Informe sobre TP de Programación Avanzada

Maximiliano Martino, Andrea Quiroz, Sofía Videla

UDESA, Maestría en Ciencias de Datos, 2024

Este trabajo práctico se enfocó en la recomendación de artículos para publicidad en internet. Nuestro objetivo era desarrollar una API capaz de sugerir productos específicos para mostrar a los usuarios, utilizando un catálogo de productos y datos sobre visitas a los productos y clics en anuncios como base.

Durante el desarrollo de nuestro proyecto, nos encontramos con diversos desafíos que nos brindaron una experiencia enriquecedora y de aprendizaje significativo. A continuación, describimos los pasos que seguimos, las dificultades que enfrentamos y las soluciones que implementamos:

1. **Generación de Datos y configuración inicial de AWS**

Comenzamos generando los datos necesarios para nuestro proyecto utilizando un Jupyter Notebook proporcionado.

Generamos 3 archivos de tipo csv:

- Advertisers: Este archivo contiene una lista de los identificadores de los anunciantes activos en nuestro sistema.
- Product Views: Se trata de una tabla donde cada fila representa una visita a un producto. Contiene información como la fecha de la visita, el identificador del anunciante y el identificador del producto.
- Ads Views: En este archivo, cada fila registra una interacción con un anuncio. Puede ser una impresión (cuando el anuncio se muestra al usuario) o un clic (cuando el usuario interactúa con el anuncio). Contiene detalles como la fecha, el identificador del anunciante y el identificador del producto.

Posteriormente, creamos una cuenta en AWS y configuramos un budget para gestionar nuestros gastos. Subimos los datos crudos a un bucket de S3.

2. **Configuración de Base de Datos:**

Establecimos una base de datos PostgreSQL en AWS para almacenar nuestros datos. Modificamos las reglas de entrada para que la base de datos fuera accesible desde cualquier dirección IP y luego utilizamos pgAdmin para interactuar con ella.

3. **Definición de Tablas y Modelos:**

Definimos las tablas de la base de datos esenciales para nuestro proyecto, TopProduct y TopCTR, con el propósito de almacenar los 20 productos mejor clasificados de cada anunciante, utilizando dos criterios distintos:

1. Top Product: Esta tabla guarda los 20 productos con mayor cantidad de visitas por día y por anunciante. Esto nos permite identificar qué productos son los más populares entre los usuarios.
2. Top CTR (Click-Through Rate): En esta tabla, registramos los 20 productos con el mejor CTR, calculado como la proporción de clics sobre impresiones. Esto nos ayuda a identificar los productos que generan mayor interés y compromiso por parte de los usuarios.

Para cada una de estas tablas, definimos los campos necesarios, que incluyen la fecha, el identificador del anunciante, el identificador del producto y su ranking en función del criterio correspondiente. Esta estructuración nos permite almacenar y consultar eficientemente los datos relevantes para nuestras recomendaciones de productos en la API.

4. Pruebas Locales de Airflow:

Nuestro próximo paso era definir nuestro data pipeline para que la salida de los modelos se almacene todos los días en nuestra base de datos en RDS. Antes de implementarlo en una instancia de AWS, decidimos realizar pruebas locales para asegurarnos de que nuestro flujo de trabajo funcionara correctamente.

4.1 Instalación y configuración de Airflow Local

Instalamos Airflow localmente y modificamos sus configuraciones para que utilice la base de datos RDS y el LocalExecutor.

En esta etapa, nos enfrentamos a problemas de recursos y tiempos de espera, los cuales solucionamos aumentando el timeout y asignando más recursos, dado que estábamos ejecutando el sistema localmente y no teníamos limitaciones de memoria y procesamiento.

4.2 Desarrollo del DAG en Airflow:

1. Creamos un DAG en Airflow con tres tareas principales:
    - Task 1: Leer datos crudos de S3, filtrar por fecha y por advertisers activos, y subir datos preprocesados nuevamente a S3.

