Laboratorio 9: AdaBoost y contenedores

Duración: 2 horas Formato: Competencia con contenedores Docker

Objetivo: Que los estudiantes implementen desde cero un modelo de **AdaBoost** con clasificadores débiles, comparen su desempeño con `sklearn.ensemble.AdaBoostClassifier` y empaqueten su solución en un **contenedor Docker** ejecutable localmente. El laboratorio concluye con una competencia basada en la **exactitud del modelo** y la **robustez de la API** expuesta desde el contenedor.

Estructura detallada de la sesión (2 horas)

Tiempo	Actividad
0:00–0:15	Repaso teórico de **boosting**, clasificadores débiles y la idea de AdaBoost.
0:15–0:25	**Kahoot 1**.
0:25–1:00	**Elemento 1 — Implementación desde cero de AdaBoost**.
1:00–1:10	**Kahoot 2**.
1:10–1:35	**Elemento 2 — Comparativa con** `AdaBoostClassifier`.
1:35–1:45	**Kahoot 3**.
1:45–2:05	**Elemento 3 — API + contenedor Docker + Makefile**.
2:05–2:15	**Kahoot 4**

Requisitos técnicos

- **Stack sugerido:** `python 3.10+`, `numpy`, `pandas`, `scikit-learn`, `FastAPI` o `Flask`, `uvicorn`.
- **Entorno de contenedores:** `Docker` instalado (Docker Desktop, Docker Engine o equivalente).
- **Automatización:** uso de `Makefile` para construir, ejecutar y detener el contenedor.
- **Repositorio local / GitHub:** estructura clara del proyecto:

```
/adaboost-container/
 model/
    adaboost_custom.py
    adaboost_sklearn.py
```

```
    preprocessor.pkl
    model.pkl

app/
    main.py
    requirements.txt

notebooks/
    np.ipynb

Dockerfile
Makefile

test_requests.py
README.md
```

- **Entregable final para evaluación:** un archivo `.tar.gz` (tarball) que contenga todo el repositorio:

  `equipo_<nombre>.tar.gz`

Elemento 1 — Implementación desde cero de AdaBoost

Objetivo: Comprender y programar el principio del algoritmo **AdaBoost (Adaptive Boosting)**, utilizando clasificadores débiles secuenciales para construir un modelo fuerte. El enfoque principal es aprender cómo el modelo ajusta los pesos de las observaciones en función de los errores previos, logrando una combinación ponderada de predictores débiles que reduce el sesgo del sistema.

1. Fundamento teórico

AdaBoost entrena de forma **secuecial** una serie de clasificadores débiles (por ejemplo, árboles de decisión de profundidad 1 o *decision stumps*). Cada clasificador h_t se ajusta sobre una distribución de pesos D_t que enfatiza los ejemplos mal clasificados por el modelo anterior.
El flujo matemático es el siguiente:

$$\epsilon_t = \sum_i D_t(i)\,[h_t(x_i) \neq y_i]$$

$$\alpha_t = \frac{1}{2}\ln\left(\frac{1-\epsilon_t}{\epsilon_t}\right)$$

$$D_{t+1}(i) = \frac{D_t(i)\,e^{-\alpha_t y_i h_t(x_i)}}{Z_t}$$

donde Z_t es un factor de normalización que garantiza que $\sum_i D_{t+1}(i) = 1$. La predicción final del modelo se obtiene mediante una **votación ponderada** de los clasificadores:

$$\hat{y}(x) = \text{sign}\left(\sum_{t=1}^{T} \alpha_t \, h_t(x)\right)$$

Este enfoque permite combinar múltiples modelos simples en un clasificador fuerte, reduciendo el **sesgo** y aumentando la capacidad predictiva del conjunto.

