(Recommender Blog)

Building A Context Aware Recommender System by Alternating Least Square Matrix Factorization in Apache Spark Cluster

Parshu Rath

January 2021 update

Keywords : Recommender system, cluster computing, context aware recommendation, Movielens dataset, Matrix factorization, Alternating Least Squares (ALS), Machine learning

Software/Tools: Python (version 2.7.15), Jupyter Notebook (version 5.7.4), IPython (version 5.8.0), Spark (version 1.6.2),

Table of Contents

1. Objectives

This works describes steps in building a context-aware recommender system (CARS) using the Movielens dataset on Python Apache Spark cluster. The goal is to incorporate contextual information (such as time of day the movie was watched) in order to deliver not just movie recommendations but also unique user experiences shaped by user context. It also takes advantage of the distributed computing power to handle processing of large scale data and thus increasing relevance of the recommendation models to the users by reducing latency.

2. Introduction

Product recommendation is ubiquitous in the virtual online shopping world that we currently live in. Whether you have purchased or just browsed items in the Amazon, Wal-Mart, Homedepot, or any other preferably large e-commerce sites, or just watched some Netflix movies, you are presented with a series of product or movies recommendations based on your purchase/viewing and browsing history. The goal of those recommendations is to entice you with a possible product (or a movie) that you may be interested and convert you from a browser to a purchaser or in the case on Netflix to a viewer. Efficiency of such conversion depend on how accurately the recommendation matches your interests for the products. Thus building an effective recommender system is essential to achieving an efficient browser to purchaser conversion ratio. There are several types of recommendation systems which are quite efficient in their own rights. An effective recommendation system could be just one of them or a combination of one or more of various recommendation systems.

As noted above there are several ways to build a recommender systems. A few of them are described below.

2.1 Collaborative Filtering Recommendation System

Collaborative filtering recommendation system uses users choices to recommend products based on similar users (collaborators). It can be either of the

(i) Item-Item collaborative filtering

(ii) User-User collaborative filtering

(iii) User-Item collaborative filtering.

2.1.1 Item-Item Collaborative Filtering

In the item-item collaborative filtering method, customer behaviors based on previously purchased items are used and recommendations are made based on item similarity.

Typically KNN (K-nearest neighbor) algorithm is used for item-based collaborative filtering which relies on item similarity either based on high dimensional Euclidean distance or just simple cosine distance defined below.

2.1.1.1 Euclidean distance

Distance between point x and y in a N-dimensional field is

$$d(x,y) = \sqrt{(y_1 - x_1)^2 + (y_2 - x_2)^2 + \cdots (y_n - x_n)^2} \quad \ldots \quad (1)$$

2.1.1.2 Cosine distance

Cosine distance is based on dot product of two vectors. Again the cosine distance between two N-dimensional vectors is given by the following formula.

$$d(x,y) = \frac{\sum_{i=1}^{n} x_i y_i}{\sum_{i=1}^{n} x_i^2 \sum_{i=1}^{n} y_i^2} \quad \ldots \quad (2)$$

2.1.2 User-User Collaborative Filtering

In the user-user collaborative filtering method user's overall purchasing behavior is used to predict the recommended items based on similar users purchasing history. Here also Euclidean or cosine-distance can be used for building recommendation system.

2.1.3 Drawbacks in item-item and user-user collaborative filtering

There are some inherent drawbacks of using either item-item or user-user collaborative system.

2.1.3.1 Popularity Bias

Items with less popularity will not be recommended by any of these systems. Only the more popular items get recommended more often (rich-gets-richer). Model is biased towards popular items.

2.1.3.2 Cold Start Issue

Any new item will not get recommended since its distance from other items are unknown. This can be alleviated by using a content based system (or a hybrid recommender system).

2.1.3.3 Scalability Problem (Curse of Dimensionality)

In-line recommendation can be very slow and intractable if not designed efficiently. This is due to the complexity of the KNN model which is in the order of $O(ndk)$, where n is number of users, d is number of items, and k is number of neighbors considered in the model. As n and d become large, the KNN model becomes more complex and computationally intensive.

2.1.4 User-Item Collaborative Filtering

User-item or model based collaborative filtering is based on the user-item interactions represented by user-item matrix. Typically this matrix is very sparse since each user only purchased small number of the thousands of items available and each item can be linked with only few customers. Therefore majority of the entries in the user-item matrix is zero. For a better representation of the user-item interaction matrix factorization method is used to factor the large sparse matrix into two smaller dense matrix.

2.2 Content Based Recommendation System

Collaborative filtering method relies on the user-item interactions, i.e. users' behaviors towards an item. It does not take into account how one user is strictly different from another or one item is strictly different from the other item. On the other hand, content based filtering takes into account user and/or item characteristics. For example, it may consider, age, sex, job or any other personal information of the users and similarly it may consider movie genre, year, actors, length of the movie etc. It may also use some unstructured information to derive a secondary attribute such as sentiment for use in the filtering process.

Content based recommender system can be built upon a classification methods (Bayesian classifier) for binary attributes such as "like", or it can also be built based on a regression model to predict rating for movie by an user.

