

LAB 3 — Continuous Training: app-iris-ct

En app-iris aprendiste a servir un modelo de Machine Learning entrenado offline mediante una API REST. Eso corresponde al nivel básico de MLOps: *deploy once, serve forever*. El problema es que el mundo real cambia: los datos de producción evolucionan, aparecen nuevas muestras etiquetadas y el modelo puede quedarse obsoleto.

En esta actividad vas a implementar **Continuous Training (CT)**: la capacidad de reentrenar el modelo, evaluar si el nuevo modelo es mejor que el actual y activarlo automáticamente solo si mejora la métrica de calidad. Al finalizar, tu servidor Iris no solo predecirá clases de flores, sino que también aprenderá de los datos nuevos sin necesidad de ser reiniciado.

Estructura del proyecto

```
app-iris-ct/
├── main.py           ← servidor FastAPI extendido (a desarrollar)
├── demo_ct.py        ← script de demostración del flujo CT
├── Dockerfile
├── requirements.txt
└── models/           ← creada automáticamente en runtime
    ├── model_active.joblib   ← modelo activo serializado
    ├── accumulated_data.joblib  ← dataset acumulado entre
    |   entrenamientos
    └── training_history.json   ← registro de versiones
 README.md
```

Objetivos de aprendizaje

Al terminar esta actividad serás capaz de:

1. Diseñar una API REST que combine inferencia y reentrenamiento en el mismo servicio.
 2. Implementar un **modelo de versiones simple** para artefactos ML.
 3. Aplicar un **gate de calidad**: solo activar un modelo nuevo si mejora la métrica anterior.
 4. Distinguir entre entrenamiento **incremental** (acumulando datos) y **desde cero**.
 5. Serializar y deserializar modelos scikit-learn con joblib de forma segura.
 6. Reflexionar sobre las limitaciones de esta arquitectura frente a MLOps de nivel 2.
-

Endpoints a implementar

Método	Ruta	Descripción
GET	/health	Estado del servicio y versión del modelo activo
POST	/predict	Predicción con el modelo activo (igual que app-iris)
POST	/train	Reentrenamiento con nuevas muestras etiquetadas
GET	/model/info	Metadata del modelo activo e historial de versiones
DELETE	/model/history	Resetea el historial (para pruebas)
TE		

Esquema de /train (request)

```
{  
    "samples": [  
        {  
            "sepal_length": 5.1,  
            "sepal_width": 3.5,  
            "petal_length": 1.4,  
            "petal_width": 0.2,  
            "label": 0  
        }  
    ],  
    "retrain_from_scratch": false  
}
```

- label acepta 0 (setosa), 1 (versicolor) o 2 (virginica).
- Se requieren **mínimo 5 muestras** por request.
- retrain_from_scratch: false → las nuevas muestras se **acumulan** al dataset anterior.
- retrain_from_scratch: true → se entrena **solo** con las muestras enviadas.

Esquema de /train (response)

```
{  
    "status": "activado",  
    "model_version": "v2.0-a3f9b1",  
    "accuracy_new": 0.9667,  
    "accuracy_previous": 0.9333,  
    "model_updated": true,  
    "message": "Nuevo modelo activado. Accuracy 0.9667 >= anterior  
(0.9333)"  
}
```

Esquema de /model/info (response)

```
{  
    "active_version": "v2.0-a3f9b1",
```

```
"trained_at": "2024-11-15T10:23:44Z",
"accuracy": 0.9667,
"n_training_samples": 120,
"algorithm": "LogisticRegression",
"history": [
  {
    "version": "v1.0-base",
    "trained_at": "2024-11-15T09:00:00Z",
    "accuracy": 0.9333,
    "n_training_samples": 120,
    "source": "bootstrap (iris dataset completo)",
    "activated": true
  },
  {
    "version": "v2.0-a3f9b1",
    "trained_at": "2024-11-15T10:23:44Z",
    "accuracy": 0.9667,
    "n_training_samples": 140,
    "source": "incremental (+20 muestras nuevas, 120 anteriores)",
    "activated": true
  }
]
```

Lógica del gate de calidad

accuracy_nuevo >= accuracy_anterior → ACTIVAR y guardar modelo
accuracy_nuevo < accuracy_anterior → RECHAZAR, mantener modelo anterior

El registro del intento de entrenamiento **siempre** queda en el historial, incluso si el modelo es rechazado. Esto permite auditar qué datos degradaron el modelo.

