
recomendations = model.transform(single_user)
reco_user_79 =recomendations.orderBy('prediction',ascending=False
reco_user_79.join(movies_titles_df, on='movie_Id').show(10)
```

```
+--------+------+------+----------+----+--------------------+
|movie_Id|rating|userId|prediction|year|               title|
+--------+------+------+----------+----+--------------------+
|   14648|   5.0|    79| 4.4529114|2003|Finding Nemo (Ful...|
|    2913|   4.0|    79| 3.9445221|2004|   Finding Neverland|
|   12497|   4.0|    79| 3.5837138|2000|         Bring It On|
|    8163|   3.0|    79|  3.225082|2004|        Two Brothers|
+--------+------+------+----------+----+--------------------+
```

UserId: 79 Actual Preference:

```
#Recommending Movies with ALS for the user 79
single_user = testing_df.filter(testing_df['userId']==79).select(['movie_Id','rating','userId'])
single_user.join(movies_titles_df, on='movie_Id').show()
```

```
+--------+------+------+----+--------------------+
|movie_Id|rating|userId|year|               title|
+--------+------+------+----+--------------------+
|    2913|   4.0|    79|2004|   Finding Neverland|
|    8163|   3.0|    79|2004|        Two Brothers|
|   12497|   4.0|    79|2000|         Bring It On|
|   14648|   5.0|    79|2003|Finding Nemo (Ful...|
+--------+------+------+----+--------------------+
```

Step 3: Does your approach work for your own preferences?

I added myself as a new user to the training data set by creating a new unique user ID. Selected some movies that I have seen among those in the training set, and add the ratings for those. The new unique id added was “11111”, and 10 movie_Id’s with their rating were created as a dataframe along with the userId (“11111”). Then this data frame was joined to the training data set and the model is allowed to train on the added training dataset with my ratings. Then check the RMSE/MAE for the New Model with your Ratings.

```
RMSE for testing data with model trained on added data is: 0.8375366920102414
[Stage 11540:=====================================>                 (141 + 8) / 200]
MAE for testing data with model trained on added data is:: 0.661
```

The RMSE for the best model is 0.837 and MAE is 0.661 for test set which means that on average the model predicts 0.837 above or below values of the original ratings matrix.

Predict recommendations for yourself: Use the model to predict recommendations for myself:

```
# recommendations for me
reco1_data.join(movies_titles_df, on='movie_Id').filter('userId = 11111').sort(desc('rating')).show(5)
```

```
[Stage 7700:============>(94 + 6) / 100][Stage 7701:>              (0 + 1) / 1]
```

```
+--------+------+---------+----+--------------------+
|movie_Id|userId|   rating|year|               title|
+--------+------+---------+----+--------------------+
|    2858| 11111|4.9714956|2000|Bounce: Bonus Mat...|
|    1207| 11111|4.9714956|1962|Experiment in Terror|
|    1172| 11111|4.9714956|1998| Krippendorf's Tribe|
|    2571| 11111|3.9771967|2002|Woodrow Wilson: A...|
|    1198| 11111|3.9771967|1971|The Cat O'Nine Tails|
+--------+------+---------+----+--------------------+
only showing top 5 rows
```

My actual preference in decreasing order of rating

```
# Movies for myself in the training data
single_user_data= (training_with_my_ratings_df[training_with_my_ratings_df.userId == '11111'])
single_user_data.join(movies_titles_df, on='movie_Id').filter('userId = 11111').sort(desc('rating')).show(5)
```

```
+--------+------+------+----+--------------------+
|movie_Id|userId|rating|year|               title|
+--------+------+------+----+--------------------+
|    2858| 11111|   5.0|2000|Bounce: Bonus Mat...|
|    1207| 11111|   5.0|1962|Experiment in Terror|
|    1172| 11111|   5.0|1998| Krippendorf's Tribe|
|    2571| 11111|   4.0|2002|Woodrow Wilson: A...|
|    1198| 11111|   4.0|1971|The Cat O'Nine Tails|
+--------+------+------+----+--------------------+
only showing top 5 rows
```

As you can see from the above, the model is able to accurately predict my preferences.

Some of the top ranked movies recommended by the ALS that I haven't seen

```
# recomendations_data.join(movies_titles_df, on='movie_Id').show(5)
array = [2959,2571,1207,296, 2858, 1172, 593, 745, 1198, 6016]
top_reco =reco1_data.join(movies_titles_df, on='movie_Id').filter('userId = 11111').sort(desc('rating'))
top_reco_yr =top_reco.filter(top_reco.movie_Id.isin(array) == False)
top_reco_yr.orderBy('rating',ascending=True).show()
```

```
+--------+------+---------+----+--------------------+
|movie_Id|userId|   rating|year|               title|
+--------+------+---------+----+--------------------+
|    3941| 11111|1.6965362|1975|The French Connec...|
|   13878| 11111|1.7531966|1975|Sanford and Son: ...|
|    5369| 11111| 1.787698|1972|Sanford and Son: ...|
+--------+------+---------+----+--------------------+
```

Final conclusion/lessons learned/future work: Learnt how to use collaborative filtering recommender system with matrix factorization to solve the issues of popular bias and item cold-start problems. Learnt to build a recommendation system for given Netflix data by leveraged Spark ML to implement distributed recommender system using Alternating Least Square (ALS).