University of New Haven
TAGLIATELA COLLEGE OF ENGINEERING
Electrical & Computer Engineering and Computer Science

Data Science (DS) - 6007

Distributed and Scalable Data Engineering

Final Project
Technical Report

SPRING 22

CONTENTS

NETFLIX
Movie Recommendation Using Collaborative Filtering

Team Members
Krishna Rohit Donepudi
Prudhvi Rao Sidduri
Pushpa Latha Vudatha

Contact :
kdone1@unh.newhaven.edu
psidd2@unh.newhaven.edu
pvuda1@unh.newhaven.edu

Group 5
Submitted On: 5/3/2022

Introduction

The rapid growth of data collection has led to a new era of information. Data is being used to create more efficient systems and this is where Recommendation Systems come into play. Recommendation Systems are a type of information filtering system as they improve the quality of search results and provide items that are more relevant to the search item or are related to the search history of the user.

'Recommender System is a system that seeks to predict or filter preferences according to the user's choices.

Recommender systems are utilized in a variety of areas including movies, music, news, books, research articles, search queries, social tags, and products in general. Moreover, companies

like Netflix and Spotify depend highly on the effectiveness of their recommendation engines for their business and success.

Introduction

The good examples of recommendation systems are:

Product Recommendations:

The most important use of recommendation systems is at online retailers. We have noted how Amazon or similar online vendors strive to present each returning user with some suggestions about what they might like to buy These suggestions are not random, but are based on the purchasing decisions made by similar customers or on their history of purchases or other techniques

Movie Recommendations:

Netflix offers its customers recommendations of movies, they might like. These recommendations are based on ratings provided by users.

News Articles:

News services have attempted to identify articles of interest to readers, based on the articles that they have read in the past. The similarity might be based on the similarity of important words in the documents, or on the articles that are read by people with similar reading tastes.

Motivation

Netflix is all about connecting people to the movies they love. To help customers find those movies, they developed a world-class movie recommendation system. Its job is to predict whether someone will enjoy a movie based on how much they liked or disliked other movies. Netflix uses those predictions to make personal movie recommendations based on each customer's unique tastes. And while Netflix's Recommendation Engine is doing well, it can always be made better.

Whenever a user accesses the Netflix service, Netflix recommendations system strives to help a user find a show or movie to enjoy with minimal effort. We estimate the likelihood that a user will watch a particular title in Netflix catalog based on several factors including:

- User interactions with Netflix service (such as user viewing history and how a user rated other titles),
- other members with similar tastes and preferences on our service, and
- information about the titles, such as their genre, categories, actors, release year, etc.

In addition to knowing what a user has watched on Netflix, to best personalize the recommendations Netflix also looks at things like:

- the time of day a user watch,
- the devices a user is watching Netflix on, and
- how long a user watches.

The recommendations system does not include demographic information (such as age or gender) as part of the decision-making process.

Now there are a lot of interesting alternative approaches to a recommendation system works that Netflix hasn't tried. In this project, we are going to study and dig into one such algorithm.

Approach

Recommendation systems use several different technologies. We can classify these systems into two broad groups.

1. **Content-based systems** - examine the properties of the items recommended. It uses a series of discrete characteristics of an item to recommend additional items with similar properties. For instance, if a Netflix user has watched many cowboy movies, then recommend a movie classified in the database as having the "cowboy" genre.

2. **Collaborative filtering systems** - recommend items based on similarity measures between users and/or items. The items recommended to a user are those preferred by similar users.

In this project we are using the most popular recommendation system, ie Collaborative Filter recommendation.

Data Analysis

This dataset is a subset of the data provided as part of the Netflix Prize. TrainingRatings.txt and TestingRatings.txt are respectively the training set and test set. Each of them has lines having the format: MovieID, UserID, Rating.

Each row represents a rating of a movie by some user. The dataset contains 1821 movies and 28978 users in all. Ratings are integers from 1 to 5. The training set has 3.25 million ratings, and the test set has 100,000. The movies data has 17769 unique movies.

