UA

Containerización de Modelos

Daniel Asensi Roch
DNI: 48776120C

30 de noviembre de 2023

Índice

1. Introducción

En la era actual de la tecnología de la información, el Deep Learning/Machine Learning ha emergido como una herramienta poderosa para resolver problemas complejos en diversos campos. Sin embargo, la implementación efectiva de arquitecturas de Deep Learning/Machine Learning en entornos de producción presenta desafíos únicos, especialmente en términos de despliegue y mantenimiento. En este contexto, los contenedores Docker ofrecen una solución robusta y flexible, permitiendo el despliegue de aplicaciones de Deep Learning/Machine Learning de manera eficiente y escalable. Este trabajo explora las técnicas y estrategias para desplegar arquitecturas de Deep Learning/Machine Learning en contenedores Docker, destacando su practicidad y eficacia.

1.1. Planteamiento del problema

El principal desafío abordado en esta práctica es el despliegue eficiente de arquitecturas de Deep Learning/Machine Learning dentro de contenedores Docker. Aunque las soluciones basadas en Docker prometen facilidad de uso y portabilidad, configurar estos entornos para aplicaciones específicas de Deep Learning/Machine Learning conlleva complejidades inherentes. Estas complejidades incluyen la gestión de dependencias, la optimización del uso de recursos (como memoria y GPU), y la configuración de entornos que soporten eficazmente la inferencia y, opcionalmente, el entrenamiento de modelos de Deep Learning/Machine Learning .

2. Metodología

La metodología adoptada para esta práctica involucra el uso de contenedores Docker para el despliegue de un modelo de Machine Learning, específicamente utilizando XGBoost para realizar predicciones de peso volumétrico. Se ha prestado especial atención a la configuración eficiente del contenedor, la gestión de recursos y la optimización del modelo para garantizar un rendimiento óptimo.

2.1. Xgboost

XGBoost (eXtreme Gradient Boosting) es un algoritmo de aprendizaje supervisado que se utiliza ampliamente en competiciones de ciencia de datos y en aplicaciones industriales. Es conocido por su eficiencia, rendimiento y flexibilidad. XGBoost es particularmente famoso por su capacidad para manejar grandes conjuntos de datos, su velocidad de ejecución y la precisión de las predicciones. También soporta regularización para evitar el sobreajuste, es capaz de manejar valores faltantes de forma interna, y permite una personalización detallada de los parámetros del modelo. [**?**]

2.1.1. Qué resuelve este modelo

El modelo XGBoost implementado se utiliza para predecir el peso volumétrico de una serie de productos ya empaquetados. Esta predicción es crucial para optimizar el proceso de envío y logística, permitiendo a las empresas calcular de manera más eficiente el espacio necesario para el transporte de productos, así como estimar costos de envío más precisos basados en el volumen y no solo en el peso físico.

2.1.2. Dataset Utilizados

El dataset utilizado para entrenar el modelo XGBoost desempeña un papel fundamental en la precisión y efectividad de las predicciones de peso volumétrico. Este conjunto de datos se compone de características específicas extraídas de los datos de los productos, cuidadosamente seleccionadas para capturar la esencia de los factores que influyen en el peso volumétrico.

2.1.3. Ficheros de pesos

Una vez entrenado, el modelo XGBoost se guarda en la carpeta artifacts como un fichero de pesos. Este fichero puede ser luego cargado para realizar inferencias sobre nuevos datos. La capacidad de almacenar el modelo entrenado facilita su uso en producción, permitiendo realizar predicciones rápidas y eficientes sin la necesidad de reentrenar el modelo cada vez.

2.2. API Desarrollada

En el marco de esta práctica, se ha desarrollado una API robusta utilizando FastAPI, un moderno framework de Python para construir APIs con altos estándares de rendimiento y fácil integración. FastAPI se seleccionó por su capacidad para manejar solicitudes asincrónicas, su soporte para tipado estático y su rápida velocidad de ejecución, lo que lo hace ideal para aplicaciones de Machine Learning en tiempo real. La API diseñada sirve como interfaz para interactuar con el modelo XGBoost, permitiendo a los usuarios obtener predicciones de peso volumétrico de manera eficiente y confiable. [**?**]

2.3. Datos de entrada

La API acepta datos de entrada en formato JSON, los cuales son necesarios para realizar las predicciones. Los datos de entrada incluyen:

- **order**: Una cadena que puede ser utilizada para identificar el pedido o lote de productos.
- **data**: Un objeto que contiene las características detalladas de los productos, incluyendo:
  - **sku**: Identificador único del producto.
  - **product_weight**: Peso del producto.

