Spark Decision Tree – Predicting Flight Delays

(In the code boxes, comments are in Green and output is in Blue)

|  |
| --- |
| import org.apache.spark.\_  import org.apache.spark.rdd.RDD  // Import classes for MLLib  import org.apache.spark.mllib.regression.LabeledPoint  import org.apache.spark.mllib.linalg.Vectors  import org.apache.spark.mllib.tree.DecisionTree  import org.apache.spark.mllib.tree.model.DecisionTreeModel  import org.apache.spark.mllib.util.MLUtils |

Below we a Scala case classes to define the Flight schema corresponding to the csv data file.

|  |
| --- |
| // define the Flight Schema  case class Flight(dofM: String, dofW: String, carrier: String, tailnum: String, flnum: Int, org\_id: String, origin: String, dest\_id: String, dest: String, crsdeptime: Double, deptime: Double, depdelaymins: Double, crsarrtime: Double, arrtime: Double, arrdelay: Double, crselapsedtime: Double, dist: Int) |

The function below parses a line from the data file into the Flight class.

|  |
| --- |
| // function to parse input into Flight class  def parseFlight(str: String): Flight = {  val line = str.split(",")  Flight(line(0), line(1), line(2), line(3), line(4).toInt, line(5), line(6), line(7), line(8), line(9).toDouble, line(10).toDouble, line(11).toDouble, line(12).toDouble, line(13).toDouble, line(14).toDouble, line(15).toDouble, line(16).toInt)  } |

Below we load the data from the csv file into a [Resilient Distributed Dataset (RDD)](https://spark.apache.org/docs/0.8.1/api/core/org/apache/spark/rdd/RDD.html). RDDs can have [transformations](https://spark.apache.org/docs/1.3.0/programming-guide.html#transformations) and [actions](https://spark.apache.org/docs/1.3.0/programming-guide.html#actions), the first() action returns the first element in the RDD.

|  |
| --- |
| // load the data into a RDD  val textRDD = sc.textFile("/user/user01/data/rita2014jan.csv")  // MapPartitionsRDD[1] at textFile  // parse the RDD of csv lines into an RDD of flight classes  val flightsRDD = textRDD.map(parseFlight).cache()  flightsRDD.take(1)  //Array(Flight(1,3,AA,N338AA,1,12478,JFK,12892,LAX,900.0,914.0,14.0,1225.0,1238.0,13.0,385.0,2475), |

We define airports as vertices. Vertices can have properties or attributes associated with them. Each vertex has the following property:

* Airport name (String)

Vertex Table for Airports

|  |  |
| --- | --- |
| ID | Property(V) |
| 10397 | ATL |

We define an RDD with the above properties that is then used for the Vertexes .

|  |
| --- |
| // create airports RDD with ID and Name  var carrierMap: Map[String, Int] = Map()  var index: Int = 0  flightsRDD.map(flight => flight.carrier).distinct.collect.foreach(x => { carrierMap += (x -> index); index += 1 })  carrierMap.toString  //res2: String = Map(DL -> 5, F9 -> 10, US -> 9, OO -> 2, B6 -> 0, AA -> 6, EV -> 12, FL -> 1, UA -> 4, MQ -> 8, WN -> 13, AS -> 3, VX -> 7, HA -> 11)  // Defining a default vertex called nowhere  var originMap: Map[String, Int] = Map()  var index1: Int = 0  flightsRDD.map(flight => flight.origin).distinct.collect.foreach(x => { originMap += (x -> index1); index1 += 1 })  originMap.toString  //res4: String = Map(JFK -> 214, LAX -> 294, ATL -> 273,MIA -> 175 ...  // Map airport ID to the 3-letter code to use for printlns  var destMap: Map[String, Int] = Map()  var index2: Int = 0  flightsRDD.map(flight => flight.dest).distinct.collect.foreach(x => { destMap += (x -> index2); index2 += 1 }) |

## Define Edges

Edges are the routes between airports. An edge must have a source, a destination, and can have properties. In our example, an edge consists of :

* Edge origin id 🡪 src (Long)
* Edge destination id 🡪 dest (Long)
* Edge Property distance 🡪 distance (Long)

Edges Table for Routes

|  |  |  |
| --- | --- | --- |
| srcid | destid | Property(E) |
| 14869 | 14683 | 1087 |

We define an RDD with the above properties that is then used for the Edges . The edge RDD has the form (src id, dest id, distance ).

