databricks MovieLens Recommendation System Comparisons

DATA 612 - Final Project

By Michael D'Acampora and Calvin Wong

In this walkthrough we compare and apply three different algorithms to two datasets used for recommendation systems. The datasets are from the MovieLens repository (https://grouplens.org/datasets/movielens/), and the ones we chose are the small version with 100k records, and the largest version with over 27m records. The data consists of four columns, `userId` , `movieId` , `rating` , and `timestamp` .

We decided to split the work up into two parts, 1) Spark Scala and 2) Python. Since Spark has an impressive Alternating Least Squares Matrix Factorization algorithm, we decided to employ this method to get experience in the Databricks environment and distributed computing in general. There is a shift in approach that is required when using Spark that is different than the similiarities that exist between R and Python. Fist, the base data structure to understand is the Resilient Distributed Dataset, or RDD. Spark takes these RDD's and distributes work across however many nodes you have dedicated to the job. The RDD is designed so Spark can quickly partition pieces as necessary. There are also DataFrames simliar to R and Python, and they exist as part of SparkSQL. Manipulating the data is a bit different, and since Spark was written in Scala, we felt it best to use the origin language and see its paradigm to better understand how Pyspark, Rspark and sparklyr work.

Scala is mostly a functional programming language and one can quickly see its fingerprints as it pertains to Spark. There is plenty of use of the `map()` function, which is used to manipulate data contained in RDDs and Dataframes. It is an anyonymous function similar to `lambda` functions in Python. Interestingly enough, after looking at Pyspark project you see that `lambda` s are often used.

Variable names:

Small -> 100k
Medium -> 1m
Large -> 10m

XL -> 27m

```
import org.apache.spark._
import org.apache.spark.SparkContext._
import org.apache.spark.sql.SparkSession
import org.apache.spark.mllib.recommendation.Rating
import org.apache.spark.ml.evaluation.RegressionEvaluator
import org.apache.spark.ml.recommendation.ALS
```

```
import org.apache.spark._
import org.apache.spark.SparkContext._
import org.apache.spark.sql.SparkSession
import org.apache.spark.mllib.recommendation.Rating
import org.apache.spark.ml.evaluation.RegressionEvaluator
import org.apache.spark.ml.recommendation.ALS
```

First up: Spark Scala with ALS

We created an Azure Databricks cluster with up to 8 worker nodes for this project. After loading our imports we can take a look at the filepaths of the files contained in workspace.

```
display(dbutils.fs.ls("/FileStore/tables/"))
```

path
dbfs:/FileStore/tables/movies.csv
dbfs:/FileStore/tables/movies.dat
dbfs:/FileStore/tables/ratings.csv
dbfs:/FileStore/tables/ratings.dat
dbfs:/FileStore/tables/ratingsMedium.dat
dbfs:/FileStore/tables/ratings_large.csv
dbfs:/FileStore/tables/ratings_large10m.dat
dbfs:/FileStore/tables/ratings_large10m_clean.csv
dbfs:/FileStore/tables/ratings_medium_clean.csv

From here we load the small and large datasets as Spark DataFrames

```
// Load data as Spark DataFrame

// 100k records
val dfSmall = spark.read
  .format("csv")
  .option("header", true)
  .load("dbfs:/FileStore/tables/ratings_small.csv")

// 1m records
val dfMed = spark.read
  .format("csv")
  .option("header", true)
  .load("dbfs:/FileStore/tables/ratings_medium_clean.csv")

// 10m records
val dfLarge = spark.read
  .format("csv")
  .option("header", true)
  .load("dbfs:/FileStore/tables/ratings_large10m_clean.csv")

// 27m records
val dfXl = spark.read
  .format("csv")
  .option("header", true)
  .load("dbfs:/FileStore/tables/ratings_large.csv")
```

```
dfSmall: org.apache.spark.sql.DataFrame = [userId: string, movieId: string ...
2 more fields]
dfMed: org.apache.spark.sql.DataFrame = [userId: string, movieId: string ... 2
more fields]
dfLarge: org.apache.spark.sql.DataFrame = [userId: string, movieId: string ...
2 more fields]
dfXl: org.apache.spark.sql.DataFrame = [userId: string, movieId: string ... 2
more fields]
```

