This tutorial will run on the MapR Sandbox, which includes Spark.

* You can download the code and data to run these examples from here: <https://github.com/caroljmcdonald/sparkmldecisiontree>
* The examples in this post can be run in the Spark shell, after launching with the spark-shell command.
* You can also run the code as a standalone application as described in the tutorial on [Getting Started with Spark on MapR Sandbox](https://mapr.com/products/mapr-sandbox-hadoop/tutorials/spark-tutorial).

Log into the MapR Sandbox, as explained in [Getting Started with Spark on MapR Sandbox](https://mapr.com/products/mapr-sandbox-hadoop/tutorials/spark-tutorial), using userid user01, password mapr. Copy the sample data file to your sandbox home directory /user/user01 using scp. Start the spark shell with:

$ spark-shell

**Load and Parse the Data from a csv File**

First, we will import the machine learning packages.  
(In the code boxes, comments are in Green and output is in Blue)

import org.apache.spark.\_

import org.apache.spark.rdd.RDD

<font color="green">\*\*// Import classes for MLLib\*\*</font>

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

In our example, each flight is an item, and we use a Scala case class to define the Flight schema corresponding to a line in the csv data file.

<font color="green">\*\*// define the Flight Schema\*\*</font>

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.

<font color="green">\*\*// function to parse input into Flight class\*\*</font>

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)

}

We use the flight data for January 2014 as the dataset. 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.

<font color="green">\*\*// load the data into a RDD\*\*</font>

val textRDD = sc.textFile("/user/user01/data/rita2014jan.csv")

<font color="blue">\*\*// MapPartitionsRDD[1] at textFile\*\*</font>

<font color="green">\*\*// parse the RDD of csv lines into an RDD of flight classes\*\*</font>

val flightsRDD = textRDD.map(parseFlight).cache()

flightsRDD.first()

//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),

**Extract Features**

To build a classifier model, first extract the features that most contribute to the classification. We are defining two classes or labels – Yes (delayed) and No (not delayed). A flight is considered to be delayed if it is more than 40 minutes late.

The features for each item consists of the fields shown below:

* Label → delayed and not delayed - delayed if delay > 40 minutes
* Features → {day\_of\_month, weekday, crsdeptime, crsarrtime, carrier, crselapsedtime, origin, dest, delayed}

Below we transform the non-numeric features into numeric values. For example, the carrier AA is the number 6. The originating airport ATL is 273.

<font color="green">\*\*// create airports RDD with ID and Name\*\*</font>

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

<font color="blue">\*\*//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)\*\*</font>

<font color="green">\*\*// Defining a default vertex called nowhere\*\*</font>

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

<font color="blue">\*\*//res4: String = Map(JFK -> 214, LAX -> 294, ATL -> 273,MIA -> 175 ...\*\*</font>

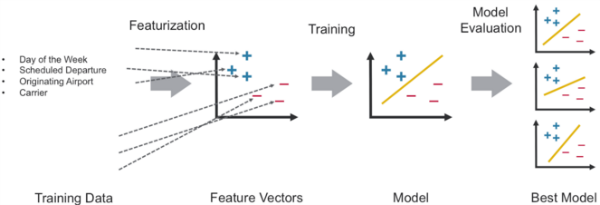
<font color="green">\*\*// Map airport ID to the 3-letter code to use for printlns\*\*</font>

var destMap: Map[String, Int] = Map()

var index2: Int = 0

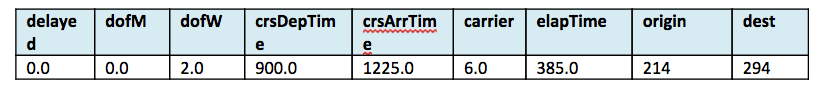
flightsRDD.map(flight => flight.dest).distinct.collect.foreach(x => { destMap += (x -> index2); index2 += 1 })

**Define Features Array**



The features are transformed and put into Feature Vectors, which are vectors of numbers representing the value for each feature.

Next, we create an RDD containing feature arrays consisting of the label and the features in numeric format. An example is shown in this table:



<font color="green">\*\*//- Defining the features array\*\*</font>

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)

<font color="blue">\*\*//res6: Array[Array[Double]] = Array(Array(0.0, 0.0, 2.0, 900.0, 1225.0, 6.0, 385.0, 214.0, 294.0))\*\*</font>

**Create Labeled Points**

From the RDD containing feature arrays, we create an RDD containing arrays of [LabeledPoints](http://spark.apache.org/docs/latest/api/scala/index.html#org.apache.spark.mllib.regression.LabeledPoint). A labeled point is a class that represents the feature vector and label of a data point.

