**Chapter 28. Recommendation**

The task of recommendation is one of the most intuitive. By studying people’s explicit preferences (through ratings) or implicit preferences (through observed behavior), you can make recommendations on what one user may like by drawing similarities between the user and other users, or between the products they liked and other products. Using the underlying similarities, recommendation engines can make new recommendations to other users.

**Use Cases**

Recommendation engines are one of the best use cases for big data. It’s fairly easy to collect training data about users’ past preferences at scale, and this data can be used in many domains to connect users with new content. Spark is an open source tool of choice used across a variety of companies for large-scale recommendations:

Movie recommendations

Amazon, Netflix, and HBO all want to provide relevant film and TV content to their users. [Netflix utilizes Spark](https://youtu.be/II8GlmbDg9M), to make large scale movie recommendations to their users.

Course recommendations

A school might want to recommend courses to students by studying what courses similar students have liked or taken. Past enrollment data makes for a very easy to collect training dataset for this task.

In Spark, there is one workhorse recommendation algorithm, Alternating Least Squares (ALS). This algorithm leverages a technique called *collaborative filtering*, which makes recommendations based only on which items users interacted with in the past. That is, it does not require or use any additional features about the users or the items. It supports several ALS variants (e.g., explicit or implicit feedback). Apart from ALS, Spark provides Frequent Pattern Mining for finding association rules in market basket analysis. Finally, [Spark’s RDD API](http://spark.apache.org/docs/latest/mllib-collaborative-filtering.html) also includes a lower-level matrix factorization method that will not be covered in this book.

**Collaborative Filtering with Alternating Least Squares**

ALS finds a 𝘬-dimensional feature vector for each user and item such that the dot product of each user’s feature vector with each item’s feature vector approximates the user’s rating for that item. Therefore this only requires an input dataset of existing ratings between user-item pairs, with three columns: a user ID column, an item ID column (e.g., a movie), and a rating column. The ratings can either be *explicit*—a numerical rating that we aim to predict directly—or *implicit*—in which case each rating represents the strength of interactions observed between a user and item (e.g., number of visits to a particular page), which measures our level of confidence in the user’s preference for that item. Given this input DataFrame, the model will produce feature vectors that you can use to predict users’ ratings for items they have not yet rated.

One issue to note in practice is that this algorithm does have a preference for serving things that are very common or that it has a lot of information on. If you’re introducing a new product that no users have expressed a preference for, the algorithm isn’t going to recommend it to many people. Additionally, if new users are onboarding onto the platform, they may not have any ratings in the training set. Therefore, the algorithm won’t know what to recommend them. These are examples of what we call the *cold start problem*, which we discuss later on in the chapter.

In terms of scalability, one reason for Spark’s popularity for this task is that the algorithm and implementation in MLlib can scale to millions of users, millions of items, and billions of ratings.

**Model Hyperparameters**

These are configurations that we can specify to determine the structure of the model as well as the specific collaborative filtering problem we wish to solve:

rank

The rank term determines the dimension of the feature vectors learned for users and items. This should normally be tuned through experimentation. The core trade-off is that by specifying too high a rank, the algorithm may overfit the training data; but by specifying a low rank, then it may not make the best possible predictions. The default value is 10.

alpha

When training on implicit feedback (behavioral observations), the alpha sets a baseline confidence for preference. This has a default of 1.0 and should be driven through experimentation.

regParam

Controls regularization to prevent overfitting. You should test out different values for the regularization parameter to find the optimal value for your problem. The default is 0.1.

implicitPrefs

This Boolean value specifies whether you are training on implicit (true) or explicit (false) (refer back to the preceding discussion for an explanation of the difference between explicit and implicit). This value should be set based on the data that you’re using as input to the model. If the data is based off passive endorsement of a product (say, via a click or page visit), then you should use implicit preferences. In contrast, if the data is an explicit rating (e.g., the user gave this restaurant 4/5 stars), you should use explicit preferences. Explicit preferences are the default.

nonnegative

If set to true, this parameter configures the model to place non-negative constraints on the least-squares problem it solves and only return non-negative feature vectors. This can improve performance in some applications. The default value is false.

