USING LOGISTIC REGRESSION, SCALA, AND SPARK

Here we explain how to do logistic regression with Apache Spark.

Logistic regression (LR) is closely related to linear regression. But instead of predicting a dependant value given some independent input values it predicts a probability and binary, yes or no, outcome.

You use linear or logistic regression when you believe there is some relationship between variables. For example, how many hours you study is obviously correlated with grades. So you could use linear or logistic regression with that.

Logistic regression outputs a 0 (false) or 1 (true). By convention if the probability of an event is > 50% then LR assigns a value of 1.

To illustrate how this can be used, we will plug data from a study of breast cancer patients at the University of Massachusetts into a simple LR model.

(This tutorial is part of our Apache Spark Guide. Use the right-hand menu to navigate.)

Breast cancer data

The description of the dataset is here. The data in Excel format is here. And the data in text file format is here.

Look at the description and Excel file. These are 50 patients who were diagnosed with breast disease and 150 who were not (the control group). The factors include age, number of miscarriages, etc. The diagnosis (1=yes 0=no) is in column D with column heading FNDX.

The good thing is all of this data is numeric and it is specifically laid out for a LR model. So all we have to do is create the required data structures to feed it into the Spark ML LR model.

To get started, download and then edit the .dat file to remove the introductory text and column headings. We will get rid of the spaces with code and also drop lines (patient records) that have missing values. We cannot put incomplete data into the model.

Comments on the data

Note that most people teaching this kind of programming first divide the input data into training and test datasets by dividing it into two pieces. But that is not exactly logical.

The purpose of the .dat file that we downloaded from the internet is to train the model. In other words, it uses least squares or other techniques to find the line that most nearly matches the input.

To put that into terms of statistics, it calculates the coefficients and intercepts to make some formula like y = mx + b where m is the coefficient, b is the intercept, and y is the predicted value. But here we have many more input variables than just x.

Having built that model, we would use that and plug in just one record from a patient who has come into the doctor's office. Then we can predict whether that patient is likely to have breast cancer. Already there are computers in doctor's offices doing exactly that.

Below is the Scala code. Run spark-shell to use the command line interpreter or use a Zeppelin notebook. We put comments in the code to explain it.

