SPARK ASSIGNMENT

COMPUTACIÓN DE ALTAS PRESTACIONES PARA BIG DATA EN LAS EMPRESAS

JUAN CARLOS PEREIRA KOHATSU

ANTONIO MARTÍNEZ PAYÁ

To do the study we chose hdfs:///GreenCab/2017/ database. In order to execute Spark the code typed was:

spark-shell --packages com.databricks:spark-csv_2.10:1.5.0 --conf spark.ui.port=50011 --num-executors 15 --executor-cores 2

import org.apache.spark.sql.SQLContext

import org.apache.spark.sql.functions._

val sqlContext = new SQLContext(sc)

val df = sqlContext.read.format("com.databricks.spark.csv").option("header", "true").option("inferSchema", "true").load("hdfs:///GreenCab/2017/")

df.registerTempTable("Taxis")

First of all, we start by querying some basic statistical parameters, with a view to know better the database. **It seems natural to know how many rows we have in the table, so before query anything:**

val ntotal = df.count()

The value of ntotal is 6369863 (Long).

1- Average and standard deviation of the variable "Trip distance" (in miles)

We did this query in two manners, the first one without using the SQL function AVG, and the second one using it. The reason of doing this query in these two manners was to practice the mapping and reducing. When we had an RDD of arrays of one element in "distances", we applied a map with the functions toString() and toFloat() to have floats and later being able to apply the function reduce((x,y) => x+y) and divide by the total to obtain the average.

val distances = sqlContext.sql("SELECT trip_distance FROM Taxis")

var total = distances.map(_(0).toString.toFloat).reduce((x,y) => x+y)/ntotal

Being the result the float 2.6248 or directly using the SQL functions:

df.select(avg($"trip_distance")).show()

df.select(stddev_samp($"trip_distance")).show()

Being the result of the average 2.6249 and the result of the standard deviation 2.7567.

2- Average of the variable "Total amount"

val outcomes = sqlContext.sql("SELECT total_amount FROM Taxis")

outcomes.map(_(0).toString.toDouble).reduce(_+_)/ntotal

Being the result the double 14.02837.

3- Average, standard deviation and maximum value of the variable "Tip amount"

df.select(avg($"tip_amount")).show()

df.select(stddev_samp($"tip_amount")).show()

df.select(max($"tip_amount")).show()

Obtaining the values: 1.1621 for the average, 2.2439 for the standard deviation and 408.0 for the maximum. At this moment, we were quite surprised with the maximum value. We also observed some strange 0 values for the variable Trip Distance, so the decision at that moment was to make the following comparation of probabilities (depending on the variable Payment Type, to investigate if there were some irregularities):

4- What are the different probabilities being 0 the variable Trip Distance depending on the payment type.

Conditioned probabilities:

val dzeros = sqlContext.sql("SELECT payment_type FROM Taxis WHERE trip_distance = 0 ")

val totalzeros = sqlContext.sql("SELECT payment_type FROM Taxis WHERE trip_distance = 0 ").count()

dzeros.map(x => (x(0).toString,1)).reduceByKey(_+_).map(x=>(x._1,x._2/totalzeros.toDouble)).collect()

Non-conditioned probabilities:

val all = sqlContext.sql("SELECT payment_type FROM Taxis)

val totalall = sqlContext.sql("SELECT payment_type FROM Taxis ").count()

all.map(x => (x(0).toString,1)).reduceByKey(_+_).map(x=>(x._1,x._2/totalall.toDouble)).collect()

Thus, the obtained probabilities were the ones in the table below, observing that there is no dependency, because the probabilities are quite similar, excepting the probability of No charging, which seems obvious, due to the fact that there are people who don't pay because they have not travelled.

PAYMENT TYPE	PROBABILITY CONDITIONED	PROBABILITY NO CONDITIONED
1	0.4719092165408738	0.4953593193448588
2	0.45872754331880516	0.4972485279510721
3	0.05283299670458169	0.004712032268197919
4	0.01595886042308919	0.0026328666723287454
5	5.713830126501541E-4	4.725376354248121E-5

5- Are there any differences in the variable Total Amount depending on the variable Store and FWD flag?