2. Lineamientos de implementación

Los estudiantes deberán crear una clase `SimpleAdaBoost` que imite el estilo de las clases de `scikit-learn`, con los métodos:

```
fit(X, y)      # Entrenamiento secuencial de clasificadores débiles
predict(X)     # Votación ponderada de predicciones
```

Se recomienda usar como clasificador base un `DecisionTreeClassifier` con `max_depth=1` (stump). El modelo deberá almacenar internamente los pesos α_t y las instancias entrenadas h_t.

3. Pseudocódigo orientativo

```
class SimpleAdaBoost:
    def __init__(self, n_estimators, base_estimator):

    def fit(self, X, y):
        # Entrenamiento secuecial de los modelos

    def predict(self, X):
        # Predicción de etiquetas
```

4. Subtareas prácticas

1. Implementar la clase `SimpleAdaBoost` siguiendo la estructura indicada.
2. Utilizar árboles de decisión con profundidad 1 como clasificadores débiles.
3. Registrar el error ϵ_t y el peso α_t en cada iteración.
4. Comparar visualmente cómo cambian los pesos de las observaciones en el tiempo.
5. Analizar cómo varía el desempeño del modelo al modificar `n_estimators`.

5. Preguntas teóricas

- ¿Qué significa que un clasificador sea "débil" dentro del contexto de AdaBoost?
- ¿Por qué los ejemplos mal clasificados ganan mayor peso en las iteraciones siguientes?
- ¿Qué efecto tiene aumentar el número de clasificadores débiles sobre el sesgo y la varianza?
- ¿Por qué AdaBoost es sensible al ruido y a los outliers?

6. Criterios de aceptación

- Implementación funcional y estable de la clase `SimpleAdaBoost`.
- Registro correcto de pesos y errores por iteración.
- Desempeño superior al de un clasificador débil individual.

- Código modular, legible y documentado.

Elemento 2 — Comparativa con `sklearn`

Objetivo: Evaluar la implementación propia de AdaBoost frente al modelo de referencia `sklearn.ensemble.AdaBoostClassifier`, analizando la precisión, estabilidad y convergencia del ensamble. El propósito es validar que la versión implementada capture correctamente el comportamiento adaptativo del algoritmo, así como comprender los efectos del número de clasificadores y la distribución de pesos.

1. Fundamento teórico

Ambos modelos comparten la misma base matemática; sin embargo, la implementación de `scikit-learn` incluye optimizaciones y tolerancias numéricas. El proceso comparativo permite evaluar:

- La tasa de convergencia del error con respecto al número de clasificadores (`n_estimators`).
- La diferencia en las fronteras de decisión entre la implementación propia y la de referencia.
- La sensibilidad del modelo a datos ruidosos o mal etiquetados.

El desempeño puede analizarse mediante métricas de clasificación tradicionales:

1. Accuracy
2. Precision
3. Recall
4. F1-Score

2. Lineamientos de implementación

1. Entrenar el modelo `SimpleAdaBoost` del Elemento 1.
2. Entrenar en paralelo el modelo de `scikit-learn` con los mismos hiperparámetros.
3. Calcular y comparar **accuracy**, **precision**, **recall** y **F1-score**.
4. Generar una gráfica de desempeño vs número de clasificadores.
5. Visualizar las fronteras de decisión en 2D para ambos modelos.

3. Subtareas prácticas

1. Evaluar ambos modelos con distintos valores de `n_estimators`.
2. Visualizar cómo disminuye el error de entrenamiento con cada iteración.
3. Comparar las fronteras de decisión y discutir sus diferencias.
4. Documentar los casos en que el modelo propio diverja o produzca resultados anómalos.
5. Registrar todas las observaciones y gráficas en el cuaderno final.

4. Preguntas teóricas

- ¿Qué ventajas ofrece el AdaBoost implementado en `scikit-learn` frente a la versión básica?
- ¿Por qué el número de clasificadores afecta la estabilidad del modelo?
- ¿Qué sucede si algunos pesos $D_t(i)$ tienden a cero o se concentran en pocos ejemplos?
- ¿Cómo se puede evitar el sobreajuste en AdaBoost?

