SparkML

Lab 03: Building a Recommendation Engine

In this lab, we'll build out a simple recommendation engine based on our 100,000 movie ratings.

Preparation

Here's we're simply going to read in three files and register them as tables. See Lab 02 for definitions of these files and how they work. While we did this in Lab 02, we want to ensure that we have their contents as DataFrames

```
import org.apache.spark.sql.functions._
// Import Spark SQL data types
import org.apache.spark.sql._
// Import mllib recommendation data types
import org.apache.spark.mllib.recommendation.{ALS, MatrixFactorizationModel, Rating}
import org.apache.spark.util.random

// input format MovieID|Title|Genres
case class Movie(movieid: Int, title: String)

// input format is UserID|Gender|Age|Occupation|Zip-code
case class User(userid: Int,  age: Int, gender: String, occupation: String, zip: String)

// function to parse movie record into Movie class
def parseMovie(str: String): Movie = {
      val fields = str.split("\\|")
      Movie(fields(0).toInt, fields(1))
 }
// function to parse input into User class
```

```
def parseUser(str: String): User = {
      val fields = str.split("\\|")
      assert(fields.size == 5)
      User(fields(0).toInt, fields(1).toInt, fields(2).toString,
            fields(3).toString, fields(4).toString)
 }

// function to parse input UserID\tMovieID\tRating
//  Into org.apache.spark.mllib.recommendation.Rating class
def parseRating(str: String): Rating= {
      val fields = str.split("\\t")
      Rating(fields(0).toInt, fields(1).toInt, fields(2).toDouble)
}

//////////////////////
// Ingest the data //
//////////////////////

// OK, now read in the user and movie info files
val users = sc.textFile("hdfs:///data/ml-100k/u.user").map(parseUser)
val usersDF = sc.textFile("hdfs:///data/ml-100k/u.user").map(parseUser).toDF()
val moviesDF = sc.textFile("hdfs:///data/ml-100k/u.item").map(parseMovie).toDF()
val fileratings = sc.textFile("hdfs:///data/ml-100k/u.data").map(parseRating)
val myratings = sc.textFile("file:/mnt/sparkclass/exercises/SparkML-In-Depth/personalRatings.t
val ratings = fileratings
val ratingsDF = fileratings.toDF()

// register the DataFrames as a temp table for SQL queries
ratingsDF.registeruTempTable("ratings")
moviesDF.registerTempTable("movies")
usersDF.registerTempTable("users")
```

Examining Our Data Along With A Few Surprises

Let's just check a bit of our data to ensure it's as we expect.

```
val usersDF = spark.read.table("users")
val moviesDF = spark.read.table("movies")
val ratingsDF = spark.read.table("ratings")

// Look at the top 10 rows of usersDF
usersDF.show(5)
```

This should look something like:

movieid	title
1	Toy Story (1995)
2	GoldenEye (1995)
3	Four Rooms (1995)
4	Get Shorty (1995)
5	Copycat (1995)

However, you may instead see a different set of movies, such as this:

movieid	title
839	Loch Ness (1995)
840	Last Man Standing...
841	Glimmer Man, The ...
842	Pollyanna (1960)
843	Shaggy Dog, The (...

We actually can't predict what output you'll get. All we know is that you'll get the 5 of the movies in the dataset.

Why Can't We Predict What Movies You'll See?

The inability to predict how your table will be ordered is one of the effects of Spark being designed for maximum parallelism. Your Spark program runs on multiple processors and cores in parallel. Unlike in MapReduce, those cores don't synchronize their efforts unless we explicitly ask them to do so.

In this case, when we asked Spark to write out our table to the Hive Warehouse, it allowed each Spark executor to write its own file to the output directory without synchronization. You can in fact see how it wrote out these tables by looking in the Hive warehouse from the shell (permission and ownership details have been omitted below for print clarity):

```
hdfs dfs -ls /user/hive/warehouse/movies
/user/hive/warehouse/movies/_SUCCESS
/user/hive/warehouse/movies/part-00000-8e9dadd4-2abb-4329-a436-624df661978a-c000.snappy.parquet
/user/hive/warehouse/movies/part-00001-8e9dadd4-2abb-4329-a436-624df661978a-c000.snappy.parquet
```

Each of these files contains half of the table in compressed binary form. Which half you see first in your `show` results is determined only by which Spark executor wrote its results first. And since the Spark executors run in parallel, the order ends up being a matter of chance.