Data prep

```
// Select appropriate columns
val ratingsDFsmall = dfSmall.select("userId", "movieId", "rating")

val ratingsDFmedium = dfMed.select("userId", "movieId", "rating")

val ratingsDFlarge = dfLarge.select("userId", "movieId", "rating")

val ratingsDFxl = dfXl.select("userId", "movieId", "rating")
```

```
ratingsDFsmall: org.apache.spark.sql.DataFrame = [userId: string, movieId: str
ing ... 1 more field]
ratingsDFmedium: org.apache.spark.sql.DataFrame = [userId: string, movieId: st
ring ... 1 more field]
ratingsDFlarge: org.apache.spark.sql.DataFrame = [userId: string, movieId: str
ing ... 1 more field]
ratingsDFxl: org.apache.spark.sql.DataFrame = [userId: string, movieId: string
... 1 more field]
```

After the dataframes are prepped, we split them up into training and test sets at an 80/20 ratio.

```
// Create your training and test data
val splitsSmall = ratingsDFsmall.randomSplit(Array(0.8, 0.2), seed = 123L)
val splitsMedium = ratingsDFmedium.randomSplit(Array(0.8, 0.2), seed = 123L)
val splitsLarge = ratingsDFlarge.randomSplit(Array(0.8, 0.2), seed = 123L)
val splitsXl = ratingsDFxl.randomSplit(Array(0.8, 0.2), seed = 123L)

val (trainingDataSmall, testDataSmall) = (splitsSmall(0), splitsSmall(1))
val (trainingDataMedium, testDataMedium) = (splitsMedium(0), splitsMedium(1))
val (trainingDataLarge, testDataLarge) = (splitsLarge(0), splitsLarge(1))
val (trainingDataXl, testDataXl) = (splitsXl(0), splitsXl(1))
```

```
splitsSmall: Array[org.apache.spark.sql.Dataset[org.apache.spark.sql.Row]] = A
rray([userId: string, movieId: string ... 1 more field], [userId: string, movi
eId: string ... 1 more field])
splitsMedium: Array[org.apache.spark.sql.Dataset[org.apache.spark.sql.Row]] =
Array([userId: string, movieId: string ... 1 more field], [userId: string, mov
ieId: string ... 1 more field])
splitsLarge: Array[org.apache.spark.sql.Dataset[org.apache.spark.sql.Row]] = A
rray([userId: string, movieId: string ... 1 more field], [userId: string, movi
eId: string ... 1 more field])
splitsXl: Array[org.apache.spark.sql.Dataset[org.apache.spark.sql.Row]] = Arra
y([userId: string, movieId: string ... 1 more field], [userId: string, movieI
d: string ... 1 more field])
trainingDataSmall: org.apache.spark.sql.Dataset[org.apache.spark.sql.Row] = [u
serId: string, movieId: string ... 1 more field]
testDataSmall: org.apache.spark.sql.Dataset[org.apache.spark.sql.Row] = [userI
d: string, movieId: string ... 1 more field]
trainingDataMedium: org.apache.spark.sql.Dataset[org.apache.spark.sql.Row] =
[userId: string, movieId: string ... 1 more field]
testDataMedium: org.apache.spark.sql.Dataset[org.apache.spark.sql.Row] = [user
Id: string, movieId: string ... 1 more field]
trainingDataLarge: org.apache.spark.sql.Dataset[org.apache.spark.sql.Row] = [u
serId: string, movieId: string ... 1 more field]
testDataLarge: org.apache.spark.sql.Dataset[org.apache.spark.sql.Row] = [userI
d: string, movieId: string ... 1 more field]
trainingDataXl: org.apache.spark.sql.Dataset[org.apache.spark.sql.Row] = [user
Id: string, movieId: string ... 1 more field]
testDataXl: org.apache.spark.sql.Dataset[org.apache.spark.sql.Row] = [userId:
string, movieId: string ... 1 more field]
```

