datafest-2017

Datafest 2017 - Monitoring High Performance Platforms using Spark MLLib

By luciano.sabenca@movile.com and flavio.clesio@movile.com

September 26, 2017

ABOUT US

Flávio Clésio

- Core Machine Learning at Movile
- MSc. in Production Engineering (Machine Learning in Credit Derivatives/NPL)
- Specialist in Database Engineering and Business Intelligence
- Blogger at Mineração de Dados (Data Mining) - http://mineracaodedados.wordpress.com (http://mineracaodedados.wordpress.com)

Luciano Sabença

- Software Developer at Movile
- BSc. in Computer Science
- Specialist in Distributed Systems
- 6+ Years in High Performance and Distributed Systems

Messaging and Billing Services

- Corporative SMS

- Mobile Content through SMS:
  - News
  - Whether report
  - Entretainment
  - Education
  - Anti-Viruses
  - Care

Movile Subscription and Billing Platform

- A distributed platform
- User's subscription management
- MISSION CRITICAL platform: can not stop under any circumstance

Main Workflow

Common problems with distributed platforms

- All things can - AND WILL - broke
- Bad deployments
- Problems with some carriers (*eg.* deployment in their side, Ramsomware, ANATEL Issues, circuit breaker, bare wire, etc.)
- Preventive Maintenance

But... How about the scale?

- 230M transactions a day (~7B transactions per month or 83B year)
- 4 Main carriers in Brazil, and the another LatAm telecom Giants

```
/*
+----------+---------+-----------+----------------+-------------------+-----------+------------------+---------
|carrier_id|  datepart|hour_of_day|avg_response_time|       first_attempt|last_attempt|successful_charges|no_credit
+----------+---------+-----------+----------------+-------------------+-----------+------------------+---------
|         1|2017-07-31|          0|           924.0|2017-07-31 00:00:...|     2017.0|           37911.0|1790648.0
|         1|2017-07-31|          1|           835.0|2017-07-31 01:00:...|     2017.0|            1500.0|1661866.0
|         1|2017-07-31|          2|           862.0|2017-07-31 02:00:...|     2017.0|            1320.0|1835148.0
|         1|2017-07-31|          3|          1093.0|2017-07-31 03:00:...|     2017.0|           12449.0|1775057.0
|         1|2017-07-31|          4|          1032.0|2017-07-31 04:00:...|     2017.0|            1231.0|1841635.0
+----------+-------+---------+----------+----------------+-------------------+-----------+-----------------+-
*/
```

Took: 1 second 293 milliseconds, at 2017-9-25 18:1

Modeling

```
/*
+------------------+
|Target            |
+------------------+
|successful_charges|
+------------------+
|           37911.0|
|            1500.0|
+------------------+
*/

/*
+------------------+
|Features          |
+----------+----------+-----------+----------------+-------------------+------------+---------+--------+----------
|carrier_id|  datepart|hour_of_day|avg_response_time|       first_attempt|last_attempt|no_credit|  errors|total_atte
+----------+----------+-----------+----------------+-------------------+------------+---------+--------+----------
|         1|2017-07-31|          0|           924.0|2017-07-31 00:00:...|      2017.0|1790648.0|145533.0|     1974
|         1|2017-07-31|          1|           835.0|2017-07-31 01:00:...|      2017.0|1661866.0|136852.0|     1800
+----------+----------+-----------+----------------+-------------------+------------+---------+--------+----------
*/

// A plain vanilla case of Supervised Learning
```

Took: 1 second 56 milliseconds, at 2017-9-25 18:1

Apache Spark

- MLlib is Apache Spark's scalable machine learning library.
- MLlib contains many algorithms and utilities, including Classification, Regression, Clustering, Recommendation, Pipelines and so on...