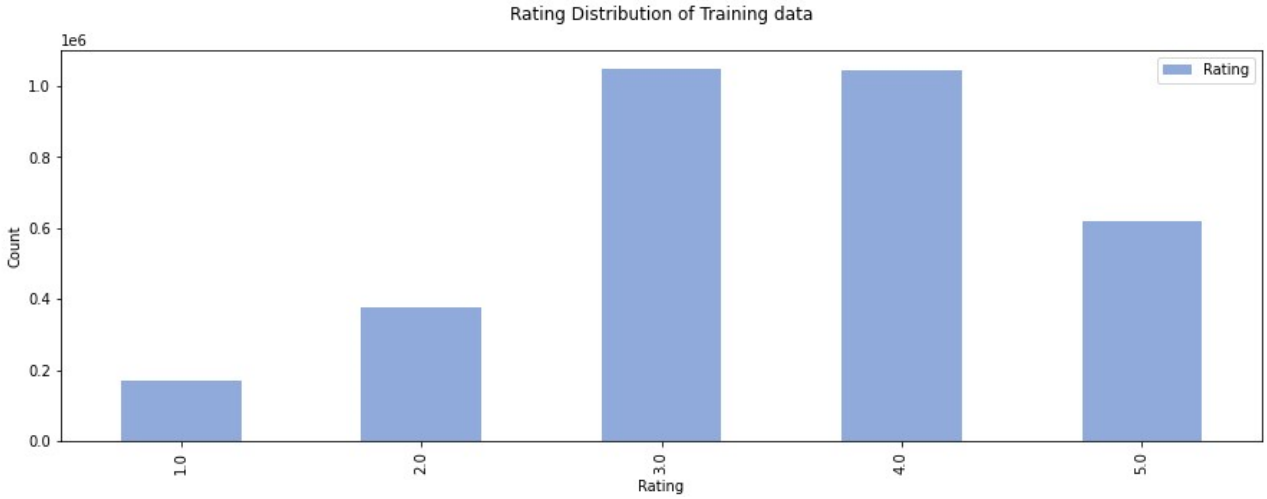
```
print("Training data set Size : ", (trainingRatings_df.count(), len(trainingRatings_df.columns)))
print("Testing data set Size : ", (testingRatings_df.count(), len(testingRatings_df.columns)))
print("Movies data set Size : ", (movieTitles_df.count(), len(movieTitles_df.columns)))
```

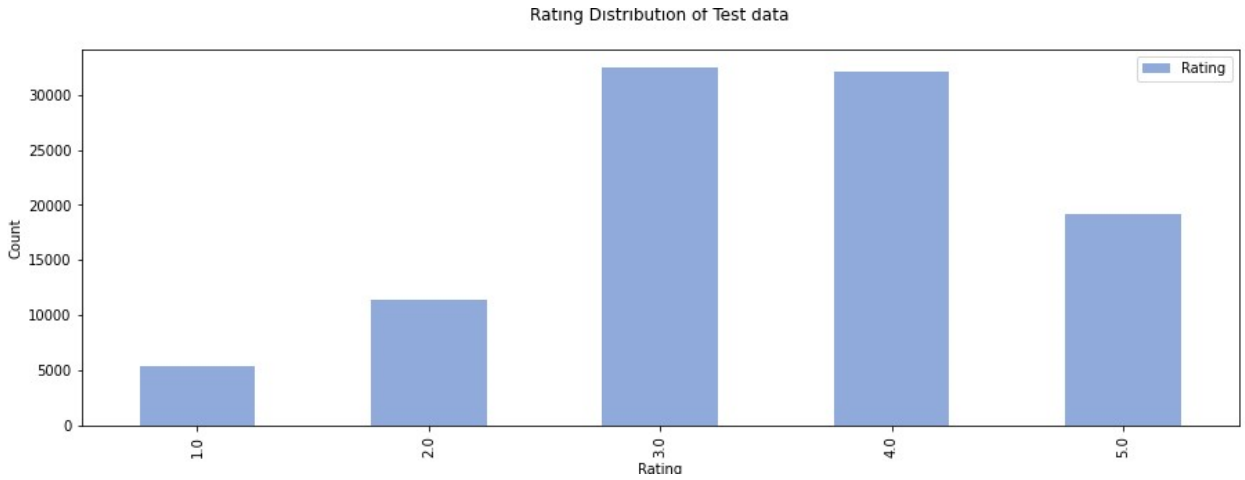
```
Training data set Size :  (3255351, 3)
Testing data set Size :  (100477, 3)
Movies data set Size :  (17769, 2)
```

```
+-------+-----------------------------+
|MovieID|Title                        |
+-------+-----------------------------+
|2      |Isle of Man TT 2004 Review   |
|3      |Character                    |
|4      |Paula Abdul's Get Up & Dance|
+-------+-----------------------------+
only showing top 3 rows

Training_Ratings list
+-------+-------+------+
|MovieID|UserID |Rating|
+-------+-------+------+
|8      |1395430|2.0   |
|8      |1205593|4.0   |
|8      |1488844|4.0   |
+-------+-------+------+
only showing top 3 rows

Testing_Ratings list
+-------+-------+------+
|MovieID|UserID |Rating|
+-------+-------+------+
|8      |2149668|3.0   |
|8      |1089184|3.0   |
|8      |2465894|3.0   |
+-------+-------+------+
only showing top 3 rows
```

