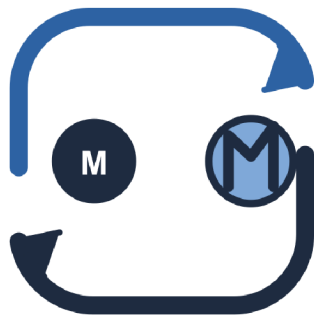


MLOps et Industrialisation des modèles IA

TP 1

Construire une expérience reproductible et versionnée

Du modèle au système ML traçable, validé et exploitable

**MLOPS**
GL-M326

TP · SOMMAIRE

Table des matières

Cadre du laboratoire	3
Objectifs pédagogiques	3
Consigne générale	3
1. Installer et prendre en main le kit — 15 min	3
2. Auditer le notebook et valider les données — 20 min	5
3. Configurer, découper et entraîner — 35 min	5
4. Évaluer et explorer les décisions — 25 min	6
5. Sauvegarder et tracer avec MLflow — 25 min	7
6. Exécuter et conserver les quatre expériences	8
Rappel — le rôle de C dans la régression logistique	8
7. Rédiger, contrôler et remettre	10
Transition vers TP2 et extension facultative	11
Annexe — contrat technique du starter	11
Fonctions à compléter	11
Sorties de l'évaluation	12
Modèle et manifeste	13
Exemple MLflow à intégrer dans main	14
Ce que les contrôles prouvent	16



Cadre du laboratoire

- **Modalité** : travail individuel.
- **Durée** : 2 h encadrées, puis 1 h de finalisation individuelle.
- **Prérequis** : Python, pandas, classification binaire et bases de Git.
- **Livrables** : `MLOps_TP1_Nom_Prenom.zip` et le rapport `reports/tp1_report.md` inclus dans cette archive.
- **Échéance de soumission** : mercredi 30 septembre 2026 à 23 h 59.
- **Retard** : aucun livrable soumis après cette échéance ne sera accepté.

Objectifs pédagogiques

À l'issue du TP, l'étudiant doit être capable de :

- transformer un notebook en expérience exécutable ;
- externaliser les paramètres, versionner le code et tracer les exécutions ;
- contrôler un split et l'absence de fuite dans le prétraitement ;
- interpréter les métriques d'un problème déséquilibré ;
- produire des preuves permettant l'inspection et la réexécution du travail.

Consigne générale

Le cas d'étude porte sur la prédiction de pannes. À partir du notebook fourni, construisez une expérience dont le code, les données, la configuration, le modèle et les métriques peuvent être inspectés. Le modèle de régression logistique sert uniquement de support expérimental : l'évaluation porte sur la démarche MLOps et les preuves produites, non sur l'amélioration du score ou le choix d'un modèle.

Le travail encadré couvre les étapes 1 à 5. La finalisation comprend les quatre expériences, la comparaison A/B, le rapport et la remise. Consultez `docs/donnees.md` pour la nature et les limites du jeu fourni. DVC, jeux de données alternatifs, quality gates, API et Model Registry ne sont pas demandés dans ce TP.

1. Installer et prendre en main le kit — 15 min

Préparation de salle. Testez l'installation sur votre poste avant la séance. La première synchronisation requiert Internet. En cas de réseau indisponible, utilisez uniquement un cache UV préparé par l'enseignant avec `uv sync --locked --offline` ; sinon, conservez le message

d'erreur et reprenez l'installation lorsqu'une connexion est disponible. Ne régénérez jamais `uv.lock`.

Téléchargez le PDF et le ZIP « Code de départ » depuis la page du TP. Le ZIP contient directement le projet, sans sous-dossier `student/`. Travaillez dans une copie personnelle et gardez le PDF accessible.

Environnements pris en charge : Linux, macOS avec Make, ou **Windows dans WSL2 Ubuntu**.

Toutes les commandes suivantes s'exécutent dans un terminal Unix; PowerShell natif ne suffit pas.