Spark RDD vs Spark DataFrame

Why we changed from RDD to Dataframe? (Solid tips)

- RDD will be deprecated at Spark 4.x
- All data science work can be done more easily in a more user-friendly API than RDDs
- Same performance as RDD and warranty of same consistency
- All Feature Engineering (Feature Extraction and Feature Selection) can be done in ML Pipelines
- Best way to assembly (wrap-up) all code in Pipelines
- A good way to perform Grid Search in our models (hyperparameters)
- Much less painful debugging

Experimental Design

Algoritihms used

- Linear Model with Stochastic Gradient (SDG)
- Lasso with SGD Model (L1 Regularization)
- Ridge Regression with SGD Model (L2 Regularization)
- Decision Trees with Regression (Regression Trees)

Configurations

- Default (*on-the-shelf*) configurations
- Grid Search for every algorithm
- No Cross Validaton
- No Random Search
- 3 entire months of data (Reason: High volatile demand and several external bias (*e.g.* media investments, marketing campaigns, etc.))

Here is the our flow. We will take the following steps to train and evaluate our model:

```
import org.apache.spark.ml.Pipeline
import org.apache.spark.ml.PipelineModel
import org.apache.spark.ml.evaluation.RegressionEvaluator
import org.apache.spark.ml.feature.VectorIndexer
import org.apache.spark.ml.feature.VectorAssembler
import org.apache.spark.ml.regression.DecisionTreeRegressionModel
import org.apache.spark.ml.regression.DecisionTreeRegressor
import org.apache.spark.ml.tuning.{ParamGridBuilder, TrainValidationSplit}
import org.apache.spark.ml.tuning.TrainValidationSplitModel

import org.apache.spark.sql.types.{StructType, StructField, StringType, IntegerType, TimestampType, DoubleType, Date
import org.apache.spark.sql.expressions.Window
import org.apache.spark.sql.functions._
import java.util.Calendar
import java.util.Date
import org.apache.spark.mllib.tree.model.DecisionTreeModel

import resource._

val ROOT = "/Users/lucianosabenca/Movile/tools/spark-notebook-0.7.0-scala-2.11.8-spark-2.1.0-hadoop-2.7.1-with-hive/

val session = SparkSession
      .builder
      .appName("DecisionTreePipeline")
      .getOrCreate()

// You can monitoring all this stuff in: http://localhost:4040/jobs/

val customSchema = StructType(Array(
    StructField("carrier_id", IntegerType, true),
    StructField("datepart", DateType, true),
    StructField("hour_of_day", IntegerType, true),
    StructField("avg_response_time", DoubleType, true),
    StructField("first_attempt", TimestampType, true),
    StructField("last_attempt", DoubleType, true),
    StructField("successful_charges", DoubleType, true),
    StructField("no_credit", DoubleType, true),
    StructField("errors", DoubleType, true),
    StructField("total_attempts", DoubleType, true)
))


val df = session.read
```

```
        .format("org.apache.spark.csv")
        .option("header", "true") //reading the headers
        .option("delimiter", ",")
        .option("mode", "DROPMALFORMED")
        .schema(customSchema)
        .csv(ROOT + "/data-sample.csv");

df.show(5)
```

Took: 927 milliseconds, at 2017-9-25 20:56

Creating some usefull fields

We will need some functions to deal with the date and extract only the Week of Month from a date.

Let's create it and apply the function to create a new DataFrame with a new column, named *week_of_month*

```
// function to extract the week of month from a timestamp
val udfWoM = (dt :Date) => {
   val cal = Calendar.getInstance()
   cal.setTime(dt)

   cal.setMinimalDaysInFirstWeek(1)
   cal.get(Calendar.WEEK_OF_MONTH).toDouble
}

// user defined function
val weekOfMonthUDF = udf(udfWoM)
```

Took: 687 milliseconds, at 2017-9-25 20:56

```
// apply UDF to extract the week of month from date time field
val rawData = df.withColumn("week_of_month", weekOfMonthUDF($"datepart"))

rawData.show(20)
```

Took: 908 milliseconds, at 2017-9-25 20:56

```
rawData.groupBy("week_of_month").count().show()
```

