UNIVERSIDAD DE CASTILLA-LA MANCHA

LABORATORIO

DOCUMENTACIÓN

Sistemas Multiagentes

LAURENTIU GHEORGHE ZLATAR
ISRAEL MATEOS APARICIO RUIZ SANTA QUITERIA
IGNACIO ROZAS LÓPEZ

17 de diciembre de 2023

Índice

1. Introducción

El presente documento tiene como fin documentar el desarrollo de la práctica de Sistemas Multiagentes. Esta consiste en montar una base de datos y una API REST para poder realizar peticiones a la misma. Los conjuntos de datos usados son datos de violencia con armas en Estados Unidos, referentes al proyecto de la asignatura de Minería de Datos. Además, debe implementarse una funcionalidad extra. En nuestro caso, esta ha sido un *frontend* para visualizar rápidamente los datos.

2. Obtención de datos y web scraping

Los conjuntos de datos para el desarrollo del proyecto son obtenidos con el *script* `src/data/get_datasets.py`. Para los *datasets* con un enlace de descarga, simplemente se obtienen mediante una petición GET al enlace. A continuación, entraremos más en detalle en dos conjuntos de datos: el de incidentes de violencia con armas, y el de datos de pobreza. Para el primero hay que hacer uso de la API de Kaggle, mientras que el segundo ha sido obtenido con técnicas de *web scraping*.

2.1. Conjunto de incidentes

El conjunto de incidentes es el *dataset* principal de nuestro proyecto. Como se encuentra alojado en la web de Kaggle y esta cuenta con una API propia, la usaremos para descargarlo automáticamente.

Para ello, necesitamos establecer el módulo `kaggle` como requisito de nuestro proyecto. La API de Kaggle nos exige una identificación, por lo que es necesario crear una cuenta en Kaggle y descargar un token identificativo en su página web, que se provee en formato *json*. Para que el *script* pueda leer el token, declaramos una variable de entorno `KAGGLE_CONFIG_DIR` con el directorio que el token. Con el fin de securizar la información, esta variable se declara en un fichero `.env` que es excluido del repositorio en el fichero `.gitignore`. Así, cada persona que use el proyecto debe establecer dónde se encuentra su token, y la información no se filtrará al repositorio remoto. Esta variable de entorno se lee mediante el uso del módulo `dotenv`.

2.2. Conjunto de datos de pobreza

El conjunto de datos de pobreza no se encuentra en un formato usable, sino que se muestra en `https://www.povertyusa.org/data`. La página permite mostrar los datos de pobreza según el estado y el año. En nuestro caso, nos interesan los años del 2015 al 2018 para todos los estados. Por tanto, iremos navegando año por año y estado por estado, recopilando toda la información que nos interese.

En la página web, nos encontramos selectores de estado y año. Cuando se selecciona alguno, nos redirige a un *endpoint* con el código estándar del estado y el año. Por tanto,

necesitamos primero estos códigos de estado. En vez de hacer una lista manualmente con todos los códigos de los estados, los obtenemos automáticamente de `https://en.wikipedia.org/wiki/ISO_3166-2:US`. Para ello, utilizamos el módulo **BeautifulSoup**, un *parser* de HTML y XML, ya que la página no carga los datos dinámicamente, sino que se encuentran en el fichero HTML. Obtenemos este fichero mediante una petición GET, y extraemos los datos que nos interesan.

Una vez que tenemos los códigos, pasamos a obtener los datos de pobreza. Estos datos se cargan dinámicamente en la página, por lo que no podemos utilizar directamente `BeautifulSoup`. Por tanto, usaremos **Selenium** para abrir una ventana con el driver de Chrome. Una vez cargados los datos gracias a ello en el HTML, lo obtenemos y lo *parseamos* con `BeautifulSoup`.

Una vez recopilados todos los datos, se exportan a un fichero `.csv` formando antes un DataFrame con **Pandas**. Este es creado con el método `from_dict` para mayor eficiancia.

3. Uso de Docker

Para evitarnos problemas de conflictos y configuraciones de entornos, hemos decidido utilizar **Docker** como la base de nuestra infraestructura.

