



PRÁCTICA OBLIGATORIA: PRODUCTIVIZACIÓN DEL MODELO APRENDIZAJE AUTOMÁTICO

Nuria Olmedilla, Lucia Poyán, Claudia Gemenó

Enero 2025

Tabla de Contenidos

Introducción.....	3
Archivos Necesarios para Configurar el Entorno con Docker	3
Pasos para Construir y Ejecutar una API Flask en Docker.....	6
Descripción y Funcionalidad del Archivo app.py para la API Flask.....	8
Proceso para Verificar y Ejecutar la API Flask en Docker.....	11
Respuesta de la API y Registro de Predicciones	14

Introducción

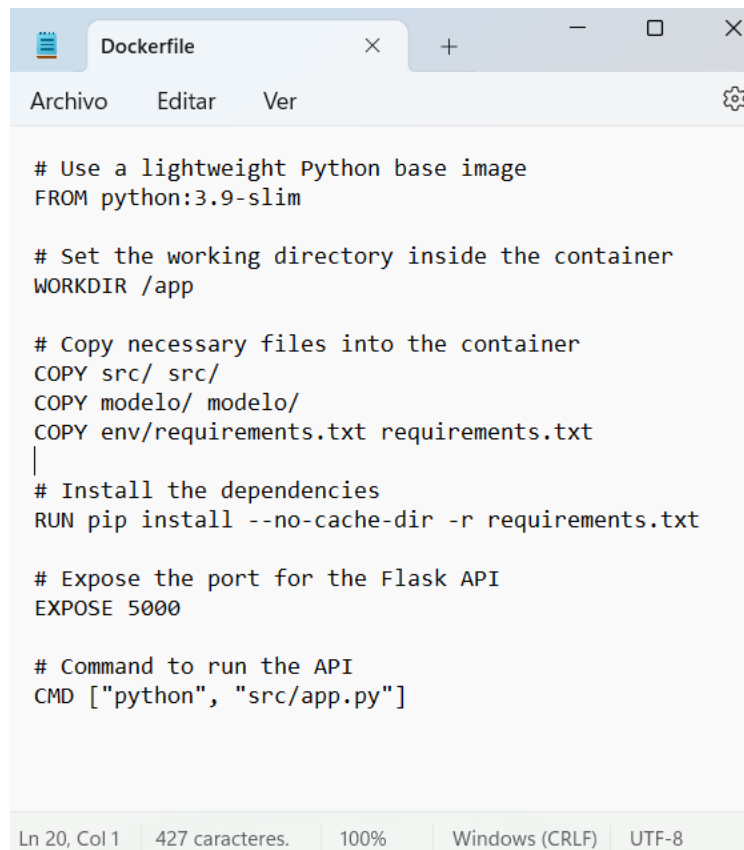
En esta práctica, el objetivo es simular la puesta en producción de un modelo de machine learning mediante la creación de una API usando Flask, y la utilización de Docker para gestionar el entorno de ejecución. Esto permitirá invocar el modelo previamente entrenado y recibir predicciones a través de solicitudes realizadas a la API. Además, se almacenarán las solicitudes realizadas a la API y las predicciones generadas en un archivo de registro, con el fin de generar un dashboard simple para el seguimiento de las predicciones y el análisis de los resultados.

Los pasos de esta práctica incluyen la configuración de un entorno de Docker, la creación de una API con Flask para interactuar con el modelo, y la implementación de un dashboard para monitorear el desempeño de la API a lo largo del tiempo. Este proyecto se llevará a cabo en un entorno Python, utilizando herramientas como Flask, Docker y librerías de visualización para generar el dashboard de seguimiento.