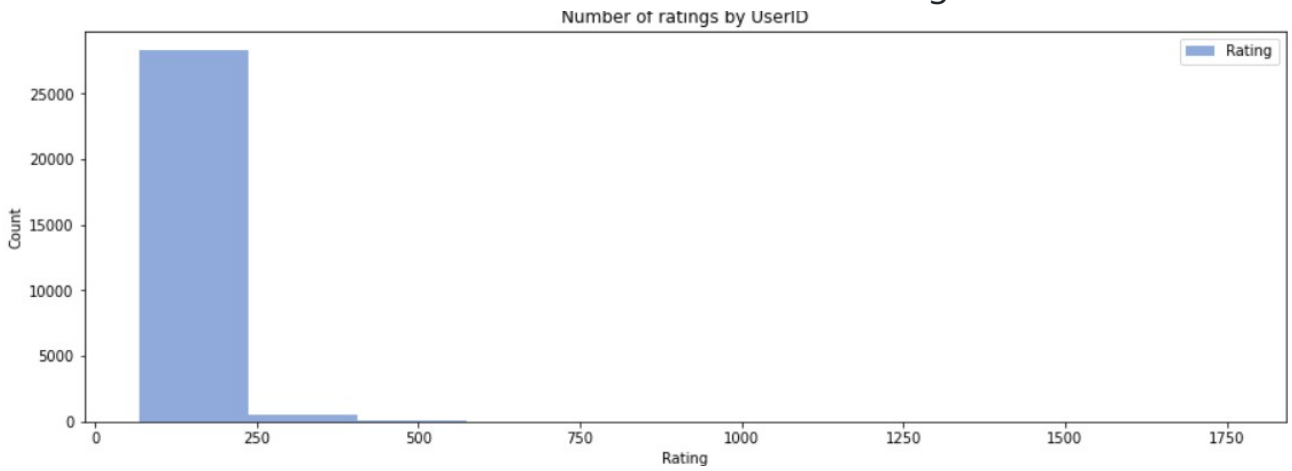
Distribution of Data

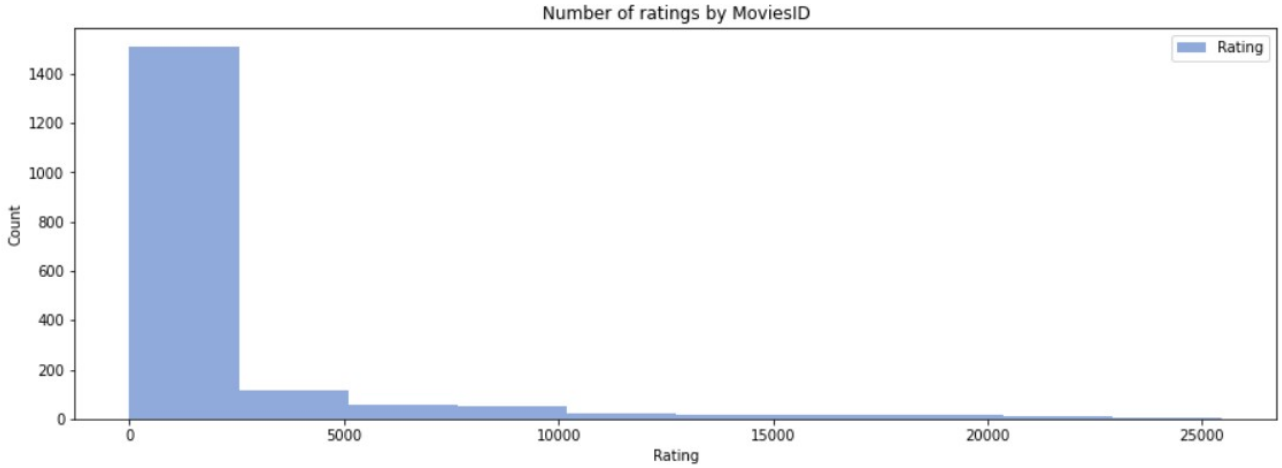
We'll start by understanding what the most frequent rating evaluations in the ratings dataset are and how are they distributed. The evaluation consists of 5 possible values between 1 and 5 (1 step increasing). We can see that 75% of the reviews are equal or greater than 3 and 50% is greater or equal to 4 stars. The low ratings are not frequent in the dataset and a 4-star rating is the most frequent evaluation.