  - **product_length**: Longitud del producto.
  - **product_width**: Ancho del producto.
  - **product_height**: Altura del producto.
  - **quantity**: Cantidad de productos.

- **conversion_rate**: Tasa de conversión utilizada para calcular el peso volumétrico a partir de las dimensiones del producto.
- **slope_correction**: Un factor de corrección aplicado al cálculo final para ajustar la predicción.

2.3.1. Salida

La respuesta de la API, tras procesar los datos de entrada, se entrega también en formato JSON. La salida incluye:

- **prediction**: El peso volumétrico predicho por el modelo, expresado como un valor decimal.
- **pesoVolumetricoEnKg**: El peso volumétrico calculado en kilogramos, basado en la predicción del modelo y ajustado por el conversion_rate.
- **pesoTotalProductos**: El peso total de los productos, suministrado en la entrada.
- **isOrder**: Un valor booleano que indica si se ha proporcionado un identificador de pedido en la entrada.

Ejemplo de salida:

```
{
"prediction":0.023986880781249997,
"pesoVolumetricoEnKg":8.88402991898148e-05,
"pesoTotalProductos":4.008,
"isOrder":true
}
```

2.4. Estructura de directorios creada

La estructura del proyecto se organiza para promover la separación de intereses y la claridad del código, facilitando el despliegue y la mantenimiento. A continuación se describen los componentes clave:

- ***/api***: Contiene los archivos esenciales de la API, incluyendo el punto de entrada main.py, que define los endpoints de FastAPI, y constants.py para variables globales.

- ***/artifacts***: Almacena los modelos de XGBoost en diferentes versiones, permitiendo la gestión eficiente de múltiples versiones de modelos y el fácil acceso a la versión más reciente.
- ***/src***: Incluye scripts auxiliares como Model.py para la definición del modelo y utils.py para funciones de utilidad, apoyando las operaciones de la API.
- ***/tests***: Dedicado a pruebas automatizadas que aseguran la calidad y funcionalidad del código.
- Archivos de Docker (*/Dockerfile* y *docker-compose.yml*): Permiten la construcción de imágenes Docker y la orquestación de servicios, respectivamente, apoyando el despliegue en contenedores.

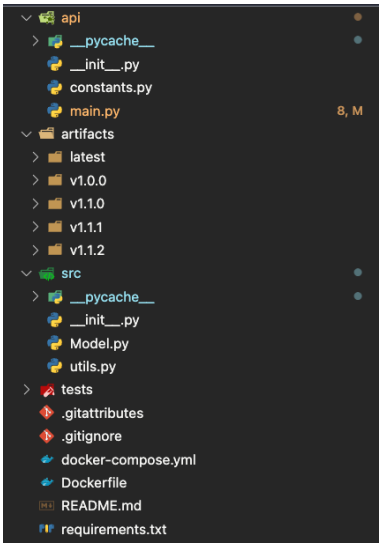

Figura 1: Estructura de directorios creada

2.4.1. Endpoints y carga de modelo a la carta

La API se ha estructurado para ofrecer flexibilidad en la selección de la versión del modelo de XGBoost a través del endpoint ***/version/predict***, donde los usuarios pueden especificar la versión deseada. Para mayor conveniencia, el endpoint ***/latest/predict*** utiliza automáticamente la versión más reciente del modelo. Estos endpoints aseguran que la API se adapte a diversas necesidades y permanezca actualizada, facilitando a los usuarios el acceso a las últimas mejoras.