Archivos Necesarios para Configurar el Entorno con Docker

1. Archivo Dockerfile

El Dockerfile es un archivo sin extensión que define los pasos para construir una imagen Docker personalizada. A continuación, se detalla su contenido y propósito:

A screenshot of a code editor window titled 'Dockerfile'. The editor has a menu bar with 'Archivo', 'Editar', and 'Ver', and a settings icon. The code content is as follows:

```
# Use a lightweight Python base image
FROM python:3.9-slim

# Set the working directory inside the container
WORKDIR /app

# Copy necessary files into the container
COPY src/ src/
COPY modelo/ modelo/
COPY env/requirements.txt requirements.txt
|
# Install the dependencies
RUN pip install --no-cache-dir -r requirements.txt

# Expose the port for the Flask API
EXPOSE 5000

# Command to run the API
CMD ["python", "src/app.py"]
```

The status bar at the bottom shows 'Ln 20, Col 1', '427 caracteres.', '100%', 'Windows (CRLF)', and 'UTF-8'.

El archivo **Dockerfile** es un componente clave para la creación de una imagen Docker personalizada. En este caso, hemos utilizado una imagen base ligera de Python 3.9 para garantizar un entorno optimizado y reducir el tamaño del contenedor. Dentro de esta configuración, se establece un directorio de trabajo (**/app**) en el contenedor, donde se alojarán todos los archivos necesarios para la ejecución de la aplicación.

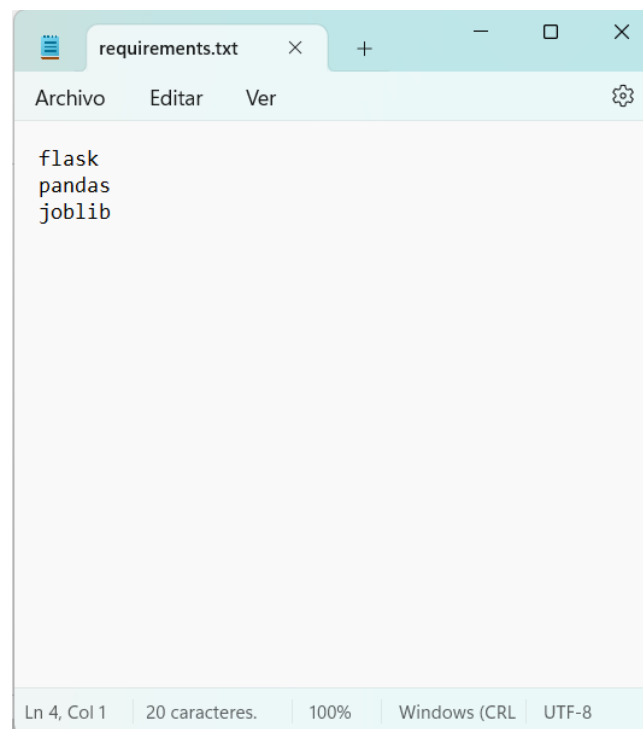
Los archivos esenciales, como el código fuente, el modelo entrenado y las dependencias, se copian al contenedor desde el entorno local. Para gestionar las dependencias, se utiliza un archivo **requirements.txt**, que contiene las librerías necesarias, como Flask, pandas y joblib. Estas dependencias se instalan de forma eficiente con el comando **pip install**, evitando el almacenamiento en caché para optimizar la instalación.

El archivo también expone el puerto 5000, necesario para que la API Flask pueda ser accesible desde fuera del contenedor. Finalmente, se define el comando que ejecutará la aplicación, garantizando que el servidor Flask inicie automáticamente cuando el contenedor se ponga en marcha.

Esta configuración asegura que la aplicación sea portable y fácil de desplegar en cualquier sistema que soporte Docker, eliminando dependencias específicas del entorno local.

2. Archivo requirements.txt

El archivo requirements.txt contiene las dependencias necesarias para que la aplicación Flask funcione. Este archivo es fundamental para que Docker instale las bibliotecas requeridas. Su contenido es el siguiente:

A screenshot of a text editor window titled 'requirements.txt'. The window has a menu bar with 'Archivo', 'Editar', and 'Ver', and a settings icon. The main text area contains three lines of code: 'flask', 'pandas', and 'joblib'. The status bar at the bottom shows 'Ln 4, Col 1', '20 caracteres.', '100%', 'Windows (CRLF)', and 'UTF-8'.