Now, let's count how those reviews are distributed among the users and movies.





User Item Similarity

Similarity-based methods determine the most similar objects with the highest values as it implies, they live in closer neighborhoods. There are multiple similarity-based metrics

- Pearson's correlation
- Spearman's correlation
- Kendall's Tau
- Cosine similarity
- Jaccard similarity

However, for our case, we'll be using Pearson's correlation because we have a sparse matrix, which means that most movies are not rated (have a rating of 0). hence, we will center all the ratings to 0 so that the default rating becomes 0.

Jaccard Distance

We could ignore values in the matrix and focus only on the sets of items rated. If the utility matrix only reflected purchases, this measure would be a good one to choose. However, when utilities are more detailed ratings, the Jaccard distance loses important information.

Cosine Distance

The cosine similarity is a metric used to find the similarity between the items/products irrespective of their size. We calculate the cosine of an angle by measuring between any two vectors in a multidimensional space. The disadvantage of cosine similarity is treating the lack of a rating as more like disliking the movie than liking it.

Pearson Correlation

Pearson Correlation is the most well-known and classic way of finding similarities in collaborative filtering-based recommendation systems.

$$Pearson\ Correlation \quad \rho_{uv} = \frac{\sum_{i \in I_{uv}} ((r_{ui} - \mu_u) * (r_{vi} - \mu_v))}{\sqrt{\sum_{i \in I_{uv}} (r_{ui} - \mu_u)^2} * \sqrt{\sum_{i \in I_{uv}} (r_{vi} - \mu_v)^2}}$$

r_{ui}	Rating of user u on item i
$\hat{r}_{ui}$	Prediction rating of user u on item i
μ_u	Average rating of user u
ρ_{uv}	Pearson correlation in between user u and v
I_{uv}	Movies rated by user u and user v
k	K nearest neighbours
u,v	user of interest
i,j	item of interest

KNNBasis Similarity

KNNBasis is a perfect go-to model and a very good baseline for recommender system development. But this is a **non-parametric, lazy** learning method. It uses a database in which the data points are separated into several clusters to make inference for new samples. KNNBasis does not make any assumptions on the underlying data distribution but it relies on **item feature similarity**. In this approach we calculated the recommendation using pearson correlation.

```
#Item Based approach
sim_options = {
    'name': 'pearson',
    'user_based': 'False'
}

clf = KNNBasic(sim_options = sim_options)
cross_validate(clf, dataset, measures=['RMSE'], cv=5, verbose=True)
```

The evaluation metric used is Root Mean Square Error. The 5 fold cross validation output is as shown below,

	Fold 1	Fold 2	Fold 3	Fold 4	Fold 5	Mean	Std
RMSE (testset)	1.0925	1.0714	1.0730	1.0905	1.0783	1.0811	0.0088
Fit time	0.04	0.03	0.02	0.03	0.02	0.03	0.01
Test time	0.02	0.02	0.02	0.02	0.02	0.02	0.00

Limitations of Knn?

Popularity bias - refers to system recommends the movies with the most interactions without any personalization

Item cold-start problem - 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.

scalability issue - refers to lack of the ability to scale to much larger sets of data when more and more users and movies added into our database.

Collaborative Filtering

Collaborative filtering tackles the similarities between the users and items to perform recommendations. Meaning that the algorithm constantly finds the relationships between the users and in-turns does the recommendations. The algorithm learns the embeddings between the users without having to tune the features. The most common technique is by performing Matrix Factorization to find the embeddings or features that make up the interest of a particular user.

Utility Matrix

In a recommendation-system application, there are two classes of entities, which we shall refer to as users and items. Users have preferences for certain items, and these preferences must be teased out of the data. The data itself is represented as a utility matrix, giving each user-item pair, a value that represents what is known about the degree of preference of that user for that item.