Since the dataframes were all string values we have to convert to numeric values for this exercise. Note the use of the `map()` function that grabs the column values and converts within the `Rating` class in Mllib. These training and test sets are created for both the small and large datasets.

```
// spark implicits needed here for serializing
import spark.implicits._

// CONVERT TEST SETS TO NUMERIC
// ==========================================
val trainingSetSmall = trainingDataSmall.map(row => {
val userId = row.getString(0)
val movieId = row.getString(1)
val ratings = row.getString(2)
Rating(userId.toInt, movieId.toInt, ratings.toDouble)
})

val trainingSetMedium = trainingDataMedium.map(row => {
val userId = row.getString(0)
val movieId = row.getString(1)
val ratings = row.getString(2)
Rating(userId.toInt, movieId.toInt, ratings.toDouble)
})

val trainingSetLarge = trainingDataLarge.map(row => {
val userId = row.getString(0)
val movieId = row.getString(1)
val ratings = row.getString(2)
Rating(userId.toInt, movieId.toInt, ratings.toDouble)
})

val trainingSetXl= trainingDataXl.map(row => {
val userId = row.getString(0)
val movieId = row.getString(1)
val ratings = row.getString(2)
Rating(userId.toInt, movieId.toInt, ratings.toDouble)
})


// CONVERT TEST SETS TO NUMERIC
// ==========================================
val testSetSmall = testDataSmall.map(row => {
val userId = row.getString(0)
val movieId = row.getString(1)
val ratings = row.getString(2)
Rating(userId.toInt, movieId.toInt, ratings.toDouble)
})

val testSetMedium = testDataMedium.map(row => {
val userId = row.getString(0)
val movieId = row.getString(1)
val ratings = row.getString(2)
Rating(userId.toInt, movieId.toInt, ratings.toDouble)
})

val testSetLarge = testDataLarge.map(row => {
val userId = row.getString(0)
val movieId = row.getString(1)
val ratings = row.getString(2)
Rating(userId.toInt, movieId.toInt, ratings.toDouble)
})

val testSetXl = testDataXl.map(row => {
val userId = row.getString(0)
val movieId = row.getString(1)
val ratings = row.getString(2)
Rating(userId.toInt, movieId.toInt, ratings.toDouble)
})
```

```
import spark.implicits._
trainingSetSmall: org.apache.spark.sql.Dataset[org.apache.spark.mllib.recommen
dation.Rating] = [user: int, product: int ... 1 more field]
trainingSetMedium: org.apache.spark.sql.Dataset[org.apache.spark.mllib.recomme
ndation.Rating] = [user: int, product: int ... 1 more field]
trainingSetLarge: org.apache.spark.sql.Dataset[org.apache.spark.mllib.recommen
dation.Rating] = [user: int, product: int ... 1 more field]
trainingSetXl: org.apache.spark.sql.Dataset[org.apache.spark.mllib.recommendat
ion.Rating] = [user: int, product: int ... 1 more field]
testSetSmall: org.apache.spark.sql.Dataset[org.apache.spark.mllib.recommendati
on.Rating] = [user: int, product: int ... 1 more field]
testSetMedium: org.apache.spark.sql.Dataset[org.apache.spark.mllib.recommendat
ion.Rating] = [user: int, product: int ... 1 more field]
testSetLarge: org.apache.spark.sql.Dataset[org.apache.spark.mllib.recommendati
on.Rating] = [user: int, product: int ... 1 more field]
testSetXl: org.apache.spark.sql.Dataset[org.apache.spark.mllib.recommendation.
Rating] = [user: int, product: int ... 1 more field]
```

Model creation

Now that we are ready to feed a model, we create on. As mentioned above we are using the Alternating Least Squares (ALS) Matrix Factorization model. `val als` instantiates the model with our parameters, and afterwards is fit to the small and large training data.