Scala code

```
import org.apache.spark.sql
import org.apache.spark.sql._
import org.apache.spark.sql.types._
import org.apache.spark.ml.classification.LogisticRegression
import org.apache.spark.mllib.linalg.{Vector, Vectors}
import org.apache.spark.ml.evaluation.BinaryClassificationEvaluator
import org.apache.spark.ml.feature.{VectorAssembler, StringIndexer}
/**
*  Missing values show up as a dot.  The dot function below
*  returns -1 if there is a dot or blank value. And it converts strings to
double.
* Then later we will
*  delete all rows that have any -1 values.
*/
def dot (s: String) : Double = {
if (s.contains(".") || s.length == 0) {
return -1
} else {
return s.toDouble
}
}
/**
```

```
*  We are going to use a Dataframe.  It requires a schema.
* So we create that below.  We use the same column names
*  as are in the .dat file.
*/
val schema = StructType (
StructField("STR", DoubleType, true) ::
StructField("OBS", DoubleType, true) ::
StructField("AGMT", DoubleType, true) ::
StructField("FNDX", DoubleType, true) ::
StructField("HIGD", DoubleType, true) ::
StructField("DEG",DoubleType, true) ::
StructField("CHK", DoubleType, true) ::
StructField("AGP1", DoubleType, true) ::
StructField("AGMN", DoubleType, true) ::
StructField("NLV", DoubleType, true) ::
StructField("LIV", DoubleType, true) ::
StructField("WT", DoubleType, true) ::
StructField("AGLP", DoubleType, true) ::
StructField("MST", DoubleType, true) ::  Nil)
/**
*  Read in the .dat file and use the regular expression
*  \s+ to split it by spaces into an RDD.
*/
val readingsRDD = spark.sparkContext.textFile("/home/walker/bbdm13.dat")
val RDD = readingsRDD.map(_.split("\\s+"))
/**
*   Run the dot function over every element in the RDD to convert them
*   to doubles, since that if the format requires by the Spark ML LR model.
*   Note that we skip the first one since that is just a blank space.
*/
val rowRDD = RDD.map(s =>
Row(dot(s(1)),dot(s(2)),dot(s(3)),dot(s(4)),dot(s(5)),dot(s(6)),
dot(s(7)),dot(s(8)),dot(s(9)),dot(s(10)),dot(s(11)),dot(s(12)),
dot(s(13)),dot(s(14))))
/**
* Now create a dataframe with the schema we described above,
*
*/
val readingsDF = spark.createDataFrame(rowRDD, schema)
/**
*  Create a new dataframe dropping all of those with missing values.
*/
var cleanDF = readingsDF.filter(readingsDF("STR") > -1 && readingsDF("OBS") >
-1 && readingsDF("AGMT")  > -1  && readingsDF("FNDX") > -1 &&
readingsDF("HIGD") > -1  && readingsDF("DEG") > -1 && readingsDF("CHK") > -1
&& readingsDF("AGP1") > -1  && readingsDF("AGMN") > -1  && readingsDF("NLV")
> -1  && readingsDF("LIV") > -1 && readingsDF("WT") > -1 &&
readingsDF("AGLP") > -1 && readingsDF("MST") > -1)
/**
*  Now comes something more complicated.  Our dataframe has the column
headings
*  we created with the schema.  But we need a column called “label” and one
called
* “features” to plug into the LR algorithm.  So we use the VectorAssembler()
to do that.
* Features is a Vector of doubles.  These are all the values like patient
age, etc. that
* we extracted above.  The label indicated whether the patient has cancer.
*/
val featureCols = Array("STR" , "OBS" , "AGMT" , "HIGD" , "DEG" , "CHK" ,
"AGP1" , "AGMN" , "NLV" , "LIV" , "WT" , "AGLP",  "MST" )
val assembler = new
VectorAssembler().setInputCols(featureCols).setOutputCol("features")
val df2 = assembler.transform(cleanDF)
/**
* Then we use the StringIndexer to take the column FNDX and make that the
label.
*  FNDX is the 1 or 0 indicator that shows whether the patient has cancer.
* Like the VectorAssembler it will add another column to the dataframe.
*/
val labelIndexer = new
StringIndexer().setInputCol("FNDX").setOutputCol("label")
val df3 = labelIndexer.fit(df2).transform(df2)
/**
*   Now we declare the LR model and run fit and transform to make
predictions.
*/
val model = new LogisticRegression().fit(df3)
val predictions = model.transform(df3)
/**
*  Now we print it out.  Notice that the LR algorithm added a “prediction”
column
*  to our dataframe.   The prediction in almost all cases will be the same as
the label.  That is
* to be expected it there is a strong correlation between these values.  In
other words
* if the chance of getting cancer was not closely related to these variables
then LR
* was the wrong model to use.  The way to check that is to check the accuracy
of the model.
*  You could use the BinaryClassificationEvaluator Spark ML function to do
that.
* Adding that would be a good exercise for you, the reader.
*/
```

```
predictions.select ("features", "label", "prediction").show()
+--------------------+-----+----------+
|            features|label|prediction|
+--------------------+-----+----------+
|[1.0,1.0,39.0,9.0...|  1.0|       1.0|
|[1.0,2.0,39.0,10....|  0.0|       0.0|
|[1.0,3.0,39.0,11....|  0.0|       0.0|
|[1.0,4.0,39.0,12....|  0.0|       0.0|
|[2.0,2.0,38.0,12....|  0.0|       0.0|
|[2.0,3.0,38.0,9.0...|  0.0|       0.0|
|[2.0,4.0,38.0,13....|  0.0|       0.0|
|[3.0,1.0,38.0,9.0...|  1.0|       1.0|
|[3.0,2.0,38.0,10....|  0.0|       0.0|
|[3.0,3.0,38.0,15....|  0.0|       0.0|
|[3.0,4.0,38.0,12....|  0.0|       0.0|
|[4.0,1.0,38.0,15....|  1.0|       1.0|
|[4.0,2.0,38.0,15....|  0.0|       0.0|
|[4.0,3.0,38.0,12....|  0.0|       0.0|
|[4.0,4.0,38.0,12....|  0.0|       0.0|
|[5.0,1.0,38.0,12....|  1.0|       1.0|
|[5.0,2.0,38.0,12....|  0.0|       0.0|
|[5.0,4.0,38.0,13....|  0.0|       0.0|
|[6.0,1.0,38.0,13....|  1.0|       1.0|
|[6.0,2.0,38.0,12....|  0.0|       0.0|
+--------------------+-----+----------+
only showing top 20 rows
```