Parte 4 — MLOps — Modelos en Producción

14 clases · Parte 4 del programa

Parte 4 — MLOps — Modelos en Producción

> *14 clases · bundle consolidado del currículo v3.*

Índice de clases

Clase 194 — Clase 194 — Versionado de datos con DVC

> *Parte: 4 — MLOps · Fuente: Huyen, Designing Machine Learning Systems cap. 6 + docs oficiales DVC 3.x. Duración estimada: 75 min.*

Objetivo

Versionar datasets pesados (>100 MB, que git rechaza) con DVC 3.x, separando qué dato se usó (puntero en git, ~200 bytes) de dónde vive el blob real (S3, GCS, Azure, disco local). Reproducir un entrenamiento de hace 3 meses con git checkout <sha> && dvc pull y entender por qué dvc.lock es a los datos lo que package-lock.json es a npm.

Resultados de aprendizaje

Al finalizar, el estudiante podrá:

- Inicializar un repo con dvc init y rastrear un dataset con dvc add data/raw.csv (entiende que git solo guarda el .dvc pointer).
- Configurar un remote (dvc remote add -d origin s3://bucket/path) y sincronizar con dvc push / dvc pull.
- Construir un pipeline reproducible declarando stages en dvc.yaml (deps, outs, params, metrics) y ejecutarlo con dvc repro.
- Comparar experimentos con dvc exp run, dvc exp show, dvc exp diff — alternativa ligera a MLflow para casos simples.
- Diagnosticar ERROR: output 'X' is already tracked by SCM y otros choques DVC ↔ git.

Temas

#	Tema	Por qué importa
1	El problema: git LFS no escala a TB	Costo por GB, throughput, y bloqueo en git
2	Modelo mental DVC: pointer en git + blob e	Git versiona el hash MD5/SHA-256; el dato
3	dvc add vs dvc.yaml stages	Trackeo manual vs pipeline declarativo.
4	Remotes (S3, GCS, Azure, SSH, local)	Donde realmente está la data; dvc remote m
5	dvc.lock — el "lockfile" de tu pipeline	Hashes congelados de inputs/outputs por st
6	dvc exp — branching-less experiments	Ejecutar 20 corridas sin ensuciar git log.

Definiciones y características

- DVC (Data Version Control): herramienta open-source que extiende git para datos/modelos. Versión actual: 3.x (rompió compat con 2.x — dvc.yaml mismo, pero comandos exp cambiaron).
- .dvc file: archivo YAML pequeño (~200 B) con md5, size, path del blob. Va a git. El blob va al remote.
- Cache local (.dvc/cache/): copia local de los blobs. dvc add mueve el archivo al cache y deja un link (reflink/hardlink/symlink) en el working tree. Por eso dvc add no duplica espacio en disco (en filesystems que soportan reflinks: APFS, XFS, btrfs, ReFS).
- Remote: backend remoto (S3/GCS/Azure/SSH/HTTP/local-path). dvc push sube cache local → remote; dvc pull baja remote → cache → working tree.
- Stage (en dvc.yaml): unidad reproducible. Tiene cmd, deps (inputs), outs (outputs), opcionalmente params (de params.yaml) y metrics. DVC re-ejecuta solo los stages cuyos hashes de deps cambiaron — como make, pero por contenido en vez de timestamp.
- dvc.lock: hashes congelados de deps/outs después de la última dvc repro exitosa. Va a git. Es lo que hace reproducible el pipeline.
- dvc exp run: ejecuta el pipeline con (opcionalmente) parámetros distintos y guarda el resultado como "experimento" — un commit interno que no contamina git log hasta que hagás dvc exp apply o dvc exp branch.

Dataset / recursos

- Dataset: seaborn.load_dataset('titanic') exportado a data/raw/titanic.csv (~60 KB — pequeño a propósito para que la clase corra sin S3 real).
- Remote demo: directorio local /tmp/dvc-remote-demo (simula S3 sin credenciales). El comando para S3 real se muestra pero no se ejecuta.
- Librerías: dvc[s3] (o dvc pelado si remote local), pandas, scikit-learn.

Ejercicios

1. Setup mínimo: inicializá un repo git + DVC (git init && dvc init). Generá data/raw/titanic.csv, hacelo trackear con dvc add data/raw/titanic.csv. Inspeccioná el .dvc resultante con cat data/raw/titanic.csv.dvc y entendé los campos md5, size, path.
2. Remote local: configurá un remote en /tmp/dvc-remote-demo con dvc remote add -d local /tmp/dvc-remote-demo. Hacé dvc push y verificá con ls /tmp/dvc-remote-demo/files/md5/ que el blob aparece como <2-char-prefix>/<resto-del-hash>.
3. Pipeline declarativo: creá dvc.yaml con dos stages — prepare (lee raw/titanic.csv, elimina nulos, escribe data/processed/clean.csv) y train (entrena un LogisticRegression, escribe model.pkl y metrics.json). Corré dvc repro y observá dvc.lock.
4. Reproducción: tocá un parámetro en params.yaml (test_size: 0.2 → 0.3). Volvé a correr dvc repro y verificá que ambos stages se re-ejecutan (porque prepare no depende de params, pero train sí — y el

output de train cambió). Compará con dvc repro --dry.

5. Experimentos sin branching: dvc exp run -S 'train.C=0.1' tres veces con valores distintos de C. Listalos con dvc exp show. Aplicá el mejor con dvc exp apply <hash>.

Homework verificable

Pipeline DVC en un repo nuevo con:

1. dvc.yaml con stages prepare → train → evaluate.
2. params.yaml con al menos test_size, random_state, model.C.
3. metrics.json con accuracy y f1 que se reporten con dvc metrics show.
4. Tres corridas dvc exp run variando model.C {0.01, 1, 100}.
5. Output de dvc exp show --no-pager adjunto como experiments.txt.

Criterio de aceptación: dvc repro corre limpio desde cero (rm -rf .dvc/cache data/processed model.pkl metrics.json && dvc pull && dvc repro) y produce los mismos hashes en dvc.lock.

Errores comunes

Síntoma / mensaje	Causa y cómo arregl				
ERROR: output 'data/r	El archivo ya está en g				
dvc push no hace nad	El cache local y el rem				
dvc repro re-ejecuta to	Modificaste un archivo				
Pierdo el cache y dvc	Probablemente el rem				
Git push lentísimo des	Alguien hizo git add da	sort -u \	xargs -I{} du -h {} 2>/d	sort -h \	tail. Después git filter-repo` o BFG pa

Preguntas frecuentes

¿DVC vs git LFS?

LFS es más simple (un binario, git lfs track "*.csv") pero (1) cobra por GB en GitHub Enterprise/GitLab, (2) bloquea git push hasta que el blob suba, (3) no tiene noción de pipeline reproducible ni de experimentos. DVC desacopla storage del provider git y agrega dvc.yaml/dvc.lock/dvc exp. Para datasets <1 GB y casos triviales, LFS alcanza; para ML serio, DVC.

¿DVC vs MLflow?

Son complementarios, no competidores. DVC versiona datos + pipeline (lado git). MLflow trackea runs (params, metrics, artifacts) y registra modelos en un model registry (Clase 195). En la práctica: DVC para reproducibilidad determinística del pipeline; MLflow para comparar 500 experimentos en un dashboard.

¿Tengo que usar S3? ¿Funciona en una notebook personal?

No, funciona con remote local (dvc remote add -d local /path/a/carpeta). Útil para aprender, o para equipos chicos con un NAS compartido. Para producción real con varios colaboradores: S3/GCS/Azure.

¿Qué pasa con datos sensibles (PII)?

DVC no encripta por sí mismo. Si el remote es S3 con SSE-KMS, ya está encriptado en reposo. Para PII fuerte: bucket privado + IAM rol por equipo + auditoría con CloudTrail. No subas datos con PII a un remote público (S3 público, GCS público) — DVC no te avisa.

¿dvc.lock lo edito a mano?

No. Lo regenera dvc repro. Editarlo manualmente rompe la garantía de reproducibilidad. Si necesitás "forzar"

un hash, usá dvc commit (con cuidado — saltea la ejecución).

¿Funciona con Jupyter notebooks?

Sí, pero el notebook en sí sigue versionado por git. DVC entra cuando un cell genera/lee artefactos pesados. Patrón común: notebook orquesta, pero las funciones heavy van a un src/, y los datasets están bajo DVC. (Para diff limpio de notebooks: nbdime o jupytext — fuera del scope de esta clase.)

Referencias

- Huyen, Chip. Designing Machine Learning Systems (O'Reilly, 2022), cap. 6 — Model Development and Offline Evaluation, sección sobre data versioning.
- DVC 3.x documentation — empezar por Get Started.
- DVC dvc.yaml reference — todos los campos de stage.
- Iterative Studio — UI hosted opcional para visualizar pipelines DVC (no requerida, gratis para repos públicos).
- Comparativa práctica DVC vs LFS vs LakeFS: Data versioning landscape 2024.

Material descargable

- Guía explicativa (PDF) — versión imprimible con todo el contenido de la clase.
- Presentación (PPTX) — deck PowerPoint listo para proyectar en clase.
- Notebook ejecutable (.ipynb) — abrilo desde el laboratorio del programa o desde Jupyter.

Clase 195 — Clase 195 — Versionado de modelos y experimentos con MLflow

> *Parte: 4 — MLOps · Fuente: Huyen, Designing Machine Learning Systems cap. 6 + 11 + docs MLflow 2.x. Duración estimada: 75 min.*

Objetivo

Trackear experimentos de ML (parámetros, métricas, artefactos, código) con MLflow Tracking, registrar modelos en el Model Registry con stages (None → Staging → Production → Archived), y entender la diferencia conceptual con DVC: DVC versiona el pipeline; MLflow versiona los runs.

Resultados de aprendizaje

Al finalizar, el estudiante podrá:

- Levantar un servidor MLflow local (mlflow ui o mlflow server) y logear runs con mlflow.start_run() + log_param/log_metric/log_artifact.
- Usar autolog (mlflow.sklearn.autolog()) para que params/metrics se capturen sin código boilerplate.
- Registrar un modelo en el Model Registry (mlflow.register_model) y transicionarlo de Staging a Production con la API o la UI.
- Cargar un modelo registrado en producción con mlflow.pyfunc.load_model("models:/MiModelo/Production").
- Configurar un backend store (PostgreSQL) y un artifact store (S3) para uso en equipo.

Temas

#	Tema	Por qué importa
1	Tracking server: backend store + artifact	Dónde van los metadatos vs los blobs (mode

2	Runs, experiments, tags	Unidad atómica + agrupación + búsqueda.
3	log_param, log_metric, log_artifact, log_m	Las 4 primitivas.
4	Autolog por framework (sklearn, PyTorch, X	Cero boilerplate cuando funciona — y sus l
5	Model Registry + stages	El camino entre "entrené esto" y "esto sir
6	MLflow Models flavor (pyfunc, sklearn, pyt	Un mismo modelo se puede cargar en N runti

Definiciones y características

- MLflow Tracking: API + UI para registrar runs (params, metrics, artifacts, código source). Cada run pertenece a un experiment (agrupación lógica, ej. "fraud-detection-v2").
- Backend store: dónde se guardan los metadatos del run (params, metrics, tags). Opciones: filesystem (default ./mlruns), SQLite, PostgreSQL, MySQL. Para equipo: Postgres.
- Artifact store: dónde se guardan los blobs (modelos, plots, datasets exportados). Opciones: filesystem local, S3, GCS, Azure Blob, HDFS. Para equipo: S3/GCS.
- Run: ejecución atómica de un experimento. Tiene un run_id (UUID), start_time, end_time, status, y N params/metrics/tags/artifacts.
- Autolog: hook que intercepta llamadas al framework (sklearn, XGBoost, PyTorch Lightning, …) y registra params/metrics sin código. Activar con mlflow.<framework>.autolog() antes del .fit().
- Model Registry: catálogo central de modelos versionados. Cada modelo registrado tiene versiones (v1, v2, …), cada versión tiene un stage (None/Staging/Production/Archived) y aliases (desde MLflow 2.5+: @champion, @challenger).
- MLflow Models: formato estándar para guardar un modelo. Define un MLmodel YAML con uno o más flavors (sklearn, pyfunc, pytorch). pyfunc es el flavor universal — cualquier modelo se sirve como predict(input_df).

Dataset / recursos

- Dataset: California Housing (sklearn.datasets.fetch_california_housing) — 20 640 filas, regresión, sin problemas de PII.
- Backend local: SQLite (mlflow server --backend-store-uri sqlite:///mlflow.db).
- Artifact local: ./mlruns/artifacts.
- Librerías: mlflow>=2.10, scikit-learn, xgboost.

Ejercicios

1. Tracking manual: entrená un LinearRegression y un RandomForestRegressor sobre California Housing. Para cada uno, abrí un run y logueá n_estimators (si aplica), max_depth, rmse_train, rmse_test. Comparalos en la UI (mlflow ui --port 5000).
2. Autolog: repetí el ejercicio anterior con mlflow.sklearn.autolog() y verificá que params + metrics + el modelo entero quedaron registrados sin código extra.
3. Sweep de hiperparámetros: corré 10 runs variando max_depth {3, 5, 10, 15, 20} y n_estimators {50, 200}. Usá mlflow.search_runs() para encontrar el mejor por rmse_test.
4. Registry: registrá el mejor modelo (mlflow.register_model(model_uri, "housing-rf")). Transicionalo a Staging, después a Production desde la API (MlflowClient.transition_model_version_stage).
5. Carga en "producción": en una celda nueva, simulá un servicio que carga el modelo Production y predice una fila: model = mlflow.pyfunc.load_model("models:/housing-rf/Production"); model.predict(X_one).

Homework verificable

Notebook + carpeta mlruns/ que contenga:

1. ≥ 10 runs en un experimento llamado homework-195.
2. Cada run con al menos model_type, rmse_test, r2_test, y el modelo logueado (mlflow.sklearn.log_model).
3. Un modelo registrado como housing-best con al menos una versión en stage Production.
4. Una celda final que carga models:/housing-best/Production y predice sobre 5 filas de test.

Criterio de aceptación: mlflow.search_runs(experiment_names=["homework-195"], order_by=["metrics.rmse_test ASC"]) devuelve la fila top, y su run_id coincide con la versión Production del modelo registrado.

Errores comunes

Síntoma / mensaje	Causa y cómo arreglar
Los runs van a ./mlruns y no aparecen en l	La UI está leyendo otro --backend-store-ur
mlflow.sklearn.autolog() no registra el mo	El modelo se loguea solo si el .fit() ocur
ConnectionRefusedError contra el tracking	El MLFLOW_TRACKING_URI apunta a un server
Artefactos vacíos en la UI cuando uso S3	El cliente no tiene credenciales AWS. Fix:
RestException: RESOURCE_ALREADY_EXISTS al	Ya existe un modelo con ese nombre. Fix: u
El stage Production está deprecated en log	Desde MLflow 2.9+ se recomienda aliases (@

Preguntas frecuentes

¿MLflow vs Weights & Biases vs Neptune?

