In [1]:

```
from pyspark.sql import SparkSession
import pyspark.sql.functions as F
from pyspark.sql.types import *

spark = SparkSession\
    .builder\
    .appName("chapter-28-ML-recommender")\
    .getOrCreate()

import os
SPARK_BOOK_DATA_PATH = os.environ['SPARK_BOOK_DATA_PATH']
```

In [2]:
```
from pyspark.ml.recommendation import ALS
from pyspark.sql import Row
ratings = spark.read\
    .text(SPARK_BOOK_DATA_PATH + "/data/sample_movielens_ratings.txt")\
    .rdd.toDF()\
    .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"
    )

training, test = ratings.randomSplit([0.8, 0.2])

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

print (als.explainParams())
alsModel = als.fit(training)               # fit to train Model on training data
predictions = alsModel.transform(test)     # transform to predict on test data
```

```
alpha: alpha for implicit preference (default: 1.0)
checkpointInterval: set checkpoint interval (>= 1) or disable checkpoint (-1). E.g. 10 means that t
he cache will get checkpointed every 10 iterations. Note: this setting will be ignored if the check
point directory is not set in the SparkContext. (default: 10)
coldStartStrategy: strategy for dealing with unknown or new users/items at prediction time. This ma
y be useful in cross-validation or production scenarios, for handling user/item ids the model has n
ot seen in the training data. Supported values: 'nan', 'drop'. (default: nan)
finalStorageLevel: StorageLevel for ALS model factors. (default: MEMORY_AND_DISK)
implicitPrefs: whether to use implicit preference (default: False)
intermediateStorageLevel: StorageLevel for intermediate datasets. Cannot be 'NONE'. (default: MEMOR
Y_AND_DISK)
itemCol: column name for item ids. Ids must be within the integer value range. (default: item, curr
ent: movieId)
maxIter: max number of iterations (>= 0). (default: 10, current: 5)
nonnegative: whether to use nonnegative constraint for least squares (default: False)
numItemBlocks: number of item blocks (default: 10)
numUserBlocks: number of user blocks (default: 10)
```

```
predictionCol: prediction column name. (default: prediction)
rank: rank of the factorization (default: 10)
ratingCol: column name for ratings (default: rating, current: rating)
regParam: regularization parameter (>= 0). (default: 0.1, current: 0.01)
seed: random seed. (default: -1517157561977538513)
userCol: column name for user ids. Ids must be within the integer value range. (default: user, curr
ent: userId)
```

In [3]:

```
# COMMAND ----------

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

```
+------+--------------+
|userId|           col|
+------+--------------+
|    28| [12, 4.854412]|
|    28|[81, 4.5314903]|
|    28|   [2, 3.90854]|
|    28| [82, 3.842551]|
|    28|[23, 3.5598414]|
|    28|[76, 3.4648323]|
|    28| [62, 3.214014]|
|    28| [26, 2.959738]|
|    28|[57, 2.8760154]|
|    28|[70, 2.7861943]|
|    26|[83, 5.9210377]|
|    26|[33, 5.4730806]|
|    26| [19, 5.261099]|
|    26| [37, 5.150425]|
|    26| [24, 5.059302]|
|    26|[12, 5.0024195]|
|    26| [88, 4.989318]|
|    26|[22, 4.9702883]|
|    26| [94, 4.957751]|
|    26|[23, 4.8820114]|
+------+--------------+
only showing top 20 rows
```

In [4]:
```
alsModel.recommendForAllItems(10)\
  .selectExpr("movieId", "explode(recommendations)").show()
```

```
+-------+---------------+
|movieId|            col|
+-------+---------------+
|     31| [21, 4.190279]|
|     31|[12, 3.6101668]|
|     31| [6, 3.0709796]|
|     31| [8, 3.0144005]|
|     31|[14, 3.0032687]|
|     31| [7, 2.8663979]|
|     31|[13, 2.4790258]|
|     31|[10, 2.4686954]|
|     31| [25, 2.211947]|
|     31| [4, 2.1117184]|
|     85|  [8, 4.849661]|
|     85| [16, 4.708046]|
|     85| [7, 3.7650425]|
|     85| [6, 3.5269358]|
|     85|    [1, 3.0007]|
|     85|[14, 2.8658288]|
|     85| [21, 2.746056]|
|     85|[19, 2.5981874]|
|     85|[20, 2.3025043]|
|     85| [10, 2.057417]|
+-------+---------------+
only showing top 20 rows
```

In [5]:
```
# COMMAND ----------

from pyspark.ml.evaluation import RegressionEvaluator
evaluator = RegressionEvaluator()\
  .setMetricName("rmse")\
  .setLabelCol("rating")\
  .setPredictionCol("prediction")
rmse = evaluator.evaluate(predictions)
print("Root-mean-square error = %f" % rmse)
```