	HP1	HP2	HP3	TW	SW1	SW2	SW3
A	4			5	1		
B	5	5	4				
C				2	4	5	
D		3					3

.1: A utility matrix representing ratings of movies on a 1–5 scale

we see an example utility matrix, representing users' ratings of movies on a 1–5 scale, with 5 the highest rating. Blanks represent the situation where the user has not rated the movie. The movie names are HP1, HP2, and HP3 for Harry Potter I, II, and III, TW for Twilight, and SW1, SW2, and SW3 for Star Wars episodes 1, 2, and 3. The users are represented by capital letters A through D.

'The goal of a recommendation system is to predict the blanks in the utility matrix'

Utility Matrix of our dataset

```
# Filling NA's with 0
uim = uim.fillna(0)
uim.head()
```

MovieID	8	28	43	48	61	64	66	92	96	111	...	17654	17660	17689	17693	17706	17725	17728	17734	17741	17742
UserID																					
7	5.0	4.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	...	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
79	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	...	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
199	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	4.0	...	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
481	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	5.0	...	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
769	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	...	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0

ALS Collaborative Filtering Implementation

In a real-world setting, most movies receive very few or even no ratings at all by users. We are looking at an extremely sparse matrix with more than 99% of entries are missing values. With such a sparse matrix, what ML algorithms can be trained and reliable to make inference? To find solutions to the question, we are effectively solving a data sparsity problem using Matrix Factorization.

In collaborative filtering, matrix factorization is the state-of-the-art solution for sparse data problem.

Matrix Factorization

Matrix factorization is an embedding. Say we have a user-movie matrix or feedback matrix, A^{NM}, the models learn to decompose into:

- A user embedding vector U, where row N is the embedding for item M.
- An item embedding vector V, where row M is the embedding for item N

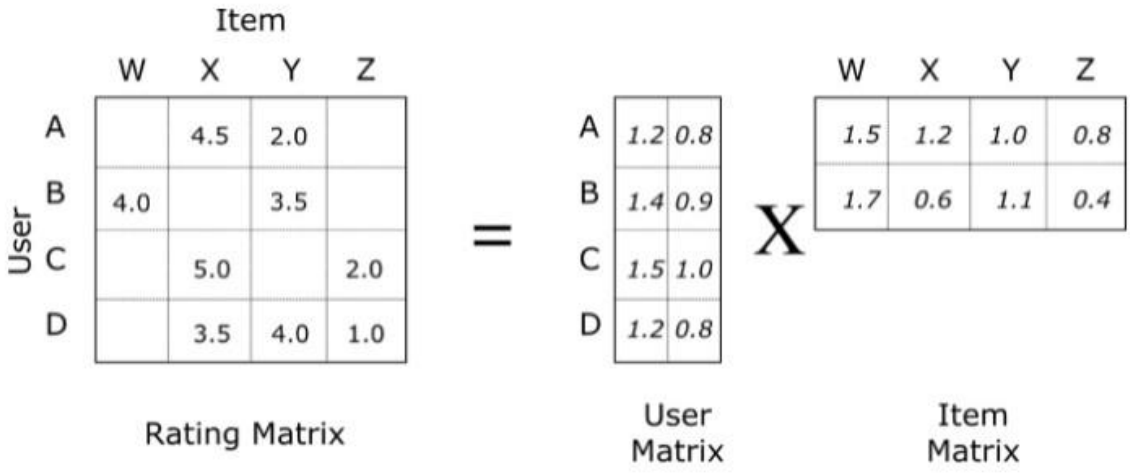
The embedding vector is learned such that by performing UV^T, an approximation of the feedback matrix, A can be formed.

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

where H is user matrix, W is item matrix

Matrix factorization is a factorization of a matrix into a product of matrices. In the case of collaborative filtering, 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.

In the sparse user-item interaction matrix, the predicted rating user u will give item i is computed as:



Matrix Factorization of Movie Ratings Data

How does matrix factorization solve our problems?

