Análisis de los Precios del Taxi en la Ciudad de Nueva York con Spark el día de Año Nuevo

Andrés Matesanz

Abstract - En este informe se presenta un análisis de la variación de precios en los taxis de la ciudad de Nueva York el primer día del año 2019 usando tecnologías Spark.

1. Introducción

Actualmente en el mundo se produce más información que en ninguna otra etapa de la historia, y cada día más. Esto es debido a la dispersión de sistemas inteligentes cada vez más pequeños y eficientes, comenzando por nuestros teléfonos móviles. Muchos de estos datos son producidos por sensores de todo tipo: temperatura, velocidad, tiempo, posicionamiento GPS… A raíz de ello han surgido una serie de portales que aglutinan y comparten gran parte de esta información, principalmente desde organismo públicos, a mayores, en los últimos años se han ido desarrollando una serie de herramientas que permiten procesar estas **grandes cantidades de datos** de forma escalable y paralela sin que el desarrollador tenga que invertir gran parte de su tiempo en este asunto, en concreto, una de las más extendidas es **Spark**.

Spark es un motor diseñado para procesar, tanto en streaming, como no, grandes volúmenes de información en varios lenguajes de programación (Java, Python, Scala y R). De esta manera un usuario puede interpretar, a través de tablas y gráficos, esos datos en tiempo real y en base a ello tomar decisiones.

En concreto, en este informe vamos a entender cómo podemos procesar y visualizar la información recogida por los **taxis de Nueva York** un día que es típicamente muy activo en todo el mundo y que coincide además con las fechas en las que nos encontramos: **el primer día del año,** en concreto, el 1 de Enero del 2019**.** La información proporcionada se encuentra en el portal de datos abiertos de la ciudad de Nueva York y con ella vamos a tratar de sacar conclusiones de cómo se comporta la ciudad en cuanto a su movilidad se refiere.

2. Ejecución

Como mencionamos más arriba la información ha sido recogida del portal de datos abiertos de la ciudad de Nueva York a través de la Comisión de Taxis y Limusinas de esa misma ciudad. Para obtener los datos debemos dirigirnos a la página

mencionada y una vez allí contamos con varias opciones: por un lado podemos descargar los datos directamente en formato CSV, por otro lado se nos ofrece utilizar la API a través de la herramienta Socrata de la librería Sodapy.

El archivo original de 2019 es bastante pesado, cuenta con 4.3 Gb de información, mientras que la API es mucho más rápida, sin embargo, sólo podremos acceder a una pequeña cantidad de esos datos de forma gratuita. Ya que el objetivo es trabajar con **grandes volúmenes de información** para este trabajo se ha utilizado el archivo original. Una vez descargado vamos a inicializar una sesión de Spark e importar el archivo CSV:

```
# initialize SparkSession
sparkSession = SparkSession\
    .builder\
    .getOrCreate()

# Import csv data
dataset = sparkSession\
    .read\
    .options(header='true', inferschema='true')\
    .format("csv")\
    .load("data/2019_Yellow_Taxi_Trip_Data.csv")
```

En este caso trabajamos con nuestra máquina local y no especificamos el número de núcleos que entran dentro del proceso, por lo que **Spark utilizará todos los recursos disponibles** de aquí en adelante. Importamos los datos con los nombres de las columnas e inferimos el esquema de datos que le corresponde a cada una de ellas (cuando se trabaja en streaming es recomendable prefijar el esquema ya que hacerlo consume un tiempo muy preciado cuando se quiere trabajar en tiempo real.

El siguiente paso es visualizar el **esquema del dataset**. El esquema define la estructuración de los datos con los que vamos a trabajar, tanto el nombre del atributo como el tipo de dato que contiene. Para ello vamos a ejecutar las siguientes líneas de código:

```
# Visualize Dataset Schema
dataset.printSchema()
```