Working the same topic, wanting to know if there are some irregularities, we propose this question. We could think that whether the trip record was held in vehicle or not could give some information.

val register = sqlContext.sql("SELECT store_and_fwd_flag FROM Taxis")

register.map(x => (x(0),1)).reduceByKey(_+_).collect()

Obtaining the values 6359715 for the non-recorded trips and 10148 for the recorded trips. Now, conditioning for the values 'Y' and 'N' and calculating the average of the total amount, the standard deviation of the total amount and the average of the tip amount, we want to study if there are irregularities. The code typed was:

val totalamount_N = sqlContext.sql("SELECT store_and_fwd_flag WHERE store_and_fwd_flag = 'N' FROM Taxis")

val totalamount_Y = sqlContext.sql("SELECT store_and_fwd_flag WHERE store_and_fwd_flag = 'Y' FROM Taxis")

totalamount_N.map(x => x(0).toString.toDouble).reduce(_+_)/6359715

totalamount_Y.map(x => x(0).toString.toDouble).reduce(_+_)/10148

val xx = sqlContext.sql("SELECT stddev_samp(total_amount) FROM Taxis WHERE store_and_fwd_flag='N'").collect()

val xx = sqlContext.sql("SELECT stddev_samp(total_amount) FROM Taxis WHERE store_and_fwd_flag='Y'").collect()

val totaltip_Y = sqlContext.sql("SELECT tip_amount FROM Taxis WHERE store_and_fwd_flag='Y'")

totaltip_Y.map(x => x(0).toString.toDouble).reduce(_+_)/10148

val totaltip_N = sqlContext.sql("SELECT tip_amount FROM Taxis WHERE store_and_fwd_flag='N'")

totaltip_N.map(x => x(0).toString.toDouble).reduce(_+_)/6359715

	store_and_fwd_flag = 'N'	**store_and_fwd_flag = 'Y'**
Total amount average	14.026779289802123	15.029848245959494
Total amount standard deviation	11.439560510740861	12.424192290518317
Tip amount average	1.1090638549467877	1.1621944049377226

The strategy that we chose gave us no evidence that the caps were cheating, due to the fact that all these values obtained are quite similar.

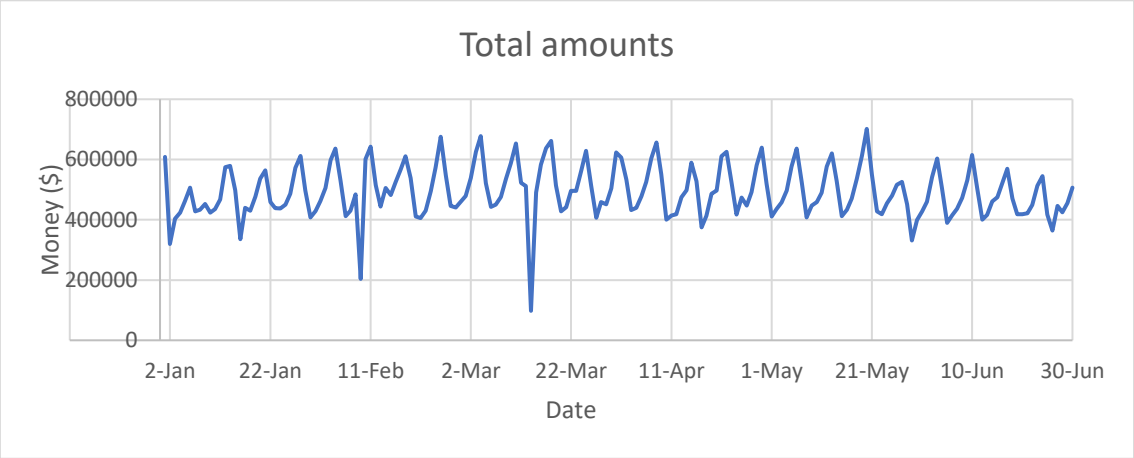
Changing the topic, we found also interesting to know how is the evolution of the amount of money during the year.

6- Describe the evolution of the total amount along the time.

