PREDICTIVE AND PREVENTIVE MAINTENANCE USING IOT, MACHINE LEARNING & APACHE SPARK

Here we explain a use case of how to use Apache Spark and machine learning. This is the classic preventive maintenance problem, one of the most common business use cases of machine learning and IoT too. We take the data for this analysis from the Kaggle website, a site dedicated to data science. This is sensor data from machines, specifically moisture, temperature, and pressure. The goal is to predict which machines needs to be taken out of service for maintenance. The code we have written is available here.

The Data Explained

The raw data is here.

These is a downside to the data that we have here, which is this is run-to-failure data. The goal of PM is not to run a machine until it breaks down. Rather is it to keep the machine in working order.

Architecture

- Apache Spark
- Apache Hadoop
- Scala
- Spark Machine Learning API

We write three programs:

1. create a logistic regression training model
2. create some sample data by taking actual data and adding noise based upon the standard deviation of that
3. feed data into model to show which vehicles need maintenance

We explain the first two steps here. In a second blog post we will explain item #3.

Create a Training Model

This program reads data and saves a **logistic regression model**. The second program then creates data given the mean, stddev, max, and minn of the variables in that training set. Then the last program runs predictions and prints out those records that are flagged with 1. With logistic regression 1 means true, which in this example means the machine requires maintenance based upon our prediction.

build.sbt

In order to compile the Scala code below you need sbt (the Scala Build tool) and this build.sbt file. This tells Scala which files to add when it builds the Jar file that we will submit to Apache Spark.

```
name         := "lr"
version      := "1.0"
organization := "com.bmc"
assemblyJarName in assembly := "bmclr.jar"
scalaVersion := "2.11.8"
mainClass := Some("com.bmc.lr")
libraryDependencies ++= Seq("org.apache.spark" %% "spark-core" % "2.3.0" %
"provided", "com.databricks" %% "spark-csv" % "1.5.0", "org.apache.spark" %%
"spark-sql" % "2.3.0" % "provided", "org.apache.spark" %% "spark-mllib" %
"2.3.0" % "provided")
resolvers += Resolver.mavenLocal
```

Note: if you get error: not found: value assemblyassemblyJarName in assembly then you need to add the sbt-assembly plugin. So in file project/assembly.sbt add:

addSbtPlugin("com.eed3si9n" % "sbt-assembly" % "0.14.8")

Run Training Model

In order to run the code below you need to have Hadoop started and then submit the job to Apache Spark like this.

The parameters are:

- jar file to read to find class class com.bmc.lr.readCSV
- location of the maintenance data .csv file
- where to store the saved model (file must not exist). You need to **hdfs fs -mkdir /maintenance** to create this folder.

```
spark-submit
```

```
--verbose
--class com.bmc.lr.readCSV
--master local
hdfs://localhost:9000/maintenance/lr-assembly-1.0.jar
hdfs://localhost:9000/maintenance/maintenance_data.csv
hdfs://localhost:9000/maintenance/maintenance_model
```

Training Model Code

Now we explain the code.

First we import Apache Spark linear algebra, machine learning, databricks, and other APIs we will need. We have to give this program a package name since scala is compiled to Java byte code and we will make a Jar file from this.

```
package com.bmc.lr
import org.apache.spark.sql.SQLContext
import org.apache.spark.SparkConf
import org.apache.spark.SparkContext
import com.databricks.spark.csv
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}
```

We make an **object** with a **main** method that we can pass objects too. We have to create the **SparkContext** and **SQLContext** specifically since we are not running in the command-line interpreter where those are created for us already. For **appName** we can set any unique value.

```
object readCSV {
def main(args: Array): Unit = {
val conf = new SparkConf().setAppName("lr")
val sc = new SparkContext(conf)
var file = args(0);
val sqlContext = new SQLContext(sc)
```

We use **databricks** to read from a .csv file to make a dataframe.

```
val df = sqlContext.read.format("com.databricks.spark.csv").option("header",
"true").option("inferSchema", "true").option("delimiter",";").load(file)
df.show()
```