|  |
| --- |
| // create routes RDD with srcid, destid , distance  //- Defining the features array  val mlprep = flightsRDD.map(flight => {  val monthday = flight.dofM.toInt - 1 // category  val weekday = flight.dofW.toInt - 1 // category  val crsdeptime1 = flight.crsdeptime.toInt  val crsarrtime1 = flight.crsarrtime.toInt  val carrier1 = carrierMap(flight.carrier) // category  val crselapsedtime1 = flight.crselapsedtime.toDouble  val origin1 = originMap(flight.origin) // category  val dest1 = destMap(flight.dest) // category  val delayed = if (flight.depdelaymins.toDouble > 40) 1.0 else 0.0  Array(delayed.toDouble, monthday.toDouble, weekday.toDouble, crsdeptime1.toDouble, crsarrtime1.toDouble, carrier1.toDouble, crselapsedtime1.toDouble, origin1.toDouble, dest1.toDouble)  })  mlprep.take(1)  //res6: Array[Array[Double]] = Array(Array(0.0, 0.0, 2.0, 900.0, 1225.0, 6.0, 385.0, 214.0, 294.0)) |

## Create Property Graph

To create a graph, you need to have a Vertex RDD, Edge RDD and a Default vertex.

Create a property graph called graph.

|  |
| --- |
| // define the graph  //Making LabeledPoint of features - this is the training data for the model  val mldata = mlprep.map(x => LabeledPoint(x(0), Vectors.dense(x(1), x(2), x(3), x(4), x(5), x(6), x(7), x(8))))  mldata.take(1)  //res7: Array[org.apache.spark.mllib.regression.LabeledPoint] = Array((0.0,[0.0,2.0,900.0,1225.0,6.0,385.0,214.0,294.0]))  // mldata0 is %85 not delayed flights  val mldata0 = mldata.filter(x => x.label == 0).randomSplit(Array(0.85, 0.15))(1)  // mldata1 is %100 delayed flights  val mldata1 = mldata.filter(x => x.label != 0)  // mldata2 is delayed and not delayed  val mldata2 = mldata0 ++ mldata1  // split mldata2 into training and test data  val splits = mldata2.randomSplit(Array(0.7, 0.3))  val (trainingData, testData) = (splits(0), splits(1))  testData.take(1)  //res21: Array[org.apache.spark.mllib.regression.LabeledPoint] = Array((0.0,[18.0,6.0,900.0,1225.0,6.0,385.0,214.0,294.0])) |

1. How many airports are there?

|  |
| --- |
| // How many airports?  // set ranges for 0=dofM 1=dofW 4=carrier 6=origin 7=dest  var categoricalFeaturesInfo = Map[Int, Int]()  categoricalFeaturesInfo += (0 -> 31)  categoricalFeaturesInfo += (1 -> 7)  categoricalFeaturesInfo += (4 -> carrierMap.size)  categoricalFeaturesInfo += (6 -> originMap.size)  categoricalFeaturesInfo += (7 -> destMap.size)  val numClasses = 2  // Defning values for the other parameters  val impurity = "gini"  val maxDepth = 9  val maxBins = 7000 |

1. How many routes are there?

|  |
| --- |
| // How many airports?  val model = DecisionTree.trainClassifier(trainingData, numClasses, categoricalFeaturesInfo,  impurity, maxDepth, maxBins)  model.toDebugString  // 0=dofM 4=carrier 3=crsarrtime1 6=origin  res20: String =  DecisionTreeModel classifier of depth 9 with 919 nodes  If (feature 0 in {11.0,12.0,13.0,14.0,15.0,16.0,17.0,18.0,19.0,20.0,21.0,22.0,23.0,24.0,25.0,26.0,27.0,30.0})  If (feature 4 in {0.0,1.0,2.0,3.0,4.0,5.0,6.0,7.0,8.0,9.0,10.0,11.0,13.0})  If (feature 3 <= 1603.0)  If (feature 0 in {11.0,12.0,13.0,14.0,15.0,16.0,17.0,18.0,19.0})  If (feature 6 in {0.0,1.0,2.0,3.0,4.0,5.0,6.0,7.0,8.0,10.0,11.0,12.0,13.0... |

1. which routes > 1000 miles distance?

|  |
| --- |
| // routes > 1000 miles distance?  val labelAndPreds = testData.map { point =>  val prediction = model.predict(point.features)  (point.label, prediction)  }  labelAndPreds.take(3)  res33: Array[(Double, Double)] = Array((0.0,0.0), (0.0,0.0), (0.0,0.0))  val wrongPrediction =(labelAndPreds.filter{  case (label, prediction) => ( label !=prediction)  })  wrongPrediction.count()  res35: Long = 11040  val ratioWrong=wrongPrediction.count().toDouble/testData.count()  ratioWrong: Double = 0.3157443157443157 |