Lab Assignment 4 **Big Data**
Due Date: 01/03/2019 Prof. P. Boncz
Jeroen Schmidt (122808560) & Dimitris Michailidis (12325929) TA: Dean De Leo
Group 10

Question 1 - Short Description of the Alternating Least Squares (ALS) method

The alternating least squares (ALS) algorithm factorizes a given matrix R into two factors U and V such that $R \approx U^T V$. In the case of collaborative filtering, U represents the users matrix and V the ratings matrix, with R being the original, complete user-reviews matrix. To solve this problem single value decomposition is usually employed. However, since R is very sparse, conventional SVD techniques cannot be used[1]. As a result, in order to find the U and V matrix, ALS minimizes the following equation:

$$\arg\min_{U,V} \sum_{\{i,j|r_{i,j}\neq 0\}} \left(r_{i,j} - u_i^T v_j\right)^2 + \lambda \left(\sum_i n_{u_i} \|u_i\|^2 + \sum_j n_{v_j} \|v_j\|^2 \right) \quad (1)$$

Where the ith element of the user matrix is denoted by u_i, the ith element of the item matrix is denoted by v_i, and the value in row i and column j of the R matrix is denoted by $r_{i,j}$. Regularization is also used and defined by the factor λ.

ALS solves the above problem as follows[1]:

- **Step 1:** Initialise V by assign the average rating for that movie as the first row, and small random numbers for the remaining entries.
- **Step 2:** Fix V, solve for U by minimising the objective function.
- **Step 3:** Fix U, solve for V by minimising the objective function on the same manner.
- **Step 4:** Repeat (2) and (3) until the following stopping criterion is satisfied: After one round of U and V updating, if the change in RMSE is less than 1 bps, the iteration stopes.

ALS is favorable over other techniques in the following cases[2]:

- When the system can parallelization.
- When the system is centered on implicit data.

Question 2 - Model Training Using Spark ALS

To train the model and test in on the given data, we use the code listed in Listing 5. The resulting output for the Mean Squared Error is the following:

```
Root mean square error = 1.14495781844
```

Listing 1: Code used to train the ALS model.

```
from pyspark.ml.recommendation import ALS
from pyspark.sql import Row

# load the training & test datasets...
def load(prefix):
```

```
        lines = spark.read.text(prefix).rdd
        parts = lines.map(lambda row: row.value.split("\t"))
        ratingsRDD = parts.map(lambda p: Row(userId=int(p[0]),
                                songId=int(p[1]), rating=float(p[2])))
        return spark.createDataFrame(ratingsRDD).cache()

training = load("s3://bads-music-dataset/train_9.txt.gz")
test = load("s3://bads-music-dataset/test_9.txt.gz")
# instantiate the ALS object
als = ALS(
    maxIter=10, rank=10, userCol="userId", itemCol="songId", ratingCol="rating",
    coldStartStrategy="drop"
    )
# train the model over the training dataset
model = als.fit(training)
# predict the ratings on the testing dataset
predictions = model.transform(test)

# Find the root mean square error (RMSE) of the predictions over the given
# ratings in the training set
from pyspark.ml.evaluation import RegressionEvaluator
rmse = RegressionEvaluator(metricName="rmse", labelCol="rating",
                                predictionCol="prediction")
print("Root mean square error = " + str(rmse.evaluate(predictions)))
```

Question 3 - Impact of the parameters `rank` and `maxIter`

The `rank` parameter refers to the number of latent(hidden) variables used to represent your original scores in the R matrix. *Rank* specifies the dimensions of variables U and V; such that $U \in R^{n,m}$ and $V \in R^{m,l}$ where `rank` is m.

We ran ALS for Rank Values of $[2, 5, 10, 15, 20]$(refer to appendix A for code). The respective RMSE scores are shown in Figure 1. We can see that the RMSE score decreases as the rank increases; this is something we expect and it shows the RMSE score increase as we increases the rank, i.e. the model is better able to capture information through its latent variables as we increase the rank. However, increasing the rank also increases the memory consumption and the needed computation time. Figure 3 shows the total needed training time for different rank values.

