Tutorial by Jordan DeLoach

1. Let's Go to Spark Shell

Download Spark, here's a link. Make sure that the examples from here, work.

2. Download Files

Have some tweets.

3. Load Files

Fire up Spark Shell. In your environment, run: `/bin/spark-shell/`.

Spark Shell is essentially a Scala environment, augmented with access to both a Spark and SQL context.

From the shell, run: `val df = spark.read.json("tweets.json")`

4. Do Some Simple Queries

With some data loaded in the Spark DataFrame format, we can now begin some basic Spark SQL.

4.1. Count

Running `df.count` on this dataset should yield 2770 tweets.

4.2. Schemas

SQL databases have tables which have schemas that provide the structure. DataFrame's have structure too, you can view that by executing `df.printSchema`.

Example subset of the results:

```
root
 |-- contributorsIDs: array (nullable = true)
 |    |-- element: string (containsNull = true)
 |-- createdAt: string (nullable = true)
 |-- currentUserRetweetId: long (nullable = true)
 |-- favoriteCount: long (nullable = true)
 |-- hashtagEntities: array (nullable = true)
 |    |-- element: struct (containsNull = true)
 |    |    |-- end: long (nullable = true)
 |    |    |-- start: long (nullable = true)
 |    |    |-- text: string (nullable = true)
 |-- id: long (nullable = true)
 |-- inReplyToScreenName: string (nullable = true)
 |-- inReplyToStatusId: long (nullable = true)
 |-- inReplyToUserId: long (nullable = true)
 |-- isFavorited: boolean (nullable = true)
 |-- isPossiblySensitive: boolean (nullable = true)
 |-- isRetweeted: boolean (nullable = true)
 |-- isTruncated: boolean (nullable = true)
 |-- lang: string (nullable = true)
 |-- mediaEntities: array (nullable = true)
```

4.3. Do Some Selects

If you want to get some information, you can execute a select statement. Executing `df.select("user.name", "text")` is partially rewarding, but it only returns what a schema of the result would look like, of the format: `org.apache.spark.sql.DataFrame = [name: string, text: string]`.

We would like to see some actually information, so if we append a `take` command to our original `select`, we can get some visible information.

Executing `df.select("user.name", "text").take(5)` gets us more of the results we are interested in.

Unfortunately, a lot of the reuslts, at least for me, are in Arabic. I do not speak Arabic and would like some results I can recognize, so I am going to select only English tweets. We can do that by ensuring the language triple equals "en."

`df.where(df("lang") === "en").select("user.name", "text").take(5)`

4.4. Write Some SQL

If you feel more comfortable writing traditional SQL, much of that is possible through registering a temporary table like so.

`df.createOrReplaceTempView("tweets")` (registerTempTable in Spark v 1.6.0)

Now, we can query the JSON data as if it was in a SQL DB.

`spark.sql("SELECT user.name, text FROM tweets WHERE lang = 'en'").take(5)`

4.5. Play Around

Now that we can do some simple SQL, let's start harnessing the power of Spark. Let's quickly take some Tweet text and do some simple clustering.

```
import org.apache.spark.ml.feature.Tokenizer
import org.apache.spark.ml.feature.HashingTF
import org.apache.spark.ml.feature.IDF
import org.apache.spark.ml.clustering.KMeans

val tokenizer = new Tokenizer().setInputCol("text").setOutputCol("words")
val wordsData = tokenizer.transform(df)
val hashingTF = new HashingTF().setInputCol("words").setOutputCol("rawFeatures")
val featurizedData = hashingTF.transform(wordsData)
val idf = new IDF().setInputCol("rawFeatures").setOutputCol("features")
val idfModel = idf.fit(featurizedData)
val rescaledData = idfModel.transform(featurizedData)
```

Now that we have some features, let's plug them into a K-Means clustering algorithm, with an arbitrary `k=15`.

```
val kmeans = new
KMeans().setK(15).setFeaturesCol("features").setPredictionCol("prediction")
val model = kmeans.fit(rescaledData)

// Shows the result
println("Final Centers: ")
model.clusterCenters.foreach(println)
```

And with that we took some Twitter JSON data, filtered with, applied TF-IDF filters and ran K-Means clustering. The best part is the scalability, the code that just worked and ran on your computer, would fly on an Hadoop cluster running YARN.