2.3 Context Aware Recommendation System (CARS)

Neither of the collaborative filtering and content based filtering take into account context of the user and the item. Adding context as an additional parameter in building the recommender system is like ly to make it more efficient. For example, one of the context could be time of the day a movie was watched. If it the watching time is during the day, it is likely that the movie is suitable for children, young adult, or it is a family movie. If the watching time is late evening or night, it is likely appropriate for adult. This kind of differentiation may not appear in a recommender system which does not take into account such a context. Similarly a customer who has purchased children's book may not have any predictive value in getting recommended for work related books. Therefore adding context make the recommendations more relevant.

Context aware recommender system can be built in three ways.

i. Context pre-filtered recommender system
ii. Recommender system built with context
iii. Context post-filtered recommender system

This blog describes a method of building a context pre-filtered recommender system.

3. Data and Methods

The Movielens dataset was accessed at http://grouplens.org/datasets/movielens/, and consist of three text files containing data on 1 million ratings from 6000 users and 4000 movies. The dataset was released in 2003 (movielens.com). This data was used for building the recommender system.

Alternating Least Square (ALS) matrix factorization implemented in Apache Spark ML was used for building the recommender system. Python Jupyter notebook was used as the scripting tool. Some information about the matrix factorization and its use in building recommender system is given below.

3.1 Matrix Factorization

Matrix factorization is a method of factorizing a very large sparse matrix into smaller matrices. Matrix factorization model in an user-item utility matrix assumes that there exists a low dimensional latent space of features where the user-item interaction can be obtained by a simple dot product of the dense vectors in that space. For example in case of user-item matrix M (*utility matrix*) of nxm dimension with n users and m items, it can be factorized into a nxl user matrix and mxl item matrix. Mathematically factorization of the utility matrix can be written as follows.

$$M = P.Q^T \quad \ldots \quad \ldots \quad \ldots \quad \ldots \quad \ldots \quad \ldots \quad \ldots \ldots \quad \ldots \quad \ldots \quad \ldots \quad \ldots \quad \ldots \quad \ldots \quad \ldots \quad \ldots \quad \ldots \quad (3)$$

where P is the nxl user matrix for n users and Q is the mxl item matrix for m items.

l is the dimension of the latent space where users and items are mapped.

Once the matrix is factorized to lower dimensions, only the smaller matrix can be used for nearest neighbor searches which make the process more efficient. For recommendation, user vector can be multiplied with an item vector to estimate rating.

Matrix factorization is similar to singular value decomposition (SVD) of matrix. However, the similarity ends there for factorizing a heavily sparse matrix. Typically SVD will be undefined for sparse matrix and using only few entries (leaving out sparse columns/rows) carries the risk of overfitting the model. Matrix imputation is an alternative choice. However, imputing data in a large matrix is prone to errors and can be computationally expensive as it increases the data significantly. Thus modeling directly with regularization to avoid overfitting offers an accurate method of factorization. For the movielens data matrix regularization a squared error on the set of ratings is minimized.

There are two methods of minimization of square errors in matrix factorization.

3.1.1 Stochastic Gradient Descent

Stochastic gradient descent was popularized by Simon Funk (https://sifter.org/~simon/journal/20061211.html) in the Netflix challenge. In this method each users rating is predicted and its error is computed. The parameter is then adjusted in the opposite direction of the gradient and the process is repeated until the gradient is at minimum. This method however can becomes slow for a very large sparse matrix where alternating least square method in a cluster offers advantage.

3.1.1 Alternating Least Square (ALS)

In matrix factorization, a known matrix M is factorized into two unknown matrices P and Q as in equation (3) above. Since both the factor matrices are unknown, the solution (minimization problem with square regularization term) is not convex. However, if one of the matrix (either $P\ or\ Q$) is held fixed, the problem reduces to a quadratic expression which can be solved optimally. In ALS, the matrices $P\ and\ Q$ are held fixed alternately and parameters are selected in opposite direction of the gradient until an optimal solution is achieved. This is typically setup to run in cluster computing environment where a solution can be arrived at quickly.

In this blog alternating least square (ALS) matrix factorization method in Apache Spark cluster is employed to build a context aware recommender system (CARS).

4. Steps in Building CARS

Below are the steps used in building the recommender system. Details of each of the steps are given in the following sections.

- **Loading the data** - the data was loaded by importing the 3 files and merged into a Pandas Dataframe.
- **Data exploration** - The dataframe was converted to a Spark DataFrame object that allowed us to perform data exploration using SQL query in Spark cluster.
- **Data visualization** - Dataframe was transformed into a Resilient Distributed Dataset (RDD) to demonstrate how visualizations can be performed quickly on large datasets.
- **Machine Learning**- Recommendations and Predictions were performed using the Spark MLlib library with the Alternating Least Squares algorithm. Models were evaluated based on root mean square errors (RMSE).
- **Contextual pre-filtering** technique was enabled before training and testing the models. Contexts included the day of the week and time of the day that the user would likely watch a movie.

- **Retrain and Predict**- Add new user ratings and re-train the model with combined dataset. Re-run the model to make recommendations and predict individual ratings
- **Dataset size**- Results of using the 1,000,000 records was also compared to using a much smaller set of records 100,000.