Examining Other Tables

While acknowledging this nondeterminism, examine the other tables you've read in using show commands such as the following:

```
usersDF.show(10)
moviesDF.show(10)
ratingsDF.show(10)
```

Analyzing The Ratings

Before we build out our recommendation engine, let's just look at a few specific movies using `filter`. If you're more used to SQL, you may prefer to use the second version.

```
moviesDF.filter(moviesDF("title").contains("Star Wars")).show
sql("""SELECT movieid, title FROM movies
       WHERE title LIKE \"Star Wars%\"""").show
```

You'll find the 1977 Star Wars has movieid 50.

movieid	title
50	Star Wars (1977)

Join the `moviesDF` DataFrame together with `ratingsDF`. You'll want the `movieid` column in `moviesDF` to match the `product` column in `ratingsDF`. Call that result `ratings_with_titleDF`, and then show the results of filtering that by the title *Star Wars*.

```
val ratings_with_titleDF = ratingsDF.join(moviesDF, ratingsDF("product") === moviesDF("movieid"
ratings_with_titleDF.show(5)
// See ratings for Star Wars now
val starwars = ratings_with_titleDF.
  filter($"title".contains("Star Wars"))
starwars.show(5)
```

You'll see something like this, although the actual rows you see may be different due to parallelism.

user	product	rating	movieid	title
290	50	5.0	50	Star Wars (1977)
79	50	4.0	50	Star Wars (1977)
2	50	5.0	50	Star Wars (1977)
8	50	5.0	50	Star Wars (1977)
274	50	5.0	50	Star Wars (1977)

Now do the following steps:

1. join in the `usersDF` DataFrame, linking the `user` field in `ratings_with_titleDF` with the `userid` field in the `userDF` DataFrame. We'll call that `denorm_ratingsDF`.
2. Filter `denorm_ratingsDF` for `title` containing *Star Wars* and assign that to val `starwars`.
3. Show and persist `starwars`.
4. Group `denorm_ratingsDF` by the field `gender` and show the average rating by gender.
5. Check those results against the average of the total corpus of movie ratings grouped by `gender`.

Here's the code to do this.

```
val denorm_ratingsDF = ratings_with_titleDF.
  join(usersDF, ratings_with_titleDF("user") === usersDF("userid"))
val starwars = denorm_ratingsDF.
  filter(denorm_ratingsDF("title").
  contains("Star Wars"))
starwars.show(5)
starwars.persist
starwars.groupBy("gender").agg(avg($"rating")).show

// check the results against the total set of movies for sanity
denorm_ratingsDF.groupBy("gender").agg(avg($"rating")).show // compare with total set
```

The Star Wars ratings are:

gender	avg(rating)
F	4.245033112582782
M	4.398148148148148

While the overall ratings are

gender	avg(rating)
F	3.5315073815073816
M	3.5292889846485322

Draw your own conclusions from this data.

Building A Recommendations Engine

Like in many supervised machine learning algorithms, we'll use a multi-step process to build our recommendation engine.

1. Randomly split our data into training and test data sets.
2. Build a Alternating Least Squares (ALS) model based on the training data set that creates a known result (in this case, a rating). See the course slides for a discussion of what ALS does and why we use it.
3. Apply that model to user data to predict a new result (i.e., make a recommendation)
4. Measure how closely the predicted results match the actual rating results in the test set.

A key point in this exercise is that while we've focused on DataFrames up until this point, the Spark Recommendation Engine tool requires `Ratings` objects, which are defined in the package `org.apache.spark.mllib.recommendation`. Conveniently, our preparation for this lab has created a `ratings` RDD made up of `Ratings` objects. As an aside, this is a common pattern with Spark machine learning libraries: usually you have to create the proper type of objects, not just DataFrames, to use them.

Step 1: Randomly Split Our Data Into Training and Test Sets

Here, we'll use the `randomSplit` method for `RDD` objects to create our training and test sets. We'll use 80% of the data for training and leave 20% for testing. We'll also cache both of those RDDs for later use.

```
al splits = ratings.randomSplit(Array(0.8, 0.2), 0L)
val trainingRDD = splits(0).cache()
val testRatingsRDD = splits(1).cache()
```

Now add our ratings to the training set

```
val trainingRatingsRDD = trainingRDD.union(myratings).cache()

 val numTraining = trainingRatingsRDD.count()
 val numTest = testRatingsRDD.count()
 println(s"Training: $numTraining, test: $numTest.")
```

You should see Training: 80018, test: 19993 or so, depending on your random number seed (we set ours to 0L).

Step 2: Build a Alternating Least Squares (ALS) Model Using The Training Set

All the hard work of parallelization and computation gets done in the ALS library. All we have to do is to set the parameters.

```
// Set up the parameters for training

val rank = 10             // number of hidden features in approximation matrices
val iterations = 10       // iterations of model to run
val lambda = 0.01         // tunable parameter controlling regularization and fitting

val model = ALS.train(trainingRatingsRDD, rank, iterations, lambda)
```

The value model returns is of type `MatrixFactorization`, which consists of two objects: `userFeatures` and `productFeatures`. These correspond to the two matrices whose dot product creates the full ratings matrix.

