SparkML

Lab 02: Ingest and Understand the ML-100K Data

Objective

In this lab, we're going to familiarize ourselves with the ML-100K data set. Our objective is to ingest and join the three tables together so that we can see how users rated a wide variety of movies. We'll be using these datasets in our next lab where we try to recommend movies to a new user based on their movie preferences.

Background

As background, MovieLens 100k (ml-100k) consists of 100,000 ratings of movies. It includes three types of files about:

1. the movies themselves (`u.item`)
2. the users who did the ratings (`u.user`)
3. the ratings created by the users (`u.data`)

All the data is in text format separated by vertical bars or tabs, depending on the file.

All of these files are preloaded into HDFS at `hdfs:///data/ml-100k. Here are the contents of that HDFS directory

```
/data/ml-100k/README
/data/ml-100k/allbut.pl
/data/ml-100k/mku.sh
/data/ml-100k/u.data
/data/ml-100k/u.genre
/data/ml-100k/u.info
/data/ml-100k/u.item
/data/ml-100k/u.occupation
/data/ml-100k/u.user
/data/ml-100k/u1.base
/data/ml-100k/u1.test
/data/ml-100k/u2.base
/data/ml-100k/u2.test
/data/ml-100k/u3.base
/data/ml-100k/u3.test
/data/ml-100k/u4.base
/data/ml-100k/u4.test
/data/ml-100k/u5.base
/data/ml-100k/u5.test
/data/ml-100k/ua.base
/data/ml-100k/ua.test
/data/ml-100k/ub.base
/data/ml-100k/ub.test
```

Lab01: Characterizing The ML-100K Users

Our first job is reading in these files and trying to understand them. Let's start with the data about the actual users stored in the file `u.user`. The user data has fields separated by vertical bars. Here's what a few lines of u.data look like:

index	age	gender	occupation	zip
1	24	M	technician	85711
2	53	F	other	94043
3	23	M	writer	32067

Let's parse this and see how many users there are in total.

```
val user_data = sc.textFile("hdfs:///data/ml-100k/u.user")
user_data.count
```

You should have 943 users.

Now let's characterize all the fields, looking for distinct values of each.

```
val user_fields = user_data.map(line => line.split("\\|"))
user_fields.take(1)
val num_users = user_fields.map(fields => fields(0)).count
val num_genders = user_fields.map(fields => fields(2)).distinct.count
val num_occupations = user_fields.map(fields => fields(3)).distinct.count
val num_zipcodes = user_fields.map(fields => fields(4)).distinct.count
println("Users: %d, genders: %d, occupations: %d, ZIP codes: %d".format(num_users, num_genders,
```

We should be seeing output that looks like

```
Users: 943, genders: 2, occupations: 21, ZIP codes: 795
```

Challenge 1

Try to figure out how many users there are by occupation. Think about it for a moment and try to write RDD code for this. Occupation is field 3 of the data set. Hint: this is going to be a little like wordcount.

Answer 1

To do a count by occupation, we'll want to create key-value pairs of the form `(occupation, 1)` and then reduce those. So the code to do this will look like this:

```
val count_by_occupation = user_fields.map(fields => (fields(3), 1)).
reduceByKey(_|_).collect()
```

or, if you don't want to bother creating tuples, you can use `countByValue` instead:

```
val count_by_occupation2 = user_fields.
    map(fields => fields(3)).
    countByValue
```

Using A Case Class for User Occupations

We know from the module on Spark SQL that we can use case classes to assign schemas to files like `u.users`. Let's do that now. We'll create a case class called `Occ` that represents each row of `u.users`.

```
case class Occ(index: Int,
    age: Int,
    gender: String,
    occupation: String,
    zip: String)
```

With that in place, we can now parse the file into a dataframe of Occ objects.

```
val user_data = sc.textFile("hdfs:///data/ml-100k/u.user") // read from HDFS by defaul
user_data.first
val user_fields = user_data.map(line => line.split("\\|"))
user_fields.take(5)
val users_array = user_fields.
    map(p => Occ(p(0).toInt, p(1).toInt, p(2), p(3), p(4)))
val users_df = users_array.toDF()
users_df.show
val occupation_count = users_df.groupBy("occupation").count.sort(desc("count"))
occupation_count.show
```

That gives us output that looks like this:

occupation	count
student	196
other	105
educator	95
administrator	79
engineer	67
programmer	66
librarian	51
writer	45
executive	32
scientist	31
artist	28
technician	27
marketing	26
entertainment	18
healthcare	16
retired	14
lawyer	12
salesman	12
none	9
homemaker	7

Ingesting Movie Names

At this point you have enough information to ingest the movie descriptions contained in the file `u.item` and the ratings contained in `u.data`. You'll need to know that the field separators for the movie descriptions are

vertical bars, while the ratings fields are separated by tabs.