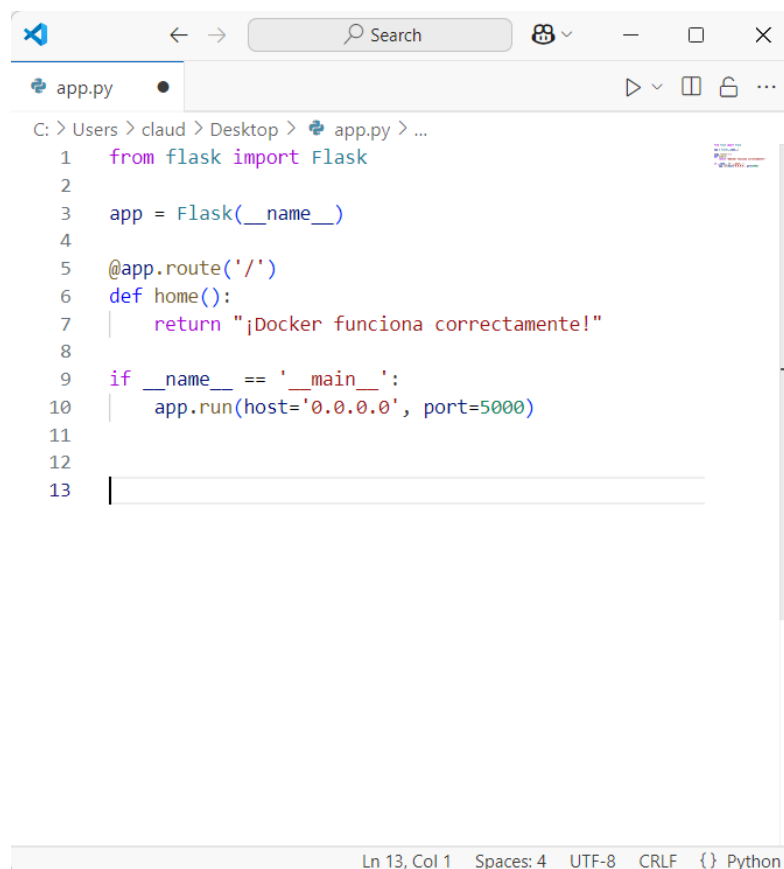
```
flask
pandas
joblib
```

En este caso, el archivo contiene tres librerías clave:

- **flask:** El framework utilizado para desarrollar la API web. Flask permite crear rutas y manejar peticiones HTTP de manera sencilla y rápida, lo que es esencial para exponer nuestro modelo como un servicio web.
- **pandas:** Una biblioteca ampliamente utilizada para la manipulación y análisis de datos. En este proyecto, se utiliza para procesar los datos recibidos a través de la API y preparar la entrada para el modelo de predicción.
- **joblib:** Librería utilizada para cargar el modelo de Machine Learning previamente entrenado y almacenado en formato **.pkl**. Joblib es eficiente en la serialización de objetos Python grandes, como modelos complejos.

3. Archivo app.py

El archivo app.py es el punto de entrada de la aplicación Flask. Para la configuración inicial, usamos un código de prueba básico para verificar que Docker y Flask están funcionando correctamente.



```
1  from flask import Flask
2
3  app = Flask(__name__)
4
5  @app.route('/')
6  def home():
7      return "¡Docker funciona correctamente!"
8
9  if __name__ == '__main__':
10     app.run(host='0.0.0.0', port=5000)
11
12
13
```

El archivo app.py es el corazón de la aplicación Flask y es responsable de definir las rutas y la lógica de la API. En su versión inicial, el archivo se utiliza principalmente para verificar

que el contenedor Docker está funcionando correctamente y que Flask está configurado adecuadamente.

Al iniciar la aplicación, Flask ejecuta el método **home()** cuando se hace una solicitud GET a la raíz (/). Este método responde con un mensaje simple que indica que la configuración básica de Docker y Flask es correcta. El servidor está configurado para escuchar en todas las interfaces de red (**host='0.0.0.0'**), lo que asegura que el contenedor pueda aceptar solicitudes de cualquier origen, y está ejecutándose en el puerto 5000, que es el estándar para aplicaciones Flask en contenedores Docker.