Este comando nos devuelve el esquema de la base de datos de la siguiente manera

```
root
 |-- VendorID: integer (nullable = true)
 |-- tpep_pickup_datetime: string (nullable = true)
 |-- tpep_dropoff_datetime: string (nullable = true)
 |-- passenger_count: integer (nullable = true)
 |-- trip_distance: double (nullable = true)
 |-- RatecodeID: integer (nullable = true)
 |-- store_and_fwd_flag: string (nullable = true)
 |-- PULocationID: integer (nullable = true)
 |-- DOLocationID: integer (nullable = true)
 |-- payment_type: integer (nullable = true)
 |-- fare_amount: double (nullable = true)
 |-- extra: double (nullable = true)
 |-- mta_tax: double (nullable = true)
 |-- tip_amount: double (nullable = true)
 |-- tolls_amount: double (nullable = true)
 |-- improvement_surcharge: double (nullable = true)
 |-- total_amount: double (nullable = true)
 |-- congestion_surcharge: double (nullable = true)
```

Podemos observar cada una de las columnas que conforman el dataset: tenemos la distancia recorrida, la hora de recogida y de finalización, el número de pasajeros, la cantidad de la propina etc… Sin embargo, para nuestro propósito no nos es necesaria toda esta información, por lo que realizamos nuestro primer **preprocesamiento de los datos** y nos quedamos tan sólo con unos pocos de estos atributos.

```
# Filtered Dataset
filt = dataset.select("tpep_pickup_datetime",
"tpep_dropoff_datetime", "trip_distance", "fare_amount")
filt = filt.filter(dataset.trip_distance != 0.00)
filt.show(10)
```

Aquí suceden varias cosas: en primer lugar filtramos tan sólo el momento de la recogida, el momento de la bajada, la distancia del viaje y el precio. A continuación para evitar ruido, **eliminamos aquellos viajes cuya distancia sea 0**, hay carreras que por el motivo que sea no llegaron a iniciar la marcha. Por último mostramos los diez primeros datos resultantes tal y como son en el dataset original.

```
+--------------------+---------------------+-------------+-----------+
|tpep_pickup_datetime|tpep_dropoff_datetime|trip_distance|fare_amount|
+--------------------+---------------------+-------------+-----------+
|01/01/2019 12:46:...| 01/01/2019 12:53:...|          1.5|        7.0|
|01/01/2019 12:59:...| 01/01/2019 01:18:...|          2.6|       14.0|
|01/01/2019 12:21:...| 01/01/2019 12:28:...|          1.3|        6.5|
```

```
|01/01/2019 12:32:...| 01/01/2019 12:45:...|          3.7|       13.5|
|01/01/2019 12:57:...| 01/01/2019 01:09:...|          2.1|       10.0|
|01/01/2019 12:24:...| 01/01/2019 12:47:...|          2.8|       15.0|
|01/01/2019 12:21:...| 01/01/2019 12:28:...|          0.7|        5.5|
|01/01/2019 12:45:...| 01/01/2019 01:31:...|          8.7|       34.5|
|01/01/2019 12:43:...| 01/01/2019 01:07:...|          6.3|       21.5|
|01/01/2019 12:58:...| 01/01/2019 01:15:...|          2.7|       13.0|
+--------------------+---------------------+-------------+-----------+
```

La distancia se encuentra en **millas**, el precio en **dólares** americanos y la fecha de recogida se encuentra en formato *MM/DD/YYYY HH:MM:SS*. Más adelante veremos las complicaciones de trabajar con datos temporales.

De momento aún tenemos todas y cada una de las instancias recogidas en el dataset, por ello, vamos a ver de cuantas se tratan antes y después de realizar el filtrado por día.

```
# Counting Dataset length and width
print(filt.count(), len(filt.columns))
```

De esta forma obtenemos que inicialmente contamos con 44131885 viajes recogidos a lo largo del año 2019, sin embargo, como ya hemos dicho, a nosotros sólo nos interesan los datos del día 1 de enero, por ello vamos a realizar nuestro segundo filtro.

```
# Convert datetime to 24h
data_frame = filt.withColumn("tpep_pickup_datetime",
F.from_unixtime(F.unix_timestamp(F.col(("tpep_pickup_datetim
e")), "MM/dd/yyyy hh:mm:ss aa"), "yyyy-MM-dd HH:mm"))

data_frame = data_frame.withColumn("tpep_dropoff_datetime",
F.from_unixtime(F.unix_timestamp(F.col(("tpep_dropoff_dateti
me")), "MM/dd/yyyy hh:mm:ss aa"), "yyyy-MM-dd HH:mm"))
```

Para ello en primer lugar **convertimos las fechas a un formato más conveniente**, concretamente el que establece la norma ISO, modificando la hora para que utilice el **formato preestablecido de 24h**. Por suerte existe una función integrada dentro de Pyspark que nos permite hacer precisamente eso "*unix_timestamp",* la alternativa "*to_timestamp"* nos devuelve la misma información pero en formato de 12h AM/PM.