5. Loading Movielens 1 Million rows Data

Below is a code chunk that loads all the necessary python packages.

5.1 Code chunk1

```
# Required Python Packages
from math import sqrt
import pandas as pd
import numpy as np
from scipy import spatial
import scipy
import matplotlib.pyplot as plt
import datetime
import os
from operator import add

#Set environments
import os
import sys
spark_home = os.environ.get('SPARK_HOME', None)
if not spark_home:
    raise ValueError('SPARK_HOME environment variable is not set')
sys.path.insert(0, os.path.join(spark_home, 'python'))
sys.path.insert(0, os.path.join(spark_home, 'C:/spark/python/lib/py4j-0.9-src.zip'))
execfile(os.path.join(spark_home, 'python/pyspark/shell.py'))

Welcome to
      ____              __
     / __/__  ___ _____/ /__
    _\ \/ _ \/ _ `/ __/  '_/
   /__ / .__/\_,_/_/ /_/\_\   version 1.6.2
      /_/

Using Python version 2.7.15 (default, Dec 10 2018 21:57:18)
SparkContext available as sc, HiveContext available as sqlContext.
```

6. Create Python Pandas dataframe from 1M movielens data

The Movielens dataset is available at http://grouplens.org/datasets/movielens/, and consist of three text files containing data on 1 million ratings from 6000 users and 4000 movies. The dataset was released in 2003 (movielens.com).

The three files text files with the data are:

i. *Ratings.dat* consisting of userid, movieid, timestamp and ratings (between 1 to 5)
ii. *Users.dat* consisting of demographic data on the users
iii. *Movies.dat* consisting of movieid, movietitle and genres of movies

These 3 files were download, extracted, and stored in Github for use in the application.

6.1 Movielens Dataset Dimension (rows and columns) and sample rows

As shown below, the dataset imported to python pandas dataframe contained ~1 million rows and 13 columns (attributes). Figure below shows randomly selected 5 rows from the dataset.

```
In [8]: movielens.shape
Out[8]: (1000209, 13)

In [9]: #movielens.head()
        movielens.iloc[np.random.randint(1000209, size=5)]
```

Out[9]:

	userid	movieid	rating	timestamp	movietitle	gender	age	occupation	zipcode	date	time	weekday	timeofday
272270	3653	2505	3	966452927	8MM (1999)	M	18	15	96661	2000-08-16	15:08:47	Weekday	afternoon
605978	398	1249	5	1044567822	Nikita (La Femme Nikita) (1990)	M	25	17	55454	2003-02-06	16:43:42	Weekday	afternoon
213644	2909	3386	5	971882777	JFK (1991)	M	25	7	43228	2000-10-18	11:26:17	Weekday	morning
427735	5675	2524	3	958742610	Towering Inferno, The (1974)	M	35	14	30030	2000-05-19	09:23:30	Weekend	morning
86064	1150	3736	2	975196037	Big Carnival, The (1951)	F	25	20	75226	2000-11-25	18:47:17	Weekend	evening

Figure 1: Example of 5 rows in movielens pandas dataframe

7. Data Exploration and Visualization in Spark Cluster

Pandas dataframe was exported to Spark cluster for data exploration using SQL queries in Spark.

7.1 Create Spark Dataframe and print some rows

```
movie_df.show(5)
```

```
+------+-------+------+---------+--------------------+------+---+----------+-------+----------+--------+-------+---------+
|userid|movieid|rating|timestamp|          movietitle|gender|age|occupation|zipcode|      date|    time|weekday|timeofday|
+------+-------+------+---------+--------------------+------+---+----------+-------+----------+--------+-------+---------+
|     1|   1193|     5|978300760|One Flew Over the...|     F|  1|        10|  48067|2000-12-31|17:12:40|Weekend|afternoon|
|     1|    661|     3|978302109|James and the Gia...|     F|  1|        10|  48067|2000-12-31|17:35:09|Weekend|afternoon|
|     1|    914|     3|978301968| My Fair Lady (1964)|     F|  1|        10|  48067|2000-12-31|17:32:48|Weekend|afternoon|
|     1|   3408|     4|978300275|Erin Brockovich (...|     F|  1|        10|  48067|2000-12-31|17:04:35|Weekend|afternoon|
|     1|   2355|     5|978824291|Bug's Life, A (1998)|     F|  1|        10|  48067|2001-01-06|18:38:11|Weekend|  evening|
+------+-------+------+---------+--------------------+------+---+----------+-------+----------+--------+-------+---------+
only showing top 5 rows
```

Figure 2: Randomly selected 5 rows of user 1 in movielens Spark dataframe

7.2 Data Summary

Table below summarizes the 'rating' and 'age' attributes in the imported movielens dataset.

```
#Display summary from the table
movie_df.select("rating", "age").describe().show()
```

```
+-------+------------------+------------------+
|summary|            rating|               age|
+-------+------------------+------------------+
|  count|           1000209|           1000209|
|   mean| 3.581564453029317| 29.73831369243828|
| stddev|1.1171018453732595|11.751982567744237|
|    min|                 1|                 1|
|    max|                 5|                56|
+-------+------------------+------------------+
```

