

Análisis de Datos de Rideshare en tiempo real y Batch con Machine Learning

Fernando Zegarra Maratuech
Gonzalo Nicolas Coaguila Valdivia

Octubre 2025

Índice

1. Introducción	2
2. Modelo de Machine learning	2
2.1. Lectura y limpieza de datos	2
2.2. Características	2
2.3. Modelo	2
2.4. Apache Spark	3
3. Arquitectura y Pipeline del Proyecto	3
3.1. Producers de Kafka	3
3.2. Spark Streaming: Consumo y Predicción	3
3.3. Consumers de Kafka	4
3.4. Flujo Completo de Datos	4
3.5. Flujo de Datos	4
3.6. Ejemplo de Salida de Predicción	5
4. HotSpots	5
4.1. Detección de hotSpots	5
4.2. Agregación y Definición de Hotspot	5
4.3. Salida	5
5. Implementación y Análisis de Dashboard Interactivo de Viajes	6
5.1. Procesamiento de Datos en real-time pipeline	6
5.2. Procesamiento de Datos en Batch	6
6. Resultados	7
6.1. Visualizacion de datos real time	7
6.2. Visualizacion de datos Batch	7
7. Conclusiones	7
7.1. Enlace al Repositorio	8

1. Introducción

El proyecto consistió en analizar datos históricos de viajes de Uber y Lyft, y en generar predicciones de precios en tiempo real. Se utilizaron las siguientes tecnologías:

- **Spark Streaming** para procesar datos en tiempo real.
- **Kafka** para transportar eventos de viaje hacia Spark Streaming.
- **Machine Learning** mediante un modelo `PipelineModel` para predecir precios.
- **S3** para almacenar modelos y datos.

2. Modelo de Machine learning

El modelo escogido es un random forest, del cual su objetivo es predecir el precio de los viajes que nos proporcione Kafka. El proceso se divide en los siguientes pasos.

2.1. Lectura y limpieza de datos

En esta sección se descarta todo registro que no tenga el valor de precio y también se cambia el valor del precio ya que en el dataset está como variable string y se tiene que cambiar de tipo a double ya que es un valor que se tiene que predecir y los precios son valores numéricos.

2.2. Características

Spark MLlib necesita que todas las variables de entrada sean de tipo numérico y también se tiene que transformar los datos en un formato que el modelo pueda entender.

- Tratamiento de nulos: en las columnas numéricas como temperatura o distancia, si hay datos faltantes se rellenan con el promedio de su columna.
- Tratamiento de texto: En las columnas categóricas se le asigna un número único a cada categoría, luego pasa por un `OneHotEncoder` para que el número pase a vectores binarios, este paso es vital para evitar errores en las categorías.

2.3. Modelo

El modelo de random forest es un ensamblado de múltiples árboles de decisión de los cuales hace un promedio de sus resultados para obtener una predicción más precisa y estable. El número de árboles en nuestro random forest es de 100 árboles diferentes y cada árbol puede tener una profundidad máxima de 10 niveles. Lo aprendido del random forest será guardado en un S3 para poder usarlo después en el real time pipeline y hacer las predicciones correspondientes de los viajes que ingresen.

2.4. Apache Spark

La elección de Apache Spark es fundamental por su capacidad de procesamiento distribuido, permitiendo manejar datasets masivos que desbordarían la memoria de una sola máquina al dividirlos y procesarlos en un clúster. Esta arquitectura optimiza el algoritmo Random Forest al entrenar múltiples árboles simultáneamente en paralelo, reduciendo drásticamente los tiempos de ejecución frente a métodos secuenciales. Además, su sistema de Pipelines encapsula tanto la lógica de limpieza como el modelo en un único artefacto guardado, garantizando que las transformaciones de datos se repliquen automáticamente en producción sin errores, asegurando así una solución robusta y escalable.

3. Arquitectura y Pipeline del Proyecto

El proyecto consistió en procesar datos de viajes de múltiples servicios de transporte compartido (Uber, Lyft e Indrive) y generar predicciones de precios en tiempo real utilizando Spark Streaming y un modelo de Machine Learning. A continuación se detalla la arquitectura y el flujo de datos implementado.

3.1. Producers de Kafka

Se implementaron tres productores de Kafka independientes, uno para cada servicio de rideshare:

- **Producer Uber:** Genera eventos en formato JSON con información de cada viaje de Uber, incluyendo tipo de vehículo, distancia, hora, día, mes, temperatura, precipitación, surge multiplier y coordenadas de ubicación.
- **Producer Lyft:** Similar al productor de Uber, genera eventos JSON para cada viaje de Lyft con los mismos campos relevantes.

Cada productor publica los datos en un tópico específico de Kafka llamado `rides_data`:

3.2. Spark Streaming: Consumo y Predicción

Se creó un flujo de Spark Streaming que consume los datos del tópico de Kafka. Este flujo realiza las siguientes tareas:

1. **Consumo de eventos:** Spark Streaming se conecta a Kafka y lee los eventos JSON de cada tópico en tiempo real, utilizando la configuración de `startingOffsets="latest"` para procesar solo los datos nuevos.
2. **Transformación y limpieza:** Cada evento es parseado y validado. Se asegura que los campos obligatorios no sean nulos, que las distancias y precios tengan valores consistentes y que los tipos de datos sean los correctos para el modelo.
3. **Aplicación del modelo de Machine Learning:** Se carga un modelo `PipelineModel` entrenado previamente en Random Forest desde S3. Cada evento se transforma con el modelo para predecir el precio del viaje, generando el campo `predicted_price`.