<font color="green">\*\*//Making LabeledPoint of features - this is the training data for the model\*\*</font>

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)

<font color="blue">\*\*//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]))\*\*</font>

Next the data is split to get a good percentage of delayed and not delayed flights. Then it is split into a training data set and a test data set

<font color="green">\*\*// mldata0 is %85 not delayed flights\*\*</font>

val mldata0 = mldata.filter(x => x.label == 0).randomSplit(Array(0.85, 0.15))(1)

<font color="green">\*\*// mldata1 is %100 delayed flights\*\*</font>

val mldata1 = mldata.filter(x => x.label != 0)

<font color="green">\*\*// mldata2 is delayed and not delayed\*\*</font>

val mldata2 = mldata0 ++ mldata1

<font color="green">\*\*// split mldata2 into training and test data\*\*</font>

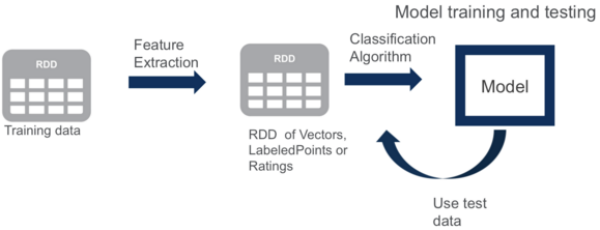
val splits = mldata2.randomSplit(Array(0.7, 0.3))

val (trainingData, testData) = (splits(0), splits(1))

testData.take(1)

<font color="blue">\*\*//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]))\*\*</font>

**Train the Model**



Next, we prepare the values for the [parameters that are required for the Decision Tree:](http://spark.apache.org/docs/latest/mllib-decision-tree.html)

* categoricalFeaturesInfo specifies which features are categorical and how many categorical values each of those features can take. This is given as a map from feature index to the number of categories for that feature. The first categorical feature, categoricalFeaturesInfo = (0 -> 31) specifies that feature index 0 (which represents the day of the month) has 31 categories (values {0, ..., 31}). The second , categoricalFeaturesInfo = (1 -> 7), represents days of the week, and specifies that feature index 1 has 7 categories. The carrier categorial feature is index 4 and the value can go from 0 to the number of distinct carriers , and so on.
* maxDepth: Maximum depth of a tree.
* maxBins: Number of bins used when discretizing continuous features.
* impurity: Impurity measure of the homogeneity of the labels at the node.

The model is trained by making associations between the input features and the labeled output associated with those features. We train the model using the DecisionTree.trainClassifier method which returns a DecisionTreeModel.

<font color="green">\*\*// set ranges for 0=dofM 1=dofW 4=carrier 6=origin 7=dest\*\*</font>

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

<font color="green">\*\*// call DecisionTree trainClassifier with the trainingData , which returns the model\*\*</font>

val model = DecisionTree.trainClassifier(trainingData, numClasses, categoricalFeaturesInfo,

impurity, maxDepth, maxBins)

<font color="green">\*\*// print out the decision tree\*\*</font>

model.toDebugString

<font color="green">\*\*// 0=dofM 4=carrier 3=crsarrtime1 6=origin \*\*</font>

<font color="blue">\*\*res20: String =

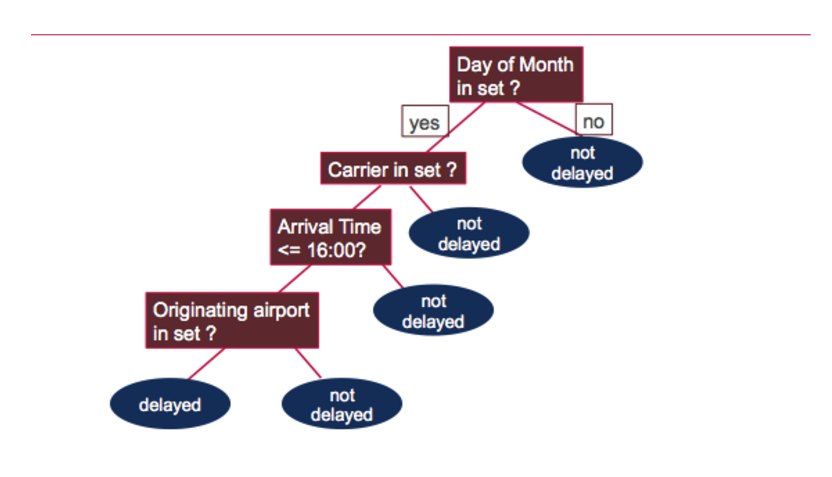
DecisionTreeModel classifier of depth 9 with 919 nodes

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 <= 0="" 6="" 1603.0)="" if="" (feature="" in="" {11.0,12.0,13.0,14.0,15.0,16.0,17.0,18.0,19.0})="" {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...<="" strong="">\*\*</font>

Model.toDebugString prints out the decision tree, which asks the following questions to determine if the flight was delayed or not:



**Test the Model**

Next we use the test data to get predictions.Then we compare the predictions of a flight delay to the actual flight delay value, the label. The wrong prediction ratio is the count of wrong predictions / the count of test data values, which is 31%.

\_<font color="green">\*\*// Evaluate model on test instances and compute test error\*\*</font>\_

val labelAndPreds = testData.map { point =>

val prediction = model.predict(point.features)

(point.label, prediction)

}

labelAndPreds.take(3)

<font color="blue">\*\*res33: Array[(Double, Double)] = Array((0.0,0.0), (0.0,0.0), (0.0,0.0))\*\*</font>

val wrongPrediction =(labelAndPreds.filter{

case (label, prediction) => ( label !=prediction)

})

wrongPrediction.count()

<font color="blue">\*\*res35: Long = 11040\*\*</font>

val ratioWrong=wrongPrediction.count().toDouble/testData.count()

<font color="blue">\*\*ratioWrong: Double = 0.3157443157443157\*\*</font>

**Want to learn more?**

* [Free On Demand Spark Training](http://learn.mapr.com/)
* <http://spark.apache.org/docs/latest/mllib-decision-tree.html>

In this blog post, we showed you how to get started using Apache Spark’s MLlib machine learning decision trees for classification. If you have any further questions about this tutorial, please ask them in the comments section below.