5. Criterios de aceptación

- Comparación funcional entre el modelo propio y el de `scikit-learn`.
- Reporte de métricas claras y visualizaciones interpretables.
- Código reproducible, modular y libre de errores.
- Discusión teórica coherente y evidencia en el cuaderno final.

Elemento 3 — API, contenedor Docker y automatización con Makefile

Objetivo: Exponer el modelo de AdaBoost entrenado mediante una **API REST** simple, empaquetar la aplicación en un **contenedor Docker** ejecutable en local y automatizar las tareas principales (`build`, `run`, `stop`, `package`) con un **Makefile**. La meta es que el docente pueda evaluar cada solución extrayendo un `tar.gz`, ejecutando un par de comandos `make` y probando la API de forma reproducible.

1. Requisitos mínimos de la API

La API debe estar implementada en `FastAPI` o `Flask` (se recomienda `FastAPI`) y ubicarse en `app/main.py`. Se requieren al menos los siguientes endpoints:

- `/health` — Verificación de estado:

```
 /health
{
    "status": "ok"
}
```

- `/info` — Información del equipo y del modelo:

```
{
    "team": "Equipo_XYZ",
    "model": "AdaBoostClassifier",
    "base_estimator": "DecisionTreeClassifier(max_depth=1)",
    "n_estimators": 50,
    "preprocessing": {
        "pclass": ???,
        "sex": ???,
        "age": ???,
```

```
        "sibsp": ???,
        "parch": ???,
        "fare": ???,
        "embarked": ???
    }
}
```

- `/predict` — Endpoint de predicción. Debe recibir un JSON con las características de un único registro y devolver la predicción generada por el modelo entrenado. El formato de entrada debe respetar los nombres de las variables crudas establecidas para la práctica, de modo que todas las APIs sean compatibles con las pruebas automáticas.
  Ejemplo de solicitud:

  ```
   /predict
  {
      "features": {
          "pclass": 3,
          "sex": "male",
          "age": 22.0,
          "sibsp": 1,
          "parch": 0,
          "fare": 7.25,
          "embarked": "S"
      }
  }
  ```

  Ejemplo de respuesta:

  ```
  {
      "prediction": 1
  }
  ```

El servicio debe encargarse internamente de aplicar el mismo proceso de preprocesamiento y transformación que se utilizó durante el entrenamiento del modelo, incluyendo imputación, codificación y/o escalado de variables. La predicción no debe depender de que el cliente realice estas transformaciones antes de enviar los datos.
Se recomienda serializar y reutilizar los objetos de preprocesamiento (`encoder`, `scaler`, `imputer`, etc.) junto con el modelo principal, asegurando coherencia entre las fases de entrenamiento y despliegue. El detalle de las transformaciones esperadas y los ejemplos de implementación se describen en el **Apéndice D — Notas sobre Preprocesamiento y Escalado**.

2. Contenedor Docker (Dockerfile)

El proyecto debe incluir un archivo `Dockerfile` en la raíz del repositorio que defina la imagen necesaria para ejecutar la API. *Véase el Anexo B para la estructura sugerida.*

3. Makefile para automatizar `build/run/stop/package`

El repositorio debe incluir un archivo `Makefile` en la raíz, que permita automatizar las operaciones básicas. *Véase el Anexo C para la estructura sugerida.*

4. Flujo esperado para la evaluación (desde el punto de vista del docente)

El docente evaluará cada entrega siguiendo un flujo estándar:

1. Extraer el tarball recibido:

```
tar -xzvf equipo_<nombre>.tar.gz
```

2. Entrar al directorio raíz del proyecto (p.ej. `adaboost-container/`).
3. Construir la imagen y levantar el contenedor:

```
make build
make run
```

4. Probar manualmente los endpoints:

```
curl http://localhost:8000/health
curl http://localhost:8000/info
curl -X POST http://localhost:8000/predict \
     -H "Content-Type: application/json" \
     -d '{"features": [0.1, 1.2, -0.3, 2.4]}'
```

