Trabajo práctico final
Programación Avanzada
2024

Aviso preliminar

Todos los servicios de AWS requeridos para este trabajo entran en el *free tier* o bien tienen un costo muy bajo por hora de uso. De todas maneras, si se usan estos servicios de manera imprudente podrían en algunas semanas generar costos no tan despreciables. Recomendamos tener en cuenta los tiempos en los que los diferentes servicios están siendo utilizados y revisar periódicamente su resumen de costos en AWS.

Enunciado

En este trabajo práctico se ejercitarán y probarán diferentes herramientas que les permitan compartir un servicio que han creado. Para eso, desarrollaremos un sistema de recomendación de productos como solución de un producto AdTech y lo convertiremos en un servicio accesible mediante una API, es decir en un servicio accesible por interfaz de programación de aplicaciones.

Nuestro problema a resolver es la recomendación de artículos en publicidades por internet.
Cuando un usuario entra a una página web, muchas veces aparecen varias publicidades en la misma que no son estáticas ni únicas, sino que dependen fuertemente de quién está ingresando, a qué hora, desde qué dispositivo, etc.
Esta situación, que se resuelve en milisegundos, involucra muchos actores. Casi nunca es el sitio mismo el que gestiona las publicidades, sino que ofrece sus espacios de publicidad a otra empresa que resolverá el problema de qué mostrar.
La situación real consta de muchos participantes: usuario, sitio web, ad exchange, bidders, advertisers, etc. En esta aplicación vamos a simplificar las interacciones parándonos en el punto de vista de una empresa de adtech (media agency) que tiene de clientes a varias empresas que quieren publicitar sus productos. Nuestro problema será entonces, dado un catálogo de productos y un usuario con su historia de navegación, indicar qué producto se recomienda mostrarle al usuario.

Para esto nos interesan dos procesos: El proceso de generar las recomendaciones y el proceso de servir las recomendaciones.

Para generar las recomendaciones vamos a contar con algunos datos de navegación de los usuarios. Contaremos con *logs* que nos indiquen qué productos vieron los usuarios en las webs de nuestros clientes (advertisers) y contaremos con logs que nos indiquen qué publicidades fueron mostradas a los usuarios y cuáles fueron clickeadas. A partir de esta información es que podremos generar las nuevas recomendaciones con los modelos detallados más adelante.

Para servir las recomendaciones contaremos con las mismas ya computadas en nuestra base de datos y la api presentará diferentes *endpoints* para acceder en tiempo real a las recomendaciones precomputadas.

Para solucionar este problema tendrán que programar y disponibilizar:

1. Un pipeline de procesamiento de datos mediante Airflow que corra en AWS EC2 que todas las noches haga el cómputo necesario para que los resultados sean servidos de manera automática el resto del día.
2. Una API accesible por HTTP que esté dockerizada y se disponibilice vía AWS ECS o AWS App Runner (a elección del grupo). Esencialmente, este servicio recibirá requests sobre qué producto mostrar en un ad a un usuario durante su navegación y responderá con la lista de recomendaciones.
3. Una base de datos disponible en AWS RDS en donde el pipeline de datos escribirá las recomendaciones y de donde la API leerá.

1. Pipeline de datos

Vamos a desarrollar un pipeline para el procesamiento de datos y la generación de las recomendaciones, que correrá diariamente y disponibilizará los resultados en una base de datos para ser accedida por la API.

El pipeline deberá trabajar con los datos crudos disponibles desde un bucket de S3. También utilizará S3 para ir escribiendo los resultados intermedios (las salidas de cada uno de los procesos del pipeline). Por último, el producto final de todo el pipeline será volcado en la base de datos PostgreSQL disponible en AWS RDS.

Datos:

- Log de vistas de productos en la página del cliente. Los mismos consistirán de sucesivas líneas, una por cada evento de vista de un producto.
  Cada línea contendrá los campos: advertiser_id, product_id, fecha
- Log de vistas de ads en las diferentes páginas web. Los mismos consistirán de sucesivas líneas, una por cada evento posible sobre los ads.
  Cada línea contendrá los campos: advertiser_id, product_id, type, fecha. Siendo type un campo que indica si el evento fue “impression” o “click”.
- Lista de advertiser_id que se encuentran activos en nuestra plataforma.

Tareas del pipeline:

- FiltrarDatos: Los logs crudos contienen datos sobre advertisers que ya no se encuentran activos en nuestra plataforma. Esta primera tarea debe limpiar los logs del día, dejando solamente las líneas que se refieran a clientes activos.
- TopCTR: Esta tarea computa el modelo TopCTR. Por cada advertirser activo debe devolver los 20 (o menos) productos con mejor click-through-rate.
- TopProduct: Esta tarea computa el modelo TopProduct. Por cada advertirser activo debe devolver los 20 (o menos) productos más vistos en la web del cliente.
- DBWriting: Esta tarea debe tomar los resultados de los modelos y los debe escribir en la base de datos PostgreSQL disponible en AWS RDS.