Aunque en esta etapa el código solo valida el entorno de ejecución, el archivo se actualizará más adelante para manejar las solicitudes de predicción de modelos de Machine Learning, utilizando el archivo **.pkl** cargado previamente. Este archivo se convierte en la puerta de entrada para que los usuarios interactúen con el modelo a través de la API RESTful.

Pasos para Construir y Ejecutar una API Flask en Docker

1. Abrir la terminal y navegar a la raíz del proyecto

El primer paso consiste en abrir la terminal y acceder al directorio donde se encuentran los archivos del proyecto. Para ello, utilizamos el comando `cd` en la terminal. En nuestro caso, la ruta del proyecto es:

```
claud@c1o MINGW64 /  
$ cd "C:/DATOS/DATOS/CUNEF/MASTER/APRENDIZAJE AUTOMATICO/Practica productivizaci  
on del modelo"
```

Este comando cambia el directorio de trabajo de la terminal al directorio raíz del proyecto, que contiene todos los archivos necesarios, como el `Dockerfile`, el archivo `requirements.txt` y el archivo `app.py`.

2. Construir la imagen Docker

Una vez que estamos en la raíz del proyecto, el siguiente paso es construir la imagen Docker a partir del `Dockerfile`. Para ello, usamos el siguiente comando:

```

claud@clo MINGW64 /c/DATOS/DATOS/CUNEF/MASTER/APRENDIZAJE AUTOMATICO/Practica pro
ductivizacion del modelo
$ docker build -t flask-app .
[+] Building 0.6s (11/11) FINISHED                                docker:desktop-linux
=> [internal] load build definition from Dockerfile                0.0s
=> => transferring dockerfile: 481B                                0.0s
=> [internal] load metadata for docker.io/library/python:3.9-slim 0.5s
=> [internal] load .dockerignore                                  0.0s
=> => transferring context: 2B                                       0.0s
=> [internal] load build context                                  0.0s
=> => transferring context: 204B                                     0.0s
=> [1/6] FROM docker.io/library/python:3.9-slim@sha256:caaf1af9e23adc6149 0.0s
=> CACHED [2/6] WORKDIR /app                                       0.0s
=> CACHED [3/6] COPY src/ src/                                       0.0s
=> CACHED [4/6] COPY modelo/ modelo/                               0.0s
=> CACHED [5/6] COPY requirements.txt requirements.txt             0.0s
=> CACHED [6/6] RUN pip install --no-cache-dir -r requirements.txt 0.0s
=> exporting to image                                              0.0s
=> => exporting layers                                              0.0s
=> => writing image sha256:48e05d824012edf139f1a694b692c4a9d012bea7d961c0 0.0s
=> => naming to docker.io/library/flask-app                        0.0s

View build details: docker-desktop://dashboard/build/desktop-linux/desktop-linux/
vskfsyc6vhukysxewmo7a0jbt

```

Este comando le indica a Docker que construya una imagen a partir del Dockerfile presente en el directorio actual (el punto . al final se refiere al directorio en el que nos encontramos).

El parámetro **-t flask-app** asigna un nombre a la imagen generada, en este caso **flask-app**. Al ejecutar este comando, Docker sigue una serie de pasos: descarga la imagen base de Python, establece el directorio de trabajo dentro del contenedor, copia los archivos necesarios (**como src/, modelo/ y requirements.txt**), instala las dependencias especificadas en el archivo requirements.txt, y finalmente expone el puerto 5000, que es el puerto donde la API de Flask escuchará las solicitudes entrantes.