4. **Salida hacia Kafka:** Las predicciones se envían a un tópico de salida llamado `rides_predictions`, permitiendo que otro consumidor use las predicciones y los datos para la visualización.
5. **Checkpointing en S3:** Se configuran checkpoints en S3 para garantizar tolerancia a fallas y permitir reinicios de Spark Streaming sin pérdida de eventos.

3.3. Consumers de Kafka

Se implementaron dos consumidores que reciben los datos procesados:

- **Consumer principal:** Consume los datos colocados en el tópico de `rides_data` y utiliza los pesos guardados en el modelo de nuestro modelo de machine learning mediante Spark Streaming para empezar a generar predicciones y luego las guarda en `rides_predictions`.
- **Consumer Dashboard tiempo real:** consume las predicciones de `rides_predictions` para empezar a generar nuestros hotspots y nuestro mapa.
- **Consumer Dashboard Batch:** consume las predicciones de `rides_predictions` para generar un JSON y lo almacena en S3 para luego ser utilizado en el Dashboard de Batch.

3.4. Flujo Completo de Datos

En resumen, el flujo de datos completo es el siguiente:

1. Los dos producers generan eventos JSON de cada viaje y los publican en el respectivo tópico.
2. Spark Streaming consume estos eventos, realiza transformaciones y validaciones, y aplica el modelo de ML para predecir el precio.
3. Las predicciones se publican en el tópico `rides_predictions`.
4. El segundo consumidor lee los resultados, para la visualización y los hotspots.
5. El tercer consumidor lee el tópico `rides_predictions`, para la visualización de datos del batch.
6. Los checkpoints se almacenan en S3, asegurando que el sistema sea tolerante a fallas.

3.5. Flujo de Datos

1. Los eventos JSON se publican en Kafka.
2. Spark Streaming los consume y transforma los datos.
3. El modelo de ML predice el precio del viaje.
4. Los resultados se envían nuevamente a Kafka.

3.6. Ejemplo de Salida de Predicción

```
1 {  
2   "cab_type": "Lyft",  
3   "short_summary": "moderate rain",  
4   "name": "Sedan",  
5   "distance": 18.96,  
6   "hour": 18,  
7   "day": 23,  
8   "month": 11,  
9   "temperature": 28.2,  
10  "precipIntensity": 5.13,  
11  "surge_multiplier": 2.2,  
12  "latitude": 42.314603,  
13  "longitude": -71.104891,  
14  "predicted_price": 28.73  
15 }
```

4. HotSpots

El propósito de los hotSpots es procesar un flujo continuo de datos de predicciones de viajes (ingestados vía Kafka) para identificar, en tiempo real, áreas geográficas específicas y ventanas temporales donde existe una concentración inusual de solicitudes de taxi. Esto permite la toma de decisiones operativas dinámicas.

4.1. Detección de hotSpots

Se detecta hotspots cruzando dimensiones espaciales y temporales. Espacialmente, utiliza una técnica de Gridding que discretiza coordenadas continuas en celdas de aprox. 200 metros, agrupando viajes cercanos mediante una fórmula matemática de redondeo tras filtrar zonas fuera del área de interés. Temporalmente, aplica una Ventana Deslizante de 2 minutos que se actualiza cada minuto. Esta estrategia permite identificar concentraciones dinámicas de demanda que surgen y desaparecen rápidamente, transformando coordenadas exactas dispersas en zonas de densidad medibles y comparables en tiempo real.

4.2. Agregación y Definición de Hotspot

El proceso de identificación se consolida agrupando el DataFrame simultáneamente por tres dimensiones clave: la ventana temporal activa y las celdas discretizadas de latitud y longitud. Sobre estos grupos se aplica un filtro de umbral lógico (conteo mayor a 1), determinando que cualquier cuadrante de aproximadamente 200 metros que registre actividad por encima de este límite dentro de los dos minutos evaluados se clasifica operativamente como un "Hotspot", sirviendo este valor como una línea base demostrativa escalable para producción

4.3. Salida

Previo a la emisión de resultados, se ejecuta un post-procesamiento que calcula el centro geométrico de cada celda y genera un zone_id único para facilitar la proyección

exacta en mapas de visualización. Finalmente, el flujo de datos se configura en modo complete, lo que instruye a Spark a reescribir la tabla completa de resultados en la consola tras cada ciclo de procesamiento, mostrando siempre una instantánea actualizada de los conteos en todas las ventanas temporales activas.