Once the training data is fit, we use that model to predict against the test data.

```
// Create model
val als = new ALS()
.setMaxIter(5)
.setRegParam(0.01)
.setUserCol("user")
.setItemCol("product")
.setRatingCol("rating")

// Fit models
val smallModel = als.fit(trainingSetSmall)
val mediumModel = als.fit(trainingSetMedium)
val largeModel = als.fit(trainingSetLarge)
val xlModel = als.fit(trainingSetXl)

// Predict on the testSets
val smallPredictions = smallModel.transform(testSetSmall).na.drop()
val mediumPredictions = smallModel.transform(testSetMedium).na.drop()
val largePredictions = largeModel.transform(testSetLarge).na.drop()
val xlPredictions = xlModel.transform(testSetXl).na.drop()
```

```
als: org.apache.spark.ml.recommendation.ALS = als_572e686559a0
smallModel: org.apache.spark.ml.recommendation.ALSModel = als_572e686559a0
mediumModel: org.apache.spark.ml.recommendation.ALSModel = als_572e686559a0
largeModel: org.apache.spark.ml.recommendation.ALSModel = als_572e686559a0
xlModel: org.apache.spark.ml.recommendation.ALSModel = als_572e686559a0
smallPredictions: org.apache.spark.sql.DataFrame = [user: int, product: int
... 2 more fields]
mediumPredictions: org.apache.spark.sql.DataFrame = [user: int, product: int
... 2 more fields]
largePredictions: org.apache.spark.sql.DataFrame = [user: int, product: int
... 2 more fields]
xlPredictions: org.apache.spark.sql.DataFrame = [user: int, product: int ... 2
more fields]
```

Evaluation

We calculated Root Mean Squared Error (RMSE) and Mean Absolute Error (MAE) for each of the three datasets with the outputs below.

```
// Evaluate models
val evaluatorRMSE = new RegressionEvaluator()
.setMetricName("rmse")
.setLabelCol("rating")
.setPredictionCol("prediction")

val evaluatorMAE = new RegressionEvaluator()
.setMetricName("mae")
.setLabelCol("rating")
.setPredictionCol("prediction")

// Calculate RMSE and MAE on both predictions
val rmseSmall = evaluatorRMSE.evaluate(smallPredictions)
val rmseMedium = evaluatorRMSE.evaluate(mediumPredictions)
val rmseLarge = evaluatorRMSE.evaluate(largePredictions)
val rmseXL = evaluatorRMSE.evaluate(xlPredictions)

val maeSmall = evaluatorMAE.evaluate(smallPredictions)
val maeMedium = evaluatorMAE.evaluate(mediumPredictions)
val maeLarge = evaluatorMAE.evaluate(largePredictions)
val maeXL = evaluatorMAE.evaluate(xlPredictions)
```

```
evaluatorRMSE: org.apache.spark.ml.evaluation.RegressionEvaluator = regEval_71
4189caf032
evaluatorMAE: org.apache.spark.ml.evaluation.RegressionEvaluator = regEval_d1c
00e0cd841
rmseSmall: Double = 1.0797295239135771
rmseMedium: Double = 1.5841092407769148
rmseLarge: Double = 0.8228531542346977
rmseXL: Double = 0.8414888221626028
maeSmall: Double = 0.8121119030481311
maeMedium: Double = 1.219735398310547
maeLarge: Double = 0.6336740600406011
maeXL: Double = 0.6422385254035295
```

As you can see there is a significant drop in RMSE and MAE (i.e. increase in accuracy) as the model gets to feed on more data. We observed 23.8% better accuracy for RMSE between the small (100k) and large (1m) datasets. Similary for the same two datasets we saw a 22.0% improvement for MAE.

Next up: Modeling in Python

Let's compare Spark's ALS alogrithm with some others from Python's sci-kit surprise library. We have chosen Singular Vector Decomposition (SVD), Non-negative Matrix Factorization (NMF), and k-Nearest Neighbors (KNN). Considering memory constraints we can only run the models on the small (100k), medium (1m), and large (10m) datasets. Each model is run twice to see if we can improve the error score; any more than that and we will encounter memory issues.