1. Model learns to factorize rating matrix into user and movie representations, which allows model to predict better personalized movie ratings for users.
2. 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

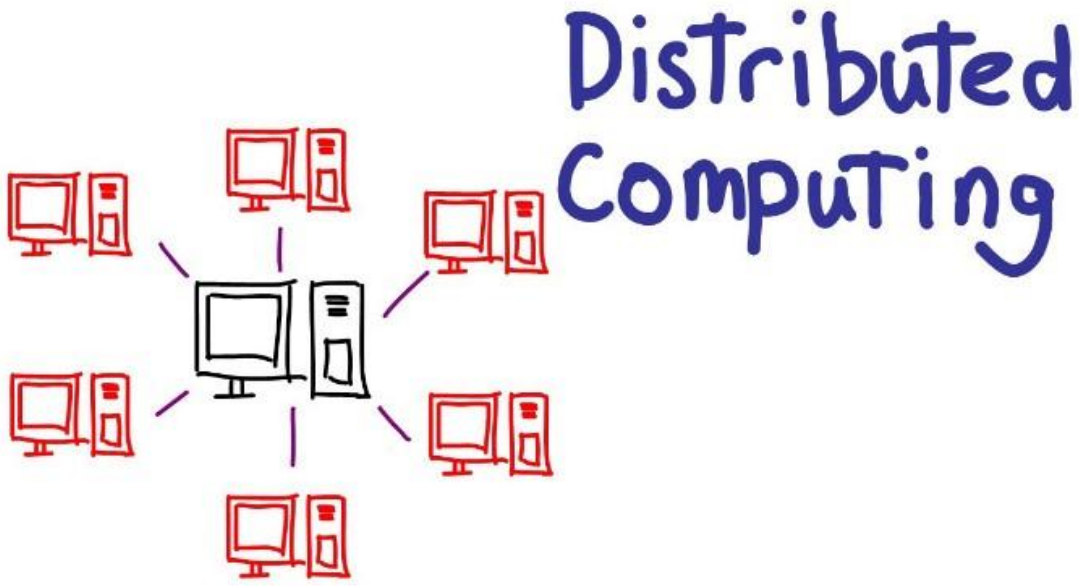
Although Funk SVD was very effective in matrix factorization with single machine during that time, it's not scalable as the amount of data grows today. With terabytes or even petabytes of data, it's impossible to load data with such size into a single machine. So, we need a machine learning model (or framework) that can train on a dataset spreading across from a cluster of machines.

Alternating Least Square (ALS) with Spark ML

Alternating Least Square (ALS) is also 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.

spark.ml currently 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. The implementation in spark.ml has the following parameters:
PySpark is the collaboration of Apache Spark and Python. Apache Spark is an open-source cluster-computing framework, built around speed, ease of use, and streaming analytics whereas Python is a general-purpose, high-level programming language.



Scaling Machine Learning Applications With Distributed Computing

System Configuration

We used Amazon EMR cluster with m4.xlarge configuration with 3 clusters.

The following are a set of instructions to run pyspark notebook on the master cluster.

```
sudo pip3 install pyyaml ipython jupyter ipyparallel pandas boto3 seaborn
sudo pip3 install scipy scikit-learn surprise
which jupyter

sudo vi ~/.bashrc
export PYSPARK_DRIVER_PYTHON=/usr/local/bin/jupyter
```

```
export PYSPARK_DRIVER_PYTHON_OPTS="notebook --no-browser --ip=0.0.0.0 --port=8888"

source ~/.bashrc
pyspark

# copy the  token

open jupyter notebook in <Master public DNS>:8888
Ex: ec2-54-226-239-4.compute-1.amazonaws.com:8888
```

ALS Implementation

Some high-level ideas behind ALS are:
Its objective function is slightly different than Funk SVD: ALS uses L2 regularization while Funk uses L1 regularization
Its training routine is different: ALS 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

```
# Create ALS model
als = ALS(
         maxIter=10,
         userCol="UserID",
         itemCol="MovieID",
         ratingCol="Rating",
         nonnegative = True,
         implicitPrefs = False,
         coldStartStrategy="drop"
)
```

Just like other machine learning algorithms, ALS has its own set of hyper-parameters. We probably want to tune its hyper-parameters via hold-out validation or cross-validation.

```
# Define evaluator as MSE
mse_evaluator = RegressionEvaluator(
           metricName="mse",
           labelCol="Rating",
           predictionCol="prediction")
```

```
# Build cross validation using CrossValidator
cv = CrossValidator(estimator=als, estimatorParamMaps=param_grid, evaluator=rmse_evaluator, numFolds=5)
```

```
#Fit cross validator to the 'train' dataset
model = cv.fit(trainingRatings_df)
```