MLflow es open-source, self-hosted, y el estándar de facto en empresas que no quieren un SaaS externo. W&B y Neptune tienen mejores UIs y features colaborativos (reports, sweeps integrados), pero son SaaS con costo por usuario. Para aprender y para deploys self-hosted: MLflow. Para equipos de research grandes con presupuesto: W&B.

¿MLflow vs DVC?

Resuelven cosas distintas. DVC (Clase 194): versionado de datos + pipeline reproducible (vive en el repo git). MLflow: tracking de runs + model registry (vive en un server). Se usan juntos: el dvc.yaml define cómo correr el pipeline; dentro del script, MLflow logea cada run.

¿Debo usar mlflow server o mlflow ui?

mlflow ui es la UI sobre un backend filesystem (local). mlflow server levanta también una REST API para que otros procesos/máquinas registren runs remotamente. Para equipo: server. Para vos solo: ui alcanza.

¿pyfunc o el flavor nativo (sklearn, pytorch)?

Si vas a cargar el modelo desde Python y usar features específicas (acceso a .coef_, .feature_importances_): flavor nativo. Si vas a servirlo detrás de una API REST o desde otro lenguaje: pyfunc — abstrae todo a predict(DataFrame) → array.

¿Qué pasa si pierdo mlflow.db?

Perdés metadatos (params, metrics, run history). Los artefactos sobreviven si están en S3. Backup: pg_dump si usás Postgres, o copiar mlflow.db si SQLite. En producción: backend Postgres con backups gestionados por la nube.

¿Cómo versionar el código junto al run?

MLflow registra automáticamente el commit git (mlflow.source.git.commit tag) si corrés desde un repo git

limpio. Si tenés cambios sin commitear, el tag dice dirty. Política sana: el CI registra runs solo desde commits firmados.

Referencias

- Huyen, Chip. Designing Machine Learning Systems (O'Reilly, 2022), cap. 6 (experimentos) y cap. 11 (model serving).
- MLflow Docs — Tracking + Model Registry + Models.
- mlflow.sklearn.autolog — qué se captura automáticamente.
- MLflow Model Registry — stages, aliases (2.5+), webhooks.
- Comparativa MLflow vs W&B vs Neptune (2024) — tomar con grano de sal (parte interesada).

Material descargable

- Guía explicativa (PDF) — versión imprimible con todo el contenido de la clase.
- Presentación (PPTX) — deck PowerPoint listo para proyectar en clase.
- Notebook ejecutable (.ipynb) — abrilo desde el laboratorio del programa o desde Jupyter.

Clase 196 — Clase 196 — Feature stores (Feast)

> *Parte: 4 — MLOps · Fuente: Huyen cap. 6 + Feast docs 0.40+. Duración estimada: 70 min.*

Objetivo

Resolver el problema más caro de ML en producción —training/serving skew: que las features que ve el modelo en producción sean distintas a las que vio en training— centralizando definiciones de features en un feature store. Usar Feast para definir entidades + feature views, materializar al online store (Redis/SQLite), y servir features con get_online_features en <10 ms.

Resultados de aprendizaje

Al finalizar, el estudiante podrá:

- Explicar la diferencia entre offline store (parquet/BigQuery, para training) y online store (Redis/DynamoDB, para serving low-latency).
- Definir Entity, FeatureView, FileSource y registrar el repo con feast apply.
- Generar un training dataset point-in-time correct con get_historical_features (evita data leakage temporal).
- Materializar features (feast materialize-incremental) y consumirlas en serving con get_online_features.
- Reconocer cuándo NO usar feature store (ML con <5 features estables, equipo de 1, datos batch puro).

Temas

#	Tema	Por qué importa
1	Training/serving skew — el bug más caro de	Modelo bueno en offline, malo en producció
2	Offline vs online store	Latencia distinta, datos idénticos (en teo
3	Entity + FeatureView + FileSource	Modelo de datos de Feast.
4	Point-in-time joins	Mismo timestamp para feature y label → sin
5	materialize / materialize-incremental	Offline → online en batch programado.
6	Feast vs construir uno propio	Cuándo justifica la dependencia.

Definiciones y características

- Feature store: sistema que (1) define features una vez y las sirve en training + producción con garantía de equivalencia, (2) provee point-in-time correctness para evitar leakage, (3) baja latencia en serving.
- Training/serving skew: la causa #1 de modelos que rinden bien en CV y mal en producción. Surge cuando el pipeline de feature engineering offline y online divergen (lenguajes distintos, librerías distintas, bugs distintos).
- Offline store: dónde viven los datos históricos para training. Feast soporta File (parquet local), BigQuery, Snowflake, Redshift, Spark.
- Online store: dónde viven los valores actuales para serving low-latency. Soporta SQLite (dev), Redis, DynamoDB, Bigtable, Postgres.
- Entity: dimensión de negocio sobre la que se computan features (ej. user_id, driver_id, product_sku). Tiene un nombre y un tipo (String, Int64).
- FeatureView: agrupa features que (1) vienen de la misma fuente y (2) se computan para la misma entidad. Define ttl (cuánto vive el feature antes de ser stale).
- Point-in-time join: dado (entity_id, event_timestamp) para cada fila del training set, Feast devuelve el valor del feature vigente en ese timestamp — no el más reciente. Evita meter información del futuro en training.
- Materialize: copia features de offline → online. materialize-incremental solo lo que cambió desde la última corrida.

Dataset / recursos

- Dataset: drivers de un servicio tipo Uber — driver_id, event_timestamp, conv_rate, acc_rate, avg_daily_trips. Generado sintéticamente en el notebook.
- Offline store: parquet local en data/.
- Online store: SQLite local en data/online_store.db.
- Librerías: feast>=0.40, pandas, pyarrow.

Ejercicios

1. Setup mínimo: feast init driver_repo. Inspeccioná feature_store.yaml, example_repo.py. Corré feast apply y verificá feast feature-views list.
2. Training dataset histórico: armá un entity_df con driver_id y event_timestamp para 5 momentos distintos del día. Pedile a Feast el feature driver_hourly_stats:conv_rate con get_historical_features. Confirmá manualmente que el valor devuelto es el último anterior al timestamp pedido (no el futuro).
3. Materialización + serving: feast materialize-incremental $(date +%Y-%m-%d). Después: store.get_online_features(features=['driver_hourly_stats:conv_rate'], entity_rows=[{'driver_id': 1001}]).to_dict(). Medí latencia con %timeit (debería ser <2 ms).
4. TTL en acción: configurá ttl=timedelta(days=1). Materializá datos viejos de 3 días atrás. Pedí features online → debería devolver None (porque expiró). Cambiá ttl=timedelta(days=7) y reintentá.
5. Skew check: comparé el feature offline (parquet) y el online (SQLite) para el mismo driver_id. Si difieren después de materialize, hay un bug.

Homework verificable

Repo Feast con:

1. Una Entity customer_id y una segunda merchant_id.
2. Tres FeatureView: customer_stats (avg_purchase, n_purchases_7d), merchant_stats (avg_rating, n_disputes_30d), transaction_features (amount, hour_of_day).
3. Un script build_training.py que arma el training set point-in-time correct para una tarea de fraud

detection.

4. Un script serve.py que, dado un (customer_id, merchant_id), devuelve el vector de features listo para el modelo.
5. README del repo Feast con instrucciones para feast apply + materialize + serve.

Criterio de aceptación: el vector de features que devuelve serve.py para un (customer_id, merchant_id, timestamp) coincide exactamente con la fila correspondiente del training set generado por build_training.py (mismo customer/merchant/timestamp).

Errores comunes

Síntoma / mensaje	Causa y cómo arreglar
get_historical_features devuelve NaN para	El event_timestamp del entity_df está fuer
get_online_features devuelve None aunque a	TTL expirado, o materializaste hasta now()
feast apply falla con No module named exam	El cwd no es el feature_repo/. Feast carga
Training tarda muchísimo con offline=BigQu	Feast no agrega filtros por timestamp en e
Mi modelo en producción usa features disti	Probablemente tu pipeline de transformació

Preguntas frecuentes

¿Necesito un feature store?

No siempre. Si: ≥3 modelos comparten features, equipo ≥5 personas, requerís serving <50 ms, o tenés bug recurrente de training/serving skew → sí. Si: 1 modelo, 1 persona, batch scoring nocturno → es overkill. Empezá sin feature store y migrá cuando duela.

¿Feast vs Tecton vs Hopsworks?

Feast es open-source, self-hosted, no cobra. No hace compute (vos generás los parquets/tablas). Tecton y Hopsworks son SaaS managed que incluyen compute (definís transformaciones SQL/Python y ellos las corren). Para empezar: Feast. Para empresas grandes con presupuesto y muchas features: Tecton.

¿Feast hace feature engineering?

No por default. Feast sirve features — no las computa. Las OnDemandFeatureView (request-time) y las Stream Feature Views (con Spark) le agregan compute, pero el patrón principal es: vos generás parquet, Feast lo sirve.

¿Por qué point-in-time joins son tan importantes?

Porque la alternativa naive (LEFT JOIN ... ON entity_id) trae el valor más reciente del feature — que pudo ocurrir después del label. Ejemplo: predecís churn del usuario X el 1 de marzo, pero la feature n_logins_7d la tomás de hoy → metés información del futuro en training, y el modelo "predice" perfecto en CV y rompe en prod.

¿SQLite online store sirve para producción?

No: 1 escritor, no es multi-host. Para producción usá Redis (más común), DynamoDB (si ya estás en AWS), o Bigtable (si estás en GCP). SQLite es para desarrollo local y tests.

¿Cómo testear que no hay skew?

Job de comparación periódica: tomar N entities random, computar features offline (mismo código que el training) y online (vía Feast), assertear igualdad con tolerancia. Si falla → alerta. Es el equivalente de "smoke test" para feature stores.

Referencias

- Huyen, Chip. Designing Machine Learning Systems (O'Reilly, 2022), cap. 6 — sección Feature Engineering y Feature Store.
- Feast docs — empezar por Quickstart.
- Point-in-time joins explained (Feast blog) — el por qué con ejemplos.
- Tecton vs Feast (2024 comparison) — vendor-biased, leer con criterio.
- MLOps at Reasonable Scale (Tagliabue, 2022) — cuándo NO usar feature store.

Material descargable

- Guía explicativa (PDF) — versión imprimible con todo el contenido de la clase.
- Presentación (PPTX) — deck PowerPoint listo para proyectar en clase.
- Notebook ejecutable (.ipynb) — abrilo desde el laboratorio del programa o desde Jupyter.

Clase 197 — Clase 197 — CI/CD para ML con GitHub Actions

> *Parte: 4 — MLOps · Fuente: Huyen cap. 10 + docs GitHub Actions + CML.dev. Duración estimada: 80 min.*

Objetivo

Automatizar el ciclo lint → test → entrenar → evaluar → comparar → desplegar con GitHub Actions. Cada PR dispara entrenamiento sobre el slice actual de datos, postea métricas como comentario, y bloquea merge si la métrica empeoró >X%. Sin CI/CD, "el modelo nuevo es mejor" es opinión; con CI/CD, es un workflow check o .

Resultados de aprendizaje

Al finalizar, el estudiante podrá:

- Escribir un workflow .github/workflows/ml.yml con jobs lint, test, train, evaluate.
- Usar CML (Continuous Machine Learning) para postear plots/metrics en el comentario del PR.
- Cachear dependencias (actions/setup-python con cache: pip) y pesos de modelos (actions/cache) para que el CI no tarde 20 min.
- Usar secrets y OIDC para deploy seguro a AWS/GCP sin claves long-lived.
- Configurar branch protection + required checks que bloqueen merge si las métricas degradan.

Temas

#	Tema	Por qué importa
1	Anatomía de un workflow: triggers, jobs, s	Vocabulario base.
2	Matrix builds (strategy.matrix)	Probar 3 versiones de Python × 2 de PyTorc
3	Caché de pip + datasets pesados	CI <5 min vs CI >30 min.
4	CML: reportar métricas en PR	Trae al code review los números, no solo e
5	OIDC para deploy (sin secret long-lived)	Federation con AWS/GCP/Azure.
6	Branch protection + required status checks	Convierte el lint/test/eval en gate, no en

Definiciones y características

- Workflow: archivo YAML en .github/workflows/*.yml. Disparado por on: (push, pull_request, schedule,

workflow_dispatch).
- Job: agrupación de steps que corren en el mismo runner (VM/container). Jobs corren en paralelo por default; usar needs: [job1] para serializar.
- Step: comando individual. Puede ser shell (run:) o una action reutilizable (uses: actions/setup-python@v5).
- Runner: máquina donde corren los jobs. ubuntu-latest (gratis para repos públicos, 2 vCPU/7 GB RAM), windows-latest, macos-latest, o self-hosted (GPU, recursos custom).
- CML (Continuous Machine Learning): CLI de Iterative.ai que postea comentarios en PR con tablas/plots desde el workflow. Hace que las métricas sean visibles al reviewer sin abrir Actions tab.
- OIDC (OpenID Connect): GitHub emite un token efímero firmado que AWS/GCP/Azure aceptan para asumir un rol. Reemplaza guardar AWS_ACCESS_KEY_ID como secret. Una vulnerabilidad menos.
- Required status checks: configurable en Settings → Branches → Branch protection. Lista de jobs cuyo nombre debe terminar en para permitir merge.

Dataset / recursos

- Modelo del ejemplo: RandomForestClassifier sobre Iris (corre en <5 s — ideal para CI).
- Repo template: estructura src/, tests/, .github/workflows/, params.yaml, metrics.json.
- Librerías: scikit-learn, pytest, ruff (lint), cml (npm package).

Ejercicios

1. Workflow mínimo (lint + test): creá .github/workflows/ci.yml con dos jobs en paralelo: lint (corre ruff check) y test (corre pytest). Push y verificá que aparecen los dos checks en el PR.
2. Job de training: agregá un job train que corre python src/train.py, sube model.pkl y metrics.json como actions/upload-artifact. Solo en PRs (if: github.event_name == 'pull_request').
3. Reporte CML: agregá un step que use iterative/setup-cml@v2, escribe report.md con cat metrics.json como tabla, y postea con cml comment create report.md. El comentario aparece en el PR.
4. Comparación contra main: en el mismo job, git fetch origin main && python src/train.py desde main, guardás metrics_main.json, y agregás al reporte una tabla con Δaccuracy, Δf1.
5. Branch protection: en Settings → Branches → Add rule → main, marcá Require status checks to pass before merging y seleccioná lint, test, train. PR con tests rotos = no podés mergear.