Import the data

Interesting to note that some of these files were `.csv` and other were `.dat` , and were separated by `::` instead of a comma, which was difficult to parse in spark. What we did was use pandas to parse the `.dat` file and saved it to Databricks be used in the spark section above.

```
%python
import numpy as np
import pandas as pd

# Load data -> small, medium and large ratings data
pdSmall = pd.read_csv("/dbfs/FileStore/tables/ratings_small.csv")


pdMedium = pd.read_csv("/dbfs/FileStore/tables/ratingsMedium.dat",
                       sep="::",
                       names=['userId',  'movieId', 'rating', 'timestamp'])

pdLarge = pd.read_csv("/dbfs/FileStore/tables/ratings_large10m.dat",
                       sep="::",
                       names=['userId',  'movieId', 'rating', 'timestamp'])

#pdXL = pd.read_csv("/dbfs/FileStore/tables/ratings_large.csv")

pdMedium.head()
```

```
/local_disk0/tmp/1562781367646-0/PythonShell.py:10: ParserWarning: Falling bac
k to the 'python' engine because the 'c' engine does not support regex separat
ors (separators > 1 char and different from '\s+' are interpreted as regex); y
ou can avoid this warning by specifying engine='python'.
  import resource
/local_disk0/tmp/1562781367646-0/PythonShell.py:14: ParserWarning: Falling bac
k to the 'python' engine because the 'c' engine does not support regex separat
ors (separators > 1 char and different from '\s+' are interpreted as regex); y
```

```
ou can avoid this warning by specifying engine='python'.
  import traceback
Out[1]:
   userId  movieId  rating  timestamp
0       1     1193       5  978300760
1       1      661       3  978302109
2       1      914       3  978301968
3       1     3408       4  978300275
4       1     2355       5  978824291
```

```
%python
# Take a quick look at the size of each dataset to make sure our imports are
correct
print(pdSmall.shape, pdMedium.shape, pdLarge.shape)
```

```
(100836, 4) (1000209, 4) (10000054, 4)
```

Serialize the data for scikit-surprise

The data needs to be formatted a certain way in order for the scikit surprise's algorithms to process them, which is what we are doing in the code chunk below.

```
%python
from surprise import Reader
from surprise import Dataset
from surprise import SVD
from surprise import NMF
from surprise import KNNBasic
from surprise.model_selection import cross_validate


# Convert files back to string values for Scikit-surprise API
pdSmall = pdSmall[['userId', 'movieId', 'rating']]
pdMedium = pdMedium[['userId', 'movieId', 'rating']]
pdLarge = pdLarge[['userId', 'movieId', 'rating']]

# Define a reader for a custom dataset
reader = Reader(rating_scale=(1,5))

# These datasets are the model-ready
smallData = Dataset.load_from_df(pdSmall, reader)
mediumData = Dataset.load_from_df(pdMedium, reader)
largeData = Dataset.load_from_df(pdLarge, reader)
```

Singular Value Decomposition

Equivalent to Probabilistic Matrix Factorization (http://papers.nips.cc/paper/3208-probabilistic-matrix-factorization.pdf)

```
%python
import random
random.seed(123)

svd = SVD()

print("====================")
print("DATASET: SMALL (100K)")
cv_svd_small = cross_validate(svd, smallData, measures=['RMSE', 'MAE'], cv=2,
verbose=True)
print("====================" + "\n")

print("DATASET: MEDIUM (1M)")
cv_svd_medium = cross_validate(svd, mediumData, measures=['RMSE', 'MAE'], cv=2,
verbose=True)
print("====================" + "\n")

print("DATASET: LARGE (10M)")
cv_svd_large = cross_validate(svd, largeData, measures=['RMSE', 'MAE'], cv=2,
verbose=True)
print("====================")
```

```
====================
DATASET: SMALL (100K)
Evaluating RMSE, MAE of algorithm SVD on 2 split(s).

                  Fold 1  Fold 2  Mean    Std
RMSE (testset)    0.8913  0.8903  0.8908  0.0005
MAE (testset)     0.6882  0.6843  0.6862  0.0019
Fit time          3.37    3.37    3.37    0.00
Test time         0.53    0.54    0.53    0.01
====================

DATASET: MEDIUM (1M)
Evaluating RMSE, MAE of algorithm SVD on 2 split(s).

                  Fold 1  Fold 2  Mean    Std
RMSE (testset)    0.9029  0.9032  0.9030  0.0002
MAE (testset)     0.7111  0.7120  0.7115  0.0005
Fit time          33.87   34.46   34.17   0.30
```

```
Test time         5.14    5.10    5.12    0.02
====================

DATASET: LARGE (10M)
Evaluating RMSE, MAE of algorithm SVD on 2 split(s).

                  Fold 1  Fold 2  Mean    Std
RMSE (testset)    0.8258  0.8262  0.8260  0.0002
MAE (testset)     0.6350  0.6354  0.6352  0.0002
Fit time          341.00  352.47  346.73  5.74
Test time         94.27   81.77   88.02   6.25
====================
```

Non-negtative Matrix Factorization collaborative filtering algorithm.