Sous Windows, installez WSL2 Ubuntu avant le TP (`wsl --install -d Ubuntu` dans PowerShell administrateur, redémarrage si demandé), puis ouvrez Ubuntu. Sous Ubuntu, si nécessaire :

TERMINAL · BASH

```
sudo apt update
sudo apt install -y git make unzip curl
```

Sous macOS, les outils de ligne de commande peuvent être installés avec `xcode-select --install`. Vérifiez `git --version` et `make --version`. Si uv manque :

TERMINAL · BASH

```
curl -LsSf https://astral.sh/uv/install.sh | sh
```

Rouvrez le terminal, vérifiez `uv --version`, puis placez-vous dans le dossier du ZIP. Sous WSL, le téléchargement Windows est généralement accessible sous `/mnt/c/Users/<utilisateur>/Downloads`; copiez le ZIP dans votre espace Linux.

TERMINAL · BASH

```
mkdir -p "$HOME/ensias-tps/MLOps/TP1_NOM_PRENOM"
# Remplacer KIT.zip par le nom exact du fichier téléchargé.
unzip KIT.zip -d "$HOME/ensias-tps/MLOps/TP1_NOM_PRENOM"
cd "$HOME/ensias-tps/MLOps/TP1_NOM_PRENOM"
uv python install 3.13
make install
make config
make report
make doctor
make test
make inspect-data
```

À vérifier : `.python-version`, `uv.lock`, `data/`, `docs/contrat.md` et le squelette `maintenance_ml/modeling/train.py`.

- Python 3.13 est imposé; `make doctor` affiche l'interpréteur réellement utilisé.
- `make test` valide le kit, pas votre solution.
- `make train` échoue initialement avec `NotImplementedError` : c'est normal.
- En cas de problème réseau, conservez l'erreur; ne régénérez pas `uv.lock`.

Preuve : capture commentée de doctor et des tests, versions dans le rapport.

2. Auditer le notebook et valider les données — 20 min

TERMINAL · BASH

```
make notebook
```

Ouvrez `notebooks/0.01-starter.ipynb` dans Jupyter. Dans une cellule temporaire, `import sys; print(sys.executable)` doit afficher le Python du `.venv` du kit. Redémarrez le kernel et exécutez toutes les cellules deux fois. Le notebook résout `../data/...` depuis `notebooks/`, contrairement au programme lancé à la racine.

Dans le rapport, relevez cinq défauts réels. Pour chacun : **observation** → **risque** → **correction** → **preuve**. Cherchez notamment paramètres en dur, chemins fragiles, split sans seed, métriques seulement affichées, provenance absente et modèle écrasé. Mentionnez aussi les bonnes pratiques déjà présentes. **N'inventez pas de fuite**.

`make check-data` est le contrôle de **votre** implémentation : exécutez-le après avoir complété `validate(config)` selon `docs/contrat.md`. La commande `make inspect-data`, elle, est disponible dès le départ et ne remplace pas ce travail.

TERMINAL · BASH

```
make check-data
```

Attendu : 5 000 lignes, 4 800 non-pannes, 200 pannes, aucune donnée manquante, deux classes. La faible prévalence des pannes justifie une comparaison avec une baseline majoritaire qui prédit systématiquement la classe 0 (« aucune panne »), afin d'interpréter l'accuracy, le recall et le F1-score.

3. Configurer, découper et entraîner — 35 min

`make config` copie la configuration suivante sans écraser votre fichier :

CONFIGURATION · YAML

```
seed: 42
test_size: 0.25
decision_threshold: 0.50
analysis_thresholds: [0.01, 0.05, 0.10, 0.30, 0.50, 0.70]
business_costs:
  false_negative: 10
  false_positive: 2
mlflow:
  tracking_uri: http://127.0.0.1:5000
  experiment_name: tp1-predictive-maintenance
model:
  type: logistic_regression
  C: 1.0
  max_iter: 1000
```

`decision_threshold: 0.50` est le **seuil de référence pédagogique** utilisé pour les métriques principales et la comparaison des quatre expériences. Il ne constitue pas une recommandation opérationnelle : les seuils de `analysis_thresholds` servent à étudier les compromis sur ce jeu de données.

Complétez `prepare` et `train` :

- exclure `machine_id` et la cible des variables explicatives ;
- utiliser `train_test_split(..., random_state=seed, stratify=y)` ;
- reprendre le `ColumnTransformer` et le `Pipeline` du notebook ;
- lire tous les paramètres expérimentaux dans le YAML.