Al suprimir los segundos “ss” los datos se agrupan convenientemente por minuto, lo cual nos facilitará más adelante la visualización.

Una vez realizado esto con las dos columnas relativas al tiempo de recogida y de bajada vamos a **añadir una nueva columna** a partir de dividir el precio del viaje entre la distancia, de esta manera obtendremos una medida más fiable de los precios de los taxis. El dataset resultante de esta modificación y su esquema son los siguientes:

```
# Add price/distance ratio column
data_frame = data_frame.withColumn("price_distance_ratio",
data_frame.fare_amount / data_frame.trip_distance)
```

```
+-------------------+--------------------+-------------+-----------+--------------------+
|tpep_pickup_datetime|tpep_dropoff_datetime|trip_distance|fare_amount|price_distance_ratio|
+-------------------+--------------------+-------------+-----------+--------------------+
|    2019-01-01 00:46|     2019-01-01 00:53|          1.5|        7.0|   4.666666666666667|
|    2019-01-01 00:59|     2019-01-01 01:18|          2.6|       14.0|   5.384615384615384|
|    2019-01-01 00:21|     2019-01-01 00:28|          1.3|        6.5|                 5.0|
|    2019-01-01 00:32|     2019-01-01 00:45|          3.7|       13.5|  3.6486486486486487|
|    2019-01-01 00:57|     2019-01-01 01:09|          2.1|       10.0|   4.761904761904762|
|    2019-01-01 00:24|     2019-01-01 00:47|          2.8|       15.0|   5.357142857142858|
|    2019-01-01 00:21|     2019-01-01 00:28|          0.7|        5.5|   7.857142857142858|
|    2019-01-01 00:45|     2019-01-01 01:31|          8.7|       34.5|  3.9655172413793105|
|    2019-01-01 00:43|     2019-01-01 01:07|          6.3|       21.5|   3.412698412698413|
|    2019-01-01 00:58|     2019-01-01 01:15|          2.7|       13.0|   4.814814814814815|
+-------------------+--------------------+-------------+-----------+--------------------+
```

```
root
 |-- tpep_pickup_datetime: string (nullable = true)
 |-- tpep_dropoff_datetime: string (nullable = true)
 |-- trip_distance: double (nullable = true)
 |-- fare_amount: double (nullable = true)
 |-- price_distance_ratio: double (nullable = true)
```

Una vez realizados todos estos pasos ya podemos ejecutar el filtrado, que consiste en seleccionar **únicamente los datos comprendidos entre el día 1 y 2**, siendo este último excluyente.

```
# Take Only data from 1st of January

data_frame_january = data_frame.\
where(F.col("tpep_pickup_datetime").\
between("2019-01-01",  "2019-01-02"))

print(data_frame_january.count())
```

Como podemos observar hemos contabilizado el número total de instancias restantes tras el filtrado del día **1 de Enero**. El resultado es que tenemos **187.250** carreras realizadas en tan sólo un día en la Ciudad de Nueva York

El siguiente y último paso es **convertir el dataset a panda**s, una librería muy útil para el manejo de tablas y que combina perfectamente con otras librerías diseñadas para la visualización de datos, tales como seaborn y matplotlib. Y podemos comenzar a analizar los resultados obtenidos.

```
#Convert to pandas to show results
pd_dataset = data_frame_january.toPandas()
pd_dataset['tpep_pickup_datetime'] =
pd_dataset['tpep_pickup_datetime'].astype('datetime64[ns]')
pd_dataset.head()
```

Para que podamos manejar datos de tiempo con librerías de **visualización** como **Matplotlib o Seaborn** debemos convertir convenientemente de nuevo los datos de fecha a formato ISO. Una vez tenemos los datos en formato **Pandas Dataframe** pasamos al Análisis de Resultados.

3. Análisis de Resultados

A continuación analizamos los resultados obtenidos del procesamiento de datos.