Homework verificable

Repo público en GitHub con:

1. Workflow .github/workflows/ml.yml con jobs lint, test, train, report.
2. CML comment funcionando: PR muestra tabla con accuracy, f1, Δaccuracy_vs_main.
3. Cache de pip configurado (actions/setup-python con cache: pip).
4. Un PR abierto donde el modelo degrada accuracy en >3% — el job report debe fallar (exit 1) por umbral configurable en params.yaml.
5. Captura del PR mostrando el comentario CML y los checks rojos.

Criterio de aceptación: el reviewer puede decidir merge/no-merge mirando solo el PR (no Actions tab), y el merge está bloqueado por branch protection cuando algún check falla.

Errores comunes

Síntoma / mensaje	Causa y cómo arreglar
Resource not accessible by integration al	El job no tiene permiso pull-requests: wri
El cache de pip nunca se reutiliza	El cache-key cambia en cada run (probablem
Workflow no se dispara en PRs de forks	Es comportamiento de seguridad: PRs de for

Process completed with exit code 137	OOM (out of memory). El runner gratis tien
OIDC token unavailable al asumir rol AWS	Falta permissions: { id-token: write } en
El job train tarda 25 min en cada push	Falta caché de datasets y modelos pre-entr

Preguntas frecuentes

¿Conviene entrenar el modelo real en CI?

Depende del tamaño. Si el training corre en <10 min en CPU: sí, en GitHub Actions. Si requiere GPU o tarda horas: usá CI solo para correr smoke tests (entrenar 1 epoch en data sample) y dispará el training real en un runner self-hosted con GPU, o en SageMaker/Vertex AI con workflow_dispatch.

¿GitHub Actions vs GitLab CI vs Jenkins?

GH Actions: integrado con GitHub, marketplace enorme de actions, gratis hasta cierto límite para repos públicos. GitLab CI: equivalente para GitLab, con auto-devops más opinado. Jenkins: self-hosted, máxima flexibilidad, mucho mantenimiento. Para equipos en GitHub: Actions sin pensar.

¿Qué hago con secrets de larga vida (API keys, DB passwords)?

(1) Si podés: OIDC + IAM role (zero secrets). (2) Si no: secrets de repo o de organization, rotados cada 90 días vía secrets-manager. (3) Nunca: hardcoded en el YAML ni en el código.

¿Cómo evito que un PR malicioso fugue secrets via CI?

PRs de forks NO ven secrets por default (correcto). Para repos privados con muchos contributors: Settings → Actions → General → Fork pull request workflows from outside collaborators: Require approval for all outside collaborators. Revisar el diff del workflow antes de approve.

¿CML es necesario o uso comentarios manuales con gh pr comment?

CML está diseñado para ML (sabe armar tablas de metrics, embedded plots como PNG base64). gh pr comment es genérico. Para reportes simples: cualquiera. Para comparación de runs con plots: CML.

¿Cómo manejo CI cuando uso self-hosted runners con GPU?

Etiquetá runners (runs-on: [self-hosted, gpu]). Para jobs cortos sin GPU, dejá runs-on: ubuntu-latest. Atención a seguridad: PRs de forks NO deberían correr en self-hosted (pueden ejecutar arbitrary code en tu hardware). Por default GH bloquea esto — no lo deshabilites.

Referencias

- Huyen, Chip. Designing Machine Learning Systems (O'Reilly, 2022), cap. 10 — Infrastructure and Tooling.
- GitHub Actions docs — empezar por Quickstart y Workflow syntax.
- CML.dev — Continuous Machine Learning de Iterative.ai.
- Configuring OIDC with AWS — deploy sin claves.
- awesome-actions — actions útiles del marketplace.

Material descargable

- Guía explicativa (PDF) — versión imprimible con todo el contenido de la clase.
- Presentación (PPTX) — deck PowerPoint listo para proyectar en clase.
- Notebook ejecutable (.ipynb) — abrilo desde el laboratorio del programa o desde Jupyter.

Clase 198 — Clase 198 — Docker para empaquetar modelos

> *Parte: 4 — MLOps · Fuente: Huyen cap. 11 + docs Docker + Nilsson, Docker Deep Dive. Duración estimada: 75 min.*

Objetivo

Empaquetar un modelo entrenado + su runtime (Python, deps, código) en una imagen Docker reproducible, optimizada (multi-stage build, capas cacheadas, imagen <500 MB), y segura (non-root user, no secrets baked in). El resultado se corre idéntico en tu laptop, en CI, y en producción.

Resultados de aprendizaje

Al finalizar, el estudiante podrá:

- Escribir un Dockerfile correcto para un servicio ML: base slim, multi-stage, layer caching, non-root.
- Diferenciar COPY de ADD, RUN de CMD de ENTRYPOINT, y cuándo usar cada uno.
- Reducir tamaño de imagen (de 2 GB a <500 MB) con python:3.12-slim, .dockerignore, y multi-stage builds.
- Versionar imágenes con tags semánticos (mymodel:1.2.3) y digests (@sha256:...) — y por qué :latest es trampa en producción.
- Diagnosticar image not building, image too big, slow rebuild con docker history, dive, docker scout.

Temas

#	Tema	Por qué importa
1	Imagen, capa, container, registry	Vocabulario base.
2	Dockerfile: FROM, COPY, RUN, CMD, ENTR	Las instrucciones que importan.
3	Layer caching: orden de instrucciones	Diferencia entre build de 2 min vs 30 s.
4	Multi-stage build	Separar build-deps de runtime-deps.
5	Imágenes base: python:slim vs distroless v	Trade-off tamaño / compat / debug.
6	Security: non-root user, secrets via env/m	No bakear API keys en la imagen.

Definiciones y características

- Imagen: filesystem inmutable + metadata (entrypoint, env, ports). Compuesta por capas read-only apiladas (cada RUN/COPY es una capa).
- Container: instancia ejecutándose de una imagen + capa writable encima. Efímero por default — al borrarlo, se pierde lo escrito (salvo volúmenes).
- Layer caching: si la instrucción N de tu Dockerfile no cambió ni cambiaron las anteriores, Docker reusa la capa cacheada. Regla de oro: lo que cambia poco va primero, lo que cambia mucho va al final.
- Multi-stage build: usar varias secciones FROM ... AS stage y copiar artefactos de una a otra con COPY --from=stage. Permite usar imagen grande para compilar y imagen pequeña para correr.
- .dockerignore: como .gitignore pero para docker build. Crítico — sin él, mandás todo el repo (incluido .git/, __pycache__/, datasets) como build context.
- Tag semántico (myapp:1.2.3): fija la versión exacta. Digest (myapp@sha256:abc...): fija el contenido exacto (inmutable). Producción usa digest. :latest no tiene garantía — puede cambiar entre docker pull y docker run.
- Distroless (gcr.io/distroless/python3): imagen sin shell, sin package manager, casi sin nada. Reduce attack surface; complica debug (no podés exec un shell).

Dataset / recursos

- Modelo: RandomForestClassifier entrenado en Iris, serializado con joblib.
- API: FastAPI sirviendo POST /predict.
- Herramientas: docker>=24, opcional dive, docker scout.

Ejercicios

1. Dockerfile básico: empaquetá un script predict.py que carga model.pkl y predice una fila random. FROM python:3.12-slim, instalá deps de requirements.txt, COPY . /app, CMD ["python", "predict.py"]. Build con docker build -t miml:v1 . y corré.
2. Layer caching: cambiá una línea en predict.py sin tocar requirements.txt. Rebuildá. Confirmá que la capa de pip install se reusa (mensaje "CACHED").
3. Multi-stage: separá en dos stages: builder (FROM python:3.12 AS builder, instalá deps con compiladores) y runtime (FROM python:3.12-slim, copiá solo site-packages del builder). Compará tamaño con docker images.
4. Non-root: agregá RUN useradd -m app && USER app antes del CMD. Verificá con docker run --rm miml:v3 whoami.
5. Tags y digest: hacé docker push miml:v1 a Docker Hub. Obtené el digest con docker inspect miml:v1 --format '{{index .RepoDigests 0}}'. Discutí por qué deploys de producción referencian el digest.

Homework verificable

Repo con:

1. Dockerfile multi-stage que produce imagen <500 MB para un modelo sklearn + FastAPI.
2. .dockerignore que excluye .git, __pycache__, *.ipynb, data/raw, mlruns/.
3. docker-compose.yml que levanta el servicio en :8000 con healthcheck.
4. README del repo con comandos build, run, push, y la URL/digest de la imagen pushed.
5. Output de dive miml:latest (o docker history) mostrando que ninguna capa única pesa más de 200 MB.

Criterio de aceptación: docker run --rm -p 8000:8000 miml:v1 levanta y responde a curl localhost:8000/predict -X POST -d '{"features":[5.1,3.5,1.4,0.2]}' -H 'content-type: application/json'. El contenedor corre como app, no como root (docker exec ... whoami devuelve app).

Errores comunes

Síntoma / mensaje	Causa y cómo arreglar
Imagen pesa 2.5 GB con un modelo de 50 MB	Estás usando python:3.12 (no slim) y/o tra
Cada build rebuildea todo aunque no cambié	Tenés COPY . /app antes de RUN pip install
docker build manda 3 GB de context	Falta .dockerignore. Fix: crearlo con al m
Error ModuleNotFoundError en el container	El pip install corrió en otra versión de P
Container corre como root	No agregaste USER al Dockerfile. Fix: RUN
:latest apunta a algo distinto que ayer	:latest es mutable. Otro docker push myimg
COPY no encuentra model.pkl	El path es relativo al build context (el .

Preguntas frecuentes

¿slim, alpine, o distroless?

slim (Debian-based, sin doc/locales): default razonable, ~120 MB, glibc, compatible con wheels precompilados. alpine (musl): 50 MB pero usa musl, no glibc — wheels precompilados de muchas libs Python no funcionan. distroless: la imagen runtime más chica y segura; complica debugging. Para ML: slim salvo

razón fuerte.

¿Docker o Podman?

Podman es API-compatible con Docker (alias podman ↔ docker), rootless por default, sin daemon. Para individuales: indistinto. Para producción Kubernetes: el OCI runtime que use el cluster (containerd, CRI-O). Las imágenes son OCI, no "Docker images" — funcionan en cualquier OCI runtime.

¿Cómo paso credenciales al container?

(1) Variables de entorno (docker run -e API_KEY=...) — visible en docker inspect. (2) Secrets mount (--secret, Docker Swarm/Kubernetes Secret) — preferido. (3) IAM role (en EC2/EKS) — el container hereda credenciales sin que las escribas. Nunca: ENV API_KEY=xxx en el Dockerfile, queda en una capa para siempre.

¿Tamaño del modelo dentro o fuera de la imagen?

Modelos pequeños (<200 MB): adentro, simple. Modelos grandes o que cambian seguido: afuera, en S3/MLflow, bajados en entrypoint. Trade-off: adentro = inmutable + lento de pull; afuera = imagen liviana pero acoplamiento con storage.

¿CMD vs ENTRYPOINT?

ENTRYPOINT es el "binario" fijo del container; CMD son sus "args default". Combinados: ENTRYPOINT ["python"] + CMD ["app.py"] → docker run img otro.py ejecuta python otro.py. Para imagen de modelo: ambos juntos, o solo CMD ["python", "app.py"] si querés que sea fácilmente overrideable.

¿Por qué docker scout o trivy?

Escanean la imagen contra CVE conocidas. Crítico antes de subir a producción — una python:3.12-slim de hace 6 meses tiene 200 CVE conocidas; rebuildear con la base actual elimina la mayoría. CI debería tener un step trivy image myimg:tag --severity HIGH,CRITICAL --exit-code 1.

Referencias

- Huyen, Chip. Designing Machine Learning Systems (O'Reilly, 2022), cap. 11 — Model Deployment and Prediction Service.
- Docker official docs — Best practices for writing Dockerfiles.
- Dive — TUI para inspeccionar capas y eficiencia.
- Docker Scout / Trivy — vulnerability scanning.
- distroless — imágenes mínimas de Google.

Material descargable

- Guía explicativa (PDF) — versión imprimible con todo el contenido de la clase.
- Presentación (PPTX) — deck PowerPoint listo para proyectar en clase.
- Notebook ejecutable (.ipynb) — abrilo desde el laboratorio del programa o desde Jupyter.

Clase 199 — Clase 199 — APIs con FastAPI sirviendo modelos

> *Parte: 4 — MLOps · Fuente: Huyen cap. 11 + docs FastAPI + Ramalho cap. 5. Duración estimada: 80 min.*

Objetivo

Exponer un modelo entrenado como REST API con FastAPI: validación de input con Pydantic, batching, async, healthcheck, métricas Prometheus, OpenAPI auto-generado. Target: p99 latency <100 ms y throughput >500 req/s en un solo nodo CPU.

Resultados de aprendizaje

Al finalizar, el estudiante podrá:

- Construir un servicio FastAPI con endpoints POST /predict, POST /predict-batch, GET /health, GET /metrics.
- Definir schemas de input/output con pydantic.BaseModel y obtener validación + docs gratis.
- Usar lifespan events para cargar el modelo una sola vez (no por request).
- Loadtestear con locust y medir latency p50/p95/p99 + throughput.
- Decidir entre sync def y async def según si el predict es CPU-bound o I/O-bound.

Temas

#	Tema	Por qué importa
1	ASGI vs WSGI: por qué FastAPI no es Flask	Async nativo, perf en I/O-bound.
2	Pydantic v2: validación + serialización	Schema = contrato; cambios rompen tests.
3	Lifespan: cargar modelo 1 vez	Sin esto, cada request reabre joblib.load.
4	Sync vs async para predict	CPU-bound → sync (deja al thread pool); I/
5	Batching: /predict-batch	100 predicciones en 1 request, no en 100.
6	Observabilidad: /health, /metrics, logs es	Lo que pide el oncall a las 3 AM.