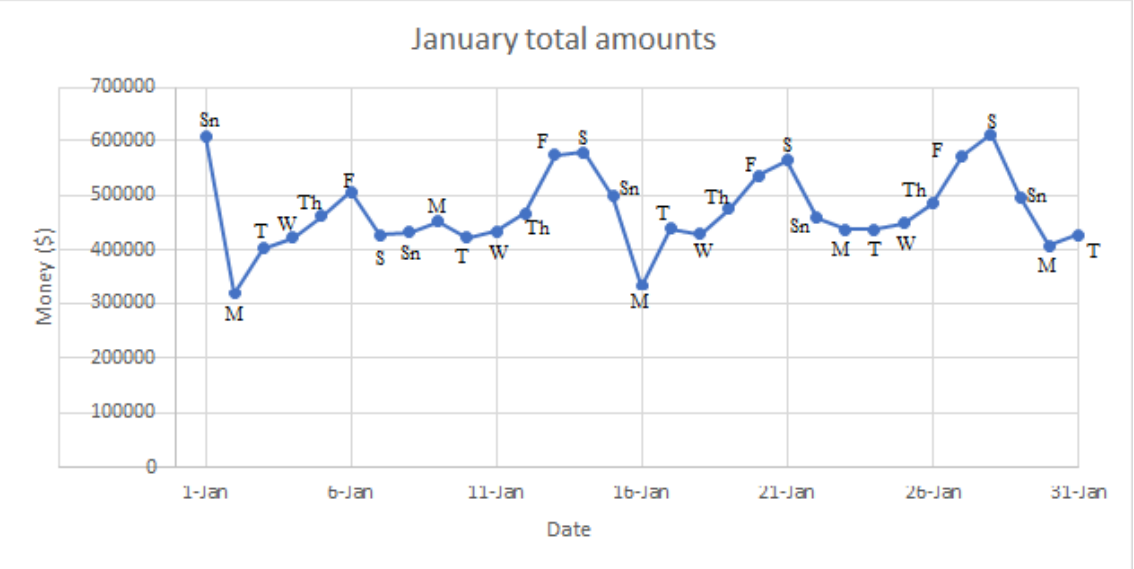
val daysamount = sqlContext.sql("SELECT month(lpep_pickup_datetime), dayofmonth(lpep_pickup_datetime) ,total_amount FROM Taxis")

daysamount.map(x => ((x(0).toString.toInt,x(1).toString.toInt),x(2).toString.toDouble)).reduceByKey(_ + _).sortBy(_._1).collect().foreach(x => println(x._1._1.toString+"/"+x._1._2.toString+";"+x._2.toString))

Using the code above we were able to print in the screen the total amount value per day. This is interesting because we can use Excel to print the graph.



That graph shows a kind of periodicity, so studying for example the month of January, there is a pattern in which we see a lot of maximums in the weekends, probably due to the fact that there are a lot of people having fun so they can't drive home, as it can be seen in the graph bellow. Another detail is that the days of minimum total amount were the ones in which there was a strong blizzard with a lot of snow (9 February and 14 of March), which makes sense.

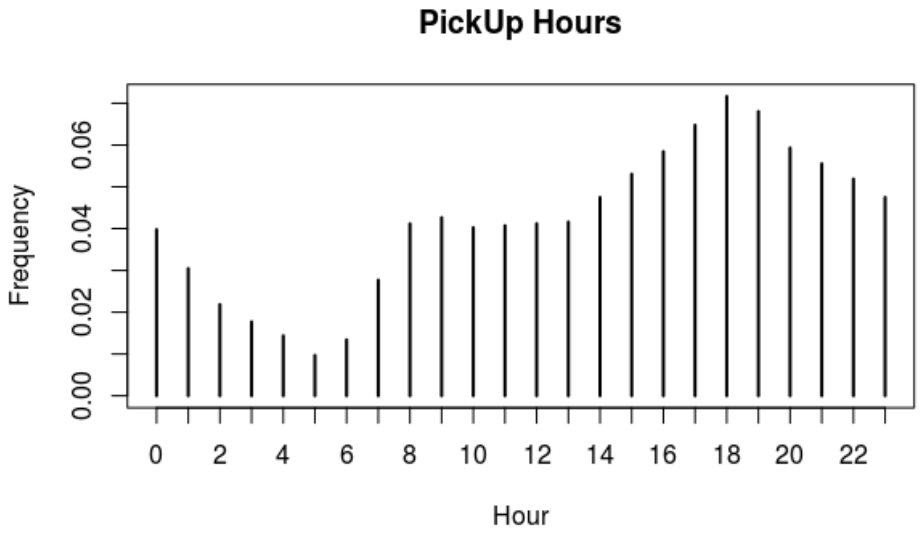


7- Try to fit the variables Total Amount, Tip Amount and Pick Up Hour in probability distributions.

