ChatGPT

Integración de Machine Learning con Contrato Inteligente en Ethereum

Este proyecto demuestra cómo combinar un modelo de *Machine Learning* entrenado en Python con un contrato inteligente en Ethereum (Solidity). Se utiliza un modelo de clasificación binaria fuera de la cadena (*off-chain*) para realizar predicciones, y luego se envía el resultado al contrato inteligente en la cadena de bloques para tomar acciones registradas en la blockchain.

Estructura del Proyecto

La estructura del proyecto está organizada en directorios para separar la lógica de *machine learning*, el backend, el contrato inteligente y las pruebas:

```
ml-smart-contract-project/
├── ml/
│   ├── train_model.py        # Script de entrenamiento del modelo ML
│   └── model.pkl             # Modelo entrenado guardado en formato pickle
├── backend/
│   ├── app.py                # Aplicación FastAPI (endpoints /predict y /
trigger)
│   └── requirements.txt      # Lista de dependencias de Python
├── contract/
│   ├── MLContract.sol        # Contrato inteligente en Solidity
│   └── deploy.py             # Script de despliegue del contrato inteligente
├── test/
│   └── test_workflow.py      # (Opcional) Script de prueba de flujo end-to-
end
└── README.md                 # Instrucciones y documentación del proyecto
```

A continuación, se detalla cada componente del proyecto con código de ejemplo y explicaciones.

1. Entrenamiento del Modelo de ML (ml/train_model.py)

En esta parte entrenamos un modelo de clasificación binaria simple usando un dataset de scikit-learn. Como ejemplo, utilizaremos el dataset de cáncer de mama (breast cancer) de scikit-learn, que es un problema de clasificación binaria (tumor maligno vs benigno). Entrenaremos un modelo de **Regresión Logística** y lo guardaremos en disco como `model.pkl` usando `joblib`.

El script de entrenamiento (`ml/train_model.py`) realiza las siguientes tareas:

- Carga el dataset de ejemplo (características y etiquetas).
- Divide los datos en entrenamiento y prueba (opcionalmente, para validar).
- Entrena un modelo de clasificación (regresión logística en este caso) con los datos de entrenamiento.

- Guarda el modelo entrenado en formato pickle (`model.pkl`) para uso posterior en el backend.

Código: ml/train_model.py

```
import joblib
from sklearn.datasets import load_breast_cancer
from sklearn.model_selection import train_test_split
from sklearn.linear_model import LogisticRegression

# Cargar el dataset de cáncer de mama de scikit-learn
data = load_breast_cancer()
X = data.data
y = data.target  # 0 = maligno, 1 = benigno (o viceversa según el dataset)

# (Opcional) Dividir en conjunto de entrenamiento y prueba para evaluar
desempeño
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2,
random_state=42)

# Entrenar un modelo de Regresión Logística (clasificación binaria)
model = LogisticRegression(max_iter=10000)  # max_iter alto para asegurar
convergencia
model.fit(X_train, y_train)

# (Opcional) Imprimir exactitud en conjunto de prueba
accuracy = model.score(X_test, y_test)
print(f"Exactitud del modelo en datos de prueba: {accuracy:.2f}")

# Guardar el modelo entrenado a disco
joblib.dump(model, "ml/model.pkl")
print("Modelo entrenado guardado en ml/model.pkl")
```

Explicación: Este script entrena una regresión logística sobre el dataset de cáncer de mama. Después de entrenar, utiliza `joblib.dump` para serializar el modelo y guardarlo como `model.pkl`. Este archivo se usará en el backend para cargar el modelo y hacer predicciones sin tener que re-entrenar cada vez.

> *Nota:* Puedes utilizar cualquier otro modelo de clasificación binaria de scikit-learn (como un árbol de decisión, `DecisionTreeClassifier`) siguiendo pasos similares. El dataset de iris también podría usarse convirtiéndolo a un problema binario, pero el de cáncer de mama ya es binario por defecto.

2. Backend con FastAPI (backend/app.py)

El backend es una aplicación web construida con **FastAPI** que expone dos endpoints principales:

- **POST** `/predict`**:** recibe un JSON con datos de entrada (las características del ejemplo a clasificar), carga el modelo entrenado y devuelve la predicción del modelo (0 o 1) en formato JSON.