Definiciones y características

- FastAPI: framework ASGI (no WSGI). Web server: uvicorn (single-process) o gunicorn -w N -k uvicorn.workers.UvicornWorker (multi-worker).
- ASGI (Asynchronous Server Gateway Interface): protocolo Python para apps async. Permite async def endpoints, websockets, streaming.
- Pydantic v2: validación + serialización con modelos tipados. ~20× más rápido que v1 (re-escrito en Rust).
- lifespan event (@asynccontextmanager): hook para inicializar recursos (cargar modelo, abrir DB) cuando el server arranca y limpiarlos al apagar. Reemplaza el deprecado @app.on_event("startup").
- Sync vs async endpoint: en FastAPI, def predict(...) (sync) corre en un thread pool — bueno para CPU-bound o libs que no son async. async def predict(...) corre en el event loop — bueno para I/O-bound (DB, HTTP a otro servicio). Si tu predict es solo model.predict(X) (CPU): sync.
- Batching: agrupar N predicciones en un solo request. Reduce overhead de HTTP (~5 ms por request) y permite vectorizar (model.predict(matrix) es ~10× más rápido que 100 model.predict(vector)).
- Healthcheck: endpoint que devuelve 200 si el servicio puede servir tráfico. K8s lo usa para livenessProbe (reiniciar pod si falla) y readinessProbe (sacar del LB si falla).

Dataset / recursos

- Modelo: cualquiera entrenado en clases previas — usamos sklearn por simplicidad.
- Librerías: fastapi, uvicorn[standard], pydantic>=2, prometheus-fastapi-instrumentator, locust (loadtest).

Ejercicios

1. API mínima: POST /predict con IrisInput(features: list[float]) → IrisOutput(class: int, proba: list[float]). Levantá con uvicorn app:app --reload. Abrí /docs (OpenAPI Swagger UI). Confirmá que probar desde la UI funciona.

2. Lifespan: cargá el modelo en un lifespan y guardalo en app.state.model. Verificá que el modelo se carga UNA vez (print al inicio) aunque hagas 100 requests.
3. Batching: agregá POST /predict-batch con BatchInput(rows: list[list[float]]). Medí latencia de 100 predicciones individuales vs 1 batch de 100.
4. Async vs sync: simulá un predict que llama a una API externa (await httpx.AsyncClient().get(...)). Compará def (bloquea thread pool) vs async def (libera event loop). Loadtest con 200 concurrent users.
5. Observabilidad: agregá prometheus-fastapi-instrumentator → /metrics. Healthcheck /health que devuelve {"status": "ok", "model_loaded": bool}. Loggeá cada request con logger.info (JSON).

Homework verificable

Servicio FastAPI + Dockerfile (Clase 198) con:

1. Endpoints POST /predict, POST /predict-batch, GET /health, GET /metrics, GET /docs.
2. Pydantic models con validación: features debe tener exactamente 4 floats positivos.
3. lifespan que carga model.pkl una vez al startup.
4. locustfile.py que simula 100 users concurrentes durante 60 s.
5. Reporte de loadtest con p50/p95/p99 latency y RPS, justificando si el target (<100 ms p99, >500 RPS) se cumple.

Criterio de aceptación: curl -X POST localhost:8000/predict -d '{"features":[-1,2,3,4]}' -H 'content-type:application/json' devuelve HTTP 422 (validación rechaza valor negativo). El loadtest cumple p99 <100 ms en al menos un workload.

Errores comunes

Síntoma / mensaje	Causa y cómo arreglar
Latency altísima (p99 >500 ms) con un mode	Estás cargando joblib.load("model.pkl") de
Error RuntimeError: This event loop is alr	Mezclaste asyncio.run(...) dentro de un en
422 Unprocessable Entity en requests que p	Pydantic v2 es estricto: list[float] recha
Single worker satura 1 CPU	uvicorn app:app corre 1 worker. Fix: gunic
/docs está activo en producción	Por default FastAPI expone /docs, /redoc,
Healthcheck devuelve 200 aunque el modelo	Healthcheck simplista. Fix: chequear app.s

Preguntas frecuentes

¿FastAPI o Flask?

Para servir modelos hoy: FastAPI. Pros: async nativo, validación con Pydantic (no request.json + checks manuales), OpenAPI gratis, performance superior en benchmarks (~3× Flask). Flask sigue siendo elegible para apps simples o equipos con experiencia previa.

¿Cuándo async def y cuándo def?

Regla: si el endpoint solo hace CPU-bound (cargar modelo, predecir): def (FastAPI lo manda al thread pool, no bloquea el event loop). Si hace I/O-bound (await DB, await HTTP): async def. Si mezclás requests (sync HTTP) en un async def: bloqueás el loop. Cambialo a httpx.AsyncClient.

¿Servir con uvicorn o gunicorn?

Dev: uvicorn --reload. Prod: gunicorn -w N -k uvicorn.workers.UvicornWorker (N = 2 * cores + 1). Razón: gunicorn maneja restarts, multi-proceso, signals. En K8s con HPA + 1 worker/pod: uvicorn solo está bien.

¿Cómo manejo modelos grandes (>1 GB) que tardan en cargar?

(1) lifespan (no re-cargar). (2) readinessProbe con initialDelaySeconds: 60 — pod no recibe tráfico hasta cargar. (3) model.eval() + torch.jit.script (PyTorch) o onnxruntime (cualquier framework) — más rápido en inferencia que el framework original. (4) Para LLMs: vLLM/TGI con paged attention (Clase 165).

¿Validación strict de Pydantic me rompe clientes legacy?

Sí, si pasaban tipos laxos. Pydantic v2 puede ser permisivo con model_config = {"strict": False} o usar validators (field_validator). Lo correcto a mediano plazo: docs claras + versioning de API (/v1/predict vs /v2/predict).

¿FastAPI sirve modelos PyTorch/TF directamente?

Sí, pero para producción seria conviene un inference server dedicado: TorchServe, TensorFlow Serving, NVIDIA Triton (multi-framework, batching dinámico, GPU optimal). FastAPI queda como gateway/proxy con auth y reglas de negocio. Para casos simples sklearn/XGBoost: FastAPI alcanza.

Referencias

- Huyen, Chip. Designing Machine Learning Systems (O'Reilly, 2022), cap. 11.
- FastAPI docs — Tutorial y Advanced User Guide.
- Pydantic v2 migration guide — qué cambió desde v1.
- Locust — loadtest distribuido en Python.
- NVIDIA Triton — alternativa para producción seria.

Material descargable

- Guía explicativa (PDF) — versión imprimible con todo el contenido de la clase.
- Presentación (PPTX) — deck PowerPoint listo para proyectar en clase.
- Notebook ejecutable (.ipynb) — abrilo desde el laboratorio del programa o desde Jupyter.

Clase 200 — Clase 200 — Kubernetes para servir modelos a escala

> *Parte: 4 — MLOps · Fuente: Huyen cap. 11 + Burns et al., Kubernetes: Up and Running (3ª ed.) + docs k8s. Duración estimada: 90 min.*

Objetivo

Desplegar el contenedor de Clase 198–199 en Kubernetes con Deployment + Service + Ingress, autoescalar con HPA (CPU + custom metrics), hacer rolling updates seguros, y configurar livenessProbe/readinessProbe/resources correctamente. Aprender los 5 manifests mínimos que necesita cualquier servicio de inferencia.

Resultados de aprendizaje

Al finalizar, el estudiante podrá:

- Escribir manifests YAML para Deployment, Service, ConfigMap, Secret, HPA, Ingress.
- Diferenciar livenessProbe (reinicia pod) de readinessProbe (saca del LB) de startupProbe (gracia inicial para modelos lentos).
- Configurar resources.requests/limits y entender por qué un pod sin requests es OOM-killable y sin limits es noisy-neighbor.

- Hacer kubectl rollout/rollback y entender los maxSurge/maxUnavailable del rolling update.
- Diagnosticar CrashLoopBackOff, ImagePullBackOff, Pending, Evicted con kubectl describe y kubectl logs.

Temas

#	Tema	Por qué importa
1	Pod, Deployment, ReplicaSet, Service	Las abstracciones core.
2	Probes (liveness, readiness, startup)	Diferencia entre "el pod murió" y "el pod
3	Resources: requests vs limits	Scheduling + OOMkill control.
4	HPA: CPU + custom metrics (latency, queue	Autoescalado más allá de "CPU alta".
5	Rolling update + rollback	Deploy sin downtime y vuelta atrás.
6	Ingress + service mesh (mention)	Cómo expone tráfico externo.

Definiciones y características

- Pod: unidad de scheduling. 1+ containers que comparten network + storage. Para ML: típicamente 1 container por pod.
- Deployment: declara "quiero N réplicas de este pod con esta imagen". Gestiona un ReplicaSet que gestiona los pods. Rolling updates son responsabilidad del Deployment.
- Service: load balancer estable interno. Tipo ClusterIP (default, interno), NodePort (expone puerto en cada node), LoadBalancer (cloud LB).
- livenessProbe: si falla, K8s reinicia el container. Apuntar a /health con timeouts generosos.
- readinessProbe: si falla, K8s saca el pod del Service (no recibe tráfico) pero NO lo reinicia. Apuntar a /ready que chequee modelo cargado + deps disponibles.
- startupProbe: ventana de gracia para containers lentos en arrancar (modelos grandes). Mientras corre, liveness/readiness no se ejecutan. Una vez OK, pasan a evaluarse normalmente.
- resources.requests: lo que el scheduler reserva para el pod. Sin requests, el pod va a "Best Effort" → primero en ser killed con presión de memoria.
- resources.limits: tope duro. Si memoria > limit: OOMKill. Si CPU > limit: throttling (no kill).
- HPA (Horizontal Pod Autoscaler): escala réplicas entre minReplicas y maxReplicas según métrica (default: CPU%). Custom metrics (latency p99, queue depth) requieren metrics-server + adapter (Prometheus Adapter).
- Rolling update: estrategia default. Crea pods nuevos (maxSurge), espera readiness, mata viejos (maxUnavailable). Garantiza disponibilidad continua.
- Rollback: kubectl rollout undo deployment/<name>. Vuelve al último ReplicaSet exitoso. Casi instantáneo si los pods viejos siguen referenciados.

Dataset / recursos

- Cluster: minikube, kind, o k3d para local. Equivalente en cloud: GKE/EKS/AKS.
- Imagen: la de Clase 198 (iris-api:v1).
- Herramientas: kubectl, kustomize (built-in) o helm.

Ejercicios

1. Cluster local: kind create cluster --name ml. Pusheá la imagen local con kind load docker-image iris-api:v1 --name ml. Verificá con kubectl get nodes.
2. Deployment + Service: aplicá los YAML del notebook. kubectl get pods -w mientras los 3 pods arrancan. kubectl port-forward svc/iris-api 8000:80 y pegale con curl localhost:8000/predict.
3. Probes: cambiá livenessProbe a apuntar a /wrong-endpoint. Observá con kubectl get pods -w cómo

entra en CrashLoopBackOff. Revertí.

4. HPA: kubectl apply -f hpa.yaml con targetCPUUtilizationPercentage: 50. Generá carga con kubectl run loadtester --image=busybox -it --rm -- /bin/sh -c "while true; do wget -q -O- iris-api/predict; done". Observá kubectl get hpa -w escalar de 3 → 10.
5. Rolling update + rollback: cambiá la imagen a iris-api:v2 (versión rota a propósito). kubectl rollout status deployment/iris-api debería timeoutear. kubectl rollout undo deployment/iris-api y verificá recuperación.

Homework verificable

Cluster (local o cloud) con:

1. Manifests YAML para Deployment (3 réplicas, probes, resources), Service (ClusterIP), HPA (CPU 50%, min 3 max 10), Ingress.
2. Imagen pusheada a un registry (Docker Hub o ECR).
3. Loadtest que dispara HPA y demostrar escalado en logs/screenshots.
4. Rolling update a una versión v2 exitosa, después rollback con kubectl rollout undo.
5. README con comandos kubectl apply, port-forward, troubleshooting con describe.

Criterio de aceptación: kubectl get pods -l app=iris-api muestra 3-10 pods escalando con la carga, kubectl get hpa reporta target CPU, y curl <ingress-host>/predict funciona desde fuera del cluster.

Errores comunes

Síntoma / mensaje	Causa y cómo arreglar
Pods en Pending infinito	Cluster sin recursos para requests. Fix: k
CrashLoopBackOff	Container falla al iniciar y K8s reintenta
ImagePullBackOff	No puede bajar la imagen. Fix: chequear no
Liveness mata el pod durante startup del m	livenessProbe arranca antes que el modelo
OOMKilled silencioso	kubectl describe pod muestra "OOMKilled" e
HPA no escala aunque CPU está alta	metrics-server no está instalado o el pod

Preguntas frecuentes

¿K8s vs Cloud Run vs ECS Fargate vs Lambda?

K8s es el más flexible y portable; el más pesado en mantenimiento. Cloud Run / Fargate: contenedor managed, escalan a 0, sin K8s machinery — buen punto intermedio. Lambda (Clase 201): serverless puro, frío, máx 15 min, ideal para batch chico. Para ML serving 24/7: K8s o Cloud Run.

¿1 worker por pod o N workers por pod?

Recomendación K8s: 1 worker por pod. K8s se encarga del fleet (HPA, rolling update, restart). Múltiples workers complican observabilidad por pod, hacen restart costoso, y suelen reproducir lo que ya hace K8s.

¿GPU pods?

Necesitás un node pool con GPUs y el NVIDIA device plugin instalado. En el pod: resources.limits["nvidia.com/gpu"]: 1. La imagen base debe ser CUDA-compatible (nvidia/cuda:... o pytorch/pytorch:cuda).

¿Helm o Kustomize?

Kustomize (built-in en kubectl): overlays por entorno (dev/staging/prod) sin templating string-based. Helm:

package manager con templates Go, más expresivo pero también más complejo. Para empezar: Kustomize. Para distribuir charts (ej. instalar Prometheus): Helm.

¿Cómo manejo secrets?

Secret resource (base64, NO encripta por default). Para producción: enable encryption-at-rest en etcd, o mejor: External Secrets Operator que sincroniza de AWS Secrets Manager / Vault → K8s Secrets. Nunca commitear secrets a git, ni siquiera "fake" — usá kubectl create secret o secrets externos.

¿Service mesh (Istio/Linkerd) es necesario?

No para empezar. Resuelve: mTLS automático entre pods, retries/timeouts/circuit breaking, canary releases (Clase 204) sin tocar código, observabilidad fina. Agrega complejidad (sidecar por pod, control plane). Vale la pena cuando: ≥10 microservicios, requerimiento mTLS interno, o canary serio.

Referencias

- Burns et al. Kubernetes: Up and Running (3ª ed., O'Reilly, 2022).
- Kubernetes docs — Workloads, Services, Scheduling, Preemption and Eviction.
- Probes guide — patrones correctos.
- Horizontal Pod Autoscaler walkthrough.
- Kind quickstart — cluster local en segundos.

Material descargable

- Guía explicativa (PDF) — versión imprimible con todo el contenido de la clase.
- Presentación (PPTX) — deck PowerPoint listo para proyectar en clase.
- Notebook ejecutable (.ipynb) — abrilo desde el laboratorio del programa o desde Jupyter.