Attendu : 3 750 lignes train (150 pannes), 1 250 test (50 pannes). Le preprocessing voit 3 750 lignes au `fit`, jamais 5 000.

Initialisez le dépôt personnel (le ZIP ne contient pas de `.git`) :

TERMINAL · BASH

```
git init
# Si nécessaire, renseigner votre identité locale avec git config user.name
# et git config user.email avant le premier commit.
git add .
git commit -m "TP1 : démarrage et configuration"
```

Committez de nouveau le programme quand l'implémentation est terminée. Avant chaque run enregistré, `git status --short` doit être vide. Le `.gitignore` exclut les fichiers générés, tout en conservant code, données et configuration. Aucun compte GitHub ni push n'est demandé.

4. Évaluer et explorer les décisions — 25 min

Complétez `evaluate` :

- quatre métriques et matrice TN/FP/FN/TP au seuil 0.50 ;
- distributions train/test et analyse des six seuils ;
- coût $10 \times \text{FN} + 2 \times \text{FP}$, fréquence des alertes, sans réentraîner.

Comparez à « toujours 0 » : 1 200/1 250 (96 %) mais recall et F1 nuls. C'est le point à expliquer; ne cherchez pas artificiellement un meilleur score.

Les valeurs 10 (FN) et 2 (FP) sont des **poids relatifs de décision pour cet exercice**, sans unité monétaire ni validation métier. Elles servent à comparer des scénarios, pas à justifier une politique industrielle.

Limite essentielle : avec 50 pannes de test, une panne détectée fait varier le recall de 2 points. L'analyse sur test reste exploratoire. Conservez 0.50 comme seuil de référence pédagogique pour les métriques principales des quatre runs. Pour recommander un seuil en production, il faudrait le sélectionner sur une validation distincte, puis l'évaluer sur un test indépendant comportant assez de pannes. Ce protocole supplémentaire n'est pas demandé ici.

Le split aléatoire stratifié est adapté au jeu pédagogique, dont les observations sont ici supposées indépendantes et issues de la même distribution. En maintenance réelle, séparez les données par période ou par machine lorsque cela reflète le déploiement; sinon, des observations trop proches peuvent surestimer la performance.

Preuve : baseline, distributions, matrice et tableau des seuils commentés; justifiez une métrique principale et discutez les coûts hypothétiques.

5. Sauvegarder et tracer avec MLflow — 25 min

Dans un second terminal, depuis la racine du kit :

```
TERMINAL · BASH
```

```
make mlflow-server
```

Gardez-le ouvert. Dans le premier :

```
TERMINAL · BASH
```

```
make server-check
```

Ouvrez `http://127.0.0.1:5000`. En cas de port occupé, vérifiez quel service est lancé avant de le fermer; le serveur du TP utilise SQLite `mlflow.db` et `mlartifacts/` dans le dossier courant. Il n'est pas installé globalement.

Complétez `main` : création des dossiers, sauvegarde du Pipeline, JSON, création du run, paramètres/métriques, manifeste, journalisation des fichiers. L'annexe fournit l'ordre et un exemple d'instrumentation. N'utilisez pas un manifeste factice si une étape échoue; corrigez la cause avant de poursuivre.

TERMINAL · BASH

```
git add maintenance_ml configs
git commit -m "TP1 : pipeline et preuves"
make train
```

Attendu : un run FINISHED, le modèle et le manifeste dans `models/`, les quatre fichiers JSON dans `reports/`. Vérifiez dans MLflow les paramètres, les métriques et les fichiers sous `evidence/`. Conservez une capture réelle qui montre le `run_id`. Le hash Git est obligatoire, pas une extension.

6. Exécuter et conserver les quatre expériences

Terminez cette étape et le rapport après la séance si nécessaire. Modifiez uniquement `model.C` dans le YAML entre expériences. Committez chaque changement avant de lancer. Ne modifiez ni le code ni le YAML entre A et B.