5. Detener y limpiar el contenedor:

```
make stop
make clean
```

5. Subtareas prácticas

1. Implementar la API en `app/main.py` con los endpoints `/health`, `/info`, `/predict`.
2. Serializar el modelo entrenado (`model.pkl`) e integrarlo en el flujo de predicción.
3. Crear un `Dockerfile` funcional que permita construir la imagen.
4. Crear un `Makefile` con las reglas `build`, `run`, `status`, `stop`, `clean` y `package`.
5. Generar el archivo `equipo_<nombre>.tar.gz`
6. Verificar que el flujo de evaluación funcione en otra máquina (o al menos en un entorno limpio).

6. Criterios de aceptación

- La API responde correctamente en `/health`, `/info` y `/predict`.
- El contenedor se construye sin errores ejecutando `make build`.
- El servicio se levanta y expone el puerto 8000 ejecutando `make run`.
- El modelo cargado en el contenedor corresponde al AdaBoost entrenado en los elementos anteriores.
- El `Makefile` funciona como se especifica y el tarball entregado es extraíble y reproducible.
- El `README.md` describe claramente cómo ejecutar y probar el proyecto.

Evaluación de la práctica

La calificación total del laboratorio se compone de la evaluación técnica (implementación, comparativa y despliegue) y del desempeño en la competencia. Cada elemento contribuye al puntaje final de acuerdo con los criterios de la siguiente tabla.

Criterio	**Ponderación**
Elemento 1 — Implementación desde cero de AdaBoost	40%
Elemento 2 — Comparativa con `sklearn`	20%
Elemento 3 — API, Docker y Makefile	30%
Presentación y claridad del notebook	10%
Total	**100%**

Competencia de desempeño (ranking)

Además de la evaluación base, se establecerá un **ranking de desempeño** que premiará la exactitud y robustez de los modelos expuestos.

Puesto	**1°**	**2°**	**3°**	**4°**	**5°**
Puntos adicionales	+100	+80	+60	+40	+20

El ranking se calculará en función de:

- Exactitud promedio (`accuracy`) sobre el conjunto de prueba.
- Correcto funcionamiento del endpoint `/predict`.
- Respuesta válida y sin errores HTTP (200 OK) durante la evaluación automática.

Bonus fuera de competencia

Los equipos podrán participar en pruebas opcionales de robustez y buenas prácticas para obtener puntos adicionales independientes del ranking. No son obligatorias y no afectan a quienes no participen.

Prueba bonus	**Puntos**
1. Imagen Docker menor a 300 MB	+10 pts
2. Endpoint adicional `/train` funcional para reentrenar	+10 pts
3. Visualización o log de evolución de pesos α_t durante entrenamiento	+10 pts
4. Makefile con variable de puerto configurable o script de pruebas automáticas (`make test`)	+10 pts
Total máximo posible	**+50 pts**

Nota: Los puntos de esta tabla son acumulables, y todos los equipo pueden acceder a ellos hasta el momento de la entrega de su prática.

Bonus por hitos en vivo

Durante la sesión se asignarán puntos adicionales en tiempo real a los equipos que cumplan primero ciertos logros:

Hito	**Bonus**
Primer equipo en crear el repositorio con la estructura completa de carpetas	+10 pts
Primer equipo en construir la imagen Docker sin errores	+10 pts
Primer equipo en tener el endpoint `/health` operativo	+10 pts
Primer equipo en devolver una predicción válida	+10 pts
Primer equipo en ejecutar `make run` con éxito	+10 pts
Primer equipo en entregar el `tar.gz` completamente funcional	+50 pts
Total máximo posible	**+100 pts**

Nota: Los puntos de los hitos se asignan de forma competitiva (el primero en cumplir gana). Estos bonus fomentan la agilidad, la organización del repositorio y la correcta ejecución del flujo Docker–Makefile–API.