```
Root-mean-square error = 1.544025
```

In [6]:
```
# COMMAND ----------

from pyspark.mllib.evaluation import RegressionMetrics
regComparison = predictions.select("rating", "prediction")\
  .rdd.map(lambda x: (x(0), x(1)))
metrics = RegressionMetrics(regComparison)
```

In [7]:
```
# COMMAND ----------

from pyspark.mllib.evaluation import RankingMetrics, RegressionMetrics
from pyspark.sql.functions import col, expr
perUserActual = predictions\
  .where("rating > 2.5")\
  .groupBy("userId")\
  .agg(expr("collect_set(movieId) as movies"))
```

In [8]:
```
# COMMAND ----------

perUserPredictions = predictions\
  .orderBy(col("userId"), expr("prediction DESC"))\
  .groupBy("userId")\
  .agg(expr("collect_list(movieId) as movies"))
```

In [ ]:
```
# COMMAND ----------

perUserActualvPred = perUserActual.join(perUserPredictions, ["userId"]).rdd\
  .map(lambda row: (row[1], row[2][:15]))
```

In [12]:
```
type(perUserActualvPred)
```

Out[12]:
```
pyspark.rdd.PipelinedRDD
```

In [14]:
```
perUserActualvPred.collect()
```

Out[14]:
```
[([19, 49, 89, 92], [65, 19, 44, 3, 59, 49, 13, 89, 92]),
 ([18, 36, 73], [61, 18, 73, 36, 91]),
 ([75, 55, 44], [31, 44, 42, 95, 52, 55, 75]),
 ([66, 64, 50], [83, 66, 44, 84, 63, 4, 15, 88, 60, 67, 50, 64, 82, 87]),
 ([30, 69, 22], [14, 44, 45, 90, 22, 30, 63, 69, 96]),
 ([72], [78, 87, 73, 98, 26, 85, 31, 72, 71]),
 ([43], [75, 44, 38, 33, 18, 16, 68, 22, 94, 56, 20, 43, 53]),
 ([90], [90, 49, 38, 15, 65, 60, 2, 21]),
 ([36, 8, 80], [8, 91, 89, 81, 46, 83, 36, 72, 80, 62]),
 ([51, 90], [1, 73, 90, 78, 25, 24, 51]),
 ([54], [26, 2, 4, 11, 45, 61, 33, 54, 6, 55]),
 ([87, 14], [98, 14, 87, 77]),
 ([46, 56, 55], [70, 46, 56, 55, 93, 29, 66, 72, 19, 34]),
 ([96], [54, 96, 79, 33, 83, 4, 71, 15, 56, 9]),
 ([30, 13, 68, 55, 73, 23], [28, 59, 23, 68, 61, 13, 0, 73, 30, 55]),
 ([42], [66, 90, 42, 43, 98, 37, 8, 58, 41]),
 ([30, 96, 68, 69], [84, 78, 31, 96, 10, 30, 69, 68]),
 ([49], [70, 42, 49]),
 ([96, 29], [55, 20, 38, 66, 46, 40, 45, 51, 96, 29]),
 ([66, 27, 13, 50, 23],
  [62, 70, 13, 45, 50, 43, 77, 66, 89, 6, 11, 88, 27, 23, 59]),
 ([52, 53, 93, 3, 72, 47],
  [48, 49, 36, 39, 93, 6, 83, 47, 53, 5, 3, 95, 94, 45, 72]),
 ([83, 4], [76, 35, 78, 4, 83]),
 ([9, 92], [98, 9, 17, 45, 95, 34, 59, 92, 29])]
```

In [15]:
```
ranks = RankingMetrics(perUserActualvPred)

# COMMAND ----------

ranks.meanAveragePrecision, ranks.precisionAt(5)


# COMMAND ----------
```

Out[15]:
```
(0.29808913308913304, 0.49565217391304356)
```

In [ ]:
```
```