It is always useful to fit variables in a probability distribution (specially if we want to predict).

Firstly, we programmed some code in Scala to obtain de different pick up hours for every travel in the green taxis data base:

val pick = sqlContext.sql("SELECT hour(lpep_pickup_datetime) FROM Taxis")
pick.map(x =>(x(0).toString.toInt,1)).reduceByKey(_+_).sortBy(_._1).collect().foreach(println)
The commands showed before allowed us to obtain the frequencies of picking up actions per hour. With this kind of data, it is possible to create some plots in R:



The graph of above shows that from 6pm to 5am the times passengers are picked up decrease, it makes sense because it is a night and people usually go home at that hours or they are sleeping. Then from 6am to 18am the times people are picked up per hour increases. From 8am to 13pm the function is stable, maybe it is because people is working or they are busy so they don't take taxis. After this analysis we tried to fit this variable to a distribution without success.

With total amount we had some problems since there are a lot of outliers, as it can be seen with the result of the code, that's why we selected only the ones between 0 and 100 (there were only a few outliers, so the result wouldn't change so much):

val tamount = sqlContext.sql("SELECT max(total_amount), min(total_amount) FROM Taxis")

tamount.collect()

Array([8999.91,-480.0])

val tamount = sqlContext.sql("SELECT total_amount FROM Taxis WHERE total_amount < 100 AND total_amount > 0").count()

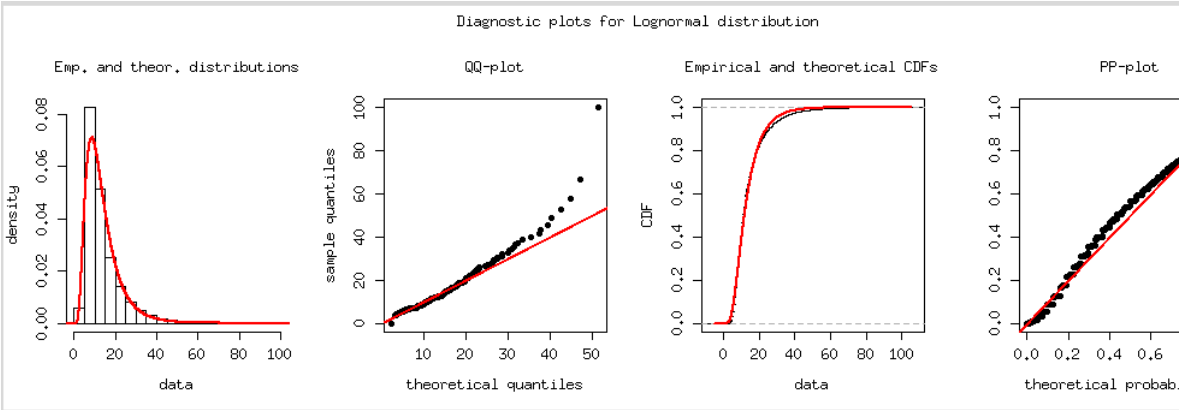
Long = 6336322

We tried to discretize the variable into intervals:

tamount.map(x => (x(0).toString).toDouble).map(x=>
*(2.5*math.ceil(x/2.5).toInt,1)).reduceByKey(_+_).sortBy(_._1).collect().foreach(x =>*
println(x._1.toString + ";" + x._2.toString))

However, R wasn't able to fit with success the variable. We think that we should let R to make these intervals to represent the variables, that's why we thought it was useful to print all the values of total amount and then process them with R. It was not possible to print the whole

dataset in the screen so we used the class PrintWriter from java.io._ . We created a file in Urraca and then transferred it to our computers using SCP protocol that uses same port as SSH.
With the whole data in R, using the library rriskDistributions, we concluded that the variable fits quite well in an lognormal distribution.