Clase 201 — Clase 201 — Serverless ML: AWS Lambda, GCP Cloud Functions

> *Parte: 4 — MLOps · Fuente: Huyen cap. 11 + docs Lambda container images + Cloud Functions 2nd gen. Duración estimada: 75 min.*

Objetivo

Desplegar un modelo como función serverless que escala de 0 a N sin gestionar servidores. Decidir cuándo serverless gana (tráfico bursty, batch chico, no podés mantener infra) y cuándo pierde (cold start crítico, modelo >1 GB, latencia <50 ms requerida).

Resultados de aprendizaje

Al finalizar, el estudiante podrá:

- Empaquetar un modelo sklearn/XGBoost como Lambda Container Image (hasta 10 GB) o Cloud Function 2nd gen (build automático con Buildpacks).
- Mitigar cold starts con: provisioned concurrency (Lambda), min-instances=1 (Cloud Functions), o snapshot-based init (SnapStart).
- Configurar API Gateway o HTTP trigger para exponer la función como REST.
- Calcular costo: $/M invocations × duration × memory, vs un pod K8s 24/7.
- Reconocer límites: timeout 15 min (Lambda), payload 6 MB sync / 256 KB async, disco efímero 512 MB /tmp (10 GB con ephemeral-storage).

Temas

#	Tema	Por qué importa
1	Modelo de ejecución: scale-to-zero, reques	Diferente a K8s/EC2 24/7.
2	Cold start vs warm	El bug que arruina la UX.
3	Package formats: ZIP (≤250 MB) vs Containe	Para modelos ML: container.
4	Cost model: invocations + GB-seconds	Cruce con K8s al ~1M req/día sostenido.
5	Provisioned concurrency / min-instances	Cambia la economía y la latencia.
6	Cuándo NO usar serverless	Modelos grandes, latencia <50 ms, stateful

Definiciones y características

- Cold start: tiempo entre llegada del request y arranque del runtime + carga del modelo en una instancia nueva. Lambda Python: ~300 ms (runtime) + tu init code. Con modelo grande: 5-15 s.
- Warm: instancia ya inicializada que recibe request → latencia = solo predict. Lambda mantiene warm ~5-15 min sin invocations (no garantizado).
- Provisioned Concurrency (Lambda): pagás por mantener N instancias warm permanentemente. Elimina cold starts a cambio de costo fijo.
- SnapStart (Lambda, Java/Python 3.12+): toma snapshot del runtime después de init y lo restaura — cold start <500 ms aún con modelos grandes.
- Cloud Functions 2nd gen (basado en Cloud Run): timeouts hasta 60 min, hasta 32 GB RAM, mejor cold start que 1st gen, y la opción --min-instances=1 para mantener warm.
- API Gateway: pone HTTP delante de Lambda. Maneja auth, throttling, CORS. Cobra extra por request — para tráfico volumétrico, Lambda Function URL (gratis) es alternativa.
- Provisioned vs On-Demand: provisioned escala a 0 manteniendo N warm. On-demand escala a 0 puro. Trade-off costo vs latency tail.

Dataset / recursos

- Modelo: LogisticRegression sobre Iris (chico — buen fit serverless).
- Tools: AWS CLI + SAM (Serverless Application Model), o gcloud functions deploy.
- Imagen base Lambda: public.ecr.aws/lambda/python:3.12.

Ejercicios

1. Lambda con container: buildeá un Dockerfile basado en public.ecr.aws/lambda/python:3.12, copiá app.py con lambda_handler(event, context), y model.pkl. Push a ECR. aws lambda create-function con --package-type Image.
2. API Gateway: creá una API HTTP y conectala a la Lambda. curl <api-url>/predict -d '{"features":[5.1,3.5,1.4,0.2]}'. Medí latencia primera vs segunda llamada (cold vs warm).
3. Provisioned Concurrency: configurá provisioned-concurrency=1 en la Lambda. Vuelve a medir — debería eliminar el cold start.
4. Cloud Functions equivalent: gcloud functions deploy iris --gen2 --runtime=python312 --trigger-http --source=. --entry-point=predict --memory=512Mi --min-instances=1. Compará cold start con Lambda.
5. Cost calc: asumí 100 req/s sostenido, modelo en RAM 512 MB, 80 ms p99. Calculá $/mes en Lambda vs un pod K8s con 4 vCPU 24/7. Encontrá el punto de cruce.

Homework verificable

1. Lambda function (container image) que sirve predict con cold start <3 s y warm <80 ms.
2. API Gateway HTTP exponiendo la Lambda.
3. CloudWatch alarm que dispara si Errors > 5 en 5 min, o Duration P99 > 1000 ms.

4. Script python cost_analysis.py que dado req_per_second_mean, req_size, mem_mb, duration_ms calcula $/mes en (a) Lambda on-demand, (b) Lambda con PC=2, (c) ECS Fargate equivalente, (d) K8s con 3 pods t3.medium.
5. Recomendación escrita: para tu workload, qué opción elegir y por qué.

Criterio de aceptación: curl <api>/predict responde en <100 ms (warm) y <3 s (cold), el cost analysis identifica el punto de cruce correcto (típicamente serverless gana <~500k requests/día sostenido).

Errores comunes

Síntoma / mensaje	Causa y cómo arreglar
Cold start de 30 s	Carga del modelo dentro del handler. Fix:
Task timed out after 3.00 seconds	Default Lambda timeout es 3 s. Fix: aws la
Imagen ZIP excede 250 MB descomprimido	Probablemente trayendo numpy+scipy+pandas.
Pérdida de estado entre invocations	Esperado: cada instancia es efímera. Fix:
Costos explotaron	(1) Provisioned concurrency olvidada. (2)
errorMessage: Unable to import module	Falta una dep o estructura incorrecta. Fix

Preguntas frecuentes

¿Cuándo serverless conviene para ML?

(1) Tráfico bursty con valles largos (1000 req/min picos, 0 req/min noche). (2) Modelo chico (<1 GB) y latencia tolerante (>200 ms p99 OK). (3) Equipo sin DevOps. (4) Predicciones batch ad-hoc (S3 trigger). Suma ~70% de los casos típicos en startups.

¿Cuándo NO?

(1) Latencia estricta <50 ms — cold start es matador. (2) Modelo grande (>3 GB) — init carísimo. (3) Throughput sostenido >1000 req/s 24/7 — sale más caro que K8s. (4) Stateful: caches calientes, sessions, websockets persistentes.

¿Lambda o SageMaker Serverless Inference?

SageMaker Serverless: built-in para ML (deploy con un click desde model registry, batch transform). Misma idea de scale-to-zero. Latencia similar. Pros: integración con SageMaker; cons: vendor-locked y más caro por request. Para ML puro en AWS: SageMaker. Para mezcla ML + lógica de negocio: Lambda.

¿GPU en serverless?

Lambda no tiene GPU. Cloud Run/Functions 2nd gen tampoco (al momento de escribir). Para inferencia GPU serverless: Modal, Banana, Replicate (3rd party SaaS). Sino: K8s con GPU nodes.

¿Cómo monitoreo cold start vs warm?

CloudWatch: métrica InitDuration (solo en cold), Duration (siempre). Lambda Insights agrega CPU/mem/network. Para correlacionar con latencia user-facing: X-Ray tracing.

¿Container Image vs Layer?

Lambda Layers: hasta 5 layers de 250 MB cada uno (compartidos entre funciones). Container Image: hasta 10 GB, todo en uno. Para ML moderno: Container Image siempre — más simple, sin límite efectivo.

Referencias

- Huyen, Chip. Designing Machine Learning Systems (O'Reilly, 2022), cap. 11.

- AWS Lambda container images for Python.
- Lambda SnapStart for Python.
- Cloud Functions 2nd gen.
- Modal — serverless con GPU para ML.

Material descargable

- Guía explicativa (PDF) — versión imprimible con todo el contenido de la clase.
- Presentación (PPTX) — deck PowerPoint listo para proyectar en clase.
- Notebook ejecutable (.ipynb) — abrilo desde el laboratorio del programa o desde Jupyter.

Clase 202 — Clase 202 — Monitoreo: data drift, model drift, alertas

> *Parte: 4 — MLOps · Fuente: Huyen cap. 8 + Evidently AI docs + NannyML + Gama et al., A Survey on Concept Drift Adaptation (ACM, 2014). Duración estimada: 80 min.*

Objetivo

Detectar antes que el negocio: (a) data drift (la distribución de features cambió), (b) prediction drift (la distribución de predicciones cambió), (c) concept drift (la relación X→y cambió). Configurar alertas que avisen antes de que la métrica de negocio caiga, no después.

Resultados de aprendizaje

Al finalizar, el estudiante podrá:

- Distinguir data drift (P(X) cambia), prediction drift (P(ŷ) cambia), concept drift (P(y|X) cambia) y elegir el test correcto para cada uno.
- Detectar drift con tests estadísticos: PSI (Population Stability Index), K-S (continuas), chi-cuadrado (categóricas), Wasserstein (más sensible que K-S en colas).
- Usar Evidently AI para generar reportes HTML con todos los tests + visualizaciones, y NannyML para estimar performance sin labels en producción (CBPE).
- Configurar alertas en Grafana / CloudWatch / Slack vía webhook cuando el drift score supera el umbral.
- Reconocer cuándo el "drift" es ruido (re-test con bonferroni) vs señal (acción).

Temas

#	Tema	Por qué importa
1	3 tipos de drift y por qué importan distin	Confundirlos lleva a "rentrenar" sin neces
2	PSI: umbral 0.1 / 0.2	El estándar de la industria; simple, inter
3	K-S, chi-cuadrado, Wasserstein	Tests con propiedades distintas — elegir s
4	Performance estimation sin labels (CBPE)	Cómo saber si el modelo empeora antes de t
5	Reference window vs current window	El qué se compara con qué.
6	Alertas: umbral + cooldown + canal	Sin diseño → alert fatigue.

Definiciones y características

- Data drift (covariate shift): P_train(X) ≠ P_prod(X). Ej: en training los users tenían 18-65 años, en prod aparecen muchos >70. El modelo todavía es válido si la relación X→y no cambió, pero las predicciones serán menos confiables en zonas no vistas.
- Prediction drift: P_train(ŷ) ≠ P_prod(ŷ). La distribución de salidas cambió. A veces es síntoma de data

drift, a veces es legítimo (estacionalidad). No siempre malo.
- Concept drift (real drift): P(y|X) cambió — la relación entre features y target. Ej: durante COVID, "salir a la calle" pasó de actividad normal a anómala. Aún sin data drift, el modelo deja de servir. Este es el más grave, y el más difícil de detectar sin labels.
- PSI (Population Stability Index): Σ (p_i - q_i) * ln(p_i / q_i) sobre bins. Umbrales clásicos: <0.1 estable, 0.1-0.2 cambio menor, >0.2 shift significativo (acción).
- K-S test (Kolmogorov-Smirnov): estadístico = sup|F_ref(x) - F_cur(x)|. p-value bajo → distros distintas. Sensible al centro de la distribución.
- Chi-cuadrado: para variables categóricas. Compara frecuencias observadas vs esperadas.
- Wasserstein distance (Earth Mover's Distance): "cuánta tierra hay que mover" para transformar una distro en otra. Más sensible que K-S a diferencias en cola.
- Reference window: muestra del periodo de training (o un slice estable previo).
- Current window: muestra del periodo a evaluar (último día, última semana). Tamaño del window afecta sensibilidad (chico → ruidoso, grande → lento).
- CBPE (Confidence-Based Performance Estimation, NannyML): estima accuracy o AUC de producción usando solo las probabilidades del modelo (no labels). Asume bien-calibrado.

Dataset / recursos

- Dataset training: California Housing — usado como reference.
- Dataset "producción": California Housing con shift sintético (escalar MedInc × 1.5 para simular inflación, o filtrar HouseAge > 30 para simular nuevo segmento).
- Librerías: evidently, scipy, nannyml, scikit-learn.

Ejercicios

1. PSI manual: bineá MedInc en 10 deciles (con bordes del reference). Calculá p_ref, p_cur y aplicá la fórmula PSI. Verificá que sin shift PSI < 0.05; con shift × 1.5 PSI > 0.5.
2. K-S vs Wasserstein: agregale outliers a 5% de la muestra de producción (multiplicá esos por 100). K-S podría no detectar (la mediana no cambió mucho); Wasserstein sí. Reproducí ambos casos.
3. Reporte Evidently: Report(metrics=[DataDriftPreset()]) sobre reference vs current. Guardalo como HTML, abrilo, identificá qué features driftearon y con qué test.
4. Concept drift sin labels (CBPE): con NannyML, fit el estimador sobre reference + predicciones. Aplicalo a current. Compará estimated_accuracy vs actual_accuracy (calculable porque tenés y en este ejercicio).
5. Alerta: escribí un script que (a) calcule PSI por feature, (b) si alguna >0.2 emite un POST a un webhook Slack/Discord, (c) usá cooldown de 4 h con un archivo last_alert.txt para evitar spamming.

Homework verificable

Sistema de monitoreo que produce:

1. Job diario (Cron / Airflow / GH Actions schedule) que toma el snapshot de producción de las últimas 24 h y lo compara con el reference window.
2. Reporte Evidently HTML guardado en S3 / GCS (timestamped).
3. PSI dashboard en Grafana o Streamlit que muestra evolución por feature en los últimos 30 días.
4. Alerta a Slack si: PSI > 0.2 en cualquier feature, O drift score global > umbral, O CBPE estimated_accuracy cayó >5 puntos.
5. Runbook (runbook.md) con 3 escenarios: (a) drift de feature legítimo (cambio negocio) → retrainings, (b) drift por bug en pipeline → fix upstream, (c) concept drift → A/B test modelo nuevo.

Criterio de aceptación: simular un shift introducido a propósito (multiplicar feature × 2) dispara la alerta en <24

h. El runbook tiene pasos accionables, no descripciones genéricas.

Errores comunes

Síntoma / mensaje	Causa y cómo arreglar
PSI = 0.3 pero el modelo sigue bien	Es data drift sin concept drift — la distr
Tests dan "drift" en TODO siempre	n muy grande → cualquier diferencia es "si
Alerta dispara 50 veces / día	Falta cooldown y/o el threshold es muy agr
Reference window se actualiza cada deploy	Si el reference cambia con cada release, p
Detecto drift pero no sé qué hacer	Falta runbook. Fix: para cada feature crít
CBPE da accuracy muy distinta a la real	El modelo está mal calibrado — CBPE asume

Preguntas frecuentes

¿Cada cuánto retrainear?

Hay 3 estrategias: (1) schedule fijo (cada lunes) — simple pero ciego. (2) trigger por drift (PSI > 0.3) — reactivo. (3) trigger por performance degradation (CBPE estima accuracy caída >X%) — mejor. La industria converge a (3) con fallback a (2). Ver Clase 203.