**Training Parameters**

The training parameters for alternating least squares are a bit different from those that we have seen in other models. That’s because we’re going to get more low-level control over how the data is distributed across the cluster. The groups of data that are distributed around the cluster are called *blocks*. Determining how much data to place in each block can have a significant impact on the time it takes to train the algorithm (but not the final result). A good rule of thumb is to aim for approximately one to five million ratings per block. If you have less data than that in each block, more blocks will not improve the algorithm’s performance.

numUserBlocks

This determines how many blocks to split the users into. The default is 10.

numItemBlocks

This determines how many blocks to split the items into. The default is 10.

maxIter

Total number of iterations over the data before stopping. Changing this probably won’t change your results a ton, so this shouldn’t be the first parameter you adjust. The default is 10. An example of when you might want to increase this is that after inspecting your objective history and noticing that it doesn’t flatline after a certain number of training iterations.

checkpointInterval

Checkpointing allows you to save model state during training to more quickly recover from node failures. You can set a checkpoint directory using SparkContext.setCheckpointDir.

seed

Specifying a random seed can help you replicate your results.

**Prediction Parameters**

Prediction parameters determine how a trained model should actually make predictions. In our case, there’s one parameter: the cold start strategy (set through coldStartStrategy). This setting determines what the model should predict for users or items that did not appear in the training set.

The cold start challenge commonly arises when you’re serving a model in production, and new users and/or items have no ratings history, and therefore the model has no recommendation to make. It can also occur when using simple random splits as in Spark’s CrossValidator or TrainValidationSplit, where it is very common to encounter users and/or items in the evaluation set that are not in the training set.

By default, Spark will assign NaN prediction values when it encounters a user and/or item that is not present in the actual model. This can be useful because you design your overall system to fall back to some default recommendation when a new user or item is in the system. However, this is undesirable during training because it will ruin the ability for your evaluator to properly measure the success of your model. 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. drop and nan (the default) are the only currently supported cold-start strategies.

**Example**

This example will make use of a dataset that we have not used thus far in the book, the MovieLens movie rating dataset. This dataset, naturally, has information relevant for making movie recommendations. We will first use this dataset to train a model:

*// in Scala*

**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)

*# in Python*

**from** **pyspark.ml.recommendation** **import** ALS

**from** **pyspark.sql** **import** Row

ratings = spark.read.text("/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)

predictions = alsModel.transform(test)

We can now output the top 𝘬 recommendations for each user or movie. The model’s recommendForAllUsers method returns a DataFrame of a userId, an array of recommendations, as well as a rating for each of those movies. recommendForAllItems returns a DataFrame of a movieId, as well as the top users for that movie:

*// in Scala*

alsModel.recommendForAllUsers(10)

.selectExpr("userId", "explode(recommendations)").show()

alsModel.recommendForAllItems(10)

.selectExpr("movieId", "explode(recommendations)").show()

*# in Python*

alsModel.recommendForAllUsers(10)\

.selectExpr("userId", "explode(recommendations)").show()

alsModel.recommendForAllItems(10)\

.selectExpr("movieId", "explode(recommendations)").show()

**Evaluators for Recommendation**

When covering the cold-start strategy, we can set up an automatic model evaluator when working with ALS. One thing that may not be immediately obvious is that this recommendation problem is really just a kind of regression problem. Since we’re predicting values (ratings) for given users, we want to optimize for reducing the total difference between our users’ ratings and the true values. We can do this using the same RegressionEvaluator that we saw in [Chapter 27](https://www.safaribooksonline.com/library/view/spark-the-definitive/9781491912201/ch27.html#s6c4---regression). You can place this in a pipeline to automate the training process. When doing this, you should also set the cold-start strategy to be drop instead of NaN and then switch it back to NaN when it comes time to actually make predictions in your production system:

*// in Scala*

**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")

*# in Python*

**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)

**Metrics**

Recommendation results can be measured using both the standard regression metrics and some recommendation-specific metrics. It should come as no surprise that there are more sophisticated ways of measuring recommendation success than simply evaluating based on regression. These metrics are particularly useful for evaluating your final model.

