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.

Alternating Least Squares (ALS) is an iterative optimization process where for every iteration try to arrive closer and closer to a factorized representation of our original data. Just like SVD, ALS is a matrix factorization algorithm but can run itself in a parallel fashion. ALS is implemented in Apache Spark ML and built for a large-scale collaborative filtering. Like most factorization methods, ALS does a pretty good job at solving scalability and sparseness of incomplete data, also, it is a simple algorithm and scales well to very large datasets.

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

path	name	size
dbfs:/FileStore/tables/MovieLens_Recommendation_System_Comparisons-ab1b6.scala	MovieLens_Recommendation_System_Comparisons-ab1b6.scala	15984
dbfs:/FileStore/tables/movies.csv	movies.csv	181870
dbfs:/FileStore/tables/ratings.csv	ratings.csv	380355

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")
```

```
org.apache.spark.sql.AnalysisException: Path does not exist: dbfs:/FileStore/tables/ratings_smal
l.csv;
```

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")
```

```
Command skipped
```

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))
```

```
  Command skipped
```

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.

```scala
// 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)
})
```

```
  Command skipped
```

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()
```

```
  Command skipped
```

Evaluation

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

For performance evaluation, the Mean Absolute Error (MAE) and the Rooted Mean Squared Error (RMSE) are two widely used scores. The choice between the above two criteria depends on the particular application. Currently in practice, MAE is common for Collaborative Filtering (CF) algorithms with discrete ratings, while RMSE is popular for CF approaches that generate real valued outputs. However, we included both scores to evaluate our findings.

```
// 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)
```

```
  Command skipped
```

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()
```

```
  Command skipped
```

```
%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)
```

```
  Command skipped
```

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)
```

```
  Command skipped
```

Singular Value Decomposition

Singular Value Decomposition (SVD) is a matrix factorization technique that is used to reduce the number of features of a data set by reducing space dimensions from N to K where K < N. It is based on the matrix factorization technique that utilizes a Latent factor model instead of calculating the similarities to predict ratings, the Latent factor trains a model on some known data, by transforming both items and users to the same Latent factor space, thus making them directly comparable. This can be demonstrated by the following equation where A is factored into the product of three matrices A = UDV T where the columns of U and V are orthonormal and the matrix D is diagonal with positive real entries. SVD like the KNN model can be computationally expensive to run. SVD is an effective optimization algorithm and generally a higher performing model. However, its ability to scale drops rapidly with a growing matrix size. We encountered this issue when processing the 20M dataset.

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("====================")
```

```
  Command skipped
```

Non-negative Matrix Factorization

Non-negative matrix factorization (NMF or NNMF), also non-negative matrix approximation is a group of algorithms in multivariate analysis and linear algebra where a matrix V is factorized into (usually) two matrices W and H, with the property that all three matrices have no negative elements. This non-negativity makes the resulting matrices easier to inspect. However, non-negativity also becomes the limitation when dealing with datasets.

```
%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("===================")
```

```
  Command skipped
```

k-NN Collaborative Filtering

K-nearest neighbors (KNN) falls in the supervised learning family of algorithms. The KNN classifier is a non-parametric and instance-based algorithm. For instance, a labelled dataset consisting of training observations (x, y) is used to capture the relationship between x and y. The goal is to learn a function h: X→Y so that given an unseen observation x, h(x) can confidently predict the corresponding output y.

The disadvantages of the KNN occurs because of the minimal training phase which comes both at a higher memory and computational cost. Since the entire data set must be stored, there is a high computational cost during test time as classifying a given observation requires a run-down of the whole data set.

```
%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("===================")
```

```
  Command skipped
```

The Results are in...

RMSE Table

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

CONCLUSION

The hypothesis prior to our analysis was that the MovieLens dataset, which is extremely sparse, would work much better on ALS as the dataset grew. There are also opportunities to tune ALS to perform better through iterative parametric testing. By picking both the proper number of features and the proper algorithmic parameter values, there should be improved scoring. However, we are under a time constraint and could not attempt parametric tuning.

The results show an overall improvement in performance for all algorithms and mathematical improvement models as the data size grew. ALS produced the best rating among the algorithms with an RMSE of 0.8229 with the 10M data set, which was the largest dataset in our testing that multiple algorithms could process without encountering memory issues. Surprisingly ALS was also the poorest performer with the smallest datasets, with an RMSE of 1.5841 on the 1M dataset. The other surprise was the lowest RMSE did not come from our largest dataset. Our hypothesis was the model would always improve as the data set grew. The less complex algorithms like NMF and KNN outperformed ALS on the smallest (100k) dataset. SVD and NMF performed well at the 1M because ALS was the poorest performer at this range.

The RMSE values indicate how well these algorithms predicts ratings for non-rated items. Based on these results, we suggest utilizing ALS out of the four algorithms for a recommender system for large data production cases. When complexity and scarcity become a factor, Spark has the ability to cluster process and apply the ALS under a much more effective system. Therefore, execution time and input data size are null factors as it pertains to big data.

Future

A possible area for future research would be to investigate whether the non-parametric model would produce proportionally better results, both complexity and RMSE considered, than the more complex algorithms such as SVD and ALS. The basis in NMF is composed of vectors with positive elements while the basis in SVD can have positive or negative values.

ALS offers two advantages, the ability to parallelize much easier than other models and how it converges quicker. We would attempt parametric tuning to improve the model and determine if the RSME can be reduced. Due to time constraints, we were only able to run the model once. By running the algorithm several times for testing different hyperparameter values, we may be able to show definitive big data capabilities.

References

Zakka, K. (2016, July 13). A Complete Guide to K-Nearest-Neighbors with Applications in Python and R. Retrieved July 14, 2019, from https://kevinzakka.github.io/2016/07/13/k-nearest-neighbor/ (https://kevinzakka.github.io/2016/07/13/k-nearest-neighbor/)

Huang, S. (2018, January 23). Introduction to Recommender System. Part 1 (Collaborative Filtering, Singular Value Decomposition). Retrieved July 14, 2019, from https://hackernoon.com/introduction-to-recommender-system-part-1-collaborative-filtering-singular-value-decomposition-44c9659c5e75 (https://hackernoon.com/introduction-to-recommender-system-part-1-collaborative-filtering-singular-value-decomposition-44c9659c5e75)

Kohler, V. (2018, July 10). ALS Implicit Collaborative Filtering - Rn Engineering. Retrieved from https://medium.com/radon-dev/als-implicit-collaborative-filtering-5ed653ba39fe (https://medium.com/radon-dev/als-implicit-collaborative-filtering-5ed653ba39fe)

```
Command skipped
```