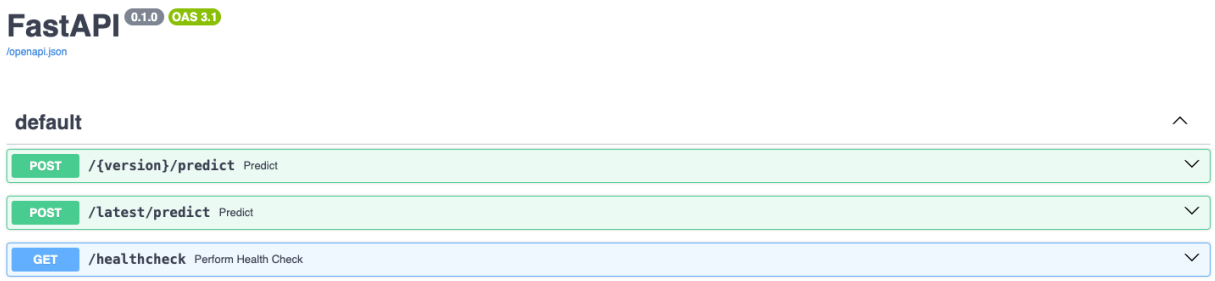

Figura 2: Endpoints creados

2.4.2. Infraestructura cargada en HuggingFace

La infraestructura del proyecto se ha aprovechado de la plataforma Hugging Face, conocida por su robusta comunidad y repositorios centrados en Machine Learning, para alojar y gestionar modelos. https://huggingface.co/spaces/Daniirxch/AIIA

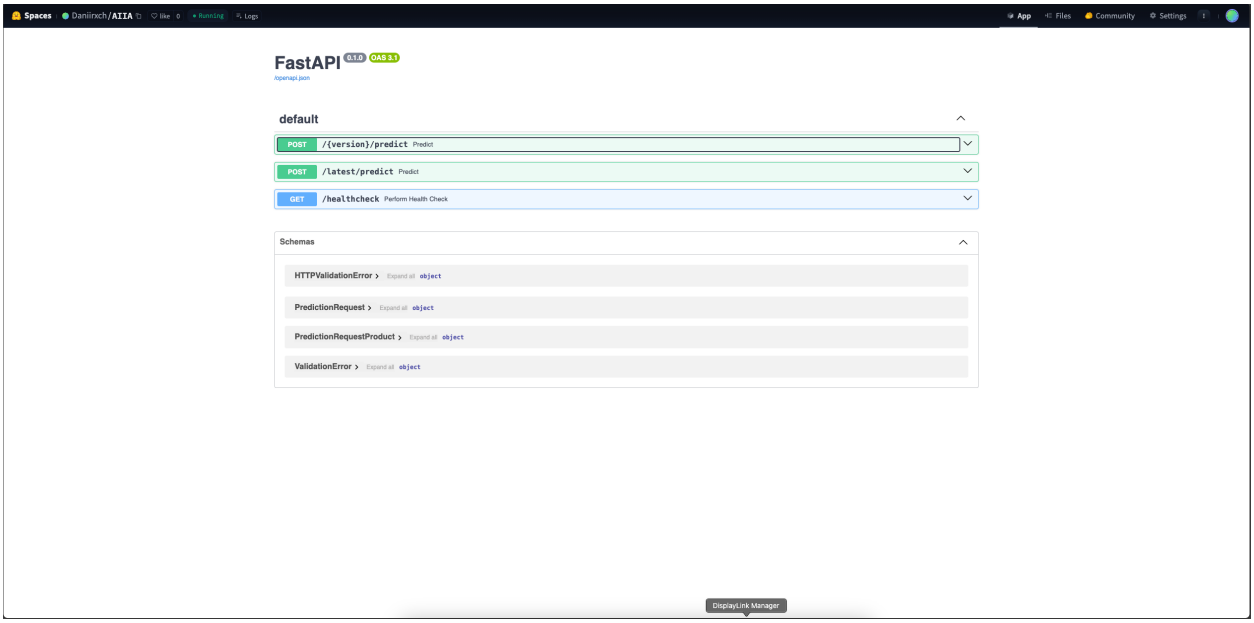

Figura 3: Endpoints creados

2.5. Dockers

La implementación de Docker en el proyecto proporciona un entorno consistente y aislado para la ejecución de la API de XGBoost, asegurando que la aplicación sea portátil y fácil de desplegar en cualquier sistema que soporte Docker.