**Regression Metrics**

We can recycle the regression metrics for recommendation. This is because we can simply see how close each prediction is to the actual rating for that user and item:

*// in Scala*

**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)

*# in Python*

**from** **pyspark.mllib.evaluation** **import** RegressionMetrics

regComparison = predictions.select("rating", "prediction")\

.rdd.map(**lambda** x: (x(0), x(1)))

metrics = RegressionMetrics(regComparison)

**Ranking Metrics**

More interestingly, we also have another tool: ranking metrics. A RankingMetric allows us to compare our recommendations with an actual set of ratings (or preferences) expressed by a given user. RankingMetric does not focus on the value of the rank but rather whether or not our algorithm recommends an already ranked item again to a user. This does require some data preparation on our part. You may want to refer to [Part II](https://www.safaribooksonline.com/library/view/spark-the-definitive/9781491912201/part02.html#part2) for a refresher on some of the methods. First, we need to collect a set of highly ranked movies for a given user. In our case, we’re going to use a rather low threshold: movies ranked above 2.5. Tuning this value will largely be a business decision:

*// in Scala*

**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"))

*# in Python*

**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"))

At this point, we have a collection of users, along with a truth set of previously ranked movies for each user. Now we will get our top 10 recommendations from our algorithm on a per-user basis. We will then see if the top 10 recommendations show up in our truth set. If we have a well-trained model, it will correctly recommend the movies a user already liked. If it doesn’t, it may not have learned enough about each particular user to successfully reflect their preferences:

*// in Scala*

**val** perUserPredictions **=** predictions

.orderBy(col("userId"), col("prediction").desc)

.groupBy("userId")

.agg(expr("collect\_list(movieId) as movies"))

*# in Python*

perUserPredictions = predictions\

.orderBy(col("userId"), expr("prediction DESC"))\

.groupBy("userId")\

.agg(expr("collect\_list(movieId) as movies"))

Now we have two DataFrames, one of predictions and another the top-ranked items for a particular user. We can pass them into the RankingMetrics object. This object accepts an RDD of these combinations, as you can see in the following join and RDD conversion:

*// in Scala*

**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)

*# in Python*

perUserActualvPred = perUserActual.join(perUserPredictions, ["userId"]).rdd\

.map(**lambda** row: (row[1], row[2][:15]))

ranks = RankingMetrics(perUserActualvPred)

Now we can see the metrics from that ranking. For instance, we can see how precise our algorithm is with the mean average precision. We can also get the precision at certain ranking points, for instance, to see where the majority of the positive recommendations fall:

*// in Scala*

ranks.meanAveragePrecision

ranks.precisionAt(5)

*# in Python*

ranks.meanAveragePrecision

ranks.precisionAt(5)

**Frequent Pattern Mining**

In addition to ALS, another tool that MLlib provides for creating recommendations is frequent pattern mining. *Frequent pattern mining*, sometimes referred to as *market basket analysis*, looks at raw data and finds association rules. For instance, given a large number of transactions it might identify that users who buy hot dogs almost always purchase hot dog buns. This technique can be applied in the recommendation context, especially when people are filling shopping carts (either on or offline). Spark implements the FP-growth algorithm for frequent pattern mining. See [the Spark documentation](https://spark.apache.org/docs/latest/ml-frequent-pattern-mining.html#fp-growth) and ESL 14.2 for more information about this algorithm.

**Conclusion**

In this chapter, we discussed one of Spark’s most popular machine learning algorithms in practice—alternating least squares for recommendation. We saw how we can train, tune, and evaluate this model. In the next chapter, we’ll move to unsupervised learning and discuss clustering.