Now we build the **features** (input variables) and **labels** (output variables). Since we are doing logistic regression there is only 1 label: broken (1 or 0).
We create a model and then run the train method. Finally we save it to the Hadoop file system.

```
val featureCols =  Array("lifetime", "pressureInd", "moistureInd",
"temperatureInd")
val assembler = new
VectorAssembler().setInputCols(featureCols).setOutputCol("features")
```

```
val labelIndexer = new
StringIndexer().setInputCol("broken").setOutputCol("label")
val df2 = assembler.transform(df)
val df3 = labelIndexer.fit(df2).transform(df2)
val model = new LogisticRegression().fit(df3)
model.save(args(1))
println("Training model saved as $args(1)")
}
}
```

Generate Data

We use the next program to create sample data, drawing on the original data file and generate a range of values based on the max, min, and standard deviation distribution of the data.

The arguments to this program are:

- how many records to create
- where is Hadoop core-site.xml
- input data file

We do not specify the output file, since Hadoop does not create one file. Instead we hard-code a folder below. After the code we explain how to view the data.

```
spark-submit
--class com.bmc.lr.generateData
--master local
hdfs://localhost:9000/maintenance/lr-assembly-1.0.jar
1000
/usr/local/sbin/hadoop-3.1.0/etc/hadoop/core-site.xml
hdfs://localhost:9000/maintenance/maintenance_data.csv
```

When the program runs stdout looks something like this. If you run 1,000 records it will take some minutes to run.

```
018-05-15 09:25:46 INFO  MemoryStore:54 - Block broadcast_15_piece0 stored as
bytes in memory (estimated size 6.6 KB, free 364.6 MB)
2018-05-15 09:25:46 INFO  BlockManagerInfo:54 - Added broadcast_15_piece0 in
memory on ip-172-31-13-71.eu-west-1.compute.internal:35220 (size: 6.6 KB,
free: 366.1 MB)
2018-05-15 09:25:46 INFO  SparkContext:54 - Created broadcast 15 from
broadcast at DAGScheduler.scala:1039
2018-05-15 09:25:46 INFO  DAGScheduler:54 - Submitting 1 missing tasks from
ResultStage 9 (MapPartitionsRDD at describe at generateData.scala:76) (first
15 tasks are for partitions Vector(0))
2018-05-15 09:25:46 INFO  TaskSchedulerImpl:54 - Adding task set 9.0 with 1
tasks
2018-05-15 09:25:46 INFO  TaskSetManager:54 - Starting task 0.0 in stage 9.0
(TID 9, localhost, executor driver, partition 0, ANY, 7754 bytes)
```