2.5.1. Dockerfile

El Dockerfile detalla los pasos para crear una imagen Docker para la aplicación. Utiliza python:3.9.6-slim como imagen base para minimizar el tamaño de la imagen final. El direc-

torio de trabajo se establece en /app, donde el código de la aplicación y las dependencias se copian y se instalan. Se instalan paquetes esenciales y se limpia el cache de apt para mantener el tamaño de la imagen bajo. Las variables de entorno DEFAULT_CONVERSION_RATE y DEFAULT_SLOPE_CORRECTION se establecen para su uso en la aplicación. El comando final utiliza uvicorn para servir la aplicación FastAPI, exponiendo la API en el puerto 7860.

```
FROM --platform=linux/amd64 python:3.9.6-slim

WORKDIR /app

COPY ./requirements.txt /app/requirements.txt

RUN apt-get update && \
    apt-get -y upgrade && \
    apt-get -y autoremove && \
    apt-get install -y \
        build-essential \
        libpq-dev && \
    apt-get clean && rm -rf /var/lib/apt/lists/* && \
    pip install --no-cache-dir --upgrade -r /app/requirements.txt

COPY ./src /app/src
COPY ./api /app/api
COPY ./artifacts /app/artifacts

ENV DEFAULT_CONVERSION_RATE=270
ENV DEFAULT_SLOPE_CORRECTION=1.13

CMD ["uvicorn", "api.main:app", "--host", "0.0.0.0", "--port", "7860"]
```

2.5.2. Docker-Compose

El archivo docker-compose.yml define la configuración para desplegar la aplicación como un servicio, volumetricXGBoost, utilizando la imagen construida a partir del Dockerfile. Configura las mismas variables de entorno que el Dockerfile y establece límites de recursos para la CPU y la memoria, asegurando que el contenedor no consuma recursos excesivos del host. Los puertos se mapean para que la API sea accesible fuera del contenedor en el puerto 7860. La sección de dispositivos está comentada para permitir la utilización opcional de la GPU en caso de disponibilidad.

```
version: '3.8'
services:
  volumetricXGBoost:
    image: xgboost-volumetricweight:latest
```

```
build: .
environment:
  DEFAULT_CONVERSION_RATE: 270
  DEFAULT_SLOPE_CORRECTION: 1.13
deploy:
  resources:
    limits:
      cpus: '0.50' # Limita a 50% de la CPU
      memory: 4G   # Limita a 4 GB de RAM
devices:
  - "/dev/nvidia0:/dev/nvidia0"
ports:
  - "7860:7860"
command: ["uvicorn", "api.main:app", "--host", "0.0.0.0", "--port", "7860"]
```

3. Resultados

Los resultados de la práctica se presentan en términos del despliegue y la funcionalidad operativa de la API de predicción de peso volumétrico, tanto en un entorno local como en la plataforma Hugging Face.

3.1. Despliegue en Local

El despliegue local de la API se realizó utilizando Docker, lo que permitió una configuración de entorno reproducible y aislada que es fácilmente escalable. La construcción de la imagen Docker y la orquestación con Docker Compose facilitaron la puesta en marcha de la API sin inconvenientes, probando así la eficacia de la implementación del Dockerfile y el docker-compose.yml. Los tests locales confirmaron que la API respondía correctamente a las solicitudes, y la funcionalidad de los endpoints para diferentes versiones de modelos de XGBoost funcionó según lo esperado. El despliegue local sirvió como una validación importante de la aplicación antes del despliegue en un entorno de producción más amplio.