Figure 3: Summary of "rating" and "age" columns in the data

7.3 Top 10 users with most ratings

Table below shows 10 users with most ratings.

```
#Top 10 users with most ratings
sqlContext.createDataFrame(sqlContext.sql("SELECT userid, COUNT(*) AS ratings_count FROM ratings_data_all GROUP BY userid \
ORDER BY ratings_count DESC LIMIT 10").collect()).show()
```

```
+------+-------------+
|userid|ratings_count|
+------+-------------+
|  4169|         2314|
|  1680|         1850|
|  4277|         1743|
|  1941|         1595|
|  1181|         1521|
|   889|         1518|
|  3618|         1344|
|  2063|         1323|
|  1150|         1302|
|  1015|         1286|
+------+-------------+
```

Figure 4: Top 10 users with most ratings

7.4 Number of movies by weekday and weekend

```
# Number of movies by weekday and weekend
sqlContext.createDataFrame(sqlContext.sql("SELECT weekday, COUNT(distinct(movieid)) AS movie_count FROM ratings_data_all GROUP B
Y weekday").collect()).show()
```

```
+-------+-----------+
|weekday|movie_count|
+-------+-----------+
|Weekend|       3557|
|Weekday|       3666|
+-------+-----------+
```

Figure 5: Movies watched on weekdays and weekends

7.5 Number of movies watched by time of day

```
# Number of movies watched by time of day
sqlContext.createDataFrame(sqlContext.sql("SELECT timeofday, COUNT(distinct(movieid)) AS movie_count FROM ratings_data_all GROUP
BY timeofday").collect()).show()
```

```
+---------+-----------+
|timeofday|movie_count|
+---------+-----------+
|    night|       3489|
|  morning|       3376|
|  evening|       3521|
|afternoon|       3584|
+---------+-----------+
```

Figure 6: Movies watched by time of the day

7.6 Users with context

Table below shows 3 movies watched by the same user and the corresponding time the movies were watched.

```
# select needed columns
m_v = movie_df.map(lambda x: (x[0], x[2], x[4], x[5], x[11], x[12]))
sqlContext.createDataFrame(m_v.take(3), schema=('userid', 'rating', 'movietitle', 'gender', 'weekday',
```

```
'timeofday')).show()
#schema=('userid', 'rating', 'movietitle', 'gender', 'weekday', 'timeofday')
```

```
+------+------+--------------------+------+-------+---------+
|userid|rating|          movietitle|gender|weekday|timeofday|
+------+------+--------------------+------+-------+---------+
|     1|     5|One Flew Over the...|     F|Weekend|afternoon|
|     1|     3|James and the Gia...|     F|Weekend|afternoon|
|     1|     3| My Fair Lady (1964)|     F|Weekend|afternoon|
+------+------+--------------------+------+-------+---------+
```

Figure 7: Example of a user's movie watching behavior

7.7 Most watched movies

```
# Function to show histogram
def histogram(movies):
    count = map(lambda x: x[0], movies)
    movietitle = map(lambda x: x[1], movies)
    plt.barh(range(len(count)), count, color = 'lightblue')
    plt.yticks(range(len(count)), movietitle)
    f = plt.gcf()
    f.set_size_inches(8, 5)
    #display(f)

    #Most Watched Movies
    histogram(x.take(5))
    plt.xlabel('Number of times watched')
```

Below is a plot of top 5 most watched moviews.

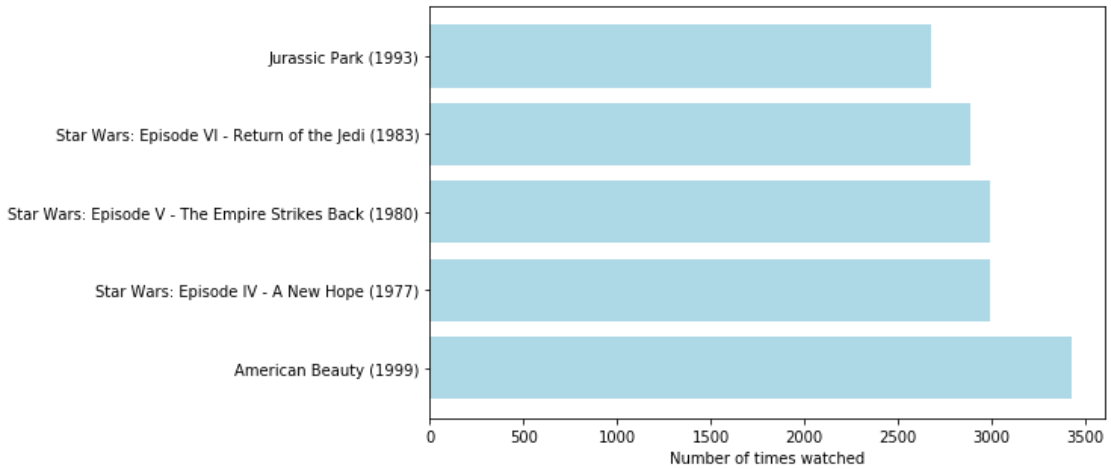

Figure 10: Top 10 most watched movies in the dataset

7.8 Ratings for movies

The plot below shows histogram of ratings in the dataset.

