Collaborative Filtering

Collaborative filtering

Collaborative filtering is commonly used for recommender systems. These techniques aim to fill in the missing entries of a user-item association matrix. `spark.ml` currently supports model-based collaborative filtering, in which users and products are described by a small set of latent factors that can be used to predict missing entries. `spark.ml` uses the alternating least squares (ALS) algorithm to learn these latent factors. The implementation in `spark.ml` has the following parameters:

- *numBlocks* is the number of blocks the users and items will be partitioned into in order to parallelize computation (defaults to 10).
- *rank* is the number of latent factors in the model (defaults to 10).
- *maxIter* is the maximum number of iterations to run (defaults to 10).
- *regParam* specifies the regularization parameter in ALS (defaults to 1.0).
- *implicitPrefs* specifies whether to use the *explicit feedback* ALS variant or one adapted for *implicit feedback* data (defaults to `false` which means using *explicit feedback*).
- *alpha* is a parameter applicable to the implicit feedback variant of ALS that governs the *baseline* confidence in preference observations (defaults to 1.0).
- *nonnegative* specifies whether or not to use nonnegative constraints for least squares (defaults to `false`).

Note: The DataFrame-based API for ALS currently only supports integers for user and item ids. Other numeric types are supported for the user and item id columns, but the ids must be within the integer value range.

Explicit vs. implicit feedback

The standard approach to matrix factorization based collaborative filtering treats the entries in the user-item matrix as *explicit* preferences given by the user to the item, for example, users giving ratings to movies.

It is common in many real-world use cases to only have access to *implicit feedback* (e.g. views, clicks, purchases, likes, shares etc.). The approach used in `spark.ml` to deal with such data is taken from Collaborative Filtering for Implicit Feedback Datasets. Essentially, instead of trying to model the matrix of ratings directly, this approach treats the data as numbers representing the *strength* in observations of user actions (such as the number of clicks, or the cumulative duration someone spent viewing a movie). Those numbers are then related to the level of confidence in observed user preferences, rather than explicit ratings given to items. The model then tries to find latent factors that can be used to predict the expected preference of a user for an item.

Scaling of the regularization parameter

We scale the regularization parameter `regParam` in solving each least squares problem by the number of ratings the user generated in updating user factors, or the number of ratings the product received in updating product factors. This approach is named “ALS-WR” and discussed in the paper “Large-Scale Parallel Collaborative Filtering for the Netflix

Prize”. It makes regParam less dependent on the scale of the dataset, so we can apply the best parameter learned from a sampled subset to the full dataset and expect similar performance.

Cold-start strategy

When making predictions using an ALSModel, it is common to encounter users and/or items in the test dataset that were not present during training the model. This typically occurs in two scenarios:

1. In production, for new users or items that have no rating history and on which the model has not been trained (this is the “cold start problem”).
2. During cross-validation, the data is split between training and evaluation sets. When using simple random splits as in Spark’s CrossValidator or TrainValidationSplit, it is actually very common to encounter users and/or items in the evaluation set that are not in the training set

By default, Spark assigns NaN predictions during ALSModel.transform when a user and/or item factor is not present in the model. This can be useful in a production system, since it indicates a new user or item, and so the system can make a decision on some fallback to use as the prediction.

However, this is undesirable during cross-validation, since any NaN predicted values will result in NaN results for the evaluation metric (for example when using RegressionEvaluator). This makes model selection impossible.

Spark allows users to set the coldStartStrategy parameter to “drop” in order to drop any rows in the DataFrame of predictions that contain NaN values. The evaluation metric will then be computed over the non-NaN data and will be valid. Usage of this parameter is illustrated in the example below.

Note: currently the supported cold start strategies are “nan” (the default behavior mentioned above) and “drop”. Further strategies may be supported in future.

Examples

Scala Java Python R

In the following example, we load ratings data from the MovieLens dataset, each row consisting of a user, a movie, a rating and a timestamp. We then train an ALS model which assumes, by default, that the ratings are explicit (implicitPrefs is false). We evaluate the recommendation model by measuring the root-mean-square error of rating prediction.

Refer to the ALS Scala docs for more details on the API.

```scala
import org.apache.spark.ml.evaluation.RegressionEvaluator
import org.apache.spark.ml.recommendation.ALS

case class Rating(userId: Int, movieId: Int, rating: Float, timestamp: Long)
def parseRating(str: String): Rating = {
  val fields = str.split("::")
  assert(fields.size == 4)
  Rating(fields(0).toInt, fields(1).toInt, fields(2).toFloat, fields(3).toLong)
}

val ratings = spark.read.textFile("data/mllib/als/sample_movielens_ratings.txt")
  .map(parseRating)
  .toDF()
val Array(training, test) = ratings.randomSplit(Array(0.8, 0.2))
```

»

```
// Build the recommendation model using ALS on the training data
val als = new ALS()
  .setMaxIter(5)
  .setRegParam(0.01)
  .setUserCol("userId")
  .setItemCol("movieId")
  .setRatingCol("rating")
val model = als.fit(training)

// Evaluate the model by computing the RMSE on the test data
// Note we set cold start strategy to 'drop' to ensure we don't get NaN evaluation metrics
model.setColdStartStrategy("drop")
val predictions = model.transform(test)

val evaluator = new RegressionEvaluator()
  .setMetricName("rmse")
  .setLabelCol("rating")
  .setPredictionCol("prediction")
val rmse = evaluator.evaluate(predictions)
println(s"Root-mean-square error = $rmse")

// Generate top 10 movie recommendations for each user
val userRecs = model.recommendForAllUsers(10)
// Generate top 10 user recommendations for each movie
val movieRecs = model.recommendForAllItems(10)
```

Find full example code at "examples/src/main/scala/org/apache/spark/examples/ml/ALSExample.scala" in the Spark repo.

If the rating matrix is derived from another source of information (i.e. it is inferred from other signals), you can set `implicitPrefs` to `true` to get better results:

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