2018-05-15 09:25:47 INFO DAGScheduler:54 - Parents of final stage: List(ShuffleMapStage 16)
2018-05-15 09:25:47 INFO DAGScheduler:54 - Missing parents: List(ShuffleMapStage 16)
2018-05-15 09:25:47 INFO BlockManagerInfo:54 - Removed broadcast_21_piece0 on ip-172-31-13-71.eu-west-1.compute.internal:35220 in memory (size: 6.6 KB, free: 366.1 MB)
2018-05-15 09:25:47 INFO DAGScheduler:54 - Submitting ShuffleMapStage 16 (MapPartitionsRDD at describe at generateData.scala:76), which has no missing parents
2018-05-15 09:25:47 INFO ContextCleaner:54 - Cleaned accumulator 189
2018-05-15 09:25:47 INFO ContextCleaner:54 - Cleaned accumulator 258
2018-05-15 09:25:47 INFO ContextCleaner:54 - Cleaned accumulator 399
2018-05-15 09:25:47 INFO ContextCleaner:54 - Cleaned accumulator 75
2018-05-15 09:25:47 INFO MemoryStore:54 - Block broadcast_26_piece0 stored as bytes in memory (estimated size 9.4 KB, free 364.4 MB)
2018-05-15 09:25:47 INFO BlockManagerInfo:54 - Added broadcast_26_piece0 in memory on ip-172-31-13-71.eu-west-1.compute.internal:35220 (size: 9.4 KB, free: 366.1 MB)
2018-05-15 09:25:47 INFO SparkContext:54 - Created broadcast 26 from broadcast at DAGScheduler.scala:1039
2018-05-15 09:25:47 INFO BlockManagerInfo:54 - Removed broadcast_7_piece0 on ip-172-31-13-71.eu-west-1.compute.internal:35220 in memory (size: 23.4 KB, free: 366.1 MB)
2018-05-15 09:25:47 INFO DAGScheduler:54 - Submitting 1 missing tasks from ShuffleMapStage 16 (MapPartitionsRDD at describe at generateData.scala:76) (first 15 tasks are for partitions Vector(0))
2018-05-15 09:25:47 INFO TaskSchedulerImpl:54 - Adding task set 16.0 with 1 tasks
2018-05-15 09:25:47 INFO TaskSetManager:54 - Starting task 0.0 in stage 16.0 (TID 16, localhost, executor driver, partition 0, ANY, 8308 bytes)
2018-05-15 09:25:47 INFO Executor:54 - Running task 0.0 in stage 16.0 (TID 16)
2018-05-15 09:25:47 INFO ContextCleaner:54 - Cleaned accumulator 211
2018-05-15 09:25:47 INFO BlockManagerInfo:54 - Removed broadcast_22_piece0 on ip-172-31-13-71.eu-west-1.compute.internal:35220 in memory (size: 23.4 KB, free: 366.2 MB)
2018-05-15 09:25:47 INFO FileScanRDD:54 - Reading File path: hdfs://localhost:9000/maintenance/maintenance_data.csv, range: 0-72679, partition values:
2018-05-15 09:25:47 INFO MemoryStore:54 - Block broadcast_27_piece0 stored as bytes in memory (estimated size 6.6 KB, free 365.7 MB)
2018-05-15 09:25:47 INFO BlockManagerInfo:54 - Added broadcast_27_piece0 in memory on ip-172-31-13-71.eu-west-1.compute.internal:35220 (size: 6.6 KB, free: 366.2 MB)
2018-05-15 09:25:47 INFO SparkContext:54 - Created broadcast 27 from broadcast at DAGScheduler.scala:1039

```
2018-05-15 09:25:47 INFO  DAGScheduler:54 - Submitting 1 missing tasks from
ResultStage 17 (MapPartitionsRDD at describe at generateData.scala:76) (first
15 tasks are for partitions Vector(0))
2018-05-15 09:25:47 INFO  TaskSchedulerImpl:54 - Adding task set 17.0 with 1
tasks
2018-05-15 09:25:47 INFO  TaskSetManager:54 - Starting task 0.0 in stage 17.0
(TID 17, localhost, executor driver, partition 0, ANY, 7754 bytes)
2018-05-15 09:25:47 INFO  Executor:54 - Running task 0.0 in stage 17.0 (TID
17)
2018-05-15 09:25:47 INFO  ShuffleBlockFetcherIterator:54 - Gett
```

Code

We start with the imports.

```
import org.apache.spark.SparkConf
import org.apache.spark.sql.SQLContext
import org.apache.spark.SparkContext
import org.apache.commons.math3.distribution.NormalDistribution
import org.apache.spark.sql.SQLContext
import java.io.DataOutputStream
import java.io.BufferedWriter
import org.apache.hadoop.fs.FSDataOutputStream
import java.io.OutputStreamWriter
import org.apache.hadoop.conf.Configuration
import org.apache.hadoop.fs.Path
import org.apache.hadoop.fs.FileSystem
import java.util.Date
import java.text.SimpleDateFormat
```

And this must be an object as we mentioned above.

```
object generateData {
```

We use org.apache.commons.math3.distribution.NormalDistribution to generate random numbers from a standard deviation based upon the data in each column and subject to a mean and max. In other words we need to simulate engines operating all all levels: normal, broken, and soon to require maintenance.

```
def generateData (mean: Double, stddev: Double, max: Double, min:Double) :
Double = {
var x:NormalDistribution = new NormalDistribution(stddev,mean)
var y:Double = x.sample()
while( (y >= max) || (y <= min) ) {
y = x.sample()
}
return y
}
def createData(x: org.apache.spark.sql.DataFrame) : Double = {
```

```
var y:Array = x.collect();
var mean:Double = y(1)(1).toString.toDouble;
var stddev:Double = y(2)(1).toString.toDouble;
var min:Double = y(3)(1).toString.toDouble;
var max:Double = y(4)(1).toString.toDouble;
return generateData(mean,stddev,max,min);
}
```