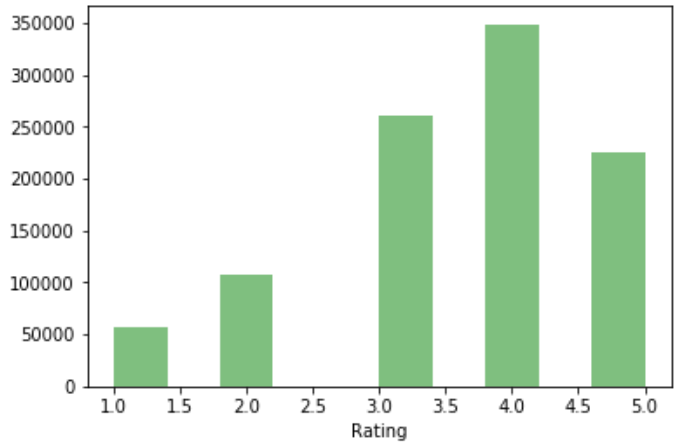

Figure 11: Rating score histogram

7.9 Ratings by user

Plot below shows number of ratings for 10 users.

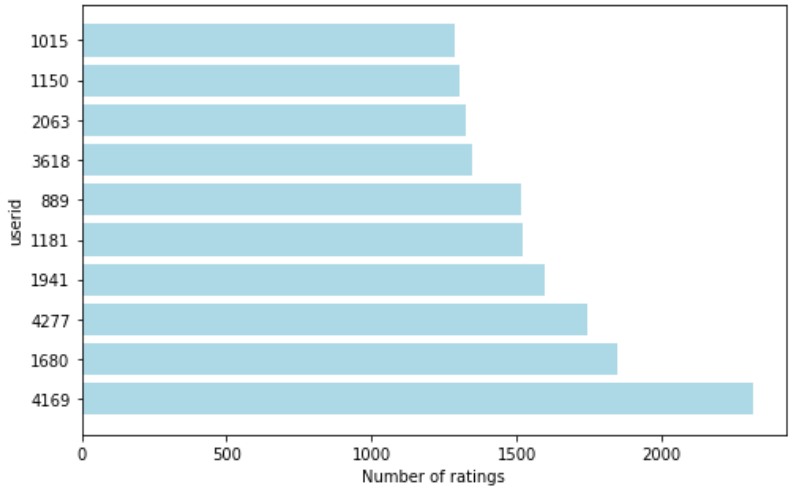

Figure 12: Sample of users and their total ratings

7.10 Movie watching (weekday and weekend)

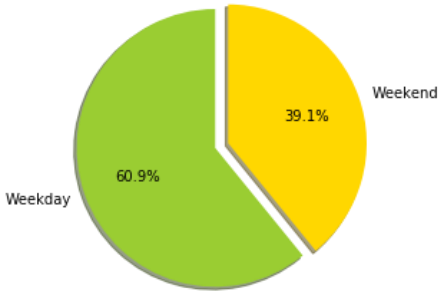

Figure 13: Percentages of movies watched on weekdays vs weekends

7.11 Movie watching (Time of the day)

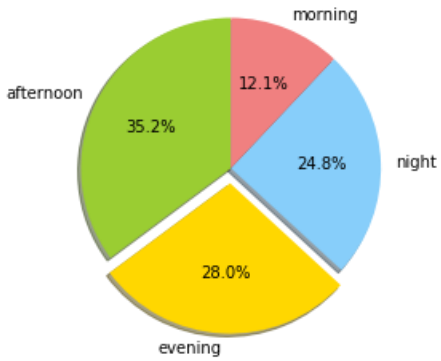

Figure 14: Percentages of movies watched at various times of the day

8. Build Recommender System

8.1 Create RDD

```
#movie_RDD.take(3)
sqlContext.createDataFrame(movie_RDD.take(3)).show()
```

```
+------+-------+------+---------+--------------------+------+---+----------+-------+----------+--------+-------+---------+
|userid|movieid|rating|timestamp|          movietitle|gender|age|occupation|zipcode|      date|    time|weekday|timeofday|
+------+-------+------+---------+--------------------+------+---+----------+-------+----------+--------+-------+---------+
|     1|   1193|     5|978300760|One Flew Over the...|     F|  1|        10|  48067|2000-12-31|17:12:40|Weekend|afternoon|
|     1|    661|     3|978302109|James and the Gia...|     F|  1|        10|  48067|2000-12-31|17:35:09|Weekend|afternoon|
|     1|    914|     3|978301968| My Fair Lady (1964)|     F|  1|        10|  48067|2000-12-31|17:32:48|Weekend|afternoon|
+------+-------+------+---------+--------------------+------+---+----------+-------+----------+--------+-------+---------+
```