¿PSI o KL divergence?

PSI es simétrica (PSI(A,B) = PSI(B,A)) y robusta; KL no. PSI domina en la industria financiera; ML moderno usa también Wasserstein o Earth Mover's Distance. Para empezar: PSI.

¿Mido drift por feature o multivariado?

Por feature es interpretable ("MedInc driftó"). Multivariado (PCA + drift sobre componentes, o Maximum Mean Discrepancy) capta correlaciones nuevas. Ideal: ambos, pero un drift de feature suele ser suficiente para disparar investigación.

¿Cómo evito alertas en feriados/temporadas?

(1) Reference window por época: comparar diciembre 2026 vs diciembre 2025, no vs julio. (2) Modelar estacionalidad explícitamente (incluir month/day_of_week como feature). (3) Suprimir alertas en feriados conocidos.

¿Evidently vs NannyML vs Whylogs vs Arize?

Evidently: open-source, reportes HTML excelentes. NannyML: open-source, especialista en CBPE (estimar performance sin labels). Whylogs: librería de profiling minimalista de WhyLabs. Arize: SaaS comercial, dashboards e integraciones empresariales. Para empezar: Evidently + NannyML.

¿Y si no tengo labels en producción?

Es lo normal en muchos casos. Estrategias: (1) CBPE (NannyML) — estima con probas. (2) Proxy labels — métricas de negocio downstream (CTR, conversión). (3) Active learning — etiquetar un subset random para validar. (4) Shadow scoring — guardás predicciones + features y cuando llegan labels (días después), evaluás.

Referencias

- Huyen, Chip. Designing Machine Learning Systems (O'Reilly, 2022), cap. 8 — Monitoring.
- Evidently AI — reportes y tests.
- NannyML — CBPE y DLE.

- Gama et al., A Survey on Concept Drift Adaptation (ACM Computing Surveys, 2014) — taxonomía clásica.
- Monitoring ML Models: Theory (Tagliabue) — visión práctica.

Material descargable

- Guía explicativa (PDF) — versión imprimible con todo el contenido de la clase.
- Presentación (PPTX) — deck PowerPoint listo para proyectar en clase.
- Notebook ejecutable (.ipynb) — abrilo desde el laboratorio del programa o desde Jupyter.

Clase 203 — Clase 203 — Reentrenamiento programado

> *Parte: 4 — MLOps · Fuente: Huyen cap. 9 + Airflow docs + Prefect docs. Duración estimada: 75 min.*

Objetivo

Convertir el reentrenamiento en un proceso programado, auditado y reversible: DAG que cada N horas/días corre pull-data → validate → train → evaluate → promote-if-better → notify, con shadow/canary (Clase 204) antes del rollout pleno. Decidir entre schedule fijo, trigger por drift y trigger por degradación según el caso.

Resultados de aprendizaje

Al finalizar, el estudiante podrá:

- Diseñar un DAG (Airflow/Prefect/Dagster) que orqueste el reentrenamiento completo.
- Implementar el patrón champion-challenger: el modelo Production sigue sirviendo hasta que el challenger demuestra ser mejor en métricas de validación + en shadow.
- Configurar tres estrategias de trigger: cron(0 2 MON), on_drift > threshold, on_performance_degraded.
- Garantizar idempotencia (re-runs no duplican datos) y observabilidad (logs estructurados, alertas en fallas).
- Diferenciar online learning (modelo actualiza con cada nueva muestra) de continual training (re-trainings periódicos).

Temas

#	Tema	Por qué importa
1	Triggers: schedule vs drift vs degradation	Sin trigger correcto, retrainings caros si
2	DAG = grafo de tareas con deps	Airflow/Prefect/Dagster son variaciones de
3	Champion-challenger	Cómo evitar regresiones en producción.
4	Idempotencia	Re-runs no deben duplicar/destruir state.
5	Online vs continual training	Trade-off frescura vs estabilidad.
6	Catastrophic forgetting	Retraining puede olvidar lo aprendido si d

Definiciones y características

- DAG (Directed Acyclic Graph): grafo de tareas con dependencias, ejecutable por un scheduler. Airflow inventó la categoría; Prefect/Dagster/Mage son alternativas modernas.
- Champion: modelo activo en producción (alias @champion en MLflow, Clase 195).
- Challenger: candidato nuevo entrenado. Promueve a champion solo si pasa todos los gates (validation metric + shadow + canary).
- Promotion gate: criterio que debe cumplirse para que el challenger reemplace al champion. Ej: f1_test >

f1_champion + 0.005 AND no_regression_in_protected_slices.

- Idempotencia: re-ejecutar la misma tarea con los mismos inputs produce el mismo output, sin efectos colaterales acumulativos. Implementación típica: usar execution_date como key, sobrescribir destino, evitar INSERT INTO sin WHERE NOT EXISTS.
- Backfill: re-ejecutar el DAG para fechas pasadas (Airflow soporta nativo). Útil para corregir bugs o llenar gaps.
- Online learning: el modelo actualiza con cada muestra/mini-batch que llega. Funciona bien con SGD-based (LR, RF online, RL). No funciona con XGBoost/RF batch.
- Continual training: re-trainings periódicos (cada hora/día/semana) sobre data acumulada. Más común y más simple que online learning.
- Catastrophic forgetting: cuando retraining sobre data reciente "olvida" patrones que aprendió antes. Mitigación: incluir data histórica en cada retraining, regularización (EWC en deep learning).

Dataset / recursos

- Pipeline target: Airflow local (Docker), o Prefect Cloud free tier, o GitHub Actions schedule.
- Librerías: apache-airflow>=2.9, prefect>=3, mlflow, evidently.

Ejercicios

1. DAG mínimo (Airflow o Prefect): 5 tareas: pull_data → validate_data → train → evaluate → promote. Cada tarea es una función Python. Schedule diario.
2. Champion-challenger: en promote, comparar challenger.f1 vs champion.f1 (leídos de MLflow). Promover solo si challenger.f1 > champion.f1 + 0.005. Sino, logear [skipped] challenger no es significativamente mejor y dejar champion intacto.
3. Idempotencia: ejecutá el DAG dos veces seguidas para la misma fecha. Verificá que no se duplican filas en data/processed/, ni se crean dos registros en MLflow con el mismo execution_date.
4. Trigger por drift: agregá una tarea inicial check_drift que (a) si PSI > 0.2 continúa el DAG, (b) si no, hace raise AirflowSkipException (skipea el resto). Resultado: solo se entrena si hay drift.
5. Backfill: simulá un bug en la tarea validate_data que ya corrió bien por una semana. Fixeá el bug, airflow dags backfill --start-date X --end-date Y. Verificá que se re-procesan los días afectados sin duplicar registros downstream.

Homework verificable

DAG en producción (o local con docker-compose) que:

1. Corre diariamente a las 02:00 UTC.
2. Pull últimos 30 días de data, valida con Great Expectations (Clase 206) — falla con alert si no pasa.
3. Re-entrena, evalúa contra hold-out + slices de fairness, registra en MLflow.
4. Compara contra champion: promueve a @challenger (alias) si métrica supera umbral.
5. Tarea shadow_test que durante 24 h compara predicciones challenger vs champion sobre tráfico real (Clase 204).
6. Tarea promote_champion (manual o auto) que reemplaza alias @champion solo si shadow OK.
7. Alertas Slack en cada falla del DAG + un resumen diario [OK] o [NO TRAIN, no drift].

Criterio de aceptación: en una semana de operación, el DAG corrió 7 veces, ≥1 promoción real ocurrió, ≥1 skip por no-mejora ocurrió, y ningún re-run duplicó datos.

Errores comunes

Síntoma / mensaje	Causa y cómo arreglar
DAG corre OK pero modelo no mejora	Probablemente estás re-entrenando solo con

Re-run del DAG duplica filas en la DB de f	Insert sin idempotencia. Fix: INSERT ... O
Catastrophic forgetting visible: f1 sobre	El nuevo retraining sobre data fresca borr
Champion-challenger nunca promueve	Umbral muy estricto, o el modelo nuevo no
DAG falla pero nadie se entera	Sin alerta en on_failure_callback. Fix: co
TaskGroup con N tareas paralelas satura el	Sin paralelismo control. Fix: en Airflow,
MLflow runs sin tag de dag_run_id	Cuando algo falla, no podés trazar qué cor

Preguntas frecuentes

¿Airflow, Prefect, Dagster, Mage, Kubeflow Pipelines?

Airflow: el estándar, máxima madurez, syntax verbosa, multi-cloud. Prefect 3: API Python moderna, hybrid execution (workers locales + cloud). Dagster: orientado a software-defined assets (data-aware). Mage: notebooks-first, UI fuerte. Kubeflow Pipelines: K8s-native, orientado a ML específicamente. Para empezar en ML: Prefect o Dagster suelen ser más rápidos que Airflow.

¿Cada cuánto retrainear?

Depende del drift rate del dominio: NLP general (cambios estacionales): mensual. Fraud (atacantes adaptativos): diario u online. Recommendation (catalog drift): horario. Empezar conservador y acelerar si las métricas justifican.

¿Cuándo NO promover automáticamente?

(1) Decisiones de alto impacto (préstamos, salud): siempre human-in-the-loop. (2) Cuando el A/B test online no es factible: requerir aprobación manual. (3) Cuando el dataset tiene mucho ruido reciente: filtrar antes de retrainar.

¿schedule_interval o trigger por evento?

Para producción crítica: ambos. Schedule fijo como red de seguridad ("al menos 1 vez/día") + trigger por drift/degradación para ser reactivo. Falla del trigger no deja al sistema sin retraining.

¿Cómo manejo training caro (días) en un DAG?

Tarea train no corre en el worker Airflow — manda job a Vertex AI / SageMaker / Ray cluster vía API. La tarea Airflow es un proxy que polea hasta completion (SageMakerTrainingOperator lo hace).

¿Y si el retraining empeora producción?

Por eso existen los gates: shadow → canary → full rollout. Si la métrica online cae después de canary 10%: rollback automático del alias @champion al modelo previo, mantenido en el registry.

Referencias

- Huyen, Chip. Designing Machine Learning Systems (O'Reilly, 2022), cap. 9 — Continual Learning and Test in Production.
- Airflow docs — DAGs y operators.
- Prefect docs — flows modernos.
- Kirkpatrick et al. Overcoming catastrophic forgetting in neural networks (PNAS, 2017) — EWC.
- SageMaker Pipelines — equivalente managed AWS.

Material descargable

- Guía explicativa (PDF) — versión imprimible con todo el contenido de la clase.
- Presentación (PPTX) — deck PowerPoint listo para proyectar en clase.

- Notebook ejecutable (.ipynb) — abrilo desde el laboratorio del programa o desde Jupyter.

Clase 204 — Clase 204 — Shadow deployment y canary releases

> *Parte: 4 — MLOps · Fuente: Huyen cap. 11 + Fowler, Continuous Delivery + Istio traffic management. Duración estimada: 75 min.*

Objetivo

Desplegar un modelo nuevo sin arriesgar producción: primero en shadow (recibe tráfico real pero sus predicciones no se devuelven al usuario), después canary (1% → 5% → 25% → 100% del tráfico). Convertir "creo que esto es mejor" en "lo medí en producción real".

Resultados de aprendizaje

Al finalizar, el estudiante podrá:

- Implementar shadow mode: doble call (champion + challenger), respuesta del champion al user, log de ambas para análisis offline.
- Configurar canary release progresivo (1% → 5% → 25% → 100%) con K8s + Istio, o con feature flag a nivel app (LaunchDarkly, Unleash).
- Definir rollback automático: si la métrica del canary degrada >X%, volver al 100% champion en <5 min.
- Implementar A/B test correcto: muestra del mismo segmento, métrica primaria pre-registrada, poder estadístico calculado.
- Diferenciar shadow (sin riesgo, costo doble) de canary (riesgo limitado, costo nuevo solo).

Temas

#	Tema	Por qué importa
1	Shadow vs canary vs blue-green vs A/B	4 estrategias, propósitos distintos.
2	Implementación: app-level vs infra-level	Feature flag (LaunchDarkly) vs Istio/AWS A
3	Métrica de "salud" del canary	Latency, error rate, business KPI proxy.
4	Rollback automático	Sin esto, canary es solo "esperar a que al
5	Análisis de shadow data	Comparación distribucional (KS) + métricas
6	A/B test riguroso	Effect size + poder + duración mínima.

Definiciones y características

- Shadow deployment: el modelo nuevo recibe el mismo tráfico que el viejo, predice, pero su predicción NO se devuelve al usuario. Se loggea para análisis. Ventaja: cero riesgo. Costo: 2× compute.
- Canary release: % del tráfico real es servido por el modelo nuevo. Empezar 1%, escalar gradual. Si métricas OK al 100%, el canary se convierte en champion. Si no, rollback.
- Blue-green: dos versiones desplegadas simultáneamente (blue=actual, green=nueva). Switch del LB de blue→green en un momento puntual. Rollback = switch back. Sin gradualidad, pero rollback instantáneo.
- A/B test: comparación estadística entre dos variantes con asignación random por usuario (sticky). Mide impacto en KPI de negocio (no solo en accuracy). Requiere poder estadístico calculado.
- Sticky assignment: el mismo usuario siempre cae en la misma variante (hash por user_id). Evita que un usuario vea ambos modelos y "engañe" la medición.
- Rollback automático: cuando alguna métrica clave del canary cae bajo un umbral (latency p99 > X, error

rate > Y%, conversion rate cae > Z%), el sistema vuelve el peso del canary a 0 sin intervención humana.
- Guardrail metric: métrica que NO querés que empeore aunque la primaria mejore. Ej: latency, errors. Si guardrail rompe → rollback aunque "el modelo prediga mejor".

Dataset / recursos

- Stack ejemplo: FastAPI + Istio (canary) o feature flag in-process (shadow simple).
- Librerías: scipy.stats (A/B test), numpy.

Ejercicios

1. Shadow en proceso: en el FastAPI de Clase 199, agregá la lógica: cargá model_champion y model_challenger. En /predict, predeci con ambos, devolvé solo champion, log los dos en JSON. Después de 1000 requests, analizá la distribución de diferencias.
2. Canary con feature flag: implementá un toggle CANARY_PERCENT (env var). Si random.random() < CANARY_PERCENT/100, usá challenger. Sticky: hash por user_id para que el mismo user vea siempre lo mismo dentro del test.
3. Canary con Istio: VirtualService con weight: 95 / 5. kubectl apply y verificá distribución de tráfico con kubectl logs.
4. Rollback automático: agregá un sidecar (Python script) que cada 60 s consulta Prometheus: si latency_p99{model=challenger} > 200ms, cambia el weight a 100 / 0 automáticamente.
5. A/B test riguroso: calculá tamaño de muestra para detectar δ = 0.02 en accuracy con α=0.05, power=0.8. Corré el A/B test el tiempo necesario para acumular ese N. Reportá p-value + CI de la diferencia.