- **POST** `/trigger`: recibe un JSON con una predicción (0 o 1) como parámetro, e invoca al contrato inteligente en Ethereum llamando a una función del mismo con la predicción. Para esto utilizamos la biblioteca `web3.py` para interactuar con una red Ethereum local (por ejemplo Ganache o Hardhat).

En `backend/app.py` se realiza lo siguiente:

- Carga el modelo entrenado `model.pkl` al iniciar (para no cargarlo en cada petición).
- Define el modelo de datos de entrada usando Pydantic (por ejemplo, un listado de características llamado `features`).
- Implementa el endpoint `/predict` que convierte la entrada a un array numpy, realiza la predicción con el modelo y retorna la respuesta.
- Configura la conexión a la blockchain local (HTTP provider de Ganache/Hardhat) con web3.py, incluyendo la dirección del contrato desplegado y su ABI (interfaz).
- Implementa el endpoint `/trigger` que toma la predicción, la convierte a booleano y realiza una transacción llamando a la función `actualizarEstado` del contrato inteligente, seguida de una llamada a `actuar`. Finalmente retorna un JSON indicando que el trigger se ejecutó.

Código: backend/app.py

```
from fastapi import FastAPI
from pydantic import BaseModel
import numpy as np
import joblib
from web3 import Web3

app = FastAPI()

# Cargar el modelo entrenado al iniciar la aplicación
model = joblib.load("ml/model.pkl")

# Definir la estructura esperada del JSON de entrada para /predict
class InputData(BaseModel):
    features: list[float]  # Lista de características numéricas de entrada

# Definir la estructura esperada del JSON de entrada para /trigger
class TriggerData(BaseModel):
    prediction: int  # Predicción (0 o 1) que se enviará al contrato

# Configurar conexión a la red Ethereum local (Ganache o Hardhat)
w3 = Web3(Web3.HTTPProvider("http://127.0.0.1:8545"))
if not w3.isConnected():
    raise Exception("No se pudo conectar a la red Ethereum local. Asegúrate
de que Ganache/Hardhat esté en ejecución.")

# **IMPORTANTE**: La dirección y ABI del contrato deben ser actualizados
# después de desplegar el contrato inteligente con deploy.py.
contract_address = "0xYourDeployedContractAddress"  # ← actualizar con la
dirección desplegada
contract_abi = [
    {
```

```
      "inputs": [{"internalType": "bool","name": "_prediccion","type":
"bool"}],
      "name": "actualizarEstado",
      "outputs": [],
      "stateMutability": "nonpayable",
      "type": "function"
    },
    {
      "inputs": [],
      "name": "actuar",
      "outputs": [],
      "stateMutability": "nonpayable",
      "type": "function"
    },
    {
      "inputs": [],
      "name": "lastPrediction",
      "outputs": [{"internalType": "bool","name": "","type": "bool"}],
      "stateMutability": "view",
      "type": "function"
    },
    {
      "anonymous": false,
      "inputs": [{"indexed": false,"internalType": "bool","name":
"prediccion","type": "bool"}],
      "name": "PredictionUpdated",
      "type": "event"
    },
    {
      "anonymous": false,
      "inputs": [],
      "name": "ActionExecuted",
      "type": "event"
    }
]
contract = w3.eth.contract(address=contract_address, abi=contract_abi)

# Usar la primera cuenta de Ganache/Hardhat para las transacciones (privada
en local)
default_account = w3.eth.accounts[0]
w3.eth.defaultAccount = default_account

@app.post("/predict")
def predict(input_data: InputData):
    # Convertir la lista de características a un array de numpy 2D (una sola
muestra)
    X = np.array([input_data.features])
    # Realizar la predicción con el modelo cargado
    pred = model.predict(X)[0]  # predicción 0 o 1
    return {"prediction": int(pred)}
```

```
@app.post("/trigger")
def trigger(data: TriggerData):
    pred_bool = bool(data.prediction)
    # Llamar a la función actualizarEstado del contrato con la predicción
    tx1 = contract.functions.actualizarEstado(pred_bool).transact({'from':
default_account})
    receipt1 = w3.eth.wait_for_transaction_receipt(tx1)
    # Llamar a la función actuar del contrato para tomar acción si la
predicción es true
    tx2 = contract.functions.actuar().transact({'from': default_account})
    receipt2 = w3.eth.wait_for_transaction_receipt(tx2)
    return {"status": "triggered", "prediction": data.prediction}
```