Figure 15: Example of RDDs

8.2 Context prefiltering

Prefilter with "Weekday" and "evening" as our example

In the following example context used for filtering is time of day (weekday-evening)movies were watched.

```
# Prefilter
#prefilter.take(3)
sqlContext.createDataFrame(prefilter.take(3)).show()
```

```
+------+-------+------+---------+--------------------+------+---+----------+-------+----------+--------+-------+---------+
|userid|movieid|rating|timestamp|          movietitle|gender|age|occupation|zipcode|      date|    time|weekday|timeofday|
+------+-------+------+---------+--------------------+------+---+----------+-------+----------+--------+-------+---------+
|    19|   1215|     5|980824980|Army of Darkness ...|     M|  1|        10|  48073|2001-01-29|22:23:00|Weekday|  evening|
|    19|   2836|     4|982295349|Outside Providenc...|     M|  1|        10|  48073|2001-02-15|22:49:09|Weekday|  evening|
|    39|   3793|     4|978044402|        X-Men (2000)|     M| 18|         4|  61820|2000-12-28|18:00:02|Weekday|  evening|
+------+-------+------+---------+--------------------+------+---+----------+-------+----------+--------+-------+---------+
```

Figure 16: Data selected by context

8.3 Data split into Training, Validation and Test RDD

The dataset was split into training set (60% data), test set (20% data), and validation set (20% data) for building a recommender system.

```
# Get only userid, movieid and rating
movie_rdd = prefilter.map(lambda x: (x[0], x[1], int(x[2])))

movie_rdd.take(3)

[(19, 1215, 5), (19, 2836, 4), (39, 3793, 4)]

#Split data into training and test datasets
training_RDD, validation_RDD, test_RDD = movie_rdd.randomSplit([6, 2, 2], seed=0L)
```

```
validation_for_predict_RDD = validation_RDD.map(lambda x: (x[0], x[1]))
test_for_predict_RDD = test_RDD.map(lambda x: (x[0], x[1]))

#test_for_predict_RDD.take(5)
#training_RDD.take(3)
```

8.4 Collaborative Filtering implementation by using Alternating Least Squares

(Partially adapted from: https://github.com/jadianes/spark-movie-lens/blob/master/notebooks/building-recommender.ipynb)

Spark MLlib library for Machine Learning provides a Collaborative Filtering implementation by using Alternating Least Squares.

The implementation in MLlib has the following parameters:

- **numBlocks** is the number of blocks used to parallelize computation (set to -1 to auto-configure).
- **rank** is the number of latent factors in the model.
- **iterations** is the number of iterations to run.
- **lambda** specifies the regularization parameter in ALS.
- **implicitPrefs** specifies whether to use the explicit feedback ALS variant or one adapted for implicit feedback data.
- **alpha** is a parameter applicable to the implicit feedback variant of ALS that governs the baseline confidence in preference observations.

```
from pyspark.mllib.recommendation import ALS
import math

seed = 5L
iterations = 10
regularization_parameter = 0.1
ranks = [4, 8, 12]
errors = [0, 0, 0]
err = 0
tolerance = 0.02

min_error = float('inf')
best_rank = -1
best_iteration = -1
for rank in ranks:
    model = ALS.train(training_RDD, rank, seed=seed, iterations=iterations,
                      lambda_=regularization_parameter)
    predictions = model.predictAll(validation_for_predict_RDD).map(lambda r: ((r[0], r[1]), r[2]))
    ratesAndpreds = validation_RDD.map(lambda r: ((int(r[0]), int(r[1])), float(r[2]))).join(predictions)
    error = math.sqrt(ratesAndpreds.map(lambda r: (r[1][0] - r[1][1])**2).mean())
    errors[err] = error
    err += 1
    print 'For rank %s the RMSE is %s' % (rank, round(error,4))
    if error < min_error:
        min_error = error
```

```
    best_rank = rank

print 'The best model was trained with rank %s' % best_rank
```

For rank 4 the RMSE is 0.9351
For rank 8 the RMSE is 0.9414
For rank 12 the RMSE is 0.9483
The best model was trained with rank 4

```
#Let's check our predictions
predictions.take(3)
```

userid	movieid	actual	predicted
2899	3751	5.0	4.3
5220	1554	3.0	3.5
195	1831	3.0	2.4
5468	538	5.0	3.9
3894	2396	4.0	4.1
5749	497	2.0	3.7
5788	2122	1.0	1.3
3815	1307	4.0	4.5
3109	1361	5.0	2.5
5989	2321	5.0	3.3

```
#Let's test the selected model
model = ALS.train(training_RDD, best_rank, seed=seed, iterations=iterations,
                  lambda_=regularization_parameter)
predictions = model.predictAll(test_for_predict_RDD).map(lambda r: ((r[0], r[1]), r[2]))
ratesAndpreds = test_RDD.map(lambda r: ((int(r[0]), int(r[1])), float(r[2]))).join(predictions)
error = math.sqrt(ratesAndpreds.map(lambda r: (r[1][0] - r[1][1])**2).mean())

print 'For test data (from 1M data) the RMSE is %s' % (round(error,4))
```