Instrucciones de desarrollo

Prerrequisitos

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

Arrancar el servidor en local

```
uvicorn main:app --reload
```

Accede a la documentación interactiva en: <http://localhost:8000/docs>

Arrancar con Docker

```
# Construir imagen
docker build -t iris-ct .

# Lanzar contenedor con volumen para persistir los modelos
docker run -d \
-p 8000:80 \
-v iris-ct-models:/app/models \
--name iris-ct \
iris-ct

docker run -d -p 8000:80 -v iris-ct-models:/app/models --name iris-ct
iris-ct
```

Con el volumen `-v iris-ct-models:/app/models` los modelos entrenados **sobreviven** al reinicio del contenedor.

Ejecutar la demo completa

```
# Servidor debe estar arrancado primero
python demo_ct.py --reset
```

El flag `--reset` restaura el modelo base antes de ejecutar el flujo. Salida esperada:

✿ DEMO: Iris Continuous Training API

Host: <http://localhost:8000>

PASO 0 – Health check

Servicio activo. Modelo activo: v1.0-base

[...]

PASO 2 – Reentrenamiento con muestras CORRECTAS

Nuevo modelo ACTIVADO → versión: v2.0-xxxx
 Accuracy nuevo: 0.9667 | Anterior: 0.9333

PASO 3 – Reentrenamiento con muestras RUIDOSAS

Modelo RECHAZADO como esperado. Accuracy 0.6000 < anterior 0.9667

Ejercicios de la actividad

Ejercicio 1 — Reproducir el workflow Arranca el servidor, ejecuta `demo_ct.py --reset` y comprueba que los 4 pasos funcionan correctamente. Captura la salida completa del terminal y el JSON de `/model/info` al final.

Ejercicio 2 — Explorar el historial

Tras ejecutar la demo, abre `models/training_history.json` y responde:

- ¿Cuántas versiones se han registrado?
- ¿Qué versiones fueron activadas y cuáles rechazadas?
- ¿Por qué el modelo entrenado con muestras ruidosas fue rechazado?

NOTA: Los ficheros generados están dentro del contenedor, montados en un volumen:
`iris-ct-models:/app/models`

Ejercicio 3 — Entrenamiento incremental vs. desde cero

Realiza dos llamadas a `/train` con el mismo conjunto de 10 muestras:

- Primera vez con `retrain_from_scratch: false`
- Segunda vez con `retrain_from_scratch: true`

¿Observas diferencias en el accuracy? ¿Por qué?

PREGUNTAS

Pregunta 1

Cuando arrancas el servidor por primera vez sin ningún modelo previo, el sistema crea automáticamente un modelo base mediante el proceso de *bootstrap*. Si detuvieras el servidor, borraras la carpeta `models/` y lo volvieras a arrancar, ¿obtendrías exactamente el mismo modelo con el mismo accuracy? Razona por qué sí o por qué no, y señala qué línea o parámetro del código garantiza o impide esa reproducibilidad.

Pregunta 2

El endpoint `/train` exige un mínimo de 5 muestras por request y además comprueba que haya al menos 2 clases distintas en el dataset de entrenamiento. ¿Qué pasaría si enviaras 10 muestras perfectamente etiquetadas pero todas de la clase *setosa*? ¿El modelo se entrenaría? ¿Se activaría? Describe paso a paso qué devolvería la API y por qué tiene sentido esa restricción.

Pregunta 3

Ejecuta la demo con --reset y observa la secuencia de versiones en el historial final. El modelo entrenado con muestras ruidosas aparece en el historial aunque no haya sido activado. ¿Por qué crees que se diseñó así, registrando también los modelos rechazados? ¿Qué información útil proporciona ese registro que no tendríamos si solo guardáramos los modelos activados?