Explicación:
- Se inicializa FastAPI y se carga el modelo ML desde `ml/model.pkl`. - El endpoint `/predict` espera un JSON con una lista de valores numéricos (`features`) que representan las características del ejemplo a evaluar (en el caso del modelo de cáncer de mama, se esperarían 30 valores correspondientes a las características del tumor). La función convierte esto en un arreglo numpy con forma (1, n_features) y obtiene la predicción del modelo. Devuelve un JSON con la predicción, por ejemplo `{"prediction": 1}`.
- El endpoint `/trigger` espera un JSON con la predicción ya calculada, por ejemplo `{"prediction": 1}`. Al recibirlo, convierte ese valor a booleano (`True` o `False`). Luego utiliza `web3.py` para llamar al contrato inteligente: - Primero invoca `actualizarEstado(pred_bool)` en el contrato para actualizar el estado almacenado en la blockchain con la predicción. Esto se envía como una transacción, utilizando la cuenta por defecto configurada (la primera cuenta de la red local). Se espera a que la transacción sea minada. - Luego invoca `actuar()` en el contrato. Esta función en el contrato (como veremos) emitirá un evento si la predicción almacenada fue `true`. - Finalmente, devuelve un JSON simple indicando que el disparador se ejecutó, junto con la predicción enviada. - La variable `contract_address` debe ser configurada con la dirección real del contrato una vez desplegado, y `contract_abi` debe contener la interfaz ABI del contrato compilado. (En este ejemplo, se muestra una ABI simplificada manualmente con las definiciones de las funciones y eventos relevantes). - **Seguridad:** Dado que se trata de un ejemplo educativo, el endpoint `/trigger` acepta un valor de predicción sin autenticación ni validación adicional. En una aplicación real, se debería asegurar que solo sistemas autorizados puedan invocar la actualización en la blockchain.

Una vez que el backend esté en ejecución, FastAPI también proporcionará documentación interactiva (Swagger UI) en `http://localhost:8000/docs` donde se pueden probar estos endpoints fácilmente.

3. Contrato Inteligente en Solidity (contract/MLContract.sol)

El contrato inteligente, escrito en Solidity, almacenará el resultado de la predicción y tomará una acción en caso de que la predicción sea positiva (`true`). Este contrato puede ser desplegado en una red local de Ethereum (Ganache o Hardhat) para pruebas.

Requerimientos del contrato: - Tener una función pública `actualizarEstado(bool _prediccion)` que actualice una variable de estado almacenando el resultado (la predicción). - Tener una función pública `actuar()` que realice alguna acción si la predicción almacenada es verdadera. Como ejemplo simple, podemos hacer que `actuar()` emita un evento en la blockchain cuando `lastPrediction`

es `true`. - Incluir eventos para registrar cuándo se actualiza la predicción y cuándo se ejecuta la acción, de forma que podamos monitorear estas acciones en la cadena. - Debe ser compatible con Hardhat o Remix (es decir, seguir las convenciones estándar de Solidity para poder compilarlo en esos entornos).

Código: contract/MLContract.sol

```
// SPDX-License-Identifier: MIT
pragma solidity ^0.8.0;

contract MLContract {
    bool public lastPrediction;   // almacena el último resultado de
predicción recibido

    // Evento que se emite cada vez que se actualiza el estado con una nueva
predicción
    event PredictionUpdated(bool prediccion);
    // Evento que se emite cuando se ejecuta una acción tras una predicción
positiva
    event ActionExecuted();

    // Actualiza el estado almacenando la predicción recibida
    function actualizarEstado(bool _prediccion) public {
        lastPrediction = _prediccion;
        emit PredictionUpdated(_prediccion);
    }

    // Realiza una acción si la última predicción es true.
    // En este ejemplo, simplemente emite un evento para indicar la acción.
    function actuar() public {
        if (lastPrediction) {
            // Tomar acción (ej. emitir un evento, transferir fondos, etc.)
            emit ActionExecuted();
            // Nota: Podría agregarse lógica adicional aquí en un caso real.
        }
        // Si lastPrediction es false, la función no hace nada (no emite
evento).
    }
}
```