For testing data (from 1M data) the RMSE is 0.93

9. Add new user into dataset and test recommendation model

9.1 Create new user ratings and add to the dataset

```
#Create a list of ratings for a new user (998)
new_user_ID_1 = 70000

# The format of each line is (userID, movieID, rating)
new_user_ratings_1 = [
 (70000, 242, 4), # Kolya (1996)
 (70000, 51, 3),  # Legends of the Fall (1994)
 (70000, 465, 1),  # Jungle Book, The (1994)
 (70000 , 86, 2), # Remains of the Day, The (1993)
 (70000, 222, 3), # Star Trek: First Contact (1996)
 (70000, 274, 4), # Sabrina (1995)
 (70000, 1042, 3),  # Just Cause (1995)
 (70000, 1184, 3), # Endless Summer 2, The (1994)
 (70000, 265, 2), # Hunt for Red October, The (1990)
 (70000, 302, 3) # L.A. Confidential (1997)
]
```

```
new_user_ratings_RDD_1 = sc.parallelize(new_user_ratings_1)
print 'New user ratings: %s' % new_user_ratings_RDD_1.take(10)
```

```
New user ratings: [(70000, 242, 4), (70000, 51, 3), (70000, 465, 1), (70000, 86, 2), (70000, 222, 3),
(70000, 274, 4), (70000, 1042, 3), (70000, 1184, 3), (70000, 265, 2), (70000, 302, 3)]
```

```
# Merge new user ratings to the existing RDD
data_with_new_ratings_RDD = movie_rdd.union(new_user_ratings_RDD_1)

#Train the ALS model using new dataset and all the parameters we selected before
from time import time

t0 = time()
new_ratings_model = ALS.train(data_with_new_ratings_RDD, best_rank, seed=seed,
                              iterations=iterations, lambda_=regularization_parameter)
tt = time() - t0

print "New model trained in %s seconds" % round(tt,3)
```

```
New model trained in 69.745 seconds
```

9.2 Get recommendations

```
#Getting top recommendations
new_user_ratings_ids = map(lambda x: x[1], new_user_ratings_1) # get just movie IDs
# keep just those not on the ID list
new_user_unrated_movies_RDD = movie_rdd.filter(lambda x: x[0] not in new_user_ratings_ids)\
                              .map(lambda x:(new_user_ID_1, x[0]))
```

```
# Use the input RDD, new_user_unrated_movies_RDD, with new_ratings_model.predictAll() to predict new ratings for the movies
new_user_recommendations_RDD = new_ratings_model.predictAll(new_user_unrated_movies_RDD)

# Transform new_user_recommendations_RDD into pairs of the form (Movie ID, Predicted Rating)
# Use distinct() here
new_user_recommendations_rating_RDD = new_user_recommendations_RDD.distinct().map(lambda x: (x.product, x.rating))
new_user_recommendations_rating_RDD.take(3)
```

```
[(1457, 2.097957421258823),
 (415, 2.725832320884349),
 (2529, 2.880118814695777)]
```

```
#Get movie titles
movies_titles = movie_RDD.map(lambda x: (int(x[1]),x[4]))
movies_titles.take(3)
```

```
[(1193, u"One Flew Over the Cuckoo's Nest (1975)"),
 (661, u'James and the Giant Peach (1996)'),
 (914, u'My Fair Lady (1964)')]
```

```
#Merge movie titles and recommendations for the new user so that the results are meaningful
#movies_titles = ratings_data.map(lambda x: (int(x[0]),x[1]))
new_user_recommendations_rating_title_RDD =
new_user_recommendations_rating_RDD.join(movies_titles)
new_user_recommendations_rating_title_RDD.distinct().sortBy(lambda x: x[1][0],
ascending=False).take(3)
```

```
[(240, (4.436655188048373, u'Hideaway (1995)')),
 (1421, (4.425997558110232, u'Grateful Dead (1995)')),
 (2938,
  (4.372122030145117,
   u'Man Facing Southeast (Hombre Mirando al Sudeste) (1986)'))]
```

10. Sampling data and comparison between 1M and 100K

Collaborative context filtering ALS model was also built with the data from 100K ratings data obtained fromm the 1M dataset. Prefilter was applied on the 1M dataset and 1/10th of the filtered data (approximately from 100K ratings data dfrom the 1M dartaset) was used to build an ALS model. RMSE of this model was evaluated and compared with the 1M filtered data.

One tenth (100k) of the 1M row movielens data was selected to build a recommender system and was compared to that of the one build using the entire dataset.

```
#Sample 1/10 of data (~100K)
movielens100 = movielens.sample(replace = False, frac=0.1, random_state=1 )
#Check the size of the data
```

```
movielens100.shape
#Create Spark dataframe
movie_df100 = sqlContext.createDataFrame(movielens100)
#Create RDD
movie_rdd100 = movie_df100.rdd
```