Took: 1 second 415 milliseconds, at 2017-9-25 20:56

```
val avgDataFrame = rawData.groupBy("carrier_id", "hour_of_day","week_of_month").agg(avg("avg_response_time").as("avg
                                                                                     avg("successful_charges").as
                                                                                     avg("no_credit").as("no_cred
                                                                                     avg("errors").as("errors"),
                                                                                     avg("total_attempts").as("to



avgDataFrame.show(10)
```

Took: 1 second 215 milliseconds, at 2017-9-25 20:56

```
rawData.where("carrier_id = 4 AND hour_of_day = 19 and WEEk_of_month = 5").show()
```

Took: 1 second 216 milliseconds, at 2017-9-25 20:56

The data now looks like:

```
avgDataFrame.where("carrier_id = 4 AND hour_of_day = 19 and WEEk_of_month = 5").show()
```

Took: 1 second 348 milliseconds, at 2017-9-25 20:56

We still need to agreggate summing the values from the past hours until the current hour:

```
val carrierList = List(1)

// filtering data by carrier_id
val listDataFrame = carrierList.map{id =>
    avgDataFrame.filter("carrier_id = " + id)
}


//means from beginning until current position
val wSpec = Window.partitionBy("week_of_month").orderBy("hour_of_day").rowsBetween(Long.MinValue, 0)

val summarizedList = listDataFrame.map{dataframe =>
    val df = dataframe.withColumn("avg_response_time", avg($"avg_response_time").over(wSpec))
               .withColumn("successful_charges", sum($"successful_charges").over(wSpec))
               .withColumn("no_credit", sum($"no_credit").over(wSpec))
               .withColumn("errors", sum($"errors").over(wSpec))
               .withColumn("total_attempts", sum($"total_attempts").over(wSpec))
    df //return the dataset
}

summarizedList.foreach{_.show(3)}
```

```
Took: 2 seconds 660 milliseconds, at 2017-9-25 20:56
```

At this point, we have a list for each carrier with only its data. We will create a unique dataframe again with all the data and write it to a disk. This will be our checkpoint!

```
val summarizedDataFrame = summarizedList.reduce(_ union _)
```

```
Took: 649 milliseconds, at 2017-9-25 20:56
```

```
summarizedDataFrame.count
```

```
120                                   Took: 1 second 404 milliseconds, at 2017-9-25 20:56
```

```
//write data to disk in parquet format
summarizedDataFrame.write.format("parquet").mode("overwrite").save(ROOT + "/sbs-summarized-dataframe")
```

Took: 3 seconds 930 milliseconds, at 2017-9-25 20:57

To read again the data from the disk, you just need to run the following command:

```
val summarizedDataFrame = session.read.load(ROOT + "/sbs-summarized-dataframe").cache
```

Took: 710 milliseconds, at 2017-9-25 20:57

Setting our model's labels and features

We are now ready to create our model's features and labels! Just before that, let's do a quick remember on our naming:

- **Features**: Our features are the independent variables. In our linear model, it is multiplied by the trained coeficients.
- **Label**: Is our dependent variables. Our target is predict its value.

In this example, we want to predict our **success** number using as input the following features: *hour_of_day*, *week_of_month*, *avg_response_time*, *no_credit*, *errors*, *total_attempts*.

```
// features
val assemblerSucccesful = new VectorAssembler()
                      .setInputCols(Array("hour_of_day", "week_of_month", "avg_response_time","no_credit", "errors"
                      .setOutputCol("features_success")



// creating label in log format
val dataWithLabels = summarizedDataFrame.withColumn("successful_charges_log", log($"successful_charges"))


val dataWithLabelsFiltered = dataWithLabels.filter("successful_charges_log is not null")
```

Took: 631 milliseconds, at 2017-9-25 20:57

Setting our metrics and validators

We have now to define which metrics will be usefull now! We have defined, after some tests, 3 classical metrics:

- **RMSE (Root Mean Squared Error)**:

$$RMSE = \sqrt{\frac{1}{n}\sum_{i=1}^{n}(\hat{y_i} - y_i)^2}$$

- **MSE (Mean Squared Error)**:

$$MSE = \frac{1}{n}\sum_{i=1}^{n}(\hat{y_i} - y_i)^2$$

- **Rˆ2**: Coeficient of determination.

```
def buildEvaluator(label: String, predictionCol: String): RegressionEvaluator = {
  new RegressionEvaluator()
    .setLabelCol(label)
    .setPredictionCol(predictionCol)
}

def evaluateMSE(df: DataFrame, label: String, predictionCol: String): Double = {
  val evaluator = buildEvaluator(label, predictionCol)
  evaluator.setMetricName("mse")
  evaluator.evaluate(df)
}

def evaluateR2(df: DataFrame, label: String, predictionCol: String): Double = {
  val evaluator = buildEvaluator(label, predictionCol)
  evaluator.setMetricName("r2")
  evaluator.evaluate(df)
}

def evaluateRMSE(df: DataFrame, label: String, predictionCol: String): Double = {
  val evaluator = buildEvaluator(label, predictionCol)
  evaluator.setMetricName("rmse")
  evaluator.evaluate(df)
}
```

Took: 653 milliseconds, at 2017-9-25 20:57

```
// prints result of algorithm tested
def buildStatsMaps(carrier: Double, col: Column, label: String, df: DataFrame, predictionCol: String): Map[String, 
  val calculateAcc = (exp: Double, predicted: Double) => {
    val error = (exp - predicted) / exp
    if (error > 0.1) 0 else 1
  }

  val calcAccuracyUDF = udf(calculateAcc)

  val rmse = evaluateRMSE(df, label, "prediction_log")
  val mse = evaluateMSE(df, label, "prediction_log")
  val r2 = evaluateR2(df, label, "prediction_log")

  val data = df.withColumn("result_column", calcAccuracyUDF(col, df(predictionCol)))
  val total = data.count.toDouble
  // filter prediction that got right
  val correct = data.filter("result_column = 1").count.toDouble
  val accuracy = (correct / total) * 100

  Map("rmse" -> rmse, "mse" -> mse, "r2" -> r2, "accuracy" -> accuracy)
}
```

Took: 792 milliseconds, at 2017-9-25 20:57

Building, training the model and evaluating the model

We are finally ready to build and train our model!

First, we need to split our data in test and training. We will use 10% of data to validate and 90% to train. After that, we will create the model itself, the DecisionTreeRegressor, with our label and features. We will build also a Pipeline with two stages: build the assembler vector with our data and the decision tree.

There is also some parameters which are related with the model itself. They are named: *hyperparameters*. In the decision tree model we have basically two hyperparameters:

- **MaxDepth**: How deep must be the decision tree?
- **MaxBin**: How many bins must be the decision tree?