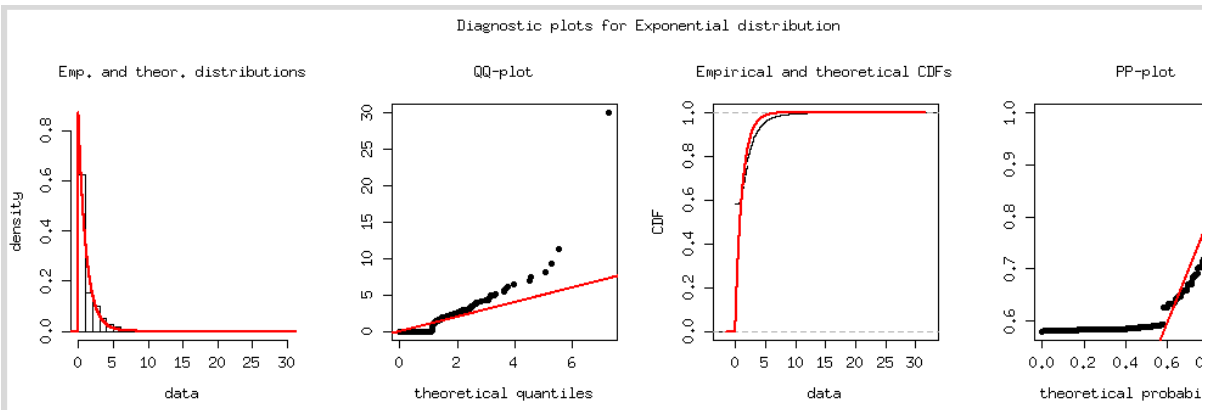


Distribution

- Normal
- Cauchy
- Logistic
- Exponential
- Chi-square
- Uniform
- Gamma
- Lognormal
- Weibull
- F
- Student's t
- Gompertz

Family	logL	AIC	BIC	Chisq(value)	Chisq(p)	AD(value)	H(AD)	KS(value)	H(KS)
Normal	-23374269.54	46748543.08	46748570.4	13400392986	0	Inf	rejected	0.16	rejected
Cauchy	-22193671.79	44387347.57	44387374.9	27983722	0	329914.45	rejected	0.16	rejected
Logistic	-22533946.32	45067896.64	45067923.96	55834340	0	224970.18	rejected	0.14	rejected
Exponential	-23061704.27	46123410.54	46123424.2	42857191	0	644379.37	rejected	0.28	rejected
Chi-square	-22715270.54	45430543.08	45430556.74	21756624134	0	369487.64	NULL	0.13	rejected
Uniform	NULL	NULL	NULL	Inf	0	Inf	NULL	0.21	rejected
Gamma	-21387329.9	42774663.79	42774691.11	28744439	0	121500.32	rejected	0.10	rejected
Lognormal	-20905633.66	41811271.33	41811298.65	23217657	0	44000.88	rejected	0.07	rejected
Weibull	-21892139.72	43784283.43	43784310.76	37317890	0	195642.83	rejected	0.12	rejected
F	-29528808.84	59057621.68	59057649	134850771	0	2841898.91	NULL	0.59	rejected
Student	-34630549.49	69261100.99	69261114.65	312120890	0	5910734.37	NULL	0.78	rejected
Gompertz	-22745814.45	45491632.89	45491660.22	41642625	0	425673.91	NULL	0.23	rejected

We did the same for the variable tip amount concluding that it fits quite well in an exponential distribution:



Distribution

- Normal
- Logistic
- Exponential
- Uniform
- Student's t
- Gompertz

Family	logL	AIC	BIC	Chisq(value)	Chisq(p)	AD(value)	H(AD)	KS(value)	H(KS)
Normal	-13086216.2	26172436.4	26172463.73	1.499869e+12	0	Inf	rejected	0.31	rejected
Logistic	-12205064.77	24410133.54	24410160.88	5.049661e+07	0	627183.9	rejected	0.30	rejected
Exponential	-7244993.07	14489988.14	14490001.81	Inf	0	Inf	rejected	0.58	rejected
Uniform	NULL	NULL	NULL	Inf	0	Inf	NULL	0.32	rejected
Student	-12179972.55	24359947.09	24359960.76	1.059079e+07	0	1493544.1	NULL	0.50	rejected
Gompertz	-7252561.02	14505126.05	14505153.38	Inf	0	Inf	NULL	0.58	rejected

8- Construction of a graph based on taxis traffic fluency in NY.