Filter the data based on the selected context (weekday, weekend) for context-aware recommender system.

```
#Sample 1/10 of data (~100K) of the context filtered data
#movie_rdd100 = movie_df.rdd.sample(False, 0.1, 3045)
# Prefilter
prefilter = movie_rdd100.filter(lambda x: x[11]=='Weekday' and x[12]=='evening')
# Get only userid, movieid and rating
movie_rdd100 = prefilter.map(lambda x: (x[0], x[1], int(x[2])))

print "Total context filtered ratings from 1M data set: %s \
\nOne-tenth of the filtered ratings:  %s" % (movie_rdd.count(), movie_rdd100.count())
#movie_rdd100.take(3)
```

```
Total context filtered ratings from 1M data set: 164283
One-tenth of the filtered ratings:  16596
```

```
#Split data into training and test datasets
training_RDD, validation_RDD, test_RDD = movie_rdd100.randomSplit([6, 2, 2], seed=0L)
validation_for_predict_RDD = validation_RDD.map(lambda x: (x[0], x[1]))
test_for_predict_RDD = test_RDD.map(lambda x: (x[0], x[1]))

#Collaborative Filtering with ALS
seed = 5L
iterations = 10
regularization_parameter = 0.1
ranks = [4, 8, 12]
errors = [0, 0, 0]
err = 0
tolerance = 0.02
min_error = float('inf')
best_rank = -1
best_iteration = -1
for rank in ranks:
    model = ALS.train(training_RDD, rank, seed=seed, iterations=iterations, lambda_=
regularization_parameter)
    predictions = model.predictAll(validation_for_predict_RDD).map(lambda r: ((r[0], r[1]), r[2]))
    ratesAndpreds = validation_RDD.map(lambda r: ((int(r[0]), int(r[1])), float(r[2]))).join(predictions)
    error = math.sqrt(ratesAndpreds.map(lambda r: (r[1][0] - r[1][1])**2).mean())
    errors[err] = error
    err += 1
    print 'For rank %s the RMSE is %s' % (rank, round(error, 4))
    if error < min_error:
        min_error = error
        best_rank = rank
print 'The best model was trained with rank %s' % best_rank
```

```
For rank 4 the RMSE is 3.1745
For rank 8 the RMSE is 1.7926
For rank 12 the RMSE is 1.5463
The best model was trained with rank 12
```

```
#Test the selected model with test set data
predictions = model.predictAll(test_for_predict_RDD).map(lambda r: ((r[0], r[1]), r[2]))
ratesAndpreds = test_RDD.map(lambda r: ((int(r[0]), int(r[1])), float(r[2]))).join(predictions)
error = math.sqrt(ratesAndpreds.map(lambda r: (r[1][0] - r[1][1])**2).mean())

print 'For testing data (from 100 K) the RMSE is %s' % (round(error,4))
```

```
For testing data (from 100 K) the RMSE is 1.4851
```

11. Model comparison

The 100K dataset had the filtered data with 16596 ratings. ALS model built based on this dataset had an RMSE of 1.546 and the test set RMSE of 1.485. On the other hand, the ALS model built with all the filtered data (164283 ratings) from the 1M dataset had an RMSE of 0.948 and the test set RMSE of 0.93. This shows that the model improves as the number of input ratings is increased.

12. Discussion

Large datasets become increasing difficult to process in a timely manner using standard data analytic techniques. Due to this, specialized software and hardware have been developed to allow this processing to occur in a much faster time.

Distributed computing which allows data to be distributed to different clusters and similarly processed by different clusters has been found to vastly reduce processing times.

The Spark engine on Python, which was selected in this case is one of such applications and it allows the programmer to work with a programing language that is familiar to them (Python in this case), while the implementation of the clustering, data sharing and computation is hidden from them.

Spark can be run on Java, Python, R and Scala, but for this exercise Python scripting was selected. Spark can also be run locally on a single computer, on a virtual machine using Linux, or in the cloud, either as a hosted machine (AWS) or as a hosted platform (DataBricks).

13. DataBricks Cluster

Databricks (https://databricks.com) is a fully hosted Apache Spark Cluster that allows users to run Spark in Jupyter on the cloud without having to install anything on their local machines. Databricks provides a cluster of 6GB to each user for free. Databricks was used as it allowed all of users to have a single well defined environment for programming, code testing and more importantly for troubleshooting. The created cluster however, terminated automatically after one

hour of inactivity which can lead to excessive loss of time in reloading the dataset into the clusters.

In summary, DataBricks was used which is a hosted Apache Spark cloud based platform for this project due to its ease of use ans environment stability. Some of the initial programing work was carried out in a local spark cluster, but transferred all this to DataBricks for the final deployment

14. Conclusion

Recommender systems are becoming ubiquitous due to the large amount of possible choices, the increase in the ability to capture both implicit and explicit data about users and items and computing power. However, the large amount of data that is collected makes it very difficult to analyze this data in a timely manner. Distributed data storage and computing (cluster computing) has come to the rescue and allows large datasets to be computed much more efficiently. Spark allows the use of such cluster by allowing a similar user-friendly interface (Python in this instance) while it hides and takes care of the complexities of the cluster computing.

However, this does come with a price. Systems setup to run cluster computing have to be finely tuned in terms of software, hardware, memory, etc and they are not easy to setup or manage. Hosted services such as DataBricks are probably preferred as they are already setup, do not require an installation, and they take care of all the management and configuration issues (zero management). They also allow the user to launch, dynamically scale up or down, and terminate clusters with just a few clicks?

```
#Stop spark cluster
sc.stop()
#
tend = time() - tstart
print "Total computation time: %s" %tend
```

15. Acknowledgements

This work was done in collaboration with Prashanth Padebettu, Adejare Windokun and Xingjia Wu.