3. Ejecutar el contenedor Docker

Después de haber construido la imagen, el siguiente paso es ejecutar el contenedor Docker. Para ello, utilizamos el siguiente comando:

```

claud@clo MINGW64 /c/DATOS/DATOS/CUNEF/MASTER/APRENDIZAJE AUTOMATICO/Practica pro
ductivizacion del modelo
$ docker run -p 5000:5000 flask-app
* Serving Flask app 'app'
* Debug mode: off
WARNING: This is a development server. Do not use it in a production deployment.
Use a production WSGI server instead.
* Running on all addresses (0.0.0.0)
* Running on http://127.0.0.1:5000
* Running on http://172.17.0.2:5000
Press CTRL+C to quit

```

Este comando inicia el contenedor utilizando la imagen **flask-app** que acabamos de crear. El parámetro **-p 5000:5000** vincula el puerto 5000 del contenedor al puerto 5000 de la máquina local, lo que permite que la API Flask, que está escuchando en el puerto 5000 dentro del contenedor, sea accesible desde el navegador en el mismo puerto en la máquina local.

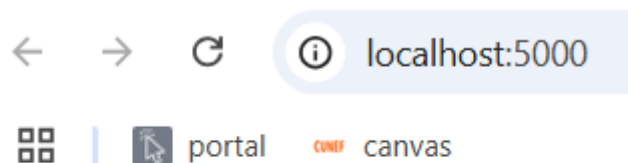
Una vez que ejecutamos este comando, Docker inicia el contenedor y Flask comenzará a funcionar en el puerto 5000.

4. Verificar que la API está corriendo

Con el contenedor en ejecución, es momento de verificar que la API Flask está funcionando correctamente. Para ello, abrimos un navegador web y accedemos a la siguiente URL:

<http://localhost:5000>

Si todo está configurado correctamente, veremos el siguiente mensaje:



¡Docker funciona correctamente!

Este mensaje es el que definimos en el archivo `app.py` como respuesta a las solicitudes GET a la ruta raíz (/). Si vemos este mensaje, podemos confirmar que el contenedor y la API están funcionando correctamente.

Descripción y Funcionalidad del Archivo `app.py` para la API Flask

El archivo `app.py` es el archivo principal de una aplicación web creada con Flask, un framework ligero de Python utilizado para desarrollar aplicaciones web. Este archivo define la lógica de la aplicación y maneja las rutas que la API seguirá para recibir y responder a las solicitudes de los usuarios. En particular, se encarga de recibir un archivo, procesarlo utilizando un modelo de machine learning previamente entrenado (en este caso un archivo `glm_optimized.pkl`), hacer predicciones con el modelo, y devolver los resultados de manera estructurada al usuario.


```

app.py
C: > DATOS > DATOS > CUNEF > MASTER > APRENDIZAJE AUTOMATICO > Practica productivización del modelo > src >
1  from flask import Flask, request, jsonify
2  import pandas as pd
3  import joblib
4  import csv
5  from datetime import datetime
6
7  # Load the trained model
8  model = joblib.load("../modelo/glm_optimized.pkl")
9
10 # Create the Flask application
11 app = Flask(__name__)
12
13 # Route to receive the file and make the prediction
14 @app.route('/predict', methods=['POST'])
15 def predict():
16     try:
17         # Check if the file was sent in the request
18         if 'file' not in request.files:
19             return jsonify({'error': 'No file part'}), 400
20
21         file = request.files['file']
22
23         # Check if a file was selected
24         if file.filename == '':
25             return jsonify({'error': 'No selected file'}), 400
26
27         # Read the CSV file (or any other type of file you're using)
28         df = pd.read_csv(file)
29
30         # Make the prediction with the loaded model
31         prediction = model.predict(df)
32
33         # Save the request and prediction in a CSV file
34         with open('predictions_log.csv', mode='a', newline='') as file_log:
35             writer = csv.writer(file_log)
36             # Save the date, the received file, and the prediction
37             writer.writerow([datetime.now(), file.filename, prediction.tolist()])
38
39         # Return the prediction as a response
40         return jsonify({'prediction': prediction.tolist()})
41
42     except Exception as e:
43         return jsonify({'error': str(e)}), 400
44
45 # Start the application
46 if __name__ == '__main__':
47     app.run(debug=True, host='0.0.0.0', port=5000)
48

```

En el archivo app.py, se define una aplicación web utilizando el marco de trabajo Flask, que tiene como objetivo recibir un archivo de entrada, realizar una predicción utilizando un modelo previamente entrenado y devolver la predicción al usuario.