We also ran ALS for Max Iterations values of $[1, 2, 3, 4, 5, 6, 7, 8, 9, 10]$(refer to appendix A for code). Max Iterations parameter adds another stopping condition to the algorithm mentioned in Question 1. Apart from it stopping when the RMSE score does not reduce, this parameter will also force it to stop after the total number of iterations reaches the set value. The RMSE scores for each value are shown in Figure 2. As we can see in this case, only 2 iterations are enough for the RMSE score to be close to the optimal for the current setting. Increasing it after that does not have a significant effect on the score. However, as shown in Figure 4, increasing the max iterations increases the total time needed for training. We can observe an almost linear relationship between them.

It should be noted that when we ran the timing benchmarks we ensured that the dataframes were first cached in memory before the models were trained and timed.

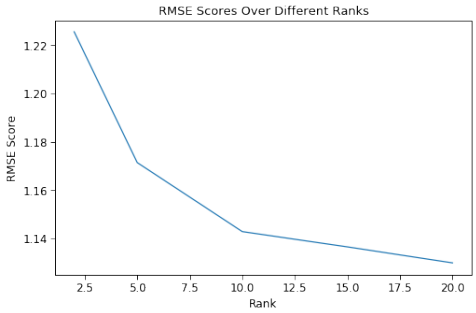

Figure 1: RMSE Scores of Different Values of Rank

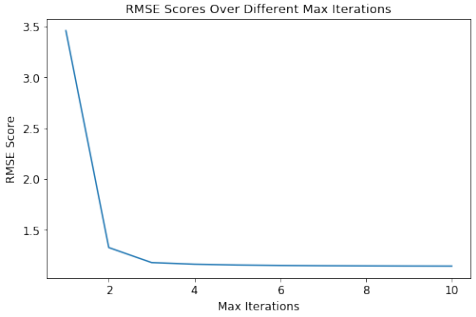

Figure 2: RMSE Scores for Different Max Iterations

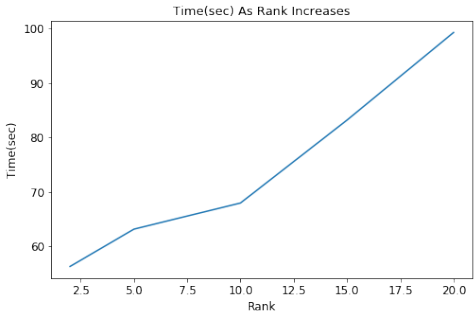

Figure 3: Time(sec) as Rank Increases

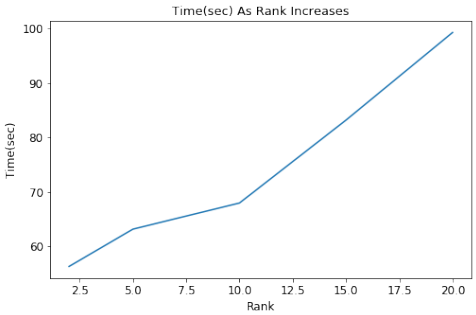

Figure 4: Time(sec) as MaxIter Increases

Question 4 - Description of the Other ALS Parameters

The constructor of the ALS object contains other parameters as well, the most important of which are presented in Table 1[1].

A short description of each one of these parameters is presented below:

- regParam: the λ regularization parameter as shown in Equation 1. Controls how much will large parameters be penalised. It can have an impact on the model's score.
- numUserBlocks/numItemBlocks: number of blocks that user and item matrix will be divided in, to control parallelism. Setting it to -1 will let MLLib automatically determine it. It impacts the performance of the ALS training process.
- implicitPrefs: controls the usage of variants of the ALS algorithm. Specifically, it determines whether to use the explicit feedback ALS or the one for implicit feedback data. Explicit feedback in this case refers to explicit preferences from the user, e.g. a user giving ratings to items. When this data is not available, implicit feedback is used instead, which refers to actions like views, clicks, purchases, etc.
- alpha: if implicit feedback is set, alpha is a constant hyperparameter that reflects how much we value observed versus unobserved implicit events[2]. It can impact the final score.