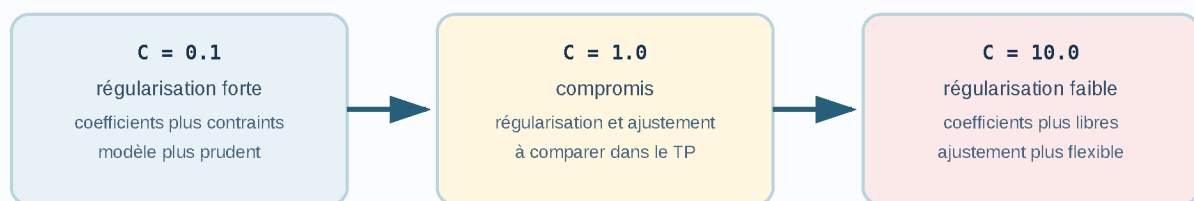
La configuration créée au départ contient `C : 1.0`. **Avant la toute première exécution**, remplacez-la par `C : 0.1`, vérifiez la valeur enregistrée, committez, puis lancez C01. Cela évite d'archiver par erreur un snapshot C01 avec `C=1.0`.

Rappel — le rôle de `C` dans la régression logistique

`C` agit **pendant l'entraînement** : c'est l'inverse de la force de régularisation. Une valeur petite impose davantage de régularisation et produit des coefficients plus contraints ; une valeur grande laisse davantage de liberté au modèle pour s'ajuster aux données. Son effet concret dépend des données, de leur mise à l'échelle, des variables et du solveur. Ce n'est pas le seuil de décision.

Deux paramètres, deux moments différents

1. Pendant l'entraînement : C règle la régularisation



2. Après l'entraînement : le seuil transforme une probabilité en alerte



C change le modèle appris ; le seuil change la décision prise à partir de ses probabilités.

Comparer C exige de garder le split et les autres paramètres constants. Comparer les seuils ne réentraîne pas le modèle.

Lecture du schéma : `C=0.1` correspond à une régularisation forte, `C=1.0` à une régularisation intermédiaire dans cette échelle, et `C=10.0` à une régularisation faible. Ce sont trois conditions expérimentales pédagogiques, pas des valeurs universellement recommandées. Le `decision_threshold` intervient seulement après `predict_proba`, pour choisir une alerte. Les deux paramètres doivent être consignés dans le manifeste, mais ils répondent à des décisions différentes.

Ordre	Label	C	Action après make train
1	C01	0.1	<code>make snapshot LABEL=C01</code>
2	A	1.0	<code>make snapshot LABEL=A</code>
3	B	1.0	<code>make snapshot LABEL=B</code>
4	C10	10.0	<code>make snapshot LABEL=C10</code>

Pour chaque changement de C :

TERMINAL · BASH

```
# Modifier model.C, puis :  
git add configs/train.yaml  
git commit -m "TP1 : configuration de l'expérience"  
make train  
# Exécuter ici la commande snapshot du tableau.
```

Entre A et B, lancez `make clean`, puis `make train` et le snapshot B. `clean` supprime uniquement les sorties courantes, sans toucher aux preuves archivées. Le snapshot refuse d'écraser un label existant; déplacez l'ancien dossier hors du kit si vous devez recommencer après une correction.

TERMINAL · BASH

```
make compare
```

Invariants A/B : hashes données/config/code/lockfile, commit, paramètres, métriques et probabilités sur le même test (tolérance absolue 1e-12). **Différences normales** : `run_id` et timestamps. Le hash binaire du modèle est une information supplémentaire : des prédictions identiques ne garantissent pas une sérialisation identique entre environnements. Ne comparez pas les manifestes bruts en exigeant une identité octet par octet.

Chaque snapshot sous `reports/runs/<label>/` conserve configuration, code, lockfile, modèle, manifeste, résultats et `mlflow_run.json` exporté du serveur. Ces preuves sont consultables sans votre base MLflow. Une réexécution chez un tiers créera de nouveaux `run_id`, sans recréer les runs historiques. Python 3.13 et `uv.lock` fixent l'environnement Python, mais pas entièrement le système, les bibliothèques numériques sous-jacentes ou le matériel : n'exigez pas une identité binaire entre machines.