Explicación: Este contrato `MLContract` tiene: - Una variable de estado pública `lastPrediction` de tipo booleano que guarda la última predicción recibida. - Una función `actualizarEstado(bool _prediccion)` que cualquiera puede llamar (public) para actualizar `lastPrediction`. Cada vez que se llama, emite un evento `PredictionUpdated` con el valor de la predicción para registro. - Una función `actuar()` que revisa `lastPrediction`. Si es `true`, emite un evento `ActionExecuted()` indicando que se tomó la acción. En un caso de uso real, esta función podría contener lógica para llevar a cabo alguna acción más significativa (por ejemplo, activar otra parte del contrato, hacer transferencias, etc.), pero aquí solo emite un evento como demostración. Si `lastPrediction` es `false`, la función termina sin hacer nada. - Marcamos el contrato con una

licencia SPDX (MIT) y la versión de pragma ^0.8.0 para compatibilidad. Este contrato es muy simple y se puede desplegar fácilmente tanto en **Remix** (pegando este código) como en **Hardhat** (incluyéndolo en la carpeta `contracts/` de un proyecto Hardhat y compilándolo allí), o compilando con herramientas en Python como veremos en el script de despliegue.

4. Script de Despliegue en Python (contract/deploy.py)

El script de despliegue en Python utiliza `web3.py` para publicar el contrato inteligente en la red Ethereum local. Antes de ejecutarlo, asegúrate de tener una instancia local de la blockchain en funcionamiento (por ejemplo, ejecutando Ganache CLI/UI o un nodo Hardhat).

Pasos que realiza `deploy.py`: - Lee el archivo `MLContract.sol` y lo compila (usando la herramienta `solcx` en Python, o asumiendo que ya tienes el ABI y bytecode precompilados). - Se conecta al proveedor HTTP local de Ethereum (Ganache o Hardhat). - Utiliza la primera cuenta de la red local para pagar la transacción de despliegue. - Envía la transacción de creación del contrato con el bytecode compilado. - Espera a que la transacción se mine y obtiene la dirección del contrato desplegado. - Muestra (imprime) la dirección del contrato, que luego deberá usarse en la configuración del backend (app.py) para que éste sepa a qué dirección llamar.

Código: contract/deploy.py

```
import json
from solcx import compile_standard, install_solc
from web3 import Web3

# Leer el código del contrato
with open("contract/MLContract.sol", "r") as file:
    contract_source_code = file.read()

# Instalar compilador Solidity versión 0.8.0 si no está instalado
install_solc("0.8.0")

# Compilar el contrato Solidity
compiled_sol = compile_standard({
    "language": "Solidity",
    "sources": {"MLContract.sol": {"content": contract_source_code}},
    "settings": {
        "outputSelection": {
            "*": {
                "*": ["abi", "metadata", "evm.bytecode", "evm.sourceMap"]
            }
        }
    }
}, solc_version="0.8.0")

# Extraer ABI y bytecode compilado
abi = compiled_sol["contracts"]["MLContract.sol"]["MLContract"]["abi"]
bytecode = compiled_sol["contracts"]["MLContract.sol"]["MLContract"]["evm"]
["bytecode"]["object"]
```

```
# Conectar a la red Ethereum local (Ganache en este ejemplo)
w3 = Web3(Web3.HTTPProvider("http://127.0.0.1:8545"))
if not w3.isConnected():
    raise Exception("Error: no se pudo conectar a Ganache/Hardhat en http://
127.0.0.1:8545")

# Seleccionar la cuenta por defecto (primera cuenta de Ganache)
w3.eth.default_account = w3.eth.accounts[0]

# Construir el contrato en Web3
MLContract = w3.eth.contract(abi=abi, bytecode=bytecode)

# Desplegar el contrato
print("Desplegando contrato...")
tx_hash = MLContract.constructor().transact()
tx_receipt = w3.eth.wait_for_transaction_receipt(tx_hash)
contract_address = tx_receipt.contractAddress
print(f"Contrato desplegado exitosamente en la dirección:
{contract_address}")

# Guardar ABI y dirección en archivos JSON para uso posterior (opcional)
with open("contract/MLContract_abi.json", "w") as abi_file:
    json.dump(abi, abi_file)
with open("contract/MLContract_address.txt", "w") as addr_file:
    addr_file.write(contract_address)
```