Homework verificable

Sistema de deployment con:

1. Modelo champion en producción sirviendo 100% del tráfico.
2. Endpoint /admin/canary que setea peso del challenger (sticky por user_id hash).
3. Métricas en Prometheus: predictions_total{model, class}, latency_seconds{model}, error_rate{model}.
4. Alerta + auto-rollback si: latency_p99{model=challenger} > 1.2 * latency_p99{model=champion} durante 5 min seguidos.
5. Dashboard Grafana con: distribución de tráfico, latency por modelo, agreement rate champion-challenger, business KPI proxy.
6. Runbook documentando los pasos: shadow 7 días → canary 1% 24 h → 5% 24 h → 25% 48 h → 100%.

Criterio de aceptación: simular un challenger con time.sleep(0.5) artificial (alta latencia). El rollback automático debe activarse en <10 min, sin pérdida de servicio para usuarios fuera del canary.

Errores comunes

Síntoma / mensaje	Causa y cómo arreglar
Canary muestra mejor accuracy pero peor bu	Métrica offline (accuracy) ≠ métrica onlin
El A/B test "ganó" pero la decisión no se	Regression to the mean + tamaño de muestra
El user ve modelos distintos en distintos	Asignación no es sticky. Fix: hash por use
Shadow no es "free" como prometieron — el	2× compute requiere 2× capacidad. Fix: dim
Canary a 5% no muestra estadística signifi	El N es chico — 5% × N usuarios típicos re
Rollback automático se dispara por ruido	Umbral muy estricto o ventana muy chica. F

Preguntas frecuentes

¿Shadow obligatorio antes de canary?

Para modelos críticos: sí. Shadow detecta bugs evidentes (modelo cae, output con NaN) sin afectar usuarios. Canary asume que el modelo funciona y mide impacto. Si saltás shadow, podés desplegar un modelo roto al 1% real.

¿Canary o blue-green?

Canary: gradual, deteccion temprana, recovery limitado. Blue-green: rollback instantáneo (un switch), sin gradualidad — el 100% se va o no se va. Para ML: canary es más común (cambios en distribución de tráfico revelan problemas que blue-green no).

¿Tengo que usar Istio?

No. Alternativas:

- AWS ALB/NLB con weighted target groups
- GCP Cloud Load Balancing weighted backends
- App-level feature flag (LaunchDarkly, Unleash) — más simple, opera fuera del infra
- Nginx con split_clients

Istio agrega valor cuando ya tenés service mesh.

¿Cómo elijo la métrica primaria del A/B?

Debe ser: (1) directly tied to business value (revenue, retention, conversion), (2) medible rápido (mejor: detectable en días, no meses), (3) una sola (multi-metric = comparison-shopping, mala práctica). Métricas técnicas (accuracy, AUC) son proxies — útiles pero no la métrica final.

¿Y si el canary mejora un segmento y empeora otro?

Análisis por segmento siempre. Si segmento A mejora 10% y segmento B empeora 5%: discutir trade-off con producto. Algunas opciones: (a) modelo distinto por segmento, (b) re-entrenar con foco en B, (c) decidir explícitamente que A pesa más.

¿Stat test correcto para A/B con accuracy?

Para proporciones (CTR, conversion): test de proporciones con corrección de continuidad. Para medias (latencia, revenue/user): t-test de Welch (no asume varianzas iguales). Para counts (clicks/usuario): Mann-Whitney o bootstrap (típicamente no-normal). Ver Partes 3 (clases 176-178) para más detalle.

Referencias

- Huyen, Chip. Designing Machine Learning Systems (O'Reilly, 2022), cap. 11 — Model Deployment y Continuous Delivery.
- Fowler, M. — CanaryRelease y BlueGreenDeployment.
- Istio Traffic Splitting tutorial.
- Kohavi, R. et al. Trustworthy Online Controlled Experiments (Cambridge, 2020) — la biblia del A/B test.
- LaunchDarkly / Unleash — feature flags as a service.

Material descargable

- Guía explicativa (PDF) — versión imprimible con todo el contenido de la clase.
- Presentación (PPTX) — deck PowerPoint listo para proyectar en clase.
- Notebook ejecutable (.ipynb) — abrilo desde el laboratorio del programa o desde Jupyter.

Clase 205 — Clase 205 — Interpretabilidad: SHAP, LIME, PDP, ICE

> *Parte: 4 — MLOps · Fuente: Molnar, Interpretable ML (2ª ed.) + Lundberg & Lee (2017, NIPS, SHAP paper) + Ribeiro et al. (2016, KDD, LIME paper). Duración estimada: 80 min.*

> *Esta clase es complementaria a la Clase 079 (SHAP profundo, Parte 1). Acá el foco es producción: cómo integrar interpretabilidad al servicio (/explain endpoint), cómo escalar SHAP a grandes datasets, y cómo presentar explicaciones a stakeholders no técnicos.*

Objetivo

Hacer interpretabilidad deployable: exponer un endpoint /explain que devuelva la atribución por feature de una predicción individual (SHAP/LIME), generar reportes globales (PDP/ICE) al promover un modelo, y entender los tres trade-offs: fidelidad vs simplicidad, global vs local, exacto vs aproximado.

Resultados de aprendizaje

Al finalizar, el estudiante podrá:

- Diferenciar local (esta predicción) vs global (modelo en general) y model-agnostic vs model-specific.
- Usar SHAP con TreeExplainer (rápido, exacto, árboles), KernelExplainer (lento, model-agnostic), DeepExplainer (redes neuronales) y Partition (recientes).
- Usar LIME para explicaciones model-agnostic en texto/imágenes/tabular.
- Generar PDP (Partial Dependence Plot) e ICE (Individual Conditional Expectation) con sklearn.inspection.
- Decidir cuándo SHAP es lo correcto y cuándo es overkill (ej. modelo lineal: usar coeficientes directamente).

Temas

#	Tema	Por qué importa
1	Local vs global, model-agnostic vs specifi	Vocabulario antes de elegir herramienta.
2	SHAP: TreeExplainer vs KernelExplainer	Exacto y rápido vs aproximado y lento.
3	LIME para texto/imágenes	Cuando SHAP no aplica.
4	PDP + ICE: efecto marginal vs heterogeneid	Lo que cambia con cada feature.
5	Endpoint /explain en producción	Latencia, caching, on-demand vs pre-comput
6	Comunicar a stakeholders no técnicos	Bar chart con top 5 features ≠ paper acade

Definiciones y características

- Interpretabilidad local: explica una predicción individual. "¿Por qué a este cliente lo categorizaste como riesgo?".
- Interpretabilidad global: explica el modelo en agregado. "¿Qué features pesan más en general?".
- Model-agnostic: funciona sin importar el modelo (LIME, KernelSHAP, PDP). Pro: portable. Con: más lento, aproximaciones.
- Model-specific: usa estructura del modelo (TreeSHAP usa árboles). Pro: rápido, exacto. Con: solo para esa familia.
- SHAP values: contribución de cada feature al cambio respecto a la predicción base, basado en teoría de juegos (valores de Shapley). Propiedades garantizadas: efficiency (suman al output), symmetry, dummy, additivity.
- TreeExplainer: para tree-based (XGBoost, LightGBM, CatBoost, sklearn RF). Polynomial time, exacto.
- KernelExplainer: agnostic. Aproxima Shapley values con regresión lineal local con kernel. O(2^n) para

exacto, K samples para aprox.
- PDP (Partial Dependence Plot): efecto promedio de una feature j sobre la predicción, marginalizando las demás. Asume independencia entre features (riesgo: si correladas, PDP miente).
- ICE (Individual Conditional Expectation): un PDP por instancia, sin promediar. Permite ver heterogeneidad (si el efecto de la feature varía por sub-grupo).
- LIME: aprende un modelo lineal/árbol simple alrededor de una instancia perturbando los inputs y observando outputs.

Dataset / recursos

- Dataset: California Housing.
- Modelos: XGBRegressor (TreeSHAP) y MLPRegressor (KernelSHAP/DeepSHAP).
- Librerías: shap>=0.45, lime, sklearn>=1.5, matplotlib.

Ejercicios

1. TreeSHAP: entrená XGB sobre California. explainer = shap.TreeExplainer(model). shap_values = explainer(X_test[:100]). shap.plots.waterfall(shap_values[0]) (local) y shap.plots.beeswarm(shap_values) (global).
2. KernelSHAP vs TreeSHAP: corré KernelExplainer con 100 background samples sobre el mismo XGB. Compará SHAP values con TreeSHAP. ¿Son iguales? ¿Cuánto tardó cada uno?
3. LIME tabular: LimeTabularExplainer sobre el mismo modelo. Explicá la misma instancia que en SHAP. Compará las features importantes.
4. PDP + ICE: sklearn.inspection.PartialDependenceDisplay.from_estimator(model, X, features=['MedInc', 'HouseAge'], kind='both'). Identificá si hay no-linealidad y/o heterogeneidad.
5. Endpoint /explain: extendé el FastAPI (Clase 199) con POST /explain que devuelve top-5 features + SHAP values + base value, para una instancia. Medí latencia — TreeSHAP debería ser <50 ms.

Homework verificable

Servicio FastAPI con:

1. POST /predict (de Clase 199).
2. POST /explain que devuelve {"prediction": float, "base_value": float, "top_features": [{"name": "MedInc", "shap": 0.23}, ...]}.
3. Endpoint GET /global-explanation que devuelve PDP + global SHAP precomputados (cacheados al startup).
4. UI HTML básica que muestra waterfall plot (SHAP) para inputs interactivos.
5. README que justifica: por qué TreeSHAP y no KernelSHAP, por qué top-5 y no top-20, por qué pre-computar global.

Criterio de aceptación: curl /explain -d '{"features":[...]}' devuelve top-5 features con SHAP, latencia <100 ms p99, y la UI muestra el waterfall correctamente.

Errores comunes

Síntoma / mensaje	Causa y cómo arreglar
KernelExplainer tarda 30 s por instancia	Esperado — es O(N samples × N features). F
SHAP values no suman a la predicción	Probablemente comparás shap_values + base_
PDP miente: la curva sugiere feature j no	feature j está correlada con otra feature;
Explicaciones distintas en cada corrida	LIME usa sampling random. Fix: fijar seed;
Top features no coinciden entre SHAP y LIM	Es esperado — son métodos distintos con as

Bar chart con SHAP global muestra "MedInc	Comunicación. Fix: traducir nombres ("MedI

Preguntas frecuentes

¿SHAP, LIME, Permutation Importance — cuál uso?

- Modelo árbol (XGBoost, LightGBM, RF): TreeSHAP sin pensar.
- Modelo agnóstico, dataset grande: Permutation Importance (sklearn) para global; LIME para local si SHAP es muy lento.
- Red neuronal: DeepSHAP o Integrated Gradients (Captum para PyTorch).
- Modelo lineal: coeficientes × valor de feature.

¿Por qué SHAP es "el estándar"?

Por las propiedades teóricas (axiomas de Shapley: efficiency, symmetry, dummy, additivity) y porque tiene implementaciones rápidas (TreeSHAP) para modelos populares. LIME es más viejo y más heurístico.

¿On-demand /explain o pre-computado?

- Pocas inferencias/día: on-demand está bien.
- Muchas inferencias, mismo input recurrente: cachear.
- Análisis batch (auditoría mensual): pre-computar en bulk con TreeSHAP — sklearn pipelines + Spark/Dask.

¿Cómo manejo SHAP para LLMs?

shap.Explainer(model) con Partition masker funciona para text. Para LLMs grandes: caro y lento. Alternativas: attention attribution, gradient-based (Captum), o prompt-level ("¿qué part del prompt fue importante?"). Área activa de research.

¿Y si el regulador me pide explicabilidad pero el modelo es Black Box?

Tres opciones: (1) Cambiá a un modelo intrínsecamente interpretable (Logistic Regression, GAM, EBM de interpret); (2) Generá SHAP por cada decisión + archivá; (3) Para fairness/compliance, también probá aequitas o fairlearn (Clase 161).

¿PDP o ALE?

PDP es más popular y simple, pero mentira con features correladas. ALE corrige eso, ligeramente más complejo. Si tus features son correladas (lo usual): ALE. Si independientes (raro): PDP alcanza.

Referencias

- Molnar, C. Interpretable Machine Learning (2ª ed., libro online gratis): https://christophm.github.io/interpretable-ml-book/
- Lundberg, S. & Lee, S. A Unified Approach to Interpreting Model Predictions (NIPS 2017) — SHAP.
- Ribeiro, M. et al. "Why Should I Trust You?": Explaining the Predictions of Any Classifier (KDD 2016) — LIME.
- SHAP docs — TreeExplainer, KernelExplainer, plots.
- InterpretML / EBM — modelos intrínsecamente interpretables (Microsoft Research).

Material descargable

- Guía explicativa (PDF) — versión imprimible con todo el contenido de la clase.
- Presentación (PPTX) — deck PowerPoint listo para proyectar en clase.
- Notebook ejecutable (.ipynb) — abrilo desde el laboratorio del programa o desde Jupyter.

Clase 206 — Clase 206 — Testing de datos: Great Expectations, Deequ

> *Parte: 4 — MLOps · Fuente: Huyen cap. 4 + Great Expectations docs (1.x) + Deequ paper. Duración estimada: 75 min.*

Objetivo

Aplicar testing como código a los datos: definir "expectations" (assertions sobre el dataset) y validarlas en cada corrida del pipeline. Detectar schema drift, outliers extremos, nulos inesperados antes de que lleguen al entrenamiento o a la predicción. Bug típico: no es que el modelo se rompe — es que la data está rota y nadie se entera.

Resultados de aprendizaje

Al finalizar, el estudiante podrá:

- Definir un Expectation Suite con Great Expectations 1.x: schema, ranges, uniqueness, null rates, regex.
- Generar un Data Docs HTML que documenta el dataset + resultados de validación (auditoría para regulador).
- Integrar GE en un pipeline DVC/Airflow: si validación falla, abortar el pipeline.
- Usar Deequ (Scala/Python via PyDeequ) para datasets grandes en Spark.
- Diferenciar data tests (sobre el dataset) de unit tests (sobre el código) — son ortogonales.