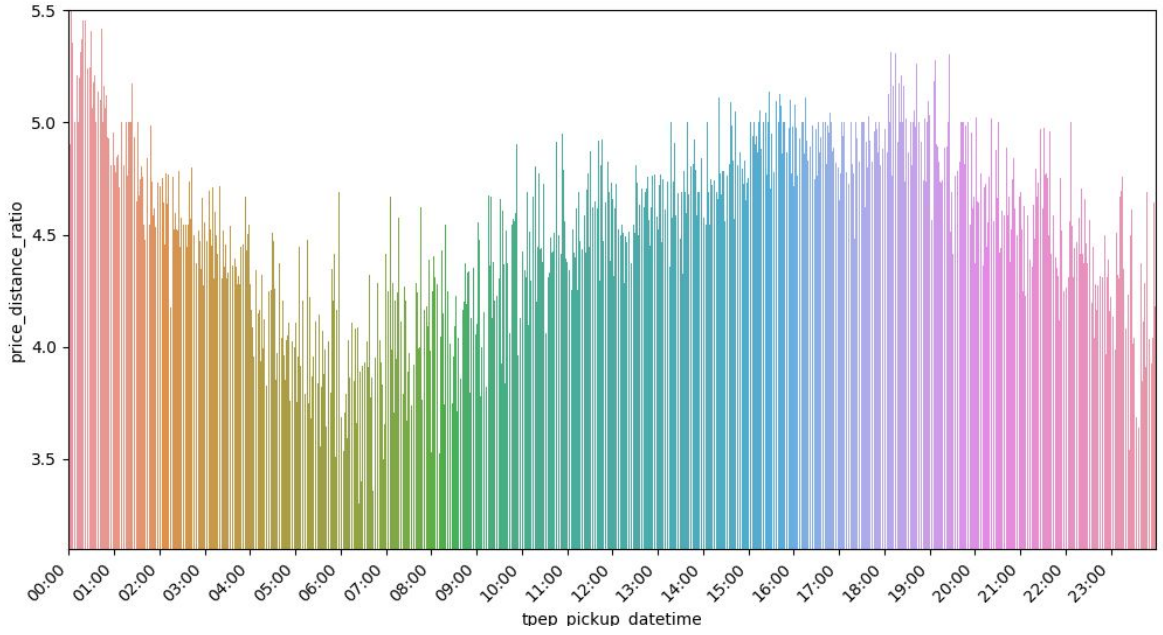

Figura 1: Ratio Precio/Distancia por minuto de los Taxis de NYC a 1 de Enero del 2019

En las gráficas resultantes podemos observar la **fluctuación de precios** a lo largo del día. Observamos que hay un pico del precio por kilómetro, figura 1, justo a las doce de la noche, cuando, suponemos, habrá un mínimo de conductores y viajeros debido a que la mayoría de los habitantes o turistas de la ciudad se encuentran celebrando el año nuevo, muy típico de esta ciudad.