- Task 2: Modelo Top Product: Para cada anunciante, seleccionar los 20 productos con más visitas y guardarlos en la tabla TopProduct de RDS.
- Task 3: Modelo Top CTR: Para cada anunciante, seleccionar los 20 productos con el mejor CTR y guardarlos en la tabla TopCTR de RDS.

Definimos la dependencia entre las tareas, asegurándonos de que Task 2 y Task 3 se ejecuten después de que Task 1 se completara con éxito.

Definimos el schedule para que el proceso corra todas las noches a las 4am, y filtre para obtener los datos del día anterior.

En esta etapa, tuvimos un inconveniente con la función to_sql de sqlalchemy. Recurrimos a internet y encontramos que tenía que ver con la versión de la libraría pandas. Con lo cual pudimos solucionar el problema re-instalando una versión anterior de la librería.

También usamos un jupyter notebook que corría un script muy similar a las Tasks para debuggear más fácilmente el proceso y para cargar datos históricos a RDS.

5. Implementación de Airflow en EC2:

Después de haber verificado el funcionamiento del DAG en Airflow local, avanzamos hacia la fase de ejecución en la nube mediante la configuración de un servidor EC2 en AWS para alojar nuestro entorno de trabajo. En la instancia EC2, procedimos a instalar Airflow y todas sus dependencias a través de un archivo requirements.txt que detallaba las versiones necesarias. Durante este proceso, nos enfrentamos a una serie de conflictos de dependencias que surgieron de inmediato, como por ejemplo, la dependencia específica de Airflow en una versión particular de Boto Core. Resolver estos conflictos implicó cierta labor de ajuste, probando diversas combinaciones de versiones hasta llegar a una solución viable. En un momento dado, identificamos que la máquina EC2 tenía instalada una versión diferente de Python a la que estábamos utilizando localmente, lo cual también abordamos reinstalando la biblioteca correspondiente. Después de esta fase de ajustes y correcciones, logramos que nuestro data pipeline se ejecutara sin contratiempos en la máquina virtual.

6. Fast API script

Para desarrollar la API, comenzamos escribiendo el código en Python. Utilizando la librería FastAPI, establecimos la conexión a la base de datos en RDS y definimos tres funciones distintas para las tres salidas requeridas por nuestra API: recomendaciones, historial y estadísticas. Durante esta etapa, no encontramos mayores dificultades dado

que las consultas para recuperar la salida de la API eran relativamente simples. Además, incorporamos mensajes de error para manejar situaciones en las que el usuario ingresara un modelo o un ID de anunciante incorrecto.

7. API dockerizada

El siguiente paso consistió en dockerizar la API. Diseñamos nuestro Dockerfile, utilizando una imagen ligera de Python como base y un requirements.txt sencillo que instalaba FastAPI, Pandas y Psycopg2. En el entrypoint, especificamos el servidor Uvicorn, utilizado por FastAPI, y el nombre de la aplicación que queríamos ejecutar. Seleccionamos el puerto 80 para la comunicación.

Por otro lado, creamos un repositorio en ECR de AWS para almacenar la imagen Docker. Luego, desde la terminal, construimos la imagen con Docker Build y la enviamos a ECR con un push. La mayor dificultad radicó en la gestión de permisos, lo que nos llevó a instalar la CLI de AWS en la terminal.

Una vez que la imagen estuvo en ECR, accedimos a App Runner de AWS y seleccionamos la imagen de ECR para crear la instancia. El despliegue fue sencillo: con solo un clic para iniciar, obtuvimos la URL para probar la API. Por ejemplo, /stats proporciona las estadísticas requeridas.

Conclusiones

En resumen, este proyecto nos permitió adquirir experiencia en el despliegue de flujos de trabajo de datos en la nube utilizando servicios de AWS y herramientas como Airflow. Aprendimos a enfrentar y resolver desafíos relacionados con la configuración de recursos, la gestión de bases de datos y la programación de tareas en un entorno distribuido. Esta experiencia nos preparó para abordar proyectos más complejos en el futuro y nos proporcionó una base sólida en el campo de la ingeniería de datos.