We will need also the evaluator to decide which of the trained models is the best model for the problem.

```
val trainedModels = carrierList.map { c =>
  val label  = "successful_charges_log"
  val features = "features_success"
  val predictionColumn = "successful_charges"
  val assembler = assemblerSucccesful
  val data = dataWithLabels.filter(s"carrier_id = $c")
  val Array(trainingData, testData) = data.randomSplit(Array(0.9, 0.1))

  // Train a DecisionTree model.
  val decisionTree = new DecisionTreeRegressor()
      .setLabelCol(label)
      .setFeaturesCol(features)
      .setPredictionCol("prediction_log")

  val pipeline = new Pipeline().setStages(Array(assemblerSucccesful, decisionTree))

  val paramGrid = new ParamGridBuilder()
     .addGrid(decisionTree.maxDepth, Array(6, 7, 8))
     .addGrid(decisionTree.maxBins, (15 to 32).toList)
     .build()

  // Select (prediction, true label) and compute test error.
  val evaluator = new RegressionEvaluator()
     .setLabelCol(label)
     .setPredictionCol("prediction_log")
     .setMetricName("rmse")


   val trainValidationSplit = new TrainValidationSplit()
     .setEstimator(pipeline)
     .setEvaluator(evaluator)
     .setEstimatorParamMaps(paramGrid)
     .setTrainRatio(0.8)

   //train a model
  val model = trainValidationSplit.fit(trainingData)

  //make predictions
  val predictions = model.transform(testData)
  val columnValue = s"prediction_$predictionColumn"

  val predictionResult = predictions.withColumn(columnValue, exp($"prediction_log"))
  val statsMap = buildStatsMaps(c, predictionResult(columnValue), label, predictionResult, columnValue)
  val bestModel = model.bestModel.asInstanceOf[PipelineModel].stages(1).asInstanceOf[DecisionTreeRegressionModel]
  println(s"maxDepth: ${bestModel.getMaxDepth}, maxBins: ${bestModel.getMaxBins}")
  (bestModel, c, statsMap)
}
```