7. Rédiger, contrôler et remettre

Complétez le modèle créé par `make report`. Pour chaque preuve, donnez une commande ou un fichier, une observation, une interprétation et une limite. Captures réelles obligatoires : installation/tests, notebook, fit/split, MLflow, analyse des seuils et comparaison A/B. Stockez-les dans `reports/figures/`, avec nom explicite et légende dans le rapport.

Les six questions sont identiques à celles du modèle de rapport. Pour chacune, appuyez-vous sur une commande, un fichier ou une capture réellement produits.

1. **Q1 — Rejeu contrôlé.** Expliquez ce que fixe `seed: 42`, puis citez deux éléments supplémentaires nécessaires pour rejouer A. Indiquez où chacun est enregistré et une cause de divergence qui peut subsister.
2. **Q2 — Provenance vérifiable.** À partir d'un snapshot A, montrez comment un lecteur relie le run MLflow au commit, au YAML, aux données et au lockfile. Distinguez ce que prouve un `run_id` de ce qu'il ne prouve pas hors de votre serveur.
3. **Q3 — Absence de fuite.** Décrivez le chemin d'une variable depuis le CSV jusqu'à `StandardScaler.fit`. Quelle preuve montre que le fit a vu 3 750 lignes, et quelle fuite resterait possible malgré cette preuve ?
4. **Q4 — Décision sous déséquilibre.** Comparez votre modèle à « toujours 0 » avec accuracy, recall, F1 et matrice de confusion. Expliquez la décision que chaque approche entraînerait pour les pannes, puis la limite de ce test.
5. **Q5 — Seuil, coût et validation.** Choisissez deux seuils de votre tableau, calculez/interprétez l'écart de FN, FP, coût et alertes. Pourquoi ce choix ne doit-il pas être promu directement en production ?
6. **Q6 — A/B et paramètre C.** Comparez A et B, puis C01 et C10. Dites ce que l'identité A/B établit, ce que `C` modifie pendant l'entraînement et pourquoi `C` ne doit pas être confondu avec `decision_threshold`.

TERMINAL · BASH

```
make test
make compare
make validate-submission
make package NAME=Nom_Prenom
```

La commande `package` vérifie d'abord la remise, puis crée le ZIP dans le dossier parent. Elle inclut les modèles et snapshots même s'ils sont ignorés par Git. Elle exclut environnement, historique Git, base MLflow, stockage serveur et caches. Ne créez pas votre remise avec `git archive`, qui omettrait les preuves générées.

Acceptation pédagogique : pipeline exécutable, configuration effectivement lue, provenance complète, quatre runs archivés, comparaison A/B, analyse des classes/seuils et rapport argumenté. Aucun minimum d'accuracy ou F1 n'est imposé. Le contrôle automatique n'évalue pas la pertinence des explications ni l'authenticité des captures : son succès est nécessaire, mais non suffisant.

Pour vérifier la portabilité, extrayez le ZIP ailleurs, installez, puis exécutez `make validate-submission` : les preuves archivées se contrôlent sans serveur. Pour **réentraîner**, initialisez un nouveau dépôt Git, committez cette copie, lancez son serveur MLflow et `make train`. Ne créez pas de nouveaux snapshots avec les labels historiques. Les anciens commits restent documentés dans les manifestes; leur historique Git complet n'est pas inclus dans la remise.

Transition vers TP2 et extension facultative

Le TP2 reprend configuration, train.py, modèle, métriques, manifeste et rapport. Conservez votre archive TP1. Les scénarios DVC, données invalides et quality gates seront introduits au TP2. Facultativement, après validation du parcours nominal, explorez des seuils sur une validation séparée ou une métrique par segment dans une copie distincte. Documentez bénéfice, preuve et limite sans remplacer les quatre expériences obligatoires. Ces explorations libres ne doivent pas être passées à `make validate-submission` : le validateur automatise uniquement le parcours nominal et ses quatre snapshots.

Annexe — contrat technique du starter

Les chemins sont relatifs à la racine du kit. Les JSON sont des objets sauf `threshold_analysis.json`, qui est une liste. Les nombres doivent être finis, les taux entre 0 et 1, les effectifs entiers. Les clés sont sensibles à la casse. `json.dumps(payload, indent=2, allow_