Explicación: Este script automatiza el despliegue del contrato: - Usa `solcx` (un paquete de Python) para compilar el contrato Solidity directamente desde el código fuente. Asegura que la versión 0.8.0 del compilador esté instalada y luego compila el código, extrayendo el *ABI* y el *bytecode* resultante. - Se conecta al nodo local de Ethereum (`http://127.0.0.1:8545`). Por defecto, Ganache CLI corre en esa dirección y puerto; si usas la aplicación de Ganache GUI podría ser `http://127.0.0.1:7545`, y Hardhat por defecto también usa 8545 con `npx hardhat node`. Asegúrate de ajustar si es necesario. - Toma la primera cuenta de la lista de cuentas disponibles (Ganache usualmente proporciona varias cuentas con fondos para pruebas). Esta será la cuenta que paga por la transacción de despliegue. - Crea un objeto de contrato en web3.py con el ABI y bytecode, y llama a `constructor().transact()` para iniciar la transacción de despliegue. Espera el *receipt* de la transacción (lo que indica que se minó un bloque y el contrato se creó). - Obtiene la dirección del contrato (`contract_address`) desde el receipt e imprime este valor. **Esta dirección es crucial**, ya que debemos copiarla en `backend/app.py` (en la variable `contract_address`) para que el backend sepa a dónde enviar las transacciones de `actualizarEstado` y `actuar`. - Finalmente, opcionalmente guarda el ABI en un archivo JSON y la dirección en un `.txt` para referencia. Esto puede ayudar para cargar la información del contrato en otros scripts (por ejemplo, el de pruebas).

Nota: También podrías desplegar el contrato utilizando Hardhat (por ejemplo con un script de Hardhat o manualmente en Remix) y solo copiar la dirección y ABI al backend; sin embargo, el propósito aquí es mostrar cómo hacerlo directamente en Python para integrar bien con el flujo del proyecto.

5. Script de Prueba de Flujo (test/test_workflow.py)

Este script opcional demuestra el flujo completo: tomar una muestra de datos, obtener una predicción del modelo ML, y luego simular las llamadas al contrato inteligente para verificar que todo funcione de extremo a extremo. En un entorno real, podríamos probar llamando directamente a los endpoints del API (por ejemplo, usando la librería `requests` para hacer POST a `/predict` y `/trigger`), pero para mantenerlo simple aquí interactuaremos directamente con el modelo y el contrato a través de web3.py.

Lo que hace `test_workflow.py`: - Carga el modelo entrenado `model.pkl`. - Toma un ejemplo del dataset (por simplicidad, el primero del conjunto de datos de cáncer de mama). - Genera la predicción usando el modelo. - Se conecta a la blockchain local y al contrato desplegado (requiere que `deploy.py` se haya ejecutado y `contract_address` esté disponible). - Llama a `actualizarEstado` en el contrato con la predicción obtenida y espera a que la transacción se mine. - Lee el valor almacenado `lastPrediction` del contrato para verificar que coincide con la predicción enviada. - Opcionalmente, llama a `actuar()` y, si la predicción fue true, espera que un evento se emita (aunque capturar eventos requeriría configuración adicional, se puede al menos ejecutar la función para ver que no falla).