Took: 25 seconds 849 milliseconds, at 2017-9-25 20:57

```
(trainedModels(0)._1).toDebugString
```

```
DecisionTreeRegressionModel (uid=dtr_65271060462a) of depth 8 with 187 nodes If (feature 3 <= 9414131.554945055) If (feature 0 <= 2.0) If (feature 0 <=
0.0) If (feature 1 <= 2.0) If (feature 1 <= 1.0) Predict: 10.476477005088364 Else (feature 1 > 1.0) Predict: 10.415594497916002 Else (feature 1 > 2.0) If
(feature 1 <= 4.0) If (feature 1 <= 3.0) Predict: 10.344347382700308 Else (feature 1 > 3.0) Predict: 10.344301418579894 Else (feature 1 > 4.0) Predict:
10.285411052061256 Else (feature 0 > 0.0) If (feature 1 <= 2.0) If (feature 0 <= 1.0) If (feature 1 <= 1.0) Predict: 10.626231290827675 Else (feature 1 > 1.0)
Predict: 10.624269007784182 Else (feature 0 > 1.0) Predict: 10.662297621895352 Else (feature 1 > 2.0) If (feature 1 <= 4.0) If (feature 2 <=
693.1666666666666) If (feature 1 <= 3.0) Predict: 10.560522611831995 Else (feature 1 > 3.0) Predict: 10.54153910621385 Else (feature 2 >
693.1666666666666) If (feature 1 <= 3.0) Predict: 10.507286211526067 Else (feature 1 > 3.0) Predict: 10.505859789258688 Else (feature 1 > 4.0) If (feature
0 <= 1.0) Predict: 10.400433005160421 Else (feature 0 > 1.0) Predict: 10.461686010002468 Else (feature 0 > 2.0) If (feature 1 <= 3.0) If (feature 4 <=
1760690.6428571432) If (feature 1 <= 2.0) Predict: 10.970208771890446 Else (feature 1 > 2.0) Predict: 10.971072234607037 Else (feature 4 >
1760690.6428571432) If (feature 0 <= 4.0) If (feature 3 <= 7927643.412087912) Predict: 11.000449854350665 Else (feature 3 > 7927643.412087912)
Predict: 10.995420174411938 Else (feature 0 > 4.0) Predict: 11.027757323635491 Else (feature 1 > 3.0) If (feature 5 <= 9793410.593406593) If (feature 4
<= 1760690.6428571432) If (feature 2 <= 1666.7678571428567) Predict: 10.754645300900677 Else (feature 2 > 1666.7678571428567) Predict:
10.769768376656003 Else (feature 4 > 1760690.6428571432) Predict: 10.803111592069065 Else (feature 5 > 9793410.593406593) If (feature 4 <=
2822585.3571428573) If (feature 0 <= 4.0) Predict: 10.854107934914774 Else (feature 0 > 4.0) Predict: 10.860256448399026 Else (feature 4 >
2822585.3571428573) Predict: 10.872630933800195 Else (feature 3 > 9414131.554945055) If (feature 3 <= 2.505480692857143E7) If (feature 3 <=
1.9274162285714287E7) If (feature 1 <= 4.0) If (feature 1 <= 2.0) If (feature 0 <= 6.0) If (feature 0 <= 5.0) Predict: 11.061918909891686 Else (feature 0 >
5.0) If (feature 1 <= 1.0) Predict: 11.124499364956598 Else (feature 1 > 1.0) Predict: 11.093329362111191 Else (feature 0 > 6.0) If (feature 3 <=
1.3166652714285716E7) Predict: 11.149362725809398 Else (feature 3 > 1.3166652714285716E7) If (feature 2 <= 881.2584915084914) Predict:
11.167795756911516 Else (feature 2 > 881.2584915084914) Predict: 11.186131397264688 Else (feature 1 > 2.0) If (feature 3 <= 1.3166652714285716E7)
If (feature 0 <= 6.0) If (feature 0 <= 5.0) Predict: 10.996158416319647 Else (feature 0 > 5.0) Predict: 11.058139176728607 Else (feature 0 > 6.0) Predict:
10.985463410499072 Else (feature 3 > 1.3166652714285716E7) If (feature 3 <= 1.7455170714285716E7) If (feature 2 <= 833.1263736263736) Predict:
11.099711454774287 Else (feature 2 > 833.1263736263736) Predict: 11.04666051000607 Else (feature 3 > 1.7455170714285716E7) If (feature 0 <= 10.0)
Predict: 11.135875995538836 Else (feature 0 > 10.0) Predict: 11.112478814739536 Else (feature 1 > 4.0) If (feature 0 <= 10.0) If (feature 0 <= 6.0) Predict:
10.91398135809223 Else (feature 0 > 6.0) If (feature 3 <= 1.570961492857143E7) If (feature 0 <= 8.0) Predict: 10.936608485311394 Else (feature 0 > 8.0)
Predict: 10.939907771344616 Else (feature 3 > 1.570961492857143E7) Predict: 10.956671519703832 Else (feature 0 > 10.0) If (feature 0 <= 11.0) Predict:
10.98384130689695 Else (feature 0 > 11.0) Predict: 11.027175655624733 Else (feature 3 > 1.9274162285714287E7) If (feature 5 <=
2.6903405928571425E7) If (feature 1 <= 4.0) If (feature 1 <= 2.0) If (feature 0 <= 11.0) If (feature 0 <= 10.0) Predict: 11.194263239896117 Else (feature 0 >