5. Implementación y Análisis de Dashboard Interactivo de Viajes

5.1. Procesamiento de Datos en real-time pipeline

Este código funciona como el Consumidor del sistema y se conecta al script de Spark (el Productor) a través del bus de mensajería Apache Kafka, actuando como el puente final para la visualización. La conexión técnica ocurre específicamente en el tópico `hotSpotsgra`: mientras que el script de Spark inyecta en este canal los resultados procesados (coordenadas de las celdas y su intensidad), este script de Streamlit se suscribe al mismo canal utilizando un `KafkaConsumer` en modo polling. Mediante un bucle infinito, el código descarga los mensajes JSON recién llegados, extrae las variables clave (`lat`, `lon`, `intensity`), las acumula en un buffer de memoria temporal y refresca dinámicamente un mapa de calor, permitiendo que las zonas de alta demanda calculadas matemáticamente en el paso anterior se materialicen visualmente en tiempo real ante el usuario.

5.2. Procesamiento de Datos en Batch

Se trabajó sobre un conjunto de datos predichos (`rides_prediction.json`) con campos clave como `predicted_price`, `hour`, `distance`, `surge_multiplier`, `cab_type`, y `short_summary`. El diseño **CSS** fue mejorado a un estilo de *dashboard* moderno (cajas elevadas y encabezado con degradado).

El archivo `index.html` contiene la lógica de procesamiento (`processData`) y las funciones de dibujo (`draw...`) con la librería **Chart.js**.

Gráficos Finales Implementados

Se implementaron y optimizaron seis gráficos clave para un análisis integral de demanda y precios:

1. **Conteo de Viajes por Hora** (*Gráfico de Barras*): Identifica las **horas pico** de demanda.
2. **Precio Predicho vs. Distancia** (*Gráfico de Dispersión/Scatter*): Muestra la **correlación** entre la distancia del viaje y el precio final predicho.
3. **Precio Promedio por Hora del Día** (*Gráfico de Líneas*): Revela la **tendencia de precios** a lo largo del ciclo diario, complementando el análisis de demanda.
4. **Proporción de Viajes con Tarifa Dinámica (Surge)** (*Gráfico de Dona*): Cuantifica la **frecuencia de aplicación de sobrecargos** (`surge_multiplier > 1.0`) en el total de viajes.
5. **Conteo de Viajes por Plataforma (cab_type)** (*Gráfico de Barras Horizontales*): Compara el volumen de viajes entre las **diferentes plataformas** de vehículos.

6. **Precio Promedio por Condición Climática (short_summary)** (*Gráfico de Radar*): Analiza cómo las **condiciones ambientales** (ej. `snow`, `clear`) impactan el precio promedio del viaje, usando un formato radial para la comparación multi-variable.

6. Resultados

6.1. Visualizacion de datos real time

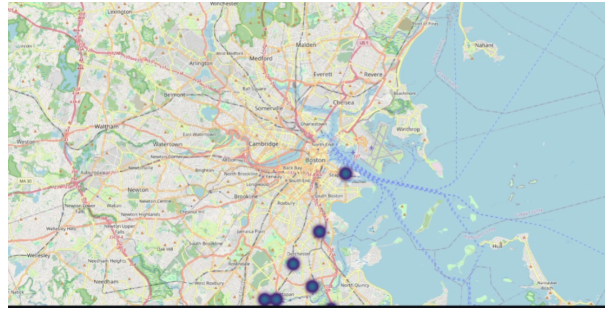


Figura 1: Graficos de los datos en Real-time

6.2. Visualizacion de datos Batch

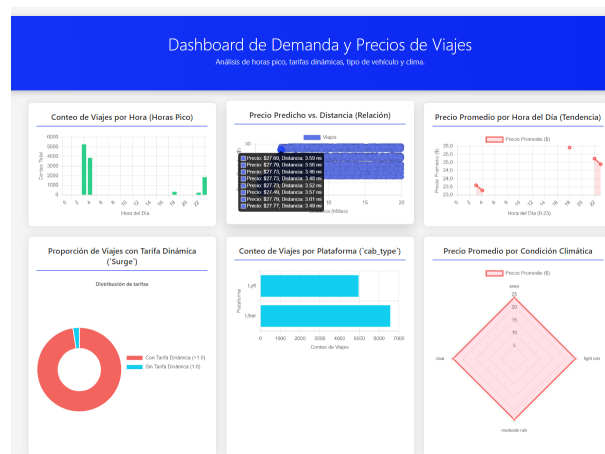


Figura 2: Graficos de los datos en Batch

7. Conclusiones

El proyecto demostró la viabilidad de una arquitectura **Big Data**, utilizando **Spark Streaming** y **Kafka** para el procesamiento distribuido de eventos de viajes en tiempo real. La aplicación del **Random Forest (PipelineModel)** aseguró predicciones de precios estables y precisas. Operacionalmente, la técnica de **Gridding** en tiempo real fue clave para la **detección dinámica de Hotspots** (concentraciones de demanda), mientras que la visualización *batch* permitió un análisis multifactorial crucial, confirmando la influencia directa de las **Horas Pico**, la **Distancia** y las **Condiciones Climáticas (short_summary)** en la variación de los precios y la demanda.

7.1. Enlace al Repositorio

El código fuente del proyecto se encuentra en: <https://github.com/Fernandozs10/Proyecto-Big-Data-MapReduce>