Código: test/test_workflow.py

```
import joblib
import numpy as np
from sklearn.datasets import load_breast_cancer
from web3 import Web3
import json

# Cargar modelo entrenado
model = joblib.load("ml/model.pkl")

# Cargar un ejemplo de datos (usaremos el primer dato del dataset de cáncer
de mama)
data = load_breast_cancer()
X_example = data.data[0]  # primer ejemplo
y_example = data.target[0]  # etiqueta real del primer ejemplo (por
curiosidad)

# Predecir usando el modelo
pred = model.predict(X_example.reshape(1, -1))[0]
print(f"Predicción del modelo para el ejemplo de prueba: {int(pred)}
(Etiqueta real: {y_example})")

# Conectar a la red Ethereum local
w3 = Web3(Web3.HTTPProvider("http://127.0.0.1:8545"))
w3.eth.default_account = w3.eth.accounts[0]

# Cargar ABI y dirección del contrato desplegado (asegúrate de haber
ejecutado deploy.py)
with open("contract/MLContract_abi.json", "r") as abi_file:
    contract_abi = json.load(abi_file)
```

```
with open("contract/MLContract_address.txt", "r") as addr_file:
    contract_address = addr_file.read().strip()

contract = w3.eth.contract(address=contract_address, abi=contract_abi)

# Llamar a la función actualizarEstado del contrato con la predicción
tx1 = contract.functions.actualizarEstado(bool(pred)).transact()
w3.eth.wait_for_transaction_receipt(tx1)
print(f"Transacción actualizarEstado enviada (predicción = {bool(pred)})")

# Leer el estado almacenado en el contrato para verificar
stored_value = contract.functions.lastPrediction().call()
print(f"Valor almacenado en contrato lastPrediction: {stored_value}")

# Llamar a la función actuar del contrato
tx2 = contract.functions.actuar().transact()
w3.eth.wait_for_transaction_receipt(tx2)
print("Función actuar ejecutada en el contrato (si lastPrediction=true, se
habrá emitido evento ActionExecuted).")
```

Explicación: Este script carga un ejemplo del dataset original y valida que el flujo funcione: - Después de predecir el valor (por ejemplo, 0 = maligno, 1 = benigno), envía ese resultado al contrato. - Tras ejecutar `actualizarEstado`, verifica mediante `lastPrediction()` que el contrato efectivamente almacenó el valor correcto (imprime el valor booleano almacenado). - Luego ejecuta `actuar()` en el contrato. Si el valor era `true`, en la blockchain se habrá emitido el evento `ActionExecuted` (aunque aquí solo indicamos que la función se ejecutó; capturar el evento requeriría escuchar los logs de eventos, lo cual se podría hacer con web3.py *filters* o mediante la interfaz de Hardhat/Remix). - Este script supone que el archivo ABI y la dirección han sido guardados por el script de despliegue. Alternativamente, podríamos haber importado directamente la misma lógica de deploy.py o configurado la dirección manualmente, pero usar los archivos generados hace el proceso más automático.

6. Instrucciones de Instalación y Ejecución

A continuación se detallan los pasos para poner en marcha el proyecto completo en un entorno local:

1. **Clonar el proyecto (o copiar la estructura)**: Asegúrate de tener los archivos organizados como en la estructura dada. Necesitarás tener instalado Python 3.8+ y Node.js (para usar Hardhat, si optas por Hardhat en lugar de Ganache).

2. **Instalar dependencias de Python**: Navega al directorio del proyecto y ejecuta:

   ```
   pip install -r backend/requirements.txt
   ```

   El archivo `requirements.txt` debería contener las bibliotecas necesarias, por ejemplo:

   ```
   fastapi
   uvicorn[standard]
   ```

```
numpy
scikit-learn
joblib
web3
py-solc-x      # (solcx for compiling solidity)
```

Esto instalará FastAPI, Uvicorn (servidor ASGI para ejecutar FastAPI), numpy, scikit-learn, joblib, web3.py y solcx, entre otras.

3. **Iniciar una red Ethereum local**: Puedes usar **Ganache** o **Hardhat**:

4. *Ganache:* Abre la aplicación de Ganache y crea una red local, o ejecuta `ganache-cli` en la terminal. Asegúrate de que esté corriendo en la URL y puerto esperados (por defecto http://127.0.0.1:8545).
5. *Hardhat:* Ve al directorio `contract/` (o la raíz) y ejecuta `npx hardhat node` para levantar un nodo local de Hardhat en http://127.0.0.1:8545. Hardhat te mostrará cuentas disponibles con sus claves privadas (se pueden usar en web3.py automáticamente como cuentas pre-fundidas).

6. Cualquiera sea la opción, toma nota de la URL (endpoint RPC) y de las cuentas disponibles. Por defecto, en Ganache y Hardhat la primera cuenta tiene saldo y es la que usamos en el script.