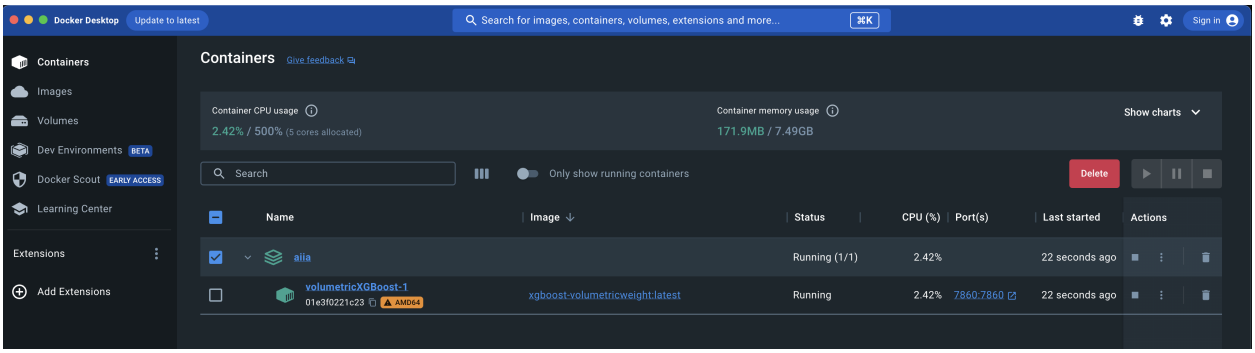

Figura 4: Despliegue en Docker

```
curl -X 'POST' \
  'http://127.0.0.1:7860/latest/predict' \
  -H 'accept: application/json' \
  -H 'x-api-key: 1234567890' \
  -H 'Content-Type: application/json' \
  -d '{"order": "Order" ,"data": [ {"sku": "030820","quantity": 2, "product_width": 140, "product_height": 10, "product_length": 40, "produ
ct_weight": 0.154 }, {"sku": "032452","quantity": 10,"product_width": 81.9,"product_height": 120,"product_length": 96.25,"product_weight":
0.37}]}'
{"prediction":0.023986880781249997,"pesoVolumetricoEnKg":8.88402991898148e-05,"pesoTotalProductos":4.008,"isOrder":true}
```

Figura 5: Ejemplo de Curl

3.2. Despliegue en Hugging Face

El despliegue en Hugging Face extendió la accesibilidad de la API, aprovechando la robustez y el alcance de la plataforma para compartir y colaborar. Al alojar la API en Hugging Face, se demostró que la infraestructura podía manejar solicitudes de predicción de manera eficiente, con la escalabilidad necesaria para soportar un número creciente de peticiones. La plataforma no solo proporcionó un medio para que los usuarios accedan y utilicen el modelo, sino que también ofreció herramientas para el seguimiento de versiones y el manejo simplificado de los modelos de aprendizaje automático. El despliegue en Hugging Face validó el rendimiento del modelo en un entorno de nube y destacó la viabilidad de la solución para aplicaciones en tiempo real. Como se muestra en el apartado 2.4.2 y en la imagen 6

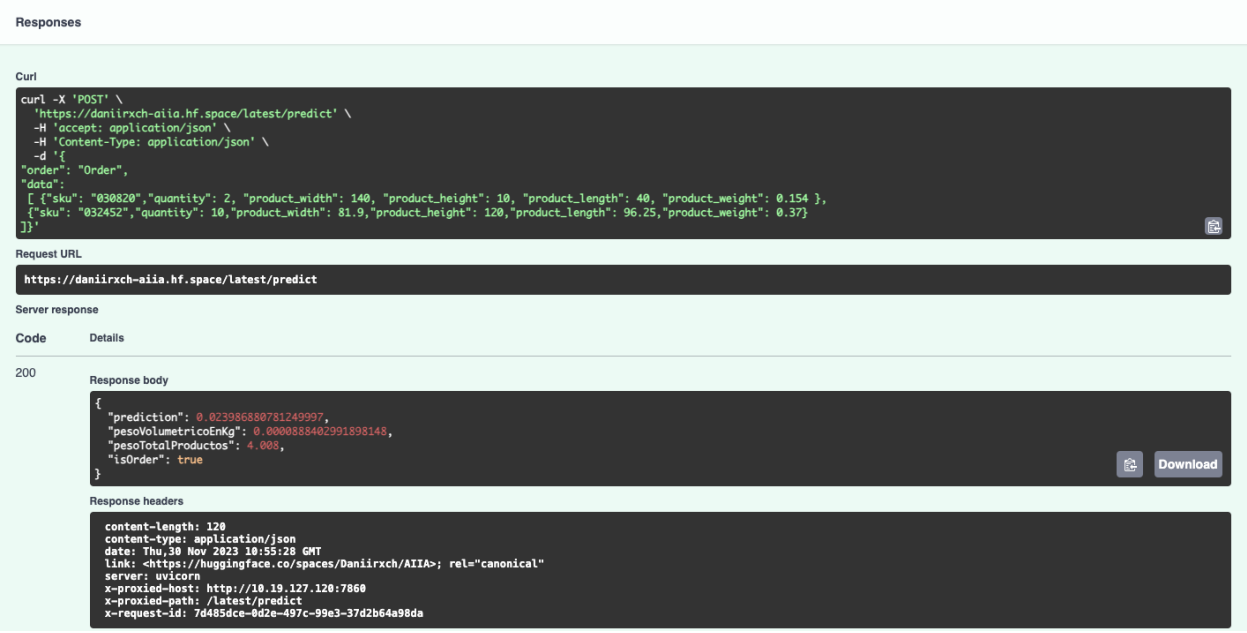

Figura 6: Ejemplo de Curl en HuggingFace