Big Data with Apache Spark (Part 2)

Lance Parlier
23 March 2017

Big Data with Apache Spark (Part 2)

A detailed real-world example of Big Data with Apache Spark.

This is an in depth look at a real-world example of Big Data with Apache Spark. In this presentation, we will look at a music recommendation system built with Apache Spark that uses machine learning. This program will suggest songs based on a user's listen history.

Big Data, Apache Spark, Recommendation Engine, Machine Learning

A Music Recommendation Engine

Before we get started, there are a few disclaimers:

- The code in the following examples is not meant to be used as examples as “clean code”. Some coding decisions may have made for the sake of simplifying an example, and may not be software engineering best practices.
- This example is an oversimplification of what is in use by very large corporations(think Pandora or Spotify). Their techniques are refined and very complex. This is just used as a base for explaining big data and machine learning.
- This is an ever-evolving field, so some techniques used here could already be outdated/depreciated, although everything should be current as of the presentation date.

The Data

The data we will be using is publically available from audioscrobbler:

- http://www-etud.iro.umontreal.ca/~bergstrj/audioscrobbler_data.html

This dataset contains the following files:

- user_artist_data.txt
  - Contains about 141,000 unique users. Each line lists the user_id, artist_id, play_count.
- artist_alias.txt
  - Each line contains 2 separate artist_ids. This is used to relate the same artists, which might have slightly different names. Example: snoop dog vs. Snoop Dogg
- artist_data.txt
  - This links the artist_ids to the artist name. Each line is an aritst_id and artist_name
  - Contains over 1.6 million unique artists

Setup

This will be a normal Scala program, which we will have everything in one file. We will split up major functionality into functions.

Imports:

```
import scala.collection.Map
import scala.collection.mutable.ArrayBuffer
import scala.util.Random
import org.apache.spark.{SparkConf, SparkContext}
import org.apache.spark.broadcast.Broadcast
import org.apache.spark.mllib.recommendation._
import org.apache.spark.rdd.RDD
```

Spark Context:

```
//Function to create the Spark Context, Returns the Spark Context
def createSparkContext(): SparkContext =
{
  val conf = new SparkConf().setAppName("LocalRecommendationSystem")
  val sc = new SparkContext(conf)
  sc
}
```

Formatting Data

We now need to format the data to be used by Spark:

```
//Function to format the data and produce the RDD[Rating]
def formatData(): RDD[Rating] =
{
  //Read in the data set of User Data
  val rawUserArtistData = sc.textFile("user_artist_data.txt")
  //Map the elements
  rawUserArtistData.map(_.split(' ')(0).toDouble).stats()
  rawUserArtistData.map(_.split(' ')(1).toDouble).stats()

  //Read in the artist alias'
  val rawArtistAlias = sc.textFile("artist_alias.txt")
  //Create a map of the artist alias
  val artistAlias = rawArtistAlias.flatMap { line =>
    val tokens = line.split('\t')
    if (tokens(0).isEmpty) {
      None
    } else {
      Some((tokens(0).toInt, tokens(1).toInt))
    }
  }.collectAsMap()
  //Create the broadcast artistAlias
  val bArtistAlias = sc.broadcast(artistAlias)
  //This creates the ratings for all of the data
  val allData = buildRatings(rawUserArtistData, bArtistAlias)
  allData
}
```

Building the Ratings

We now will build the ratings for the data:

```
//This function builds the ratings for all the data
//Function takes in RDD[String] and Broadcast[Map[Int, Int]] variable
def buildRatings(
                  rawUserArtistData: RDD[String],
                  bArtistAlias: Broadcast[Map[Int,Int]]) = {
  rawUserArtistData.map { line =>
    val Array(userID, artistID, count) = line.split(' ').map(_.toInt)
    val finalArtistID = bArtistAlias.value.getOrElse(artistID, artistID)
    Rating(userID, finalArtistID, count)
  }
}
```

A Word About Ratings

Spark's Machine Learning Library's API documentation has a great explanation of how the feedbacks/ratings work: https://spark.apache.org/docs/latest/mllib-collaborative-filtering.html:

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.mllib` 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.

The Model

In this project, we will be using an Alternating Least Squares model, which is built into Spark's Machine Learning Library:

- https://spark.apache.org/docs/1.1.0/api/java/org/apache/spark/mllib/recommendation/ALS.html

We will look for the best parameters to train the data, then train it with the parameters we found.

```
def trainImplicit(ratings: RDD[Rating], rank: Int, iterations: Int, lambda: Double, alpha: Double):
    MatrixFactorizationModel
```

Train a matrix factorization model given an RDD of 'implicit preferences' given by users to some products, in the form of (userID, productID, preference) pairs.

The parameters we will search for:

- rank: the number of latent factors in the model
- lambda: the regularization parameter in ALS
- alpha: a parameter applicable to the implicit feedback variant of ALS that governs the *baseline* confidence in preference observations.
- iterations: we will manually set this, ALS typically converges to a reasonable solution in 20 iterations or less.