Similar to SVD.

```
%python

nmf = NMF()

print("====================")
print("DATASET: SMALL (100K)")
cv_nmf_small = cross_validate(nmf, smallData, measures=['RMSE', 'MAE'], cv=2,
verbose=True)
print("====================" + "\n")

print("DATASET: MEDIUM (1M)")
cv_nmf_medium = cross_validate(nmf, mediumData, measures=['RMSE', 'MAE'], cv=2,
verbose=True)
print("====================" + "\n")

print("DATASET: LARGE (10M)")
cv_nmf_large = cross_validate(nmf, largeData, measures=['RMSE', 'MAE'], cv=2,
verbose=True)
print("====================")
```

```
====================
DATASET: SMALL (100K)
Evaluating RMSE, MAE of algorithm NMF on 2 split(s).

                  Fold 1  Fold 2  Mean    Std
RMSE (testset)    0.9618  0.9535  0.9577  0.0042
```

```
MAE (testset)     0.7375  0.7339  0.7357  0.0018
Fit time          4.58    4.45    4.51    0.06
Test time         0.40    0.39    0.40    0.00
====================

DATASET: MEDIUM (1M)
Evaluating RMSE, MAE of algorithm NMF on 2 split(s).

                  Fold 1  Fold 2  Mean    Std
RMSE (testset)    0.9280  0.9267  0.9274  0.0006
MAE (testset)     0.7327  0.7315  0.7321  0.0006
Fit time          40.42   40.67   40.55   0.13
Test time         6.59    9.18    7.88    1.30
====================

DATASET: LARGE (10M)
Evaluating RMSE, MAE of algorithm NMF on 2 split(s).

                  Fold 1  Fold 2  Mean    Std
RMSE (testset)    0.8780  0.8789  0.8784  0.0005
MAE (testset)     0.6784  0.6792  0.6788  0.0004
Fit time          407.35  411.23  409.29  1.94
Test time         93.18   84.94   89.06   4.12
====================
```

k-NN basic collaborative filtering algorithm

```
%python
# Instantiate KNN algo
knn = KNNBasic()

# Cross validate and score model on both datasets
print("====================")
print("DATASET: SMALL (100K)")
cv_knn_small = cross_validate(knn, smallData, measures=['RMSE', 'MAE'], cv=2,
verbose=True)
print("====================" + "\n")

# print("DATASET: MEDIUM (1M)")
# cv_knn_medium = cross_validate(knn, mediumData, measures=['RMSE', 'MAE'],
cv=2, verbose=True)
# print("====================" + "\n")

# print("DATASET: LARGE (10M)")
# cv_knn_large = cross_validate(knn, largeData, measures=['RMSE', 'MAE'], cv=2,
verbose=True)
# print("====================")
```

```
====================
DATASET: SMALL (100K)
Computing the msd similarity matrix...
Done computing similarity matrix.
Computing the msd similarity matrix...
Done computing similarity matrix.
Evaluating RMSE, MAE of algorithm KNNBasic on 2 split(s).

                  Fold 1  Fold 2  Mean    Std
RMSE (testset)    0.9785  0.9754  0.9769  0.0016
MAE (testset)     0.7503  0.7489  0.7496  0.0007
Fit time          0.06    0.08    0.07    0.01
Test time         3.83    3.22    3.52    0.30
====================
```

The Results are in...

RMSE Table

Model	100K	1M	10M	20M
ALS	1.0797	1.5841	0.8229	0.8415

Model	100K	1M	10M	20M
SVD	0.8908	0.9030	0.8260	NA
NMF	0.9577	0.9274	0.8784	NA
KNN	0.9769	NA	NA	NA

MAE Table

Model	100K	1M	10M	20M
ALS	0.8121	1.2197	0.6337	0.6422
SVD	0.6862	0.7115	0.6352	NA
NMF	0.7357	0.7321	0.6788	NA
KNN	0.7496	NA	NA	NA

```
The spark context has stopped and the driver is restarting. Your notebook wi
ll be automatically reattached.
```