Apéndice A — Recordatorio teórico de Boosting

- **Idea general:** El **boosting** es una técnica de ensamble que combina muchos modelos simples (llamados *clasificadores débiles*) para formar un modelo más preciso y robusto. En lugar de entrenarlos todos a la vez (como en el bagging o Random Forest), el boosting los entrena **de forma secuencial**, de modo que cada nuevo modelo intenta corregir los errores del anterior.

- **Clasificador débil:** Es un modelo con un desempeño apenas mejor que adivinar al azar, por ejemplo, un árbol de decisión muy simple con `max_depth=1` (también llamado *stump*). Aunque por sí solo no es muy potente, al combinar muchos de ellos con pesos adecuados se obtiene un modelo mucho más fuerte.

- **Reponderación adaptativa:** En cada iteración t, el algoritmo asigna un peso $D_t(i)$ a cada ejemplo del conjunto de entrenamiento. Los ejemplos que el modelo anterior clasificó mal reciben más peso, de modo que el siguiente clasificador se enfoca en ellos. Así, el modelo "aprende de sus errores" de manera sistemática.

- **Errores y pesos:** En cada paso, se calcula el error ponderado del clasificador actual:

$$\epsilon_t = \sum_i D_t(i)\,[h_t(x_i) \neq y_i],$$

  donde $h_t(x_i)$ es la predicción del clasificador débil y y_i la etiqueta real. Luego se calcula la importancia (α_t) de ese clasificador según su desempeño:

$$\alpha_t = \tfrac{1}{2} \ln\left(\frac{1-\epsilon_t}{\epsilon_t}\right)$$

  Finalmente, los pesos de los ejemplos se actualizan para la siguiente ronda:

$$D_{t+1}(i) = \frac{D_t(i) e^{-\alpha_t y_i h_t(x_i)}}{Z_t},$$

  donde Z_t es un factor de normalización para que los pesos sigan sumando 1.
- **Combinación final:** Tras entrenar T clasificadores débiles, el modelo final realiza una votación ponderada:

$$\hat{y}(x) = \text{sign}\left(\sum_{t=1}^{T} \alpha_t h_t(x)\right)$$

  Los clasificadores con menor error (α_t grande) influyen más en la predicción final.
- **Interpretación intuitiva:**
  - Cada árbol intenta "aprender de los errores" del anterior.
  - Los ejemplos difíciles se vuelven más importantes en el entrenamiento.
  - Los pesos α_t actúan como un "voto de confianza" hacia cada modelo.
  - Al combinar modelos secuenciales, el ensamble reduce el sesgo sin incrementar demasiado la varianza.

Anexo B — Guía práctica del archivo `Dockerfile`

Este archivo define el entorno de ejecución de la aplicación y permite construir una imagen reproducible del proyecto. Debe ubicarse en la raíz del repositorio y contener, al menos, los siguientes pasos:

```
FROM python:<version>

# 1. Establecer directorio de trabajo dentro del contenedor
WORKDIR /app

# 2. Copiar el contenido del repositorio local
COPY . .

# 3. Instalar las dependencias necesarias
RUN pip install --no-cache-dir -r app/requirements.txt

# 4. Exponer el puerto donde correrá el servicio
EXPOSE <port>

# 5. Definir el comando de inicio
CMD ["uvicorn", "app.main:app", "--host", "0.0.0.0", "--port", "<port>"]
```

Recomendaciones:

- Verificar que el modelo (`model.pkl`) y el script principal (`app/main.py`) se copien correctamente.
- Probar localmente con:

  ```
  docker build -t adaboost-api .
  docker run -d -p 8000:8000 adaboost-api
  ```

- Validar el correcto funcionamiento de los endpoints antes de generar el `tar.gz`.

Comando útil para inspeccionar la imagen:

```
docker images | grep adaboost
```

Anexo C — Guía práctica del archivo `Makefile`

El `Makefile` es una herramienta de automatización que permite definir "recetas" para ejecutar tareas comunes con comandos simples. En este laboratorio, se usa para construir, ejecutar y empaquetar el contenedor del modelo AdaBoost.