Pregunta 4

El gate de calidad usa la regla `accuracy_nuevo >= accuracy_anterior` para decidir si activar el modelo. Con esta regla, un modelo nuevo que obtiene exactamente el mismo accuracy que el anterior sí se activa. ¿Crees que eso es correcto o debería usarse > estricto? Argumenta tu respuesta pensando en qué ocurre cuando el dataset de entrenamiento crece con muestras de buena calidad pero el accuracy se estabiliza cerca del máximo

Antes de responder, conviene entender cómo funciona el gate de calidad en el código. Cuando se llama a `POST /train`, el servidor entrena un nuevo modelo con las muestras disponibles y evalúa su accuracy sobre un subconjunto de validación. Luego compara ese accuracy con el del modelo activo actual usando la condición:

```
model_updated = (previous_accuracy is None) or (accuracy_new >= previous_accuracy)
```

Si la condición se cumple, el nuevo modelo reemplaza al anterior y queda registrado como activado. Si no se cumple, el modelo se rechaza y el anterior sigue en producción. El operador `>=` significa que un modelo nuevo con **exactamente el mismo accuracy** que el anterior también se activa.

La pregunta te pide que reflexiones sobre si ese `>=` es la elección correcta, o si debería ser un `>` estricto que solo activara el modelo cuando hay una mejora real. Para razonarlo, piensa en qué ocurre en la práctica cuando el dataset de entrenamiento crece progresivamente con muestras de buena calidad: ¿llega un momento en que el accuracy deja de subir aunque los datos sigan mejorando? ¿Qué significaría entonces que dos versiones consecutivas obtengan el mismo accuracy: que son igualmente buenos, o que uno podría ser mejor en aspectos que la métrica no captura? ¿Qué consecuencias tendría para el sistema usar `>` estricto en ese escenario?

EJERCICIO DE PROGRAMACIÓN

Política de activación configurable

Idea central: el gate de calidad actual usa siempre la misma regla (\geq). En producción, distintos contextos requieren políticas distintas: a veces se acepta cualquier mejora mínima, otras veces se exige una mejora significativa, y otras se quiere activar el modelo solo si mejora en una clase específica problemática.

Fase de razonamiento previo (entregar antes del código)

Considera los siguientes tres escenarios hipotéticos. Elege y justifica qué política aplicarías en cada uno:

- *Escenario 1:* un modelo de detección de fraude bancario donde los falsos negativos (fraude no detectado) son mucho más costosos que los falsos positivos.
- *Escenario 2:* un modelo de recomendación de películas donde cualquier mejora marginal en accuracy es bienvenida porque el coste de un error es bajo.
- *Escenario 3:* un modelo médico de diagnóstico donde la estabilidad y previsibilidad del comportamiento importan tanto como la precisión.

A partir de ese razonamiento, el alumno debe diseñar en pseudocódigo —antes de tocar el código— la estructura de una clase `ActivationPolicy` que encapsule distintas políticas y describa qué parámetros necesitaría cada una.

Funcionalidad a implementar

Se extiende el request de `/train` con un campo `policy` que acepta tres modos:

- "any_improvement" (el \geq actual),
- "min_delta" (requiere mejorar al menos un porcentaje configurable, por ejemplo 2%), y
- "per_class_f1" (el modelo se activa solo si la F1-score de una clase específica mejora, aunque el accuracy global no lo haga).

El historial debe registrar qué política se usó en cada entrenamiento y por qué se activó o rechazó según esa política.

ENTREGABLES

1. Fichero `models/training_history.json` con al menos 4 versiones (mínimo 2 activadas, 2 rechazadas).
 2. Captura de pantalla o log de la ejecución de `demo_ct.py`.
 3. Carpeta `app-iris-ct/` completa con el código del ejercicio de programación desarrollado.
 4. Documento PDF (máx. 2 páginas) con las respuestas a las PREGUNTAS de reflexión.
-

Referencias

- [FastAPI Documentation](#)
- [scikit-learn Model Persistence](#)
- [Google MLOps: Continuous delivery and automation pipelines in ML](#)
- [Joblib documentation](#)