Most important hyper-params in Alternating Least Square (ALS):
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)

```
#Extract best model from the cv model above
best_model = model.bestModel
print("**Best Model**")
# Print "Rank"
print("  Rank:", best_model._java_obj.parent().getRank())
# Print "MaxIter"
print("  MaxIter:", best_model._java_obj.parent().getMaxIter())
# Print "RegParam"
print("  RegParam:", best_model._java_obj.parent().getRegParam())
```

```
**Best Model**
  Rank: 20
  MaxIter: 10
  RegParam: 0.05
```

Evaluation

We evaluated the performance using Root Mean Square Error and Mean square evaluators.

RegressionEvaluator

Define the evaluator, select rmse as metricName in evaluator.

```
# Define evaluator as RMSE
rmse_evaluator = RegressionEvaluator(
            metricName="rmse",
            labelCol="Rating",
            predictionCol="prediction")
```

```
# Define evaluator as MSE
mse_evaluator = RegressionEvaluator(
            metricName="mse",
            labelCol="Rating",
            predictionCol="prediction")
```

RMS error and MS errors are as shown below,

```
RMSE = rmse_evaluator.evaluate(test_predictions)
print('RMSE Error on Test data : ', RMSE)
```

```
[Stage 4065:=============================================>
```

```
RMSE Error on Test data :  0.8356966096190834
```

```
MSE = mse_evaluator.evaluate(test_predictions)
print('MSE Error on Test data : ', MSE)
```

```
[Stage 4095:====================================>
```

```
MSE Error on Test data :  0.6983888233288308
```

Results Section

Predicted ratings on test dataset with the best model from the cross validation

```
# View the predictions
test_predictions = best_model.transform(testingRatings_df)
```

```
print("Model Predicted Ratings on Test data : \n")
test_predictions.show(5, truncate=False)
```

```
Model Predicted Ratings on Test data :

+-------+-------+------+----------+
|MovieID|UserID |Rating|prediction|
+-------+-------+------+----------+
|28     |2358799|3.0   |3.861073  |
|156    |973051 |5.0   |4.066124  |
|851    |1189060|3.0   |3.5445638 |
|1100   |2376892|2.0   |2.2553442 |
|1123   |1628484|3.0   |3.4171214 |
+-------+-------+------+----------+
only showing top 5 rows
```

Predicted top 10 recommendations for the use with the User ID ‘2678’

```
total_rec_df.filter('UserId = 2678').sort('Rating', ascending=False).limit(10).show()
```

```
[Stage 4303:=================================================> (98 + 2) / 100]

+-------+------+---------+--------------------+
|MovieID|UserID|   Rating|               Title|
+-------+------+---------+--------------------+
|  12421|  2678|4.4339585|        The Crusades|
|  15480|  2678|4.3982887|Farscape: The Pea...|
|  12125|  2678|4.3975625|The Blue Planet: ...|
|  12544|  2678|4.3677964|Peter Gabriel: Pl...|
|   4238|  2678|4.3015156|           Inu-Yasha|
|   7283|  2678|4.2696924|Due South: Call o...|
|  10947|  2678| 4.256988|     The Incredibles|
|  17515|  2678|4.2543573|LeapFrog: Letter ...|
|   2160|  2678| 4.209249|Magnetic Storm: Nova|
|   7505|  2678| 4.186874|Fushigi Yugi: The...|
+-------+------+---------+--------------------+
```

Results on Own Preferences

Created dataset with a custom preference. Added my movie preferences as a new user to the data set. To do this, I created a new, unique user ID for myself.

```
my_rated_movies = [
             (6991,my_user_id,5.0),
             (12232,my_user_id,3.0),
             (7569,my_user_id,2.5),
             (16559,my_user_id,4.0),
             (8933,my_user_id,1.8),
             (2355,my_user_id,4.6),
             (12125,my_user_id,1.0),
             (2160,my_user_id,2.0),
             (209,my_user_id,3.0),
             (17515,my_user_id,3.5),
             (2985,my_user_id,0.0),
             (28,my_user_id,5.0),
             (1100,my_user_id,3.0),
             (4238,my_user_id,4.0),
             (7283,my_user_id,2.0),
             (6522,my_user_id,1.0),
             (15040,my_user_id,3.5),
             (3151,my_user_id,4.5),
             (218,my_user_id,2.5),
             (7016,my_user_id,1.5),
             (15567,my_user_id,1.8),
             (9701,my_user_id,2.8),
             (4963,my_user_id,3.0),
             (4,my_user_id,5.0),
             (1, my_user_id,4.6),

]
```