10.0) Predict: 11.217401459550587 Else (feature 0 > 11.0) Predict: 11.257005179117343 Else (feature 1 > 2.0) If (feature 5 <= 2.5315660285714284E7) If
(feature 0 <= 11.0) Predict: 11.157656674790887 Else (feature 0 > 11.0) Predict: 11.14300709019823 Else (feature 5 > 2.5315660285714284E7) If (feature
0 <= 12.0) Predict: 11.203015311901291 Else (feature 0 > 12.0) Predict: 11.198136439180159 Else (feature 1 > 4.0) If (feature 0 <= 13.0) Predict:
11.067644552391348 Else (feature 0 > 13.0) Predict: 11.117713329326238 Else (feature 5 > 2.6903405928571425E7) If (feature 2 <=
1483.0340136054417) If (feature 5 <= 2.8279224785714284E7) If (feature 1 <= 2.0) If (feature 0 <= 12.0) Predict: 11.268962519627692 Else (feature 0 >
12.0) Predict: 11.295430354214638 Else (feature 1 > 2.0) If (feature 3 <= 2.208086892857143E7) Predict: 11.25206799329152 Else (feature 3 >
2.208086892857143E7) Predict: 11.238239528097465 Else (feature 5 > 2.8279224785714284E7) If (feature 0 <= 16.0) If (feature 0 <= 13.0) Predict:
```

```
11.310953429304936 Else (feature 0 > 13.0) Predict: 11.290260945979771 Else (feature 0 > 16.0) Predict: 11.334626378612285 Else (feature 2 > 1483.0340136054417) If (feature 0 <= 15.0) Predict: 11.159890825849988 Else (feature 0 > 15.0) Predict: 11.205621663144964 Else (feature 3 > 2.505480692857143E7) If (feature 5 <= 4.0744571452380955E7) If (feature 3 <= 2.7930319285714284E7) If (feature 1 <= 4.0) If (feature 3 <= 2.637873464285714E7) If (feature 2 <= 787.3690476190477) Predict: 11.328957124400455 Else (feature 2 > 787.3690476190477) If (feature 4 <= 4501917.241758242) Predict: 11.37805043851977 Else (feature 4 > 4501917.241758242) Predict: 11.36239175736965 Else (feature 3 > 2.637873464285714E7) If (feature 1 <= 2.0) If (feature 0 <= 15.0) Predict: 11.410777068878906 Else (feature 0 > 15.0) Predict: 11.424753794721026 Else (feature 1 > 2.0) If (feature 0 <= 16.0) Predict: 11.369071590620338 Else (feature 0 > 16.0) Predict: 11.391642024324076 Else (feature 1 > 4.0) If (feature 0 <= 17.0) Predict: 11.244237102626611 Else (feature 0 > 17.0) Predict: 11.268047421510014 Else (feature 3 > 2.7930319285714284E7) If (feature 2 <= 1163.399350649351) If (feature 3 <= 3.101713607142857E7) If (feature 2 <= 789.0095238095238) If (feature 0 <= 17.0) Predict: 11.406430973271199 Else (feature 0 > 17.0) Predict: 11.434335154438102 Else (feature 2 > 789.0095238095238) If (feature 3 <= 2.951352957142857E7) Predict: 11.461787577556972 Else (feature 3 > 2.951352957142857E7) Predict: 11.487704643111902 Else (feature 3 > 3.101713607142857E7) If (feature 1 <= 2.0) If (feature 3 <= 3.257691307142857E7) Predict: 11.527483423339042 Else (feature 3 > 3.257691307142857E7) Predict: 11.564640658207047 Else (feature 1 > 2.0) If (feature 0 <= 19.0) Predict: 11.46646333881617 Else (feature 0 > 19.0) Predict: 11.497791023588952 Else (feature 2 > 1163.399350649351) If (feature 4 <= 5939958.714285714) If (feature 0 <= 20.0) Predict: 11.335272422257631 Else (feature 0 > 20.0) Predict: 11.36614230044955 Else (feature 4 > 5939958.714285714) If (feature 4 <= 6245839.333333333) If (feature 0 <= 22.0) Predict: 11.393724639009708 Else (feature 0 > 22.0) Predict: 11.410104051105758 Else (feature 4 > 6245839.333333333) Predict: 11.435041912737269 Else (feature 5 > 4.0744571452380955E7) If (feature 1 <= 2.0) If (feature 0 <= 21.0) If (feature 4 <= 7308891.714285714) If (feature 0 <= 20.0) Predict: 11.595830738257241 Else (feature 0 > 20.0) Predict: 11.610734567815989 Else (feature 4 > 7308891.714285714) Predict: 11.631752893330244 Else (feature 0 > 21.0) If (feature 0 <= 22.0) If (feature 1 <= 1.0) Predict: 11.663026049460731 Else (feature 1 > 1.0) Predict: 11.640072869390346 Else (feature 0 > 22.0) Predict: 11.68803403356037 Else (feature 1 > 2.0) If (feature 3 <= 3.571473957142857E7) If (feature 0 <= 21.0) Predict: 11.535627246997644 Else (feature 0 > 21.0) Predict: 11.517709716451803 Else (feature 3 > 3.571473957142857E7) If (feature 0 <= 22.0) Predict: 11.564056424753659 Else (feature 0 > 22.0) Predict: 11.583032382344221
```