Temas

#	Tema	Por qué importa
1	Por qué unit tests no alcanzan en data pip	El código está OK; la data llega rota.
2	Expectation Suite: schema + business rules	Catalogar lo que es "data correcta" en est
3	Profiling automático	GE puede inferir un suite inicial del data
4	Data Docs: docs ejecutables	Auditoría + onboarding sin esfuerzo extra.
5	Checkpoints e integración con pipelines	El gate que aborta el flow si la data está
6	Deequ para Spark scale	Cuando el dataset no entra en pandas.

Definiciones y características

- Expectation: assertion sobre los datos. Ej: expect_column_values_to_not_be_null('user_id'), expect_column_values_to_be_between('age', 0, 120), expect_column_distinct_values_to_be_in_set('country', ['AR', 'BR', 'CL']).
- Expectation Suite: colección de expectations versionada (YAML/JSON).
- Validation: ejecutar el suite contra un batch real → result con success: true/false por expectation.
- Checkpoint: configuración que une data + suite + acción (postear a Slack, abortar pipeline, generar docs).
- Data Docs: HTML auto-generado que muestra: schema, ejemplos, resultados de validación, evolución.
- Profiling: GE puede generar un suite inicial inspeccionando el dataset (UserConfigurableProfiler). Punto de partida — siempre revisar manualmente.
- Deequ: librería de Amazon (Scala, también PyDeequ para Python) que hace lo mismo pero sobre Spark. Diseñada para TB.
- Pandera: alternativa Python-first, integrada con pandas/polars DataFrames con sintaxis decorator.

Dataset / recursos

- Dataset: California Housing (sin shift) como reference, después con shift sintético para ver fallas.

- Librerías: great-expectations>=1.0, opcional pandera, pydeequ.

Ejercicios

1. Bootstrap de un suite: gx init para crear el proyecto. gx datasource new apuntando a CSV. Profileá el dataset para suite inicial. Revisalo a mano: descomentá las expectations razonables, borrá las absurdas.
2. Custom expectations: agregá: expect_column_values_to_be_between('MedInc', 0, 20), expect_table_row_count_to_be_between(1000, 100000), expect_column_pair_values_A_to_be_greater_than_B('AveRooms', 'AveBedrms') (más rooms que bedrooms).
3. Checkpoint: configurá un checkpoint que (a) corre el suite, (b) si falla, abre un Slack alert. Corré con gx checkpoint run my_checkpoint.
4. Data Docs: gx docs build. Abrí el HTML. Mostrá a un PM/regulador hipotético: la suite + el último validation result.
5. Pandera alternative: el mismo schema con pandera: class HousingSchema(pa.DataFrameModel): MedInc: Series[float] = pa.Field(in_range={"min_value": 0, "max_value": 20}). Compará verbosidad y velocidad.

Homework verificable

Pipeline ML con:

1. Expectation Suite Great Expectations versionada en git (YAML/JSON).
2. Checkpoint que se corre como step del pipeline DVC o Airflow (Clase 203) antes del training.
3. Data Docs HTML buildeado en CI y publicado a GitHub Pages.
4. Slack alert (con webhook stub si no tenés real) cuando una expectation falla.
5. Demonstración de un dataset alterado (NaN inyectados, outliers, schema cambiado) que el pipeline detecta y aborta, sin que el modelo se entrene con data sucia.

Criterio de aceptación: el pipeline corre clean con data buena. Inyectando NaN en MedInc: el checkpoint falla, el pipeline aborta, el Slack alerta dispara, el modelo NO se re-entrena con data corrupta.

Errores comunes

Síntoma / mensaje	Causa y cómo arreglar
gx init no encuentra Python venv correcto	GE 1.x quiere su propio gx CLI. Fix: activ
expect_column_values_to_be_in_set falla po	Tu suite quedó vieja. Fix: actualizar el s
Validation pasa pero el modelo igual rinde	Validación de forma ≠ validación de distri
Profiler genera 200 expectations, todas se	El profiler es muy permisivo. Fix: empezar
El Data Docs no muestra el último validati	data_docs no se rebuildeó. Fix: agregar up
Performance lento en 10M filas	GE sobre pandas escanea todo. Fix: para vo

Preguntas frecuentes

¿Great Expectations, Pandera o Deequ?

- GE: más completo (Data Docs, suite versioning, plugin ecosystem), pero verboso.
- Pandera: Python-first, sintaxis decorator, mejor DX para devs. No tiene Data Docs.
- Deequ / PyDeequ: para Spark, datasets grandes.

Para equipos chicos en pandas: Pandera. Para empresas con auditoría: GE. Para Spark: Deequ.

¿Cuándo data testing vs schema validation (Pydantic)?

- Pydantic / dataclasses: validación por fila / record (en API requests). Rápido, in-process.
- GE / Pandera: validación por dataset (batch). Tests sobre agregados (medias, conteos), no solo schema.

Son complementarios: API valida cada request con Pydantic; pipeline batch valida cada slice con GE.

¿Cómo manejo expectations que cambian con el tiempo (estacionalidad)?

(1) Expectations relativas: "row count entre last_week × 0.8 y last_week × 1.2". (2) Múltiples suites por época. (3) Marcar la expectation como warning (no aborta) y revisar manualmente.

¿GE rompe mi pipeline cada deploy?

Sí, si tu suite es muy estricto y la data cambia legítimamente. Fix: separar expectations críticas (abortan: PII no debe ser null, schema no debe cambiar) de alertas (warning, no abortan: distribución shift).

¿Versionar el Expectation Suite en git?

Sí siempre. Cambios al suite van por PR como cambios de código — review, CI, merge. El suite es contrato de datos = parte del contrato del producto.

¿Y datos de producción real-time (streaming)?

GE no es ideal para streaming. Alternativas: Apache Griffin, validations custom en el consumer Kafka, o aceptar que el testing es batch (validar cada 1 h sobre el último window).

Referencias

- Huyen, Chip. Designing Machine Learning Systems (O'Reilly, 2022), cap. 4 — Training Data.
- Great Expectations docs — 1.x es la versión actual (rompió compat con 0.x).
- Pandera docs — schema validation for pandas/polars.
- PyDeequ — Spark-scale data testing.
- Schelter et al. Automating large-scale data quality verification (VLDB 2018) — paper de Deequ.

Material descargable

- Guía explicativa (PDF) — versión imprimible con todo el contenido de la clase.
- Presentación (PPTX) — deck PowerPoint listo para proyectar en clase.
- Notebook ejecutable (.ipynb) — abrilo desde el laboratorio del programa o desde Jupyter.

Clase 207 — Clase 207 — Testing de modelos: invariance + behavioral tests

> *Parte: 4 — MLOps · Fuente: Ribeiro et al., Beyond Accuracy: Behavioral Testing of NLP Models with CheckList (ACL 2020, best paper) + Huyen cap. 9 + Deepchecks docs. Duración estimada: 80 min.*

Objetivo

Ir más allá de "accuracy en hold-out": tests que verifican que el modelo se comporta como debería en casos específicos. Tres familias: invariance tests ("misma predicción si reemplazo Juan por María"), directional tests ("si subo el ingreso, la proba de aprobar el préstamo no debe bajar"), minimum functionality tests ("predicción correcta sobre casos canónicos hand-crafted").

Cierra la Parte 4: con esto, el modelo en producción tiene 6 capas de protección (data tests, model tests, monitoring, shadow, canary, rollback).

Resultados de aprendizaje

Al finalizar, el estudiante podrá:

- Implementar invariance tests (perturbaciones que NO deben cambiar la predicción): swap de nombres protegidos, sinónimos, ruido pequeño.
- Implementar directional tests (perturbaciones que deben cambiar la predicción en una dirección esperada): subir ingreso → menos riesgo crediticio.
- Crear minimum functionality test sets (MFT): casos hand-crafted que cubren cada feature/segmento crítico.
- Integrar tests de modelo en pytest y correrlos en CI antes de promover (gate de Clase 197).
- Usar Deepchecks o CheckList para suites pre-armados (especialmente NLP).

Temas

#	Tema	Por qué importa
1	Accuracy ≠ corrección — los 3 tipos de tes	Modelo "bueno" puede tener bugs sistemátic
2	Invariance tests	Detecta sesgos (mismo CV con nombre cambia
3	Directional tests	Detecta inconsistencias (más experiencia →
4	MFT (Minimum Functionality)	Casos triviales que un humano resuelve sin
5	Slice-based testing	Performance por subgrupo, no solo overall.
6	Integración con CI	Tests verdes ≠ gate, requerir explícitamen

Definiciones y características

- Invariance test (INV): assert model(x) == model(perturb(x)). La perturbación es una transformación que no debería cambiar la predicción. Ej: cambiar nombre, agregar puntuación, sinónimos.
- Directional Expectation test (DIR): assert sign(model(x_modified) - model(x)) == expected_direction. Ej: si aumento income, P(default) debe bajar o quedar igual — nunca subir.
- Minimum Functionality Test (MFT): un set chico de ejemplos hand-crafted etiquetados manualmente. El modelo DEBE acertar todos. Ej: "Pésimo servicio" → negativo en un sentiment classifier.
- Slice-based testing: medir métrica por sub-grupo (gender, age, country) y reportar el peor caso, no el promedio. El promedio esconde regresiones en minorías.
- CheckList (Ribeiro et al.): framework + paper que estandarizó los 3 tipos de test (INV/DIR/MFT) para NLP. Acabó ganando best paper en ACL 2020.
- Deepchecks: librería Python con suites pre-armados para tabular, vision, NLP. Detecta data leakage, train-test drift, performance bias.
- Adversarial testing: variante de invariance — perturbaciones pequeñas adversarialmente diseñadas que rompen el modelo. cleverhans, foolbox para deep learning.

Dataset / recursos

- Tabular: Adult Income (UCI) con gender como atributo sensible.
- NLP: subset de IMDb reviews para sentiment.
- Librerías: deepchecks>=0.18, checklist, pytest, pandas.

Ejercicios

1. MFT tabular: para un modelo de crédito sobre Adult, hand-craft 20 casos: 10 obvios "should approve" (alto ingreso, educación alta, sin debts) y 10 obvios "should reject" (ingreso bajo, edad joven, sin historial). Asseré con pytest que el modelo acierta los 20.
2. Invariance — gender swap: tomá 500 registros, cambiá gender M ↔ F, dejá el resto igual. Mediá

accuracy(predictions_original == predictions_swapped). Si <99%: el modelo está usando gender (probablemente vía proxy).

3. Directional — income up: para los mismos 500, multiplicá income × 1.5. La proba predicha de >50K debería subir (o quedar igual) en >95% de los casos. Si baja en muchos: bug.
4. Slice-based: calculá accuracy por slice (gender × race × age_bucket). Reportá top 5 worst slices. Si el peor slice tiene accuracy 0.55 y el promedio 0.85: tenés un problema de fairness invisible en métricas agregadas.
5. Pytest gate: empaquetá los 4 tests anteriores en tests/test_model_behavior.py. Hacelo correr en GitHub Actions (Clase 197) como required check. Si tests rojos: PR no se mergea.

Homework verificable

Repo con:

1. tests/test_model_behavior.py con ≥4 tests: 1 MFT, 1 INV, 1 DIR, 1 slice-based.
2. CI workflow que corre pytest tests/test_model_behavior.py después de cada training.
3. deepchecks full suite ejecutado, reporte HTML guardado.
4. Reporte por slice (markdown) con métricas en gender, race, age_bucket y bottom-5 worst slices identificados.
5. Documentación de 3 bugs reales encontrados por estos tests durante el desarrollo (puede ser bug fixed; lo importante es demostrar que los tests sirven).

Criterio de aceptación: si en una iteración futuro alguien cambia el modelo y un test rompe (ej. gender swap accuracy cae de 0.99 a 0.85), el CI falla y el PR no se mergea sin discusión explícita.

Errores comunes

Síntoma / mensaje	Causa y cómo arreglar
Gender swap test "falla" pero el modelo no	Está usando un proxy (ocupación, código po
MFT pasa pero el modelo rinde mal en produ	El MFT está sesgado al subset que vos imag
Slice-based: el worst slice tiene n=12 y m	Sample size insuficiente para sacar conclu
Tests behavioral pasan todos siempre	Probablemente las perturbaciones son trivi
Directional test falla "raro"	Quizás la relación NO es monótona — ej: in
CI lento por tests	Si pytest tarda 20 min, nadie los corre lo

Preguntas frecuentes

¿Esto reemplaza al test set normal?

No, lo complementa. Hold-out test set mide performance estadística sobre distribución similar a training. Behavioral tests miden correctness sobre casos importantes hand-crafted. Necesitás ambos.

¿Cuántos behavioral tests son suficientes?

CheckList paper sugiere 1 MFT + 1 INV + 1 DIR por capability del modelo. En tabular: por cada feature crítica. En NLP: por cada fenómeno linguístico. Empezá con 10-20 tests bien diseñados, no con 1000 mediocres.

¿Esto es lo mismo que fairness testing?

Hay overlap. Fairness suele ser un subset de invariance tests (perturbar atributos protegidos) + slice-based (medir disparidad). Pero behavioral tests también cubren directional/MFT que no son sobre fairness directamente.

¿Deepchecks o escribir tests propios?

Deepchecks ofrece checks pre-armados (data leakage, integrity, performance bias) que cubren casos genéricos. Para tu dominio específico (correctness de negocio), escribís tests custom. Usar ambos.

¿Cómo manejo tests probabilísticos (no determinísticos)?

(1) Fijar seeds. (2) Para tests aproximados: assert metric > 0.99 (no == 1.0). (3) Bootstrap CI para asseré lower_bound > threshold.

¿Y modelos LLM?

Aplican igual los 3 tipos:

- MFT: prompts canónicos con respuestas esperadas exactas/reguladas.
- INV: paráfrasis del mismo prompt no debería cambiar el output dramáticamente.
- DIR: agregar "be brief" al prompt debería acortar output.

Frameworks: promptfoo, LangSmith, Phoenix (Arize).

Referencias

- Ribeiro, M. et al. Beyond Accuracy: Behavioral Testing of NLP Models with CheckList (ACL 2020) — el paper que estableció el vocabulario.
- Huyen, Chip. Designing Machine Learning Systems (O'Reilly, 2022), cap. 9 — Continual Learning and Test in Production.
- Deepchecks docs — suites para tabular/vision/NLP.
- CheckList GitHub — framework de Ribeiro.
- promptfoo / LangSmith — equivalente para LLMs.

Material descargable

- Guía explicativa (PDF) — versión imprimible con todo el contenido de la clase.
- Presentación (PPTX) — deck PowerPoint listo para proyectar en clase.
- Notebook ejecutable (.ipynb) — abrilo desde el laboratorio del programa o desde Jupyter.

Cierre de la parte

Fin del bundle consolidado de Parte 4 — MLOps — Modelos en Producción · 14 clases.