Selected some movies that, to those as training set and test set.

```
my_ratings_df = sqlContext.createDataFrame(my_rated_movies, schema=testingRatings_df_schema)
my_ratings_df.cache()
seed = 1999
my_train_df, my_test_df, my_val_df = my_ratings_df.randomSplit([6.0, 3.0, 1.0], seed)
my_test_df = my_test_df.union(my_val_df)

my_train_df.cache()
my_test_df.cache()

my_movies_df = my_ratings_df.join(movieTitles_df, on='MovieID')
my_movies_df.show()
```

```
+-------+------+------+--------------------+
|MovieID|UserID|Rating|               Title|
+-------+------+------+--------------------+
|   6991|  5000|   5.0|    A History of God|
|  12232|  5000|   3.0| Lost in Translation|
|   7569|  5000|   2.5|Dead Like Me: Sea...|
|  16559|  5000|   4.0|Red Green: Stuffe...|
|   8933|  5000|   1.8|Ikiru: Bonus Mate...|
|   2355|  5000|   4.6|Berlin: Symphony ...|
|  12125|  5000|   1.0|The Blue Planet: ...|
|   2160|  5000|   2.0|Magnetic Storm: Nova|
|    209|  5000|   3.0|Star Trek: Deep S...|
|  17515|  5000|   3.5|LeapFrog: Letter ...|
```

Founded the predictions by model.

```
# Generate n Recommendations for all users n = 10
my_recommendations = my_als_model.recommendForAllUsers(10)
my_recommendations = my_recommendations\
    .withColumn("rec_exp", explode("recommendations"))\
    .select('UserID', col("rec_exp.MovieID"), col("rec_exp.Rating"))
my_rec_df = my_recommendations.join(movieTitles_df, on='MovieID')
my_rec_df.filter('UserId = '+str(my_user_id)).sort('Rating', ascending=False).limit(10).show()
```

```
+-------+------+---------+--------------------+
|MovieID|UserID|   Rating|               Title|
+-------+------+---------+--------------------+
|   6991|  5000| 5.012259|    A History of God|
|      4|  5000|4.9514775|Paula Abdul's Get...|
|   2756|  5000| 4.798466|         Tenchi Muyo|
|   4918|  5000|4.6848273|Slayers Try DVD C...|
|   8919|  5000| 4.660586|   Cardcaptor Sakura|
|   6619|  5000| 4.568393|Slayers Next DVD ...|
|  10777|  5000| 4.393004|Battle Athletes V...|
|   5484|  5000|4.2430744|Har dil jo Pyar k...|
|   6629|  5000| 4.231745|Legend of the Dra...|
+-------+------+---------+--------------------+
```

Discussion

In a real-life scenario like Netflix, not only one method is used but rather a hybrid architecture is used. Different types of recommendation algorithms are combined to produce the highest quality of recommendations.

Different model architectures excel at different tasks. A recent study of Netflix proved a scope for deep learning in recommendation systems.

Deep Learning for Recommender Systems: A Netflix Case Study

Recommendations | Machine Learning

Publication

Feb 4, 2022 • Harald Steck, Senior Research Scientist; Linas Baltrunas, Senior Research Engineer; Ehtsham Elahi, Senior Research Scientist; Dawen Liang, Senior Data Scientist; Yves Raimond, Research/Engineering Director; Justin Basilico, Director, Recommendation Systems Research and Engineering

Even though many deep-learning models can be understood as extensions of existing (simple) recommendation algorithms, that were initially did not observe significant improvements in performance over well-tuned non-deep-learning approaches. Deep learning has ultimately resulted in large improvements to Netflix recommendations as measured by both offline and online metrics. On the practical side, integrating deep-learning toolboxes in Netflix system has made it faster and easier to implement and experiment with both deep-learning and non-deep-learning approaches for various recommendation tasks.

Conclusion

In this post, we covered how to improve collaborative filtering recommender system with **matrix factorization**. We learned that matrix factorization can solve “popular bias” and “item cold-start” problems in collaborative filtering. We also leveraged **Spark ML** to implement distributed recommender system using **Alternating Least Square (ALS)**.