Here's what the first three rows of the movie dataset, `u.item`, look like:

```
1|Toy Story (1995)|01-Jan-1995||http://us.imdb.com/M/title-exact?Toy%20Story%20(1995)|0|0|0|1|1
2|GoldenEye (1995)|01-Jan-1995||http://us.imdb.com/M/title-exact?GoldenEye%20(1995)|0|1|1|0|0|0
3|Four Rooms (1995)|01-Jan-1995||http://us.imdb.com/M/title-exact?Four%20Rooms%20(1995)|0|0|0|0
```

At present, we really only care about the item number (the first field) and the movie name (the second field). We're going to ignore the rest of the fields.

We can ingest this `u.item` file in a couple of ways. One way is to replicate what we did with the users dataset. We read the text file, scan for vertical bars, put the fields in the appropriate case class and then convert to a data frame. That code looks like this:

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

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

// Let's join the movie file in so we can get titles

val movies_df = sc.textFile("/data/ml-100k/u.item").map(parseMovie).toDF()
```

Alternatively, we can use our Spark SQL skills and ask the `read` function to read this file as a .csv file with vertical bar as our separator and with a schema that specifies only the first two fields. That code would look like this:

```
// Alternative way of reading in movies_df

import org.apache.spark.sql.types._

val schema = new StructType().
  add($"movieid".long.copy(nullable = false)).
  add($"title".string)

val movies_df = spark.read.
    schema(schema).
    option("sep", "|").
    csv("/data/ml-100k/u.item")
```

Both options produce the same results, which look like this

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

And so on for about 1682 rows.

Ingesting the Ratings

The ratings file located in `/data/ml-100k/u.data` looks like this:

```
196     242     3       881250949
186     302     3       891717742
22      377     1       878887116
244     51      2       880606923
166     346     1       886397596
298     474     4       884182806
115     265     2       881171488
253     465     5       891628467
305     451     3       886324817
```

Clearly we need to decode this structure.

First, the fields are separated by tabs. The field types in order are:

```
userid: Int
itemid: Int
rating: Int
timestamp: String
```

Once again, we could ingest this data frame either with a case class or just by calling `spark.read` with the proper arguments. Again, we'll show it both ways.

First, using a case class

```
case class RatingObj(userid: Int, itemid: Int, rating: Int, timestamp: String)
val ratings_fields = ratings_data.map(line => line.split("\\t"))
val ratings_array = ratings_fields.map(p => RatingObj(p(0).toInt, p(1).toInt, p(2).toInt, p(3)
val ratings_df = ratings_array.toDF()
```

Next, using a schema and `spark.read.csv`.

```
val ratingsschema = new StructType().
  add($"userid".long.copy(nullable = false)).
  add($"itemid".long.copy(nullable = false)).
  add($"rating".long).
  add($"timestamp".string)

val ratings_df2 = spark.read.
    schema(ratingsschema).
    option("sep", "\t").
    csv("/data/ml-100k/u.data")
```

Now we should have our ratings in a nice DataFrame

userid	itemid	rating	timestamp
196	242	3	881250949
186	302	3	891717742
22	377	1	878887116
244	51	2	880606923
166	346	1	886397596
298	474	4	884182806

Let's do a quick sanity check and compute the average ratings for each itemid. We'll even sort the results by average rating.

```
val ratings_means = ratings_df.groupBy("itemid").
  agg(avg(ratings_df("rating")).alias("avg_rating"))

ratings_means.orderBy(desc("avg_rating")).show
```

Joining Ratings With Movies

We now have the three major tables ingested. First, we'd like to join together the movies and the ratings so that we can see which movie titles were the hightest rated.

We'll use a join operation to do this, and the join code is pretty straightforward. We'll want to join the `itemid`

field in the `ratings_df` table with the `movieid` field in the `movies_df` table. Note that we use the === operator to compare the values of the fields instead of the equality of the objects involved.

We also will cache the result for future queries.