Estructura base sugerida:

```
IMAGE_NAME=adaboost-api
CONTAINER_NAME=adaboost-container
PORT=8000
TEAM_NAME=equipo_demo   # Cambiar por el identificador del equipo

build:
    # Comando para construir el contenedor definido en el dockerfile

run:
    # Comando para ejecutar el contenedor construido

status:
    # Comando para obtener el status del contenedor una vez ejecutado

stop:
    # Comando para detener (si está en ejecución)
    # Comando para eliminar el contenedor especificado

clean:
    #  Comando limpiar recursos no utilizados de Docker

package:
    # Comando para crea un paquete comprimido (.tar.gz) del proyecto, excluyendo archivo
     .
```

Convención de entrega: El archivo generado por la regla `package` debe renombrarse (o configurarse) para seguir el formato:

`equipo_<nombre>.tar.gz`

Por ejemplo: equipo_7B.tar.gz, equipo_AdaBoost1.tar.gz, etc.

Funciones esperadas:

- `make build` — construye la imagen Docker.
- `make run` — levanta el contenedor en segundo plano.
- `make status` — verifica si el contenedor está corriendo.
- `make stop` — detiene y elimina el contenedor.
- `make clean` — limpia recursos y capas huérfanas.
- `make package` — genera el archivo `tar.gz` con todo el proyecto.

Convenciones de entrega:

- El archivo resultante debe nombrarse:

  ```
  equipo_<nombre>.tar.gz
  ```

- El flujo de evaluación se basará en los comandos:

  ```
  make build
  make run
  curl http://localhost:8000/health
  make stop
  ```

- Los equipos deben verificar que el tarball funcione en un entorno limpio antes de enviarlo.

Tip adicional:

- Se pueden definir variables de entorno, por ejemplo:

  ```
  export TEAM_NAME=equipo7B
  ```

  lo que permite generar automáticamente el archivo `equipo7B.tar.gz` con `make package`.

Apéndice D — Notas sobre Preprocesamiento y Escalado

Durante el desarrollo de este laboratorio, cada equipo debe asegurarse de que la API realice internamente el mismo proceso de preprocesamiento que fue aplicado durante el entrenamiento del modelo. Esto garantiza la coherencia entre las fases de entrenamiento y predicción, evitando errores por discrepancias en los formatos de entrada.

1. Variables que requieren codificación

Las siguientes variables poseen valores categóricos o discretos, por lo que deben ser transformadas a formato numérico antes de alimentar el modelo

La transformación debe reproducir exactamente el método utilizado en el entrenamiento. Si se guardó un codificador (por ejemplo, `encoder.pkl`), este debe cargarse y aplicarse dentro del endpoint `/predict`.

2. Variables numéricas y escalado

Las variables numéricas continuas pueden requerir normalización o estandarización, dependiendo del enfoque de cada equipo. Es fundamental usar el mismo objeto de escalado que se utilizó para entrenar el modelo.

3. Imputación de valores faltantes

En el conjunto de entrenamiento pueden existir valores faltantes, por lo tanto, cada equipo debe aplicar una estrategia coherente de imputación.
La API debe cargar y reutilizar los mismos objetos de imputación utilizados en el entrenamiento, garantizando que los datos recibidos a través del endpoint `/predict` sean procesados de forma idéntica antes de realizar la predicción.

4. Flujo sugerido dentro del endpoint `/predict`

```
# Carga del preprocesador y modelo
preprocessor = joblib.load("model/preprocessor.pkl")
model = joblib.load("model/adaboost_model.pkl")

@app.post("/predict")
def predict(data: dict):
    # 1. Convertir las características crudas en DataFrame

    # 2. Aplicar el preprocesamiento completo

    # 3. Generar la predicción

    return {"prediction": int(y_pred)}
```

Este flujo ejemplifica el principio central del laboratorio: **el modelo no debe depender de que el cliente realice el preprocesamiento**. La responsabilidad de transformar los datos crudos recae completamente en la API, asegurando así su robustez y reproducibilidad.