Docker es una plataforma de código abierto diseñada para la creación y ejecución de contenedores, que permiten encapsular aplicaciones y dependencias. Los contenedores actúan como una especie de máquina virtual, con la diferencia de que los contenedores Docker comparten el sistema operativo del anfitrión, mientras que las máquinas virtuales también tienen un sistema operativo invitado que se ejecuta sobre el sistema anfitrión. Esto los hace mucho más eficientes y ligeros. Además, ya hay disponibles varias imágenes para crear contenedores con todo tipo de entornos ya configurados para su uso.

Nuestro proyecto necesita de varios servicios y *scripts*, por lo que utilizaremos **Docker Compose**. Docker Compose permite crear aplicaciones con varios contenedores, y crear una red interna entre ellos. Con un único comando, toda la infraestructura se levanta. Para ello, es necesario definir los servicios en un fichero YAML. A continuación, se detallan los servicios definidos en nuestra aplicación.

3.1. Servicios

postgres

Necesitaremos almacenar nuestros datos en una base de datos —su estructura se detallará posteriormente en este documento—. Para ello, hemos decidido utilizar **PostgreSQL** como sistema de gestión de bases de datos.

Este es el primer servicio en levantarse, montando la base de datos. El puerto desde el que se accede tanto internamente en el contenedor como su mapeo al exterior son definidos en el fichero. Además, se definen valores de la base de datos como su nombre, usuario y

contraseña como variables de entorno. Se define un volumen, que mapea un directorio en el contenedor con otro en el anfitrión, con el fin de persistir la información. Esto debe hacerse porque los contenedores de Docker borran su memoria tras la ejecución.

Por último, hemos definido un *healthcheck*, que consiste en probar el comando `pg_isready` cada segundo para comprobar cuando la base de datos está lista para recibir conexiones. Esto nos permitirá establecer dependencias con otros servicios, de manera que no comiencen su ejecución sin que la base de datos esté lista (en caso de que la necesiten).

data_loader

La imagen de este contenedor lo hemos creado nosotros mismo, definiéndola a través de un archivo *Dockerfile*, que establece cómo debe Docker crear el contenedor. Este contenedor usa Python 3.9. Partimos de esta imagen, le instalamos las dependencias y copiamos los conjuntos de datos y los *scripts* necesarios del anfitrión al contenedor. Por último, ejecutamos el *script* `load_data.py`, que carga los datos a las distintas capas de nuestra base de datos.

En la definición del servicio, establecemos una dependencia con *postgres* de manera que sólo se levante una vez la base de datos esté lista para recibir conexiones. Además, definimos las variables de entorno necesarias para la conexión con la base de datos, y creamos un volumen para *logs*.

backend

De nuevo, definimos la creación de la imagen con un fichero *Dockerfile*. Partimos de la imagen de Python 3.10 (necesitamos una funcionalidad de esta versión), instalamos las dependencias y ejecutamos el servidor. También establecemos una dependencia con *postgres* de la misma manera que en *data_loader*. Además, establecemos una dependencia con *data_loader*, de manera que sólo se levante una vez este se haya ejecutado, para evitar cargar el servidor sin datos a los que acceder.

pgadmin

Este contenedor será creado a partir de la imagen oficial de *pgadmin*, una herramienta para gestionar y administrar bases de datos en PostgreSQL. Definimos las variables de entorno necesarias para su acceso, junto con el puerto en el que se encontrará. Establecemos de nuevo una dependencia con *postgres*.

4. Base de datos

La base de datos en la que se almacenan nuestros conjuntos de datos se monta con PostgreSQL, como se ha mencionado anteriormente. Su estructura consiste en tres esquemas:

1. ***raw***. Contiene los datos crudos, como se obtienen directamente de las fuentes. Cada conjunto de datos se almacena en una tabla.

2. ***silver***. Contiene los datos preprocesados, i.e. en su estado previo a la creación de nuestras tarjetas de datos finales. Cada conjunto de datos se almacena en una tabla.

3. ***gold***. Contiene las tarjetas de datos finales que se usan en algoritmos de minería de datos. Cada una de ellas se almacena en una tabla distinta.

Para crear y poblar estos esquemas, utilizamos el *script* `load_data.py`. Este *script* realiza la siguiente secuencia de operaciones:

1. Se conecta con la base de datos.

2. Carga los datos crudos en *raw*.

3. Preprocesa los datos usando funciones del *script* `preprocess_datasets.py` (este no se explicará, por ser objeto de otra asignatura).

4. Carga los datos preprocesados en *silver*.

5. Crea las tarjetas de datos finales usando funciones del *script* `create_processed_tables.py` (también es objeto de otra asignatura).