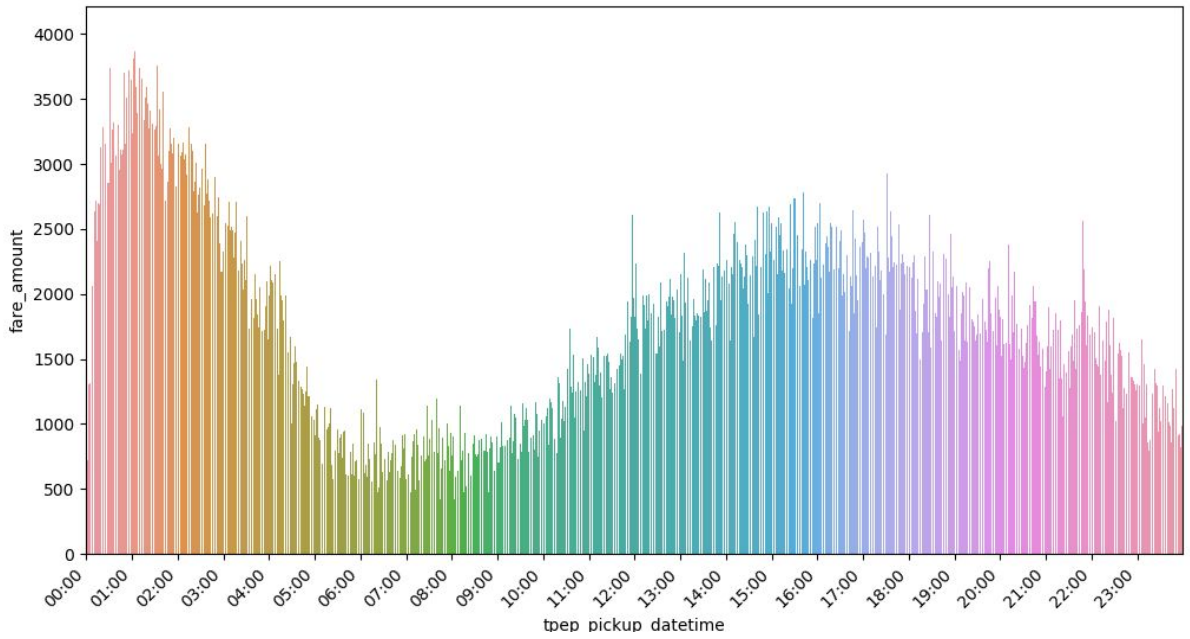

Figura 2: Recaudación total por minuto de los Taxis de NYC a 1 de Enero del 2019

Si observamos la gráfica de la recaudación total, figura 2, a las 00:00 vemos que hay una **caída muy significativa de los cobros** que en poco minutos vuelve a aumentar fuertemente, consecuencia de la reactivación del tránsito de viajeros justo después de las campanadas.

Poco a poco **va descendiendo** tanto **la actividad total** como el ratio precio/distancia hasta alcanzar un valle alrededor de las 6:00 el cual se mantiene durante unas tres o cuatro horas hasta que la ciudad vuelve a revitalizarse y se produce el efecto contrario, pero en este caso ya con subidas más suaves propias de una tarde más normal, **estabilizando su precio medio entre los 4.5$ y los 5.0$** por milla recorrida. y ya según va cayendo la noche los precios van reduciéndose de nuevo ante la falta de viajeros y la mejora en la fluidez general del transporte rodado.

4. Conclusiones

Spark facilita enormemente el manejo de grandes volúmenes de datos incluso cuando se trabaja en local, sin embargo el manejo de **series temporales** de datos sigue suponiendo un pequeño reto, aunque los resultados tienden a ser más

llamativos y más fáciles de interpretar. Los problemas surgen de las distintas formas existentes de notación de fechas que hace de ésta una tarea complicada.

Las distintas versiones de Spark y la utilización de entornos virtuales con librerías como **Pyspark** hace que la búsqueda de recursos sea un poco compleja en algunos casos. Sin embargo el poder de la información resultante de trabajar con cientos de miles de datos tiene un **valor estratégico y económico** muy elevado. Cualquier procesamiento con Spark conlleva un alto trabajo de la CPU tal y como se puede apreciar en la figura 3

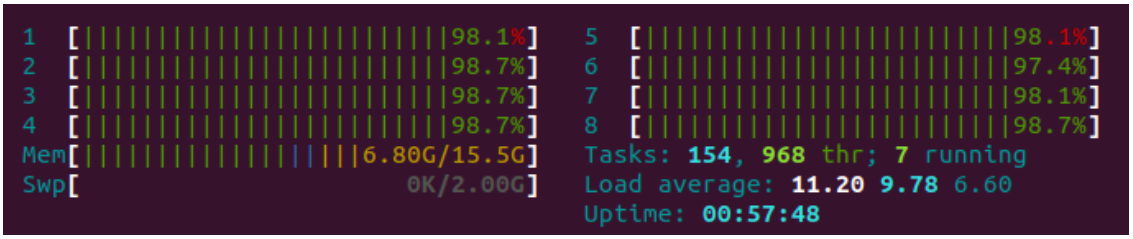

Figura 3: Carga de Trabajo de la CPU con Spark

Por último, los **portales públicos de bases de datos** suponen una gran ayuda al pequeño investigador o estudiante para conocer mejor las dificultades que un analista de datos puede encontrarse en su día a día.

Referencias

El código empleado para este trabajo puede consultarse en el siguiente enlace.