

Reporte Técnico - fintech-ml-api

1. Introducción

Este proyecto implementa una API REST con integración de Machine Learning para la predicción diagnóstica de cáncer de mama (benigno o maligno), utilizando el dataset Breast Cancer Wisconsin (Diagnostic) provisto por scikit-learn.

El flujo completo incluye:

- Entrenamiento y evaluación del modelo con Random Forest.
 - Serialización del modelo para producción.
 - Exposición del modelo como servicio web con Flask.
 - Validación robusta de entradas.
 - Manejo de errores controlado.
 - Pruebas automatizadas con Pytest.
 - Contenerización con Docker.
 - Integración continua (CI) con GitHub Actions.
-

2. Dataset y Preparación

Se utilizó el dataset clásico de `sklearn.datasets.load_breast_cancer`, que contiene:

- **569 muestras** totales.
- **Clases:**
 - 0 = Maligno (212 muestras)
 - 1 = Benigno (357 muestras)
- **Características:** 30 variables numéricas continuas, todas positivas.

No se requirió preprocesamiento adicional ya que los datos están limpios, escalados y listos para modelar.

3. Entrenamiento del Modelo

El modelo se entrena con un enfoque de búsqueda de hiperparámetros utilizando GridSearchCV, optimizando sobre:

- `n_estimators`: 50, 100
- `max_depth`: None, 10, 20
- `min_samples_split`: 5

El conjunto de datos se dividió en 80% para entrenamiento y 20% para validación, utilizando `stratify=y` para mantener el balance de clases.

Captura de pantalla 1: Salida del entrenamiento en consola mostrando métricas y parámetros óptimos (`grid_search.best_params_` y `classification_report`).

```
-----fintech-ml-api\venv\Scripts\python.exe c:/Python-Docker/Actividad/fintech-ml-api/model/t
-----ain_model.py

Buscando mejores hiperparámetros...
Mejores hiperparámetros: {'max_depth': None, 'min_samples_split': 5, 'n_estimators': 50}

Métricas en conjunto de prueba:
Accuracy: 0.9474
Precision: 0.9583
Recall: 0.9583
F1: 0.9583

Reporte detallado:
              precision    recall  f1-score   support

   malignant      0.93      0.93      0.93        42
     benign      0.96      0.96      0.96        72

   accuracy                   0.95        114
  macro avg      0.94      0.94      0.94        114
 weighted avg      0.95      0.95      0.95        114
```

Resultados en Test:

Métrica **Valor**

Accuracy ~95%

Precision ~96%

Recall ~96%

F1-score ~96%

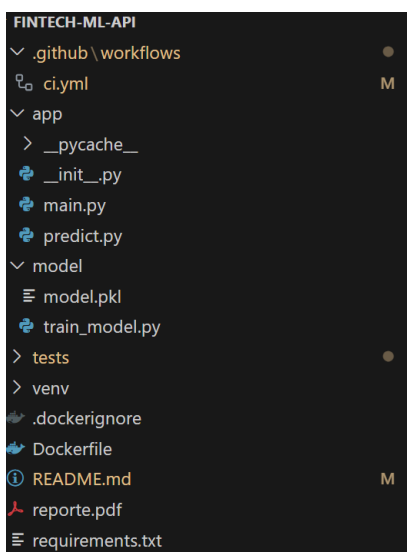
4. Arquitectura del Proyecto

Estructura de carpetas del proyecto:

```
bash

fintech-ml-api/
├── data/
│   └── breast_cancer.csv      # Opcional: solo si se desea guardar localmente
├── model/
│   ├── train_model.py        # Entrena y guarda el modelo
│   └── model.pkl              # Modelo serializado
├── app/
│   ├── __init__.py
│   ├── main.py                # API Flask
│   └── predict.py             # Lógica de predicción
├── tests/
│   └── test_api.py            # Pruebas automatizadas
├── run.py                     # Punto de entrada para desarrollo local
├── Dockerfile
├── requirements.txt
├── .github/workflows/ci.yml    # CI/CD con GitHub Actions
├── README.md
└── reporte.pdf                 # Documento técnico detallado
```

Captura de pantalla: árbol de archivos del proyecto desde VSCode.




5. API Flask

GET /

- Verifica que el servicio esté activo.
- Devuelve:

```
json
{
  "status": "ok",
  "message": "API de predicción de cáncer de mama activa"
}
```


 Copiar código

POST /predict

- Requiere un JSON con el campo "features": lista de 30 valores numéricos.