6. Carga las tarjetas de datos en *gold*.

Algo a destacar de este proceso es cómo se realiza la conexión y las posteriores operaciones con la base de datos. Para esto, utilizamos **SQLAlchemy**. SQLAlchemy es lo que se conoce como un *Object Relational Mapper*, cuya función es mapear la información en la base de datos con objetos en el lenguaje de programación que se use —en nuestro caso, Python—. De esta manera, conseguimos una mayor flexibilidad. Las ventajas del uso de SQLAlchemy se ven más claramente en la lógica del *Backend*. Por ejemplo, podemos cargar directamente un DataFrame de Pandas a la base de datos con el uso de una única función `to_sql()`.

5. Backend

Con el fin de montar una **API REST** para acceder a los datos de la capa *gold*, hemos desarrollado un servidor que gestiona la conexión con la base de datos y permite gestionar peticiones.

Una API REST es una interfaz de comunicación entre sistemas que usa el protocolo de transferencia de hipertexto HTTP para obtener datos o ejecutar operaciones sobre ellos en distintos formatos, como XML o JSON. Por tanto, permite realizar peticiones POST, GET, PUT y DELETE.

Para montar esta API, hemos decidido utilizar **FastAPI**, principalmente por su facilidad de uso. FastAPI es un *framework* para crear APIs en Python.

La estructura de nuestro *backend* se detalla a continuación.

models.py

El fichero `models.py` contiene los modelos de SQLAlchemy. Para trabajar más fácilmente con la base de datos, se ha creado un modelo por cada tabla de la capa *gold*. Estos modelos definen objetos equivalente a cada una de las tablas, mapeando la información de la base de datos en nuestro código. Este mapeo se consigue heredando nuestras clases de la base declarativa *Base* de nuestra base de datos, que se crea al conectarse con ella. Esta es una clase especial con metadatos y que permite el mapeo de objetos con tablas.

Así, en cada una de estas clases definimos los atributos equivalente a cada uno de las columnas de nuestras tablas. El uso de SQLAlchemy nos permite abstraer la información y olvidarnos de utilizar directamente sentencias SQL. Además, esto nos protege de vulnerabilidades como las inyecciones en SQL.

schemas/

De manera parecida a los modelos de SQLAlchemy, también hacemos uso de modelos en **Pydantic**, definidos en los ficheros del módulo `schemas`. Estos modelos permiten la validación, la serialización y la generación de los esquemas JSON que se utilizarán en las peticiones y respuestas de nuestra API.

Estos modelos se definen de manera muy parecida a las *dataclasses* de Python, definiendo los atributos y sus tipos. Hemos definido tanto un modelo base del que heredan los demás, ya que algunos atributos son compartidos por todas las tablas, como modelos para cada tabla. Además, según la operación serán necesarios unos datos u otros. Por esta razón, estos modelos se dividen según la operación que se vaya a hacer.

crud/

Con el fin de tener un buen nivel de abstracción en nuestro proyecto, hemos decidido crear unas clases que serán las que manejen las operaciones de nuestra API a bajo nivel. Estas se encuentran en los ficheros del módulo `crud`.

Como su nombre indica, estas clases se encargan de las operaciones básicas CRUD: Create, Read, Update, Delete. De nuevo, hemos creado una clase base, que generaliza las operaciones sin importar la tabla. Después, para cada una de las tablas se define una clase que hereda de esta. Estas clases contienen operaciones específicas para cada tabla. Por ejemplo, en nuestro caso los datos vienen agrupados por estados y años, por lo que tenemos métodos que leen los registros que coinciden con estados y años determinados.

api/

Los ficheros del módulo `api` manejan las operaciones correspondientes a cada *endpoint* de la API. Para cada tabla de la base de datos se ha definido un endpoint mediante la clase `APIRouter` de FastAPI. Esta clase se encarga de definir las distintas rutas de nuestro servidor.

Se han definido aquí las operaciones a realizar cuando se le mandan distintas peticiones a los *endpoints*, siendo estas POST, GET, PUT y DELETE, que corresponden a Create, Read, Update y Delete en la base de datos. Por tanto, en cada una de las operaciones y tablas se llama a la clase correspondiente del módulo `crud`. De esta manera, conseguimos un alto nivel de abstracción.