En primer lugar, se importa **Flask**, junto con algunas bibliotecas necesarias como **request**, **jsonify**, **pandas**, **joblib**, **csv** y **datetime**. Estas bibliotecas permiten gestionar las solicitudes HTTP, cargar el modelo entrenado, manipular los datos de entrada y registrar los resultados.

El modelo previamente entrenado (almacenado en formato **.pkl**) se carga al inicio mediante la función **joblib.load()**. Este modelo, en este caso denominado **glm_optimized.pkl**, es el que se usará para hacer predicciones con los datos proporcionados en la solicitud.

La aplicación se crea utilizando **Flask(__name__)**, lo que inicia la instancia de la aplicación web. Posteriormente, se define una ruta **/predict** que recibirá solicitudes HTTP de tipo POST. Esta ruta es la que permite a los usuarios enviar un archivo CSV y obtener una predicción como respuesta.

Dentro de la función **predict()**, se comprueba si el archivo se ha enviado correctamente. Si no es así, se devuelve un mensaje de error con un código de estado 400. Si el archivo se ha recibido, se procede a verificar que efectivamente haya sido seleccionado un archivo válido.

Una vez validado el archivo, se lee el contenido del archivo CSV utilizando **pd.read_csv()**, lo que convierte el archivo en un DataFrame de pandas. Luego, el modelo entrenado realiza la predicción sobre los datos mediante el método **model.predict()**. Esta predicción se almacena en la variable **prediction**.

El siguiente paso es guardar tanto la solicitud como la predicción en un archivo de log (**predictions_log.csv**). Esto se hace para tener un registro de las predicciones realizadas, junto con la fecha y el nombre del archivo recibido.

Finalmente, la función devuelve la predicción al usuario en formato JSON utilizando **jsonify()**, lo que permite enviar el resultado de forma estructurada y fácil de procesar. Si se produce algún error durante el proceso, el bloque **except** captura la excepción y devuelve un mensaje de error con su descripción.

El archivo termina con la instrucción **app.run()**, que ejecuta la aplicación Flask en el puerto 5000, permitiendo que sea accesible a través de la red local para realizar las solicitudes POST a la ruta **/predict**. La opción **debug=True** permite que se muestren mensajes detallados de error durante el desarrollo.

Proceso para Verificar y Ejecutar la API Flask en Docker

Una vez que hemos configurado todo y estamos listos para interactuar con la API, los siguientes pasos implican asegurarnos de que el contenedor Docker está activo y funcionando correctamente antes de proceder con el envío de solicitudes.




1. **Navegar hasta la ruta específica donde está el archivo app.py:** Debemos asegurarnos de estar en la ruta correcta dentro de nuestra máquina local donde se encuentra el archivo app.py. Para ello, utilizamos la terminal para navegar a la carpeta correspondiente.

```
claud@c1o MINGW64 /  
$ cd "C:\DATOS\DATOS\CUNEF\MASTER\APRENDIZAJE AUTOMATICO\Practica productivización del modelo\src"  
claud@c1o MINGW64 /c/DATOS/DATOS/CUNEF/MASTER/APRENDIZAJE AUTOMATICO/Practica productivización del modelo/src
```

2. **Activar el contenedor Docker:** El primer paso es asegurarnos de que el contenedor Docker que hemos creado previamente está activo y en funcionamiento. Para ello, debemos ejecutar el comando **docker ps** en la terminal. Este comando nos muestra una lista de los contenedores que están corriendo actualmente. Al ejecutarlo, buscamos en la salida el contenedor que corresponde a nuestra aplicación Flask. Si todo está correcto, veremos que el contenedor está activo y escuchando en el puerto 5000, lo que indica que la API está lista para recibir solicitudes.

```
claud@c1o MINGW64 /c/DATOS/DATOS/CUNEF/MASTER/APRENDIZAJE AUTOMATICO/Practica productivización del modelo/src  
$ docker ps  
CONTAINER ID   IMAGE          COMMAND                  CREATED        STATUS        PORTS  
9d2f97a06d5d   flask-app     "python src/app.py"     5 days ago    Up 2 hours    0.0.0.0:5000->5000/tcp
```