Deploy del pipeline:

El pipeline de Airflow debe correr sobre una instancia de AWS EC2.
Airflow viene configurado por defecto con el SequentialExecutor, que no es apto para entornos de producción. Para poder utilizar otro executor, es necesario configurar una base de datos distinta a SQLite (que también viene por defecto). En este caso vamos a utilizar la instancia RDS creada en AWS que nos provee una base de datos PostgreSQL.

Cambios a realizar en la configuración (AIRFLOW_HOME/airflow.cfg):

En la sección [core]

- Setear executor = LocalExecutor
- Setear parallelism = 2 para que se corran como máximo 2 tasks al mismo tiempo y no saturar los recursos del servidor
- (opcional) Setear load_examples = False si quieren que el scheduler no cargue todos los DAGs de ejemplo

En la sección [database] **Esto se hace antes de hacer el db migrate**

- Setear sql_alchemy_conn = postgresql+psycopg2://usuario:contraseña@rds_url donde usuario, contraseña y rds_url los obtienen a partir de su instancia RDS

En la sección [webserver]

- (opcional) Setear workers = 1 para que Gunicorn utilice un único worker y no consuma más recursos

Una vez realizados estos cambios, pueden inicializar la db de airflow.

Al levantar el webserver y el scheduler, pueden utilizar el parámetro -D para que sean levantados como procesos daemon. De esa manera los procesos van a correr en background y no les va a bloquear la consola ni se van a morir si se desloguean del server.
En caso de que quieran matar los procesos pueden utilizar top (o htop) para buscar el id del proceso (pid) y luego matarlos con kill (o desde htop directamente con F9).

2. API

La API deberá estar implementada en FastAPI y debe soportar las siguientes entradas:

MODELOS:
TOP params
CTR

Entradas API

path param ambos

- /recommendations/<ADV>/<Modelo>

  Esta entrada devuelve un JSON en dónde se indican las recomendaciones del día para el adv y el modelo en cuestión.

- /stats/

  Esta entrada devuelve un JSON con un resumen de estadísticas sobre las recomendaciones a determinar por ustedes. Algunas opciones posibles:
  - Cantidad de advertisers
  - Advertisers que más varían sus recomendaciones por día
  - Estadísticas de coincidencia entre ambos modelos para los diferentes advs.

- /history/<ADV>/ **Las 20 recomedaciones de cada dia**

  Esta entrada devuelve un JSON con todas las recomendaciones para el advertiser pasado por parámetro en los últimos 7 días.

Si no hay adv mandar 400 como error

Deploy de la API:

La API debe estar disponibilizada utilizando AWS App Runner o AWS ECS (a elección del grupo). Para ambas opciones, se requiere que la aplicación esté *dockerizada*. Por ejemplo, pueden utilizar la imagen python:3.10-slim-bullseye como base.
La instalación de FastAPI debería ser análoga a lo visto en clase. Luego se debería crear un *working directory* llamado “app” en donde pongan su código.
La API debe ser levantada usando el ASGI server uvicorn.

3. Base de datos

Nuestro pipeline diario y nuestra API se comunican mediante una base de datos en donde se guardan las recomendaciones.
La base debe ser de tipo PostgreSQL y estar alojada en AWS RDS.

Interacción con la base:

Para interactuar con la base se pide utilizar la librería *psycopg2* tanto para la escritura como para la lectura.

Deploy de la base:

Sobre el deploy no hay mayores restricciones. La base debe estar en AWS RDS y debe ser accesible tanto por el pipeline de datos como por la API para poder escribir y leer de ella.

Criterios evaluación y entregable

La entrega de trabajo práctico consiste en levantar los servicios y tenerlos disponibles para la corrección. El periodo para levantar los servicios se coordinará con el corrector para minimizar costos.

Además de tener levantado los servicios, deberán compartirnos el repositorio con el código utilizado, junto a un pequeño informe en donde detallen los pasos más importantes y las dificultades encontradas para desarrollar todo el sistema. Deberán enviar su entrega mediante el campus de la materia, utilizando la carpeta de tareas entregables llamada TP Final. Deberán hacer una sola entrega por grupo. La fecha límite para la entrega es **19-05-2024 23:59hs.**

Nota general

Este trabajo práctico tiene como motivación que ustedes encaren un proceso íntegro desde la creación del modelo (que ya hicieron) hasta la puesta en producción. Durante el proceso de resolución, se toparán con muchos errores que les impedirán avanzar. **Esto es esperable** y parte del trabajo consiste en que ustedes lidien con eso, buscando el error en internet, leyendo comentarios en foros y documentación oficial. Los docentes por supuesto estamos para acompañarlos en que este proceso no sea tortuoso, pero la idea es que usen esta instancia para ejercitar y aprender a resolver problemas de implementación.

texto