Las funciones de estas clases permiten recibir atributos, que pueden o bien ser escritos en el *endpoint* como parámetros de SQL, e.g. `/incidents/climate/?state=Alabama`, o como parte del *endpoint* en sí, e.g. `/incidents/climate/45`, dependiendo del uso que se le quiera dar. Además de las operaciones básicas por identificador (clave primaria), permitimos la lectura de datos por estado, año o ambas.

db.py y *main.py*

El fichero `db.py` se encarga de crear una conexión con la base de datos con las variables de entorno definidas en el contenedor, junto con las distintas sesiones a estas, y permite su obtención mediante la función `get_db()`. Es en este fichero en el que se forma la base declarativa de la que hacen uso los modelos de SQLAlchemy.

Finalmente, el fichero `main.py` es el fichero que se ejecuta al levantar nuestro servidor. En este fichero, se define la aplicación de FastAPI en sí, junto con las rutas a cada *endpoint* y el *middleware* de CORS. Este último es un *middleware* que implementa la especificación CORS, y que se utiliza cuando un *frontend* hace uso del servidor. En este se permiten todos los orígenes y operaciones.

6. Frontend

Como funcionalidad original al desarrollo de la práctica, se ha desarrollado un *frontend* que permite ver gráficos acerca de los datos.

El frontend consiste en una aplicación web básica mediante el uso de ficheros HTML, CSS y JavaScript. El fichero HTML presenta el marco de la aplicación web, estilizado mediante el fichero CSS. El código en JavaScript contiene las funciones para hacer cálculos con los datos y rellenar los gráficos.

La funcionalidad del *frontend*, por tanto, se concentra en el código en JavaScript. En este, se crean *arrays* para cada una de las variables de las tablas. El *frontend* realiza peticiones al *backend* para obtener los datos, y los almacena en estos *arrays*. Una vez recogidos estos datos, implementa en las funciones el cálculo de la media cuando corresponde. Finalmente, genera gráficos de barras usando la librería **Chart.js**, que proporciona distintas opciones de visualización de datos.

El *frontend* se ha diseñado así buscando la simplicidad y efectividad. De una manera muy visual, conseguimos aportar una visión global de los datos. Simplificando la interfaz al máximo, conseguimos centrarnos en lo importante.

7. Otras funcionalidades

En esta sección se incluyen funcionalidades o buenas prácticas seguidas en nuestro proyecto, que creemos que proporcionan ventajas no expuestas anteriormente.

7.1. Uso de *logs*

En los distintos *scripts* de nuestro proyecto hemos utilizado el módulo `logging` de Python para registrar el proceso que siguen. Durante el desarrollo de una aplicación, es muy importante conocer su estado en cada momento, tanto para hacer *debug* de fallos como para comprobar su correcto funcionamiento. Por tanto, registramos cada paso en un *log*, que se imprime tanto a la consola como a un fichero. Este fichero contiene una marca de fecha y momento exactos en su nombre para diferenciarlos. Gracias a guardar estos ficheros, se pueden comprobar posteriormente detalles a depurar.

7.2. Código limpio

Una de las características más importantes y comúnmente olvidadadas de una buena aplicación es la mantenibilidad del código. Para ello, durante el desarrollo de nuestro proyecto hemos incluido:

- Uso de estándares de nombrado de variables, así como nombres representativos.
- Uso de *type-checking* en cada método.
- Uso de *docstrings* en cada método.
- Uso de comentarios sólo donde lo consideramos estrictamente necesario.
- Uso de los *formateadores* `black` e `isort` para un código con buenas prácticas de estilo.

7.3. Mantenimiento del repositorio remoto

De igual manera que con la limpieza del código, es importante seguir un control de versiones adecuado y mantenerlo en un repositorio remoto. Hemos alojado nuestro proyecto en GitHub, y hemos hecho uso de lo siguiente:

- Uso de *issues* para establecer cada tarea a realizar.
- Creación de una nueva rama por cada funcionalidad a implementar o fallo a resolver.
- Uso de palabras clave en *commits* y *pull requests* para cerrar automáticamente sus *issues* asociados.

- Establecimiento de nombres a seguir en cada rama, con el fin de formar una convención interna. Este nombre es “tipo de *issue* al que esta asociado”/issue-“número de *issue* al que está asociado”/“nombre representativo”, siendo ejemplos de tipo de *issue feature*, *bugfix*...