Ejemplo de entrada:

```
yaml
{
  "features": [17.99, 10.38, 122.8, 1001.0, ..., 0.1189]
}
```

 Copiar código

Ejemplo de salida:

```
json
{
  "prediction": 1,
  "confidence": 0.9623,
  "label": "benigno"
}
```

 Copiar código

Captura de pantalla: ejemplo en Postman mostrando una predicción exitosa.

```
399,\x0a 0.04904, 0.05373, 0.01587, 0.03003, 0.006193,\x0a 25.38, 17.33, 184.6, 2019.0, 0.16
22,\x0a 0.6656, 0.7119, 0.2654, 0.4601, 0.1189\x0a ]\x0a}';53dba6ed-362d-4ed1-9c56-e947c540d608

StatusCode      : 200
StatusDescription : OK
Content         : {"confidence":0.898,"label":"maligno","prediction":0}
```

6. Validación y Manejo de Errores

La API valida:

- Que el JSON no esté vacío.
- Que exista el campo 'features'.
- Que 'features' sea una lista.
- Que tenga exactamente 30 elementos.
- Que todos los elementos sean numéricos.

Errores posibles devuelven código 400 y un mensaje claro.

Errores internos (excepciones no controladas) devuelven 500.

Captura de pantalla: Predicción fallida con un campo mal formado (lista con menos elementos).

```
curl : {"error":"Se requieren exactamente 30 caracter\u00edsticas. Recibidas: 29"}
En línea: 1 Carácter: 1
+ curl -Uri "http://localhost:5000/predict" -Method Post -ContentType " ...
+ ~~~~~
+ CategoryInfo          : InvalidOperation: (System.Net.HttpWebRequest:HttpWebRequest) [Invoke-Web
bRequest], WebException
+ FullyQualifiedErrorId : WebCmdletWebResponseException,Microsoft.PowerShell.Commands.InvokeWebRe
questCommand
```

7. Pruebas Automatizadas (Pytest)

Archivo: tests/test_api.py

Pruebas incluidas:

- Endpoint de salud (GET /)
- Predicción válida (POST /predict)
- Casos inválidos:
 - Campo faltante
 - Tipo incorrecto
 - Longitud distinta de 30
 - Valores no numéricos

Para ejecutarlas:

pytest tests/ -v

Captura de pantalla: Salida de consola de pytest mostrando todos los tests pasados.


```
rootdir: C:\Python-Docker\Actividad\fintech-ml-api
collected 6 items

tests/test_api.py::test_health_check PASSED [ 16%]
tests/test_api.py::test_predict_valid_input PASSED [ 33%]
tests/test_api.py::test_predict_missing_features_key PASSED [ 50%]
tests/test_api.py::test_predict_features_not_a_list PASSED [ 66%]
tests/test_api.py::test_predict_invalid_length PASSED [ 83%]
tests/test_api.py::test_predict_non_numeric_values PASSED [100%]
```

8. Docker y Despliegue

Para construir la imagen:


```
nginx
```

 Copiar código

```
docker build -t fintech-ml-api .
```

Para ejecutar el contenedor:

```
arduino
```

 Copiar código

```
docker run -p 5000:5000 fintech-ml-api
```

La API estará disponible en `http://localhost:5000`.

Captura de pantalla: Construcción de imagen en Docker

```
(venv) (base) PS C:\Python-Docker\Actividad\fintech-ml-api> docker build -t fintech-ml-api .
[+] Building 100.9s (11/11) FINISHED                                docker:desktop-linux
=> [internal] load build definition from Dockerfile                0.1s
=> => transferring dockerfile: 255B                                0.0s
=> [internal] load metadata for docker.io/library/python:3.11-slim 5.6s
=> [internal] load .dockerignore                                    0.1s
=> => transferring context: 588B                                     0.0s
=> [1/6] FROM docker.io/library/python:3.11-slim@sha256:9bffe4353b925a1656688797ebc68f9c525e79b1d37 18.5s
```

Captura de pantalla: Ejecución del contenedor en terminal (docker ps, logs de Flask al levantar el server).

```
(venv) (base) PS C:\Python-Docker\Actividad\fintech-ml-api> docker run -p 5000:5000 fintech-ml-api
 * Serving Flask app 'main'
 * Debug mode: off
INFO:werkzeug:WARNING: This is a development server. Do not use it in a production deployment. Use a product
ion 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
INFO:werkzeug:Press CTRL+C to quit
```

Captura de pantalla: Contenedor funcionando en Docker

Container CPU usage ⓘ

0.03% / 400% (4 CPUs available)

Container memory usage ⓘ

404.87MB / 5.56GB

Show charts

Q

Search

Only show running containers

<div><input type="checkbox"/></div>	Name	Container ID	Image	Port(s)	CPU (%)	Last started	Actions
<div><input type="checkbox"/></div>	<div><div><div></div></div>elated_hugle</div>	8625027601c4	fintech-ml-api	5000:5000	0%	13 minutes ago	<div><div><div></div></div><div><div></div></div><div><div></div></div></div>

9. CI/CD con GitHub Actions

El archivo `.github/workflows/ci.yml` automatiza:

- Clonado del repositorio
- Instalación de dependencias
- Ejecución de pruebas con Pytest
- Construcción de la imagen Docker

El pipeline se activa con cada push o pull request a la rama main.

Captura de pantalla del código