3. **Verificar que el contenedor está funcionando:** Una vez que confirmamos que el contenedor está activo con `docker ps`, necesitamos asegurarnos de que nuestra aplicación Flask está funcionando correctamente. Si no vemos el contenedor activo o no está escuchando en el puerto adecuado, puede ser necesario revisar los logs del contenedor para verificar si hubo algún problema durante el inicio. Sin embargo, si todo está en orden, ya podemos proceder al siguiente paso.

 9d2f97a06d5d 
[5000:5000](#) 

 [flask-app:latest](#)

STATUS
Running

4. **Ejecutar el archivo app.py:** Con el contenedor funcionando, el siguiente paso es ejecutar el archivo app.py. Este archivo contiene el código de la API Flask y, cuando lo ejecutamos, arranca la aplicación. Normalmente, esto se hace dentro del contenedor, pero en algunos casos, si estamos trabajando fuera del contenedor, podemos ejecutar directamente el archivo usando el comando python app.py desde la terminal. Este comando inicia el servidor Flask, que comenzará a escuchar solicitudes en el puerto 5000.

```
claud@clo MINGW64 /c/DATOS/DATOS/CUNEF/MASTER/APRENDIZAJE AUTOMATICO/Practica pr
ductivización del modelo/src
$ python app.py
C:\Users\claud\AppData\Local\Programs\Python\Python312\Lib\site-packages\sklearn
\base.py:380: InconsistentVersionWarning: Trying to unpickle estimator LogisticR
egression from version 1.3.0 when using version 1.6.0. This might lead to breaki
ng code or invalid results. Use at your own risk. For more info please refer to:
https://scikit-learn.org/stable/model_persistence.html#security-maintainability-
limitations
  warnings.warn(
* Serving Flask app 'app'
* Debug mode: on
WARNING: This is a development server. Do not use it in a production deployment.
Use a production WSGI server instead.
* Running on all addresses (0.0.0.0)
* Running on http://127.0.0.1:5000
* Running on http://192.168.1.46:5000
Press CTRL+C to quit
* Restarting with stat
C:\Users\claud\AppData\Local\Programs\Python\Python312\Lib\site-packages\sklearn\bas
e.py:380: InconsistentVersionWarning: Trying to unpickle estimator LogisticRegression
n from version 1.3.0 when using version 1.6.0. This might lead to breaking code or i
nvalid results. Use at your own risk. For more info please refer to:https://scikit-l
earn.org/stable/model_persistence.html#security-maintainability-limitations
  warnings.warn(
* Debugger is active!
* Debugger PIN: 100-711-841
C:\Users\claud\AppData\Local\Programs\Python\Python312\Lib\site-packages\sklearn\uti
ls\validation.py:2732: UserWarning: X has feature names, but LogisticRegression was
fitted without feature names
  warnings.warn(
127.0.0.1 - - [07/Jan/2025 19:54:19] "POST /predict HTTP/1.1" 400 -
C:\Users\claud\AppData\Local\Programs\Python\Python312\Lib\site-packages\sklearn\uti
ls\validation.py:2732: UserWarning: X has feature names, but LogisticRegression was
fitted without feature names
  warnings.warn(
127.0.0.1 - - [07/Jan/2025 19:55:04] "POST /predict HTTP/1.1" 200 -
```

5. **Confirmar que el servidor está en ejecución:** Al ejecutar el archivo app.py, el servidor Flask debe iniciar correctamente y comenzar a escuchar en el puerto 5000, como se indica en el código. Si todo ha salido bien, podremos ver un mensaje similar a *“Running on http://127.0.0.1:5000/ (Press CTRL+C to quit)”*, lo que confirma que la API está funcionando correctamente y está lista para recibir solicitudes.