Find Best Hyper-Parameters

Next, we will find the best hyper-parameters for the model:

```
//Function to search for the best hyperparameter combination
//Takes in the training data, cross-validation data, and the broadcasted test IDs
//Prints out and returns the best AUC value
//May throw ava.lang.OutOfMemoryError: Java heap space if RAM is not enough
def hyperparameterSearch(trainData: RDD[Rating], cvData: RDD[Rating], bTestIDs: Broadcast[Array[Int]]): ((Int, Double, Double), Double) =
{
  //Max to find the best AUC value
  var max = -999.999
  //Loop through and find the best parameters
  val evaluations = for (rank <- Array(10, 11);
                         lambda <- Array(1.0, 0.1);
                         alpha <- Array(1.0, 2.0))
    yield {
      val model = ALS.trainImplicit(trainData, rank, 10, lambda, alpha)
      val auc = areaUnderCurve(cvData, bTestIDs, model.predict)
      unpersist(model)
      //Find the max AUC value
      if(auc > max)
      {
        max = auc
      }
      ((rank, lambda, alpha), auc)
    }
  //Return max evaluations
  evaluations.sortBy(_._2).reverse.max
}
```

Building the Best Model

Now build the best model, based on the best parameters:

```
//Build the best model based on the best hyperparameter combination
def buildBestModel(bestRank: Int, bestLambda: Double, bestAlpha: Double, trainData: RDD[Rating]): MatrixFactorizationModel =
{
  //Create a model, use 5 iterations and the best rank, lambda, and alpha
  val model = ALS.trainImplicit(trainData, bestRank, 5, bestLambda, bestAlpha)
  model
}
```

Measuring the Model

We will use the area under the curve (AUC) as a measure of how precise the model is. We will calculate two different AUC values, one based on the best model, and one based on just recommending the most listened to artists(to use for comparison). *We are omitting the code for calculating the AUC, since it is fairly long, and does not help explain Big Data concepts.

```
//Get the AUC value given the test data and best model
def bestAUC(bestModel: MatrixFactorizationModel, testsData: Broadcast[Array[Int]], cvData: RDD[Rating]): Double =
{
  val auc = areaUnderCurve(cvData, testsData, bestModel.predict)
  auc
}

//Get the AUC value based on just recommending the most listened to artists
def genericAuc(cvData: RDD[Rating],  trainData: RDD[Rating], bTestIDs: Broadcast[Array[Int]]): (Double) = {
  val genericAucValue = areaUnderCurve(cvData, bTestIDs, predictMostListened(trainData))
  genericAucValue
}

//Predict the most listened to artist
def predictMostListened(train: RDD[Rating])(allData: RDD[(Int, Int)]) = {
  val bListenCount = sc.broadcast(train.map(r => (r.product, r.rating)).reduceByKey(_ + _).collectAsMap())
  allData.map {case(user, product) =>
    Rating(user, product, bListenCount.value.getOrElse(product, 0.0))
  }
}
```

More on Evaluating Metrics

Spark provides an entire page on evaluating metrics of a model:
https://spark.apache.org/docs/latest/mllib-evaluation-metrics.html

This page provides plenty of information on ways to evaluate your model:

Available metrics

Metric	Definition
Precision (Positive Predictive Value)	$PPV = \frac{TP}{TP+FP}$
Recall (True Positive Rate)	$TPR = \frac{TP}{P} = \frac{TP}{TP+FN}$
F-measure	$F(\beta) = (1+\beta^2) \cdot \left(\frac{PPV \cdot TPR}{\beta^2 \cdot PPV + TPR}\right)$
Receiver Operating Characteristic (ROC)	$FPR(T) = \int_T^{\infty} P_0(T)\,dT$ $TPR(T) = \int_T^{\infty} P_1(T)\,dT$
Area Under ROC Curve	$AUROC = \int_0^1 \frac{TP}{P} d\left(\frac{FP}{N}\right)$
Area Under Precision-Recall Curve	$AUPRC = \int_0^1 \frac{TP}{TP+FP} d\left(\frac{TP}{P}\right)$

Finally, Recommendations

We will use this function to look up artist recommendations:

```
def recommendArtists(recommendations: Array[Rating]) = {
  //Read in the artist data
  val rawArtistData = sc.textFile("artist_data.txt")

  val artistByID = rawArtistData.flatMap{line =>
    val (id, name) = line.span(_ !='\t')
    if(name.isEmpty){
      None}else{try{
      Some((id.toInt, name.trim))
    }catch{
      case e: NumberFormatException => None}}}

  recommendations.zipWithIndex.foreach{
    case (k, i) => println(i + " " + k + " " + artistByID.lookup(k.product).head)
  }
}
```

Putting It All Together

We will call the functions from the main, and get the results:

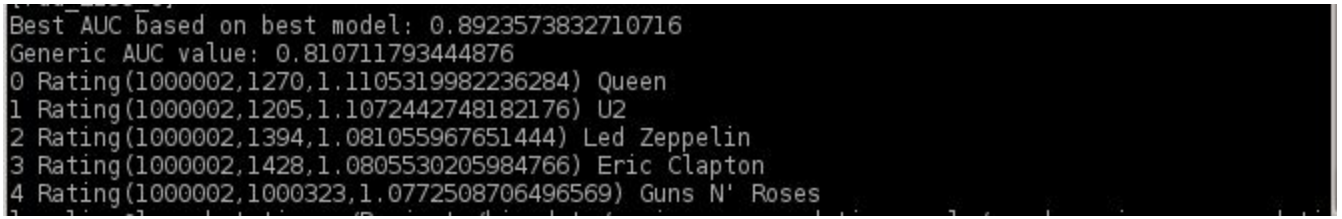
```
object MusicRecommendationLocal
{
  //Create a Spark Context
  //For simplicity, we will use this as a global variable
  val sc = createSparkContext()

  //Main method
  def main(args: Array[String]): Unit =
  {
    //This creates the ratings for all of the data from the formatData function call
    //Formats the data into a usable set
    val allData = formatData()

    //Create an array of trainData(60%), cvData(20%), and testingData(20%)f
    val Array(trainData, cvData, testingData) = allData.randomSplit(Array(0.6, 0.2, 0.2))

    //Cache the trainData, cvData, and testingData
    trainData.cache()
    cvData.cache()
    testingData.cache()

    //Create the test IDs and map them and ensure they are distinct
    val testIDs = testingData.map(_.product).distinct().collect()

    //Broadcast the test IDs
    val bTestIDs = sc.broadcast(testIDs)

    //Search for the best parameters, get them and put them into respective variables
    val bestParameters = hyperparameterSearch(trainData, cvData, bTestIDs)

    //Get the best model
    val bestModel = buildBestModel(bestParameters._1._1, bestParameters._1._2, bestParameters._1._3, trainData)
```

Putting It All Together cont.

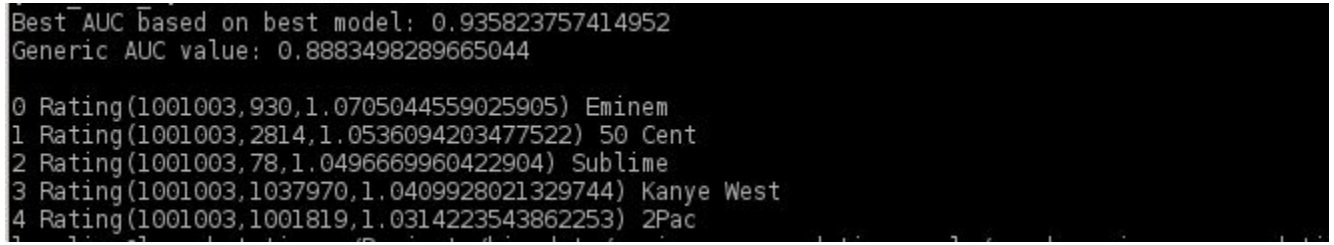
```
//Get the AUC from the best model
val bestAuc = bestAUC(bestModel, bTestIDs, cvData)
println("Best AUC based on best model: " + bestAuc)

val genericAucValue = genericAuc(cvData, trainData, bTestIDs)
println("Generic AUC value: " + genericAucValue)

//Select 5 recommendations based of of this user, who originally listened to the same Metallica
//song 320 times, and plenty of rock and roll songs
val recommendations = bestModel.recommendProducts(1000002,5)

//Select 5 recommendations based on this user, who listened to Ludacris, and several light rock bands
val recommendations2 = bestModel.recommendProducts(1001003, 5)

recommendArtists(recommendations2)
}
```

Recommendations

Recommendations for the user who liked rock and roll:



Recommendations for the user who liked hip hop:



Recommendations cont.

Results from running one more time:

```
Best AUC based on best model: 0.9344209590259456
Generic AUC value: 0.8873970530276176
0 Rating(1000002,1196,1.0332233477816375) Elvis Presley
1 Rating(1000002,1000323,1.0305955460775362) Guns N' Roses
2 Rating(1000002,1000188,1.0198661115519252) Dire Straits
3 Rating(1000002,1270,1.019088241597713 2) Queen
4 Rating(1000002,606,1.0169768400336596) The Rolling Stones

0 Rating(1001003,930,1.1148538732049502) Eminem
1 Rating(1001003,2814,1.0990355141358166) 50 Cent
2 Rating(1001003,1034635,1.0881169280065386) [unknown]
3 Rating(1001003,250,1.0805404676546455) Outkast
4 Rating(1001003,1007614,1.0714901602302) Jay-Z
```

Wrapping Up

As seen from the results, by training the models with the best hyper-parameters, we are getting fairly good AUC values.

We do get decent AUC values from just recommending the most listened to songs, this isn't customized, and would likely not be useful in many scenarios.

By using Spark's Machine Learning Library, and the built in ALS model, we save time while the built in functions do the heaving lifting. This isn't a one size fits all solution, but in this case it works well.