Took: 996 milliseconds, at 2017-9-25 20:57

Evaluating our model

We can now evaluate our model, simply print the stats already computed:

```
trainedModels.foreach{ case (m, c, statsMap) =>
                       m.write.overwrite.save(ROOT + "/trained-models-dataframe/success-c" + c)
                       println(statsMap)}
```

Took: 1 second 340 milliseconds, at 2017-9-25 20:57

Nostradamus API

A Generic API which loads all the serialized models and make them accessible using an HTTP API.

Watcher AI

We've build also Watcher-AI, a scheduler which consults our relational database and uses nostradamus to consult the models. Watcher-AI is also capable of dispare a lot of notifications to several systems.

```
                                                                               +------------------------------------
                                                                               |          OMNI Notification Strateg
                                                                               +------------------------------------
                            +---------------------------------------+          |
                            |            Nostradamus API            |          |
                            +---------------------------------------+          |        +------------------+
                            |                                       |  ---------------> | SMS Notification |
+-----------------------+   |  (Load ML Models)                     |          |        +------------------+
| Pre Treined ML Models | <---------------------                    |          |
+-----------------------+   |                    |                  |          |        +---------------------------
                            |         +------------------------+    |  ---------------> | Pushbullet (3Rd Party Mes
                            |         |                        |    |          |        +---------------------------
                            |         | Watcher-AI (Scheduler) |    |          |
                            |         |                        |    |          |        +---------------------------
                            |         +------------------------+    |  ---------------> | Zabbix (Alerts and Monito
+-----------------------+   |                     |                 |          |        +---------------------------
| Amazon Redshift       | <----------------------                   |          |
+-----------------------+   |  (SQL Query in DB)                    |          |        +-----------------------+
                            |                                       |  ---------------> | JIRA (Issue Tracking) |
                            +---------------------------------------+          |        +-----------------------+
                                                                               |
                                                                               +------------------------------------
```

Afterwards (Empirical observations about this kind of problem)

- Regularization doesnt's fits so well with our low dimensional data (*i.e.* All columns have some importance in the model)
- (Empirically for us) Linear Methods are good for extrapolation, but Decision Trees are more suitable for interpolation problems (More deterministic)

- Time Series with thresholds didn't work in the past 'cause we have several exogenous factors that makes the regular algorithms go wild (*e.g.* investments in media, freezing of carriers, maintenance in several satellite platforms, *etc*)
- We avoid (totally removed) fixed thresholds based in standard deviations 'cause when the volume naturally goes up, the lower limit doesn't make sense anymore, and we need some dynamic ajustments
- In our case, regular regression modeling it's good for regular prediction in well behaviored problems. In our problem we deal with several factors like seasonality, patterns inside the hour, number of the week, day of the week, special hours for billing in some carriers and make the whole trainning thinking in the 4 main carriers

Prelimirary Results

- First barrier of defense
- Early warning in several teams (*e.g.* infrastructure, platforms, revenue assurance, and so on)
- Catch any discrepancy in hourly fashion

Key Results

Challenges and Next Steps

- Automatic refeed and training using collected data. Analyse more data to predict possible errors with carrier
- Notify more people and specific teams (more complex problems)
- Expand the same idea for another platforms
- Link some specific behaviours and solve the problem automatically

Build: | **buildTime**-*Mon Jan 23 20:05:34 UTC 2017* | **formattedShaVersion**-*0.7.0-c955e71d0204599035f603109527e679aa3bd570* | **sbtVersion**-*0.13.8* | **scalaVersion**-*2.11.8* | **sparkNotebookVersion**-*0.7.0* | **hadoopVersion**-*2.7.1* | **jets3tVersion**-*0.7.1* | **jlineDef**-*(jline,2.12)* | **sparkVersion**-*2.1.0* | **withHive**-*true* |.