In this section, we have used Spark to obtain the necessary resources to build a graph. First of all, we obtained all the undirected edges using the following queries:

val locations = sqlContext.sql("SELECT PULocationID, DOLocationID FROM Taxis"

locations.map(x => Array(x(0).toString.toInt,x(1).toString.toInt).sortBy(+_)).map(x => ((x(0),x(1)),1)).reduceByKey(_+_).sortBy(_._1).collect()

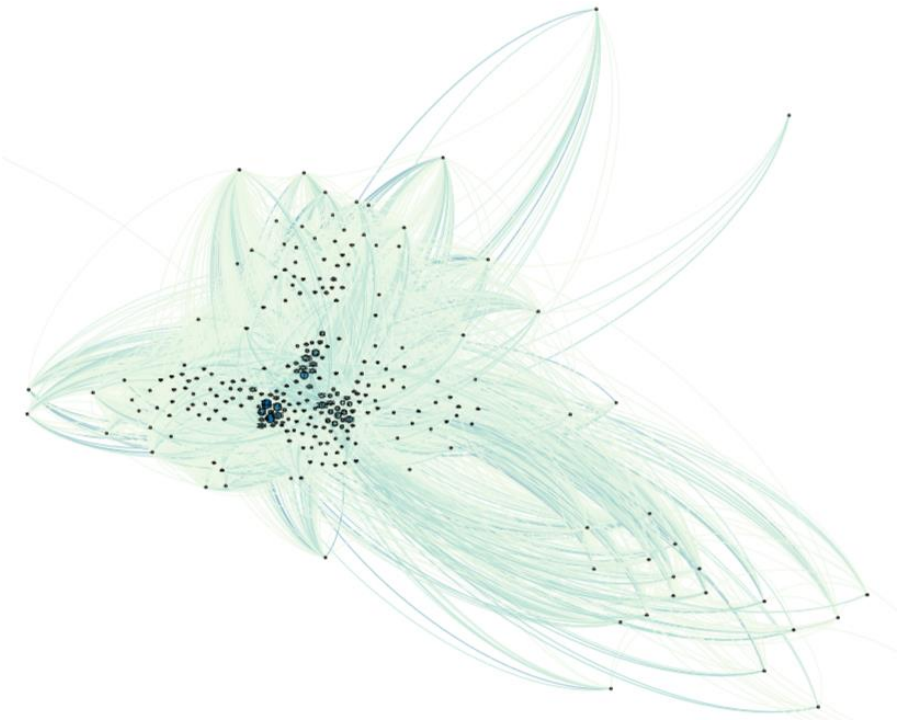
It was necessary to obtained the locations ids in an array to sort this ids because we are considering an undirected graph. For example: (1,2) and (2,1) is the same edge so after sorting we would obtained this result: (1,2) and (1,2). Then we have counted the number of edges to calculate the weight of each edge in the graph. There are 19905 edges in our graph, using count() function it is really easy to obtain this value.

Second, we obtained all the nodes of the graph using these queries:

val locaID = sqlContext.sql("SELECT DISTINCT PULocationID, DOLocationID FROM Taxis")

locaID.flatMap(x => Array(x(0).toString.toInt,x(1).toString.toInt)).distinct().sortBy(+_).count()

We have found 261 different nodes in our graph. After this step, we printed in a file the output of our queries and transferred this files to our computers using SCP protocol. We used this data to create a graph using Gephi program and obtained this picture:



Each node represents a place which has an ID, the more blue and big is the node, the more weighted grade the node has. The edges al represented with lines, the more intense color of the edge is, the more weight it has. It's possible to analyze the more important node (bigger ones). In order to do this task, we have obtained a CSV with the name places that match with that IDs (https://s3.amazonaws.com/nyc-tlc/misc/taxi+_zone_lookup.csv)