Step 3: Apply Our Model To User Data To Predict A Result (i.e., make a recommendation)

Before we actually do the prediction, which is just a method we invoke on our `model` object, let's look at a specific user. We'll arbitrarily use user 0, and first let's find out which movies he or she rated by extracting them out of `trainingRatingsRDD`.

```
// We'll test it by using it on our ratings (user 0)
val user0ratings = trainingRatingsRDD.filter(rating => rating.user == 0)
val user0ratingsDF = user0ratings.toDF()

/// these are the user 0 ratings used to train the model
user0ratingsDF.join(moviesDF,
  user0ratingsDF("product") === moviesDF("movieid")).show
```

Now let's look at the top 3 predictions for user 0.

```
val topRecForUser0 = model.recommendProducts(0, 3)
topRecForUser0.map(rating => (movieTitles(rating.product), rating.rating)).foreach(println)
```

We get back something that looks like this:

```
(Boxing Helena (1993),18.402586059612382)
(Harlem (1993),17.438677456080185)
(Joy Luck Club, The (1993),16.45317941043561)
```

Step 4. Measure How Close The Predicted Results Are To The Actual Rating Results In The Test Set

Now let's try to evaluate this model. How well does it perform? To do this, we'll have it do predictions for the entire test data set Once we have a set of predictions, we'll then compare those with the actual ratings using a Mean Absolute Error (MAE) function

We have to strip off the rating to get our predictions in our test set, so We'll use a case class to get a user-

product pair from testRatingsRDD.

```
val testUserProductRDD = testRatingsRDD.map {
  case Rating(user, product, rating) => (user, product)
}
```

Now we can just ask for a prediction for every user-product pair in the test data set.

```
// remember that testRatingsRDD is our test sample of all our data
val predictionsForTestRDD  = model.predict(testUserProductRDD)
predictionsForTestRDD.take(5).mkString("\n")
```

You'll get back something like:

```
Rating(368,320,1.9506349488976316)
Rating(3,320,3.256979002151988)
Rating(21,320,4.715583624693077)
Rating(752,752,3.453001416009629)
Rating(883,752,3.3110704894373333)
```

Now how do we compare these results with the actual ratings given by the users? We'll create a key which is the user-product pair as a tuple, and then have its value be the rating. Then we can join these two RDDs together and compare the results.

```
// prepare  predictions for comparison
val predictionsKeyedByUserProductRDD = predictionsForTestRDD.map{
  case Rating(user, product, rating) => ((user, product), rating)
}
// prepare  test for comparison
val testKeyedByUserProductRDD = testRatingsRDD.map{
  case Rating(user, product, rating) => ((user, product), rating)
}
//Join the  test with  predictions
val testAndPredictionsJoinedRDD = testKeyedByUserProductRDD.
  join(predictionsKeyedByUserProductRDD)
```

Let's look at the comparison for a few products:

```
// print the (user, product),( test rating, predicted rating)
scala> testAndPredictionsJoinedRDD.take(5).mkString("\n")

((660,47),(2.0,2.695739817626425))
((918,495),(3.0,2.731500894684755))
((151,605),(4.0,3.693202454620277))
((249,235),(4.0,3.420363946441323))
((325,616),(4.0,3.4547303609194855))
```

We're not terribly far off. However, it would be nice to know how often our predictor really gets the result wrong.

Let's look for false positives in our predicted ratings. We'll define a false positive as a movie that the user rated a 1 or lower and the predictor rated the movie a 4 or more. We'll then count those to see how badly we're doing.

```
val falsePositives =(testAndPredictionsJoinedRDD.filter{
  case ((user, product), (ratingT, ratingP)) => (ratingT <= 1 && ratingP >=4)
  })
falsePositives.take(3)
falsePositives.count
```

Finally, we'll evaluate the model by computing the Mean Absolute Error (MAE) and Mean Squared Error (MSE) between test and predictions. We'll just write our own version of that right here, although most people would use the library function to do this. The ML libraries have a variety of build-in functions to rate the quality of results.

```
//Evaluate the model using Mean Absolute Error (MAE) between test and predictions
val meanAbsoluteError = testAndPredictionsJoinedRDD.map {
  case ((user, product), (testRating, predRating)) =>
    val err = (testRating - predRating)
    Math.abs(err)
}.mean()
println("Mean Absolute Error = " + meanAbsoluteError)

//We can also calculate the Mean Squared Error (MSE) equally easily

val MSE = testAndPredictionsJoinedRDD.map{
    case ((user, product), (actual, predicted)) =>
    math.pow((actual - predicted), 2)
}.reduce(_ + _) / testAndPredictionsJoinedRDD.count
println("Mean Squared Error = " + MSE)
```

Using the fairly small number of iterations we specified above, we got an MAE of around 0.7 and a MSE of around 1. We have to expect every run to be different because the ALS algorithm initializes with a random value.

However, we can ask the question, can we improve this? What if we did more interations? Added more hidden features? Tried a different lambda? We encourage you to experiment with this algorithm and see if you can reduce the MSE and MAE from our first tries.

This step concludes the lab.