Recommender Systems

Techniques used in Spark

- Alternating Least Squares (ALS)
  - Technique used - Collaborative Filtering.
  - Recommendations based on which items, users **interacted** with in the past.
  - It does not require or use any additional features about the users or the items.
- Frequent Pattern Mining
  - Technique used - Market basket Analysis.
- lower-level matrix factorization method (for RDD)

Alternating Least Squares (ALS)

- K-dimensional feature: Dot product of User vector and Item vector.
- dot product of the two vectors approximates the user rating.

- Input data set will have the following:
  - User ID
  - Item ID
  - Rating ID (Implicit and Explicit)

- Model will produce feature vectors to predict user rating for Items that are not rated yet.

Alternating Least Squares (ALS)

Disadvantages:

- It gives preferences to things which are common and things have lot of information.
- If new items are included, this algorithm does not recommend to many people.
- *Cold start problem*: the algorithm won’t know what to recommend to new users as they may not have any ratings in the training set.

Advantages:

- Scalable:
  - It can scale millions of users, millions of items, and billions of ratings.

Hyperparameters

Params	Values	Description
rank	default value is 10	Dimension of the feature vectors learned for users and items

Hyperparameters

Params	Values	Description
rank	default value is 10	Dimension of the feature vectors learned for users and items
alpha	default of 1.0	When training on **implicit** feedback (behavioral observations), the alpha sets a baseline confidence for preference.

Hyperparameters

Params	Values	Description
rank	default value is 10	Dimension of the feature vectors learned for users and items
alpha	default of 1.0	When training on **implicit** feedback (behavioral observations), the alpha sets a baseline confidence for preference.
regParam	default is 0.1	regularization to prevent overfitting

Hyperparameters

Params	Values	Description
rank	default value is 10	Dimension of the feature vectors learned for users and items
alpha	default of 1.0	When training on **implicit** feedback (behavioral observations), the alpha sets a baseline confidence for preference.
regParam	default is 0.1	regularization to prevent overfitting
implicitPrefs	default is explicit	whether you are training on implicit (true) or explicit (false)

Hyperparameters

Params	Values	Description
rank	default value is 10	Dimension of the feature vectors learned for users and items
alpha	default of 1.0	When training on **implicit** feedback (behavioral observations), the alpha sets a baseline confidence for preference.
regParam	default is 0.1	regularization to prevent overfitting
implicitPrefs	default is explicit	whether you are training on implicit (true) or explicit (false)
nonnegative	default value is false	set to true, then return non-negative feature vectors

Training parameters

Params	Values	Description
numUserBlocks	default value is 10	This determines how many blocks to split the users into.
numItemBlocks	default value is 10	This determines how many blocks to split the items into.
maxIter	default value is 10	Total number of iterations over the data before stopping
checkpointInterval		Checkpointing allows you to save model state during training to more quickly recover from node failures.
seed		Specifying the same random seed can help you replicate your results.

Prediction Parameter

- It helps to determine **how** a trained model will make predictions.
- **Cold Start:**
  - Users or items did not appear in training set (have no ratings) and therefore the model has no recommendation to make
  - **Simple random splits** such as Spark's CrossValidator or TrainValidationSplit can also cause this issue; users or items in evaluation set which are not in training set.
  - **NaN** are assigned when user or items are not present in actual model.
  - Set the **coldStartStrategy** parameter to drop in order to drop any rows in the DataFrame of predictions that contain **NaN** values

Example

```
import org.apache.spark.ml.recommendation.ALS

val ratings = spark.read.textFile("/data/sample_movielens_ratings.txt")
.selectExpr("split(value , '::') as col")

.selectExpr("cast(col[0] as int) as userId", "cast(col[1] as int) as movieId", "cast(col[2]
as float) as rating", "cast(col[3] as long) as timestamp")

val Array(training, test) = ratings.randomSplit(Array(0.8, 0.2))

val als = new ALS() .setMaxIter(5) .setRegParam(0.01) .setUserCol("userId")
.setItemCol("movieId").setRatingCol("rating")

println(als.explainParams())

val alsModel = als.fit(training)

val predictions = alsModel.transform(test)
```

Example

```
alsModel.recommendForAllUsers(10)
.selectExpr("userId", explode(recommendations)").show()

alsModel.recommendForAllItems(10)
.selectExpr("movieId", "explode(recommendations)").show()
```

Evaluators for Recommendation

- Recommendation problem is a kind of regression problem

- We want to optimize the total difference between users' ratings and the true values. It can be done by using the **RegressionEvaluator**

- To do this , we need to set cold start strategy to "drop" instead of "NaN"

Example: Evaluators

```
import org.apache.spark.ml.evaluation.RegressionEvaluator

val evaluator = new RegressionEvaluator()
.setMetricName("rmse")
.setLabelCol("rating")
.setPredictionCol("prediction")

val rmse = evaluator.evaluate(predictions)

println(s"Root-mean-square error = $rmse")
```

Metrics (RDD-based)

- Recommendation-specific metrics (more sophisticated than simply evaluating based on regression).

- Two types of metrics (**RDD-based** API) :
  - Regression Metrics
  - Ranking Metrics

Regression Metrics

- Used to help in identifying how close each prediction is to the actual rating for that user and item.

```
import org.apache.spark.mllib.evaluation.{RankingMetrics,
RegressionMetrics}


val regComparison = predictions.select("rating", "prediction")
.rdd.map(x => (x.getFloat(0).toDouble,x.getFloat(1).toDouble))


val metrics = new RegressionMetrics(regComparison)
```

Ranking Metrics:

- Used to compare our recommendations with actual ratings of the user.
- "RankingMetric" verifies whether the algorithm actually recommends previously ranked item to an user.

```
import org.apache.spark.mllib.evaluation.{RankingMetrics, RegressionMetrics}

import org.apache.spark.sql.functions.{col, expr}

val perUserActual = predictions .where("rating > 2.5")
.groupBy("userId") .agg(expr("collect_set(movieId) as movies"))
```

- val **perUserPredictions** = predictions .orderBy(col("userId"), col("prediction").desc) .groupBy("userId") .agg(expr("collect_list(movieId) as movies"))

- It will get the top 10 recommendations displayed in our true set.

- We have two DataFrames; Prediction (perUserActual) and Top-ranked Items (perUserPredictions).
- RankingMetrics accepts the **RDD** of these combinations.

```
val perUserActualvPred = perUserActual.join(perUserPredictions,
Seq("userId")).map(row => (
row(1).asInstanceOf[Seq[Integer]].toArray,
row(2).asInstanceOf[Seq[Integer]].toArray.take(15)
))

val ranks = new RankingMetrics(perUserActualvPred.rdd)
```

- Metrics from that ranking is being displayed.
- Used to check the precision of our algorithm.

```
ranks.meanAveragePrecision
ranks.precisionAt(5)
```

Frequent Pattern Mining:

- It is referred as Market Basket Analysis.
- It will identify the raw data and recommends something which is associated to the data.
- Example : a person buying a same brand of food , will be suggested the same while the person is trying to fill the shopping cart.