```
val ratings_with_titleDF = ratings_df.
  join(movies_df, ratings_df("itemid") === movies_df("movieid")).cache
```

The Rating For Best Movie Is....

We can now compute the average rating by movie title and sort the results in descending order. Let's do this using the Spark API first:

```
ratings_with_titleDF.
  groupBy($"title").
  agg(avg($"rating").alias("avg_rating")).
  orderBy(desc("n")).show(10)
```

We see this as the result:

title	avg_rating
Great Day in Harl...	5.0
Prefontaine (1997)	5.0
Star Kid (1997)	5.0
Saint of Fort Was...	5.0
Aiqing wansui (1994)	5.0
Marlene Dietrich:...	5.0
Entertaining Ange...	5.0
Santa with Muscle...	5.0
They Made Me a Cr...	5.0
Someone Else's Am...	5.0
Pather Panchali (...	4.625

Maya Lin: A Stron...	4.5
Some Mother's Son...	4.5
Anna (1996)	4.5
Everest (1998)	4.5
Close Shave, A (1...	4.491071428571429
Schindler's List ...	4.466442953020135
Wrong Trousers, T...	4.466101694915254
Casablanca (1942)	4.45679012345679
Wallace & Gromit:...	4.447761194029851
Shawshank Redempt...	4.445229681978798
Rear Window (1954)	4.3875598086124405
Usual Suspects, T...	4.385767790262173
Star Wars (1977)	4.3584905660377355
12 Angry Men (1957)	4.344

Conclusion: the users rating these movies have terrible taste: Star Kid rates higher than Casablanca? Really?

Maybe some of the movies weren't rated many times, allowing them to have surprisingly high ratings. We can check that by adding the count of the number of ratings to the resulting table. Let's update our query.

```
ratings_with_titleDF.
  groupBy($"title").
  agg(avg($"rating").alias("avg_rating"),
      count($"rating").alias("n")).
  orderBy(desc("avg_rating")).show(25)
```

And we now see:

title	avg_rating	n
They Made Me a Cr...	5.0	1

Someone Else's Am...	5.0	1
Entertaining Ange...	5.0	1
Saint of Fort Was...	5.0	2
Star Kid (1997)	5.0	3
Great Day in Harl...	5.0	1
Prefontaine (1997)	5.0	3
Marlene Dietrich:...	5.0	1
Aiqing wansui (1994)	5.0	1
Santa with Muscle...	5.0	2
Pather Panchali (...	4.625	8
Maya Lin: A Stron...	4.5	4
Everest (1998)	4.5	2
Anna (1996)	4.5	2
Some Mother's Son...	4.5	2
Close Shave, A (1...	4.491071428571429	112
Schindler's List ...	4.466442953020135	298
Wrong Trousers, T...	4.466101694915254	118
Casablanca (1942)	4.45679012345679	243
Wallace & Gromit:...	4.447761194029851	67
Shawshank Redempt...	4.445229681978798	283
Rear Window (1954)	4.3875598086124405	209
Usual Suspects, T...	4.385767790262173	267
Star Wars (1977)	4.3584905660377355	583
12 Angry Men (1957)	4.344	125

OK I feel better now. Most of those obscure movies only had a few ratings.

By the way, We could also get this same result with a `sql` statement as follows:

```
sql("""select first(title),
              avg(rating) as avg_rating,
              count(itemid) from ratings
        group by itemid, movieid, title
         order by avg_rating DESC""").show
```

So for our final analysis, let's rank the top 10 movies by average rating, but only for those movies with 200 or more reviews. We can do that by simply adding a filter operation to the end of the chain.

```
ratings_with_titleDF.
  groupBy($"title").
  agg(avg($"rating").alias("avg_rating"),
      count($"rating").alias("n")).
  orderBy(desc("avg_rating")).where("n >= 200").show(10)
```

Now we see

title	avg_rating	n
Schindler's List ...	4.47	298
Casablanca (1942)	4.46	243
Shawshank Redempt...	4.45	283
Usual Suspects, T...	4.39	267
Rear Window (1954)	4.39	209
Star Wars (1977)	4.36	583
Silence of the La...	4.29	390
One Flew Over the...	4.29	264
To Kill a Mocking...	4.29	219
Godfather, The (1...	4.28	413

And with that result, our faith in our fellow moviegoers is restored.