[1] http://spark.apache.org/docs/2.2.0/api/python/_modules/pyspark/ml/recommendation.html
[2] https://amsterdam.luminis.eu/2016/12/04/alternating-least-squares-implicit-feedback-search-alpha/

- nonnegative: controls whether to use a non-negative constraint for the ratings matrice. If for example there is negative feedback from a user to an item and we want to also predict negative ratings, then it should be set to true. It can have impact on the score if not used wisely.
- coldStartStrategy: cold start refers to new users or items that have no ratings associated with them. Setting it to 'drop' will drop any rows in the DataFrame of predictions that contain NaN values. This decision can impact the model's score.

Table 1: Parameters of the ALS object of pyspark.ml.recommendation package.

Parameter	**Documentation Description**
regParam	regularization parameter(λ)
numUserBlocks	number of user blocks
numItemBlocks	number of item blocks
implicitPrefs	whether to use implicit preference
alpha	alpha for implicit preference
nonnegative	whether to use nonnegative constraint for least squares
coldStartStrategy	strategy for dealing with unknown or new users/items at prediction time. Supported values: 'nan', 'drop'.

References

[1] Yunhong Zhou, Dennis Wilkinson, Robert Schreiber, and Rong Pan. Large-scale parallel collaborative filtering for the netflix prize. In *International conference on algorithmic applications in management*, pages 337–348. Springer, 2008.

[2] Yehuda Koren, Robert Bell, and Chris Volinsky. Matrix factorization techniques for recommender systems. *Computer*, (8):30–37, 2009.

A Code for Question 3

Listing 2: Code Used to See Behavior of *Rank*

```
%pyspark
from pyspark.ml.tuning import ParamGridBuilder, TrainValidationSplit

data = training.union(test).cache()
rmse = RegressionEvaluator(metricName="rmse", labelCol="rating",
                           predictionCol="prediction")

paramGrid = ParamGridBuilder()\
    .addGrid(als.rank, [2,5,10,15,20]) \
    .build()

tvs = TrainValidationSplit(estimator=als,
                           estimatorParamMaps=paramGrid,
                           evaluator=rmse,
                           # 80% of the data will be used for training,
                           #20% for validation.
                           trainRatio=0.8)

model = tvs.fit(data)
model.validationMetrics
```

Listing 3: Code Used to See Behaviour of *MaxItr*

```
%pyspark
from pyspark.ml.tuning import ParamGridBuilder, TrainValidationSplit

data = training.union(test).cache()
rmse = RegressionEvaluator(metricName="rmse", labelCol="rating",
                           predictionCol="prediction")



paramGrid = ParamGridBuilder()\
    .addGrid(als.maxIter , [1,2,3,4,5,6,7,8,9,10])\
    .build()

tvs = TrainValidationSplit(estimator=als,
                           estimatorParamMaps=paramGrid,
                           evaluator=rmse,
                           # 80% of the data will be used for training,
                           #20% for validation.
                           trainRatio=0.8)

model = tvs.fit(data)
model.validationMetrics
```

Listing 4: Code Used to See Behaviour of *Rank* Execution Time

```
%pyspark
import time
times = []
for rank in [2,5,10,15,20]:

    als = ALS(maxIter=10, rank=rank, userCol="userId", itemCol="songId",
    ratingCol="rating", coldStartStrategy="drop")
    start = time.time()
    als.fit(training)
    end = time.time()

    elapse = end - start
    times.append(elapse)
```

Listing 5: Code Used to See Behaviour of *MaxItr* Execution Time

```
%pyspark
import time
times_maxitr = []
for max_itr in [1,2,3,4,5,6,7,8,9,10]:

    als = ALS(maxIter=max_itr, rank=10, userCol="userId", itemCol="songId",
    ratingCol="rating", coldStartStrategy="drop")
    start = time.time()
    als.fit(training)
    end = time.time()

    elapse = end - start
    times_maxitr.append(elapse)
```