Envío de Solicitudes POST a la API Flask para Realizar Predicciones

Una vez que la API Flask está en funcionamiento, el siguiente paso es enviar una solicitud POST a la API para realizar predicciones utilizando el modelo entrenado. Para esto, usamos el comando curl en la terminal. Este comando permite enviar el archivo que contiene los datos a predecir a la API y obtener la respuesta con las predicciones generadas por el modelo.

A continuación, se detalla el comando **curl** utilizado para enviar la solicitud de predicción:

```
claud@clo MINGW64 /
$ curl -X POST -F "file=@C:\DATOS\DATOS\CUNEF\MASTER\APRENDIZAJE AUTOMATICO\GITH
UB\proyecto_final_aprendizaje_automatico\notebook\X_train_pca.csv" http://127.0.
0.1:5000/predict
```

El comando curl que utilizamos tiene varias partes importantes.

Primero, se especifica la opción **-X POST**, que indica que estamos enviando una solicitud de tipo POST. Esto es crucial porque en una solicitud POST es donde se envían los datos al servidor, en este caso, el archivo que contiene las características sobre las que se realizará la predicción.

Luego, con el comando **-F "file=@C:/DATOS/DATOS/CUNEF/MASTER/APRENDIZAJE AUTOMATICO/GITHUB/proyecto_final_aprendizaje_automatico/notebook/X_test_pca.csv"**, indicamos el archivo que vamos a enviar. Este archivo, **X_test_pca.csv**, contiene un conjunto de datos preprocesados que sirven como entrada para el modelo. Este archivo corresponde al conjunto de prueba que se utilizó durante el proceso de evaluación del modelo.

El archivo **X_test_pca.csv** es relevante porque contiene las características necesarias para que el modelo cargado pueda generar predicciones, es decir, las variables que el modelo utilizará para determinar las predicciones de la variable objetivo. Es importante destacar que este archivo no contiene la variable objetivo, ya que el propósito es solo hacer predicciones a partir de las características de los datos.

Por último, la URL **http://127.0.0.1:5000/predict** es el punto de acceso a nuestra API Flask. El **127.0.0.1** hace referencia a la máquina local (localhost), y el puerto **5000** es donde la API está escuchando las solicitudes. La ruta **/predict** es la que hemos definido en el archivo **app.py** para recibir los datos y realizar las predicciones.

Respuesta de la API y Registro de Predicciones


Una vez que se ha enviado el archivo correctamente a la API y se han realizado las predicciones, la respuesta obtenida de la API es un conjunto de valores que corresponden a las predicciones generadas por el modelo. En este caso, los resultados obtenidos son valores booleanos, es decir, True o False. Estos valores hacen referencia a las predicciones del modelo para cada una de las instancias en el conjunto de datos.

El modelo que hemos utilizado para generar las predicciones devuelve valores binarios que están relacionados con la variable objetivo. En nuestro caso, estos valores corresponden a dos posibles estados:

- **True:** Indica que el cliente ha tenido impagos o problemas financieros.
- **False:** Indica que el cliente ha cumplido con sus pagos.

```
    true,  
    true,  
    true,  
    false,  
    true,  
    false,  
    true,  
    false,  
    true,  
    true,  
    true,  
    false,  
    false,  
    true,  
    true,  
    true,  
    true,  
    false,  
    false,  
    true,  
    true,  
    true,  
    true  
  ]  
}
```

Además de devolver las predicciones como respuesta, el proceso también incluye el registro de estas predicciones en un archivo CSV denominado predictions_log.csv.

 predictions_log.csv

Este archivo se crea y se actualiza automáticamente cada vez que se hace una solicitud a la API. En él se guarda información clave sobre la solicitud, como la fecha y hora en que se realizó, el nombre del archivo recibido, y las predicciones generadas por el modelo. De esta forma, podemos llevar un registro detallado de todas las predicciones realizadas por la API y consultar cualquier predicción pasada si es necesario.