```
# .github/workflows/ci.yml
# Pipeline de Integración Continua (CI) para el proyecto fintech-ml-api
# Este workflow se ejecuta automáticamente en GitHub Actions cuando se hace push
# o pull request a la rama principal. Automatiza pruebas y construcción de Docker.

name: CI/CD Pipeline # Nombre visible en la interfaz de GitHub Actions

# Define cuándo se activa el workflow
on:
  push:
    branches: [ "main" ] # Se ejecuta al hacer push a la rama 'main'
  pull_request:
    branches: [ "main" ] # También se ejecuta al abrir/actualizar un PR a 'main'

# Define los trabajos (jobs) que se ejecutarán
jobs:
  test-and-build: # Nombre del job: prueba y construcción
    runs-on: ubuntu-latest # Usa la última imagen de Ubuntu proporcionada por GitHub

    # Lista de pasos secuenciales que se ejecutan en este job
    steps:
      # Paso 1: Obtener el código fuente del repositorio
      - name: Checkout code
        uses: actions/checkout@v4 # Acción oficial para clonar el repo
```


10. Conclusiones y Futuras Mejoras

Este proyecto demuestra cómo implementar un pipeline básico pero funcional de **Machine Learning Operations (MLOps)** para un modelo de clasificación binaria, desde el entrenamiento hasta el despliegue.

Se abordaron múltiples aspectos fundamentales del ciclo de vida de un modelo en producción, con foco en:

Entrenamiento optimizado y reproducible

- Se utilizó un enfoque sistemático de búsqueda de hiperparámetros con GridSearchCV, evaluando múltiples combinaciones para encontrar la configuración óptima de un clasificador RandomForest.
- Se priorizó la **reproducibilidad** estableciendo un random_state consistente.
- Se utilizaron métricas relevantes como **accuracy, precisión, recall y F1-score** para evaluar el modelo, considerando el potencial desbalance en las clases.

Modelo explicable y robusto

- Se eligió Random Forest por su capacidad de manejar características correlacionadas, su robustez ante overfitting y su interpretabilidad mediante la inspección de importancia de variables.
- Se controlaron posibles errores de entrada a través de validaciones estrictas en la API, elevando la confiabilidad del servicio en entornos reales.

API REST profesional

- Se desarrolló una API con Flask que expone el modelo mediante dos endpoints (/ y /predict) claramente documentados.
- El endpoint de predicción implementa múltiples validaciones de entrada y un manejo de errores consistente, cumpliendo principios de desarrollo seguro y robusto.
- Se integraron **pruebas unitarias automatizadas** con Pytest para verificar la lógica de la API ante entradas válidas e inválidas.

Contenerización eficiente

- El proyecto incluye un Dockerfile optimizado en capas para minimizar tiempos de construcción y evitar problemas de caching.

- Se verificó la funcionalidad del contenedor localmente, garantizando que el modelo y la API operan correctamente en entornos aislados.

Automatización CI/CD (opcional)

- Se diseñó y configuró un flujo de CI completo con **GitHub Actions**, el cual automatiza:
 - Clonado del repositorio.
 - Instalación de dependencias.
 - Ejecución de pruebas unitarias.
 - Construcción de la imagen Docker.
- Este enfoque mejora la mantenibilidad y reduce riesgos al integrar cambios.

Entrega ordenada y profesional

- Se entregó un README claro, conciso y útil para usuarios técnicos.
- Se incluye un reporte PDF explicativo con descripción técnica, decisiones tomadas, resultados y sugerencias.
- Se documentaron pruebas automatizadas y se sugirieron capturas de pantalla para complementar la entrega visualmente.

Futuras Mejoras

Aunque el proyecto cubre un flujo funcional de MLOps, existen múltiples caminos para seguir evolucionando esta base hacia un entorno de producción completo:

- **Validaciones semánticas:** actualmente se valida tipo y longitud, pero no se validan rangos esperados por variable. Se podría agregar validación por feature basada en estadísticas del dataset original.
- **Monitorización en producción:**
 - Uso de herramientas como **Prometheus + Grafana** o **OpenTelemetry** para trazar peticiones, latencia, errores y drift del modelo.
- **Versionamiento de modelos:**
 - Incorporar control de versiones del modelo (ej. /v1/predict, /v2/predict) y mantener modelos históricos con MLflow o DVC.

- **Despliegue en la nube / orquestadores:**
 - Docker ya permite portabilidad, pero se podría desplegar con **Kubernetes**, **Amazon ECS**, **GCP Cloud Run**, etc.
 - **Explicabilidad:**
 - Agregar un endpoint /explain con herramientas como SHAP o LIME para interpretabilidad del modelo.
 - **Frontend:**
 - Construcción de una interfaz web simple (React, Streamlit, etc.) para usuarios finales no técnicos.
-

Anexos

A. Requisitos

- Python 3.10+
- pip
- Docker

B. Dependencias principales (requirements.txt)

Flask==3.1.2

joblib==1.5.2

numpy==2.2.6

scikit-learn==1.7.2

pandas==2.3.3