7. **Entrenar el modelo ML**: Ejecuta el script de entrenamiento para generar el archivo del modelo:

```
python ml/train_model.py
```

Esto creará el archivo `ml/model.pkl` si todo va bien. En la salida verás la exactitud del modelo en los datos de prueba (si implementaste esa parte) y el mensaje de que el modelo fue guardado.

8. **Desplegar el contrato inteligente**: Asegúrate que la red local Ethereum esté corriendo (paso 3). Luego ejecuta:

```
python contract/deploy.py
```

Este script compilará `MLContract.sol` y enviará la transacción de despliegue. Debería imprimir en la consola algo como:

```
Desplegando contrato...
Contrato desplegado exitosamente en la dirección: 0xABC123...789
```

Copia esa dirección y **actualiza la variable** `contract_address` en `backend/app.py` con este valor. (También puedes encontrar la dirección en el archivo `contract/MLContract_address.txt` generado). Asegúrate de que `contract_abi` en `app.py` coincide con el ABI del contrato (el deploy.py lo guardó en `MLContract_abi.json`, que puedes copiar y pegar, o importar directamente leyendo el JSON en `app.py` para mayor automatización).

9. **Configurar el backend**: Verifica que en `backend/app.py` has puesto la dirección del contrato desplegado. Si deseas, en lugar de hardcodear la dirección, puedes modificar `app.py` para que lea la dirección desde el archivo generado o desde una variable de entorno. Para mantener el ejemplo sencillo, en este punto simplemente edita el archivo manualmente:

```
contract_address = "0x...la_dirección_impresa_al_desplegar..."
```

Y guarda los cambios.

10. **Ejecutar el servidor FastAPI**: Inicia la aplicación backend usando Uvicorn (el servidor ASGI):

```
uvicorn backend.app:app --reload --port 8000
```

Esto levantará el servidor en http://127.0.0.1:8000 (puedes cambiar el puerto si 8000 está ocupado). La opción `--reload` recargará automáticamente si hay cambios en el código. Deberías ver en la consola un mensaje como "Uvicorn running on http://127.0.0.1:8000".

11. **Probar el endpoint /predict**: Con el servidor en marcha, abre otra terminal o usa una herramienta como *curl* o *Insomnia/Postman*. Por ejemplo, usando curl:

```
curl -X POST http://localhost:8000/predict \
     -H "Content-Type: application/json" \
     -d '{"features": [val1, val2, ..., valN]}'
```

Donde `[val1, val2, ... valN]` debe ser reemplazado por un array de números con el mismo tamaño de características que el modelo espera. Por ejemplo, para el modelo de cáncer de mama, se necesitan 30 valores numéricos. Puedes tomar alguno de los ejemplos del dataset original o cualquier valor de prueba. Si todo funciona, la respuesta será un JSON con la predicción: `{"prediction": 0}` o `{"prediction": 1}`.
Ejemplo: Supongamos que conocemos un caso de prueba (por simplicidad usamos los primeros 30 valores del primer ejemplo del dataset):

```
curl -X POST http://localhost:8000/predict \
     -H "Content-Type: application/json" \
     -d '{"features":
[17.99,10.38,122.8,1001.0,0.1184,0.2776,0.3001,0.1471,0.2419,0.07871,1.095,0.9053,8.589,1
```

La respuesta podría ser algo como `{"prediction": 0}` indicando, por ejemplo, predicción de "maligno".

12. **Probar el endpoint /trigger**: Una vez que tengas una predicción (p. ej. 0 o 1), puedes enviar ese resultado al contrato. Normalmente, el flujo sería: primero llamas a `/predict` (desde una aplicación front-end, por ejemplo) y luego tomas ese resultado y llamas a `/trigger`. Para probar manualmente, puedes hacer:

```
curl -X POST http://localhost:8000/trigger \
     -H "Content-Type: application/json" \
     -d '{"prediction": 1}'
```

(Aquí estamos enviando `1` como ejemplo; puedes usar `0` o `1` según la predicción obtenida). La respuesta debería ser `{"status": "triggered", "prediction": 1}`. Esto indica que el backend recibió tu solicitud y ejecutó las transacciones en la blockchain. En la consola donde corre Ganache/Hardhat, deberías ver los logs de las transacciones que se enviaron (llamadas a `actualizarEstado` y `actuar`). No se devuelve directamente el resultado on-chain, pero puedes verificar el estado del contrato de otras maneras:

13. Observando eventos: Ganache GUI tiene una sección de *Logs* donde puedes ver los eventos emitidos. Si enviaste `prediction: 1`, el contrato emitirá `PredictionUpdated(true)` y luego `ActionExecuted()`. Estos eventos aparecerán en los logs de la cadena.
14. Consultando manualmente el contrato: Puedes usar herramientas como Remix o incluso añadir un endpoint GET en FastAPI para leer `lastPrediction`. Por ejemplo, podríamos añadir en `app.py`:

```
@app.get("/status")
def status():
    value = contract.functions.lastPrediction().call()
    return {"lastPrediction": value}
```

Entonces una petición GET a `/status` devolvería el valor booleano almacenado en el contrato.

15. Usando el script de prueba `test_workflow.py` descrito antes para verificar que `lastPrediction` corresponde a lo enviado.

16. **(Opcional) Ejecutar el script de prueba end-to-end**: Como alternativa a los últimos pasos manuales, puedes ejecutar:

```
python test/test_workflow.py
```

Este script hará una predicción sobre un ejemplo fijo y enviará la transacción al contrato, imprimiendo en la consola los resultados. Asegúrate de haber puesto la dirección del contrato en los archivos correspondientes o de haber ejecutado `deploy.py` que guarda esos valores, para que el script de prueba funcione correctamente.

Notas Finales

- **Versión de Solidity**: El contrato está escrito para Solidity 0.8.x. Si usas una versión distinta, puede ser necesario ajustar cosas (por ejemplo, tipos o habilitar el optimizador al compilar, etc.). La versión 0.8 es moderna y soporta las características usadas (eventos, etc.).
- **Ganache vs Hardhat**: Puedes usar cualquiera. Si usas Ganache GUI, recuerda configurar la URL correcta. Para Hardhat, el proceso es similar; Hardhat proporciona cuentas que puedes usar. En Hardhat, si quieres desplegar con el script Python, asegúrate de que Hardhat esté corriendo

(`npx hardhat node`). Como alternativa, podrías integrar el despliegue del contrato usando un script de Hardhat y luego solo interactuar con él en Python.

- **Seguridad**: Este proyecto es un **ejemplo educativo**. No implementa autenticación, ni en el backend ni en el contrato (cualquiera puede llamar a las funciones públicas del contrato). En un entorno real, uno podría restringir quién puede llamar a `actualizarEstado` (por ejemplo, solo la cuenta del backend) usando patrones de autorización en el contrato (modificadores `onlyOwner`, etc.). Además, las transacciones en Ethereum cuestan gas; en este entorno local esto no es un problema (Ganache/Hardhat no cuestan dinero real), pero en una red de prueba o mainnet habría que considerar costos y seguridad de la clave privada usada por `web3.py`.
- **Extensiones posibles**: Este proyecto básico se puede ampliar de muchas formas:
- En lugar de solo emitir un evento, el contrato podría, por ejemplo, transferir algún token o Ether a alguien si la predicción es positiva, o podría alternar algún estado de un sistema mayor.
- Se podría almacenar un histórico de predicciones en el contrato (por ejemplo, usando un array o mapeo) en lugar de solo la última, dependiendo del caso de uso.
- Integrar un frontend que consuma los endpoints de FastAPI y muestre resultados al usuario, completando así una aplicación web completa.
- Usar un modelo más complejo o incluso hacer que el contrato solicite la predicción mediante *oracles* si quisiéramos que la inferencia ocurra en respuesta a un evento on-chain (esto último es más avanzado y requiere un oráculo off-chain orquestando llamadas).

Con esto, el proyecto **ml-smart-contract-project** quedaría completo. Siguiendo las instrucciones, deberías poder entrenar el modelo, desplegar el contrato, ejecutar el backend y observar cómo una predicción de machine learning se utiliza para triggerar lógica en un contrato inteligente de Ethereum. ¡Buena suerte y feliz aprendizaje!