The usual **main()** function.

```
def main(args: Array): Unit = {
val conf = new SparkConf().setAppName("lr")
val sc = new SparkContext(conf)
var records:Int = args(0).toInt;
var hdfsCoreSite = args(1)
var file = args(2)
```

Create SQLContext and read data file into a dataframe using databricks. Indicate what we want to call out column heading in the output file.

```
val sqlContext = new SQLContext(sc)
val df = sqlContext.read.format("com.databricks.spark.csv").option("header",
"true").option("inferSchema", "true").option("delimiter",";").load(file)
var header =
"lifetime;broken;pressureInd;moistureInd;temperatureInd;team;provider"
```

Ths simulation here is that we receive IoT (internet of things) data on some frequency. So we save files in folders with the format YMDdhms

Below that we write the data to the Hadoop file system.

```
val date = new Date()
var dformat:SimpleDateFormat = new SimpleDateFormat("yyyy.MM.dd.HH.mm.ss");
val csvFile =  "/maintenance/" + dformat.format(date) + ".csv"
println("writing to " + csvFile)
val fs = {
val conf = new Configuration()
conf.addResource(new Path(hdfsCoreSite))
FileSystem.get(conf)
}
val dataOutputStream: FSDataOutputStream = fs.create(new Path(csvFile))
val bw: BufferedWriter = new BufferedWriter(new
OutputStreamWriter(dataOutputStream, "UTF-8"))
println(header)
bw.write(header + "\n")
```

Generate random data as described above and save it.

```
val r = scala.util.Random
var i:Int = 0
```

```
while (i < records ) {
var pressureInd =  createData(df.describe("pressureInd"))
var moistureInd =  createData(df.describe("moistureInd"))
var temperatureInd =  createData(df.describe("temperatureInd"))
var lifetime =  createData(df.describe("lifetime"))
var str = lifetime + ";" + "0" + ";" + pressureInd + ";" + moistureInd + ";"
+ temperatureInd +";" + r.nextInt(100) + ";" + "Ford F-750"
println (str)
bw.write(str + "\n")
i = i + 1
}
bw.close
}
}
```

View the Output Data

The output file is store in Hadoop. So you must use Hadoop commands to view it. Remember that Hadoop is a distributed file system. So it assembles files in **parts**. In the example I have run here there is only part, which you can see with **hadoop fs ls**.

```
hadoop fs -ls /maintenance/2018.04.20.16.26.53.csv
Found 2 items
-rw-r--r--   3 ubuntu supergroup          0 2018-04-20 16:26
/maintenance/2018.04.20.16.26.53.csv/_SUCCESS
-rw-r--r--   3 ubuntu supergroup      17416 2018-04-20 16:26
/maintenance/2018.04.20.16.26.53.csv/part-00000-b1d37f5f-5021-4368-86fd-
d941497d8b52-c000.csv
```

To look at this file use cat.

```
hadoop fs -cat
/maintenance/2018.04.20.16.26.53.csv/part-00000-b1d37f5f-5021-4368-86fd-
d941497d8b52-c000.csv
team,provider,pressureInd,moistureInd,temperatureInd,label,prediction
63,Ford F-750,107.60039392741436,89.98427587791616,48.217222871678814,0.0,1.0
98,Ford F-750,43.28868205264517,127.8055095809048,96.48049423573129,0.0,1.0
23,Ford F-750,122.53982028285051,127.73394439569482,98.44610180531744,0.0,1.0
81,Ford F-750,147.2665064979327,108.80626610625283,101.79608087222353,0.0,1.0
58,Ford F-750,61.40860126097286,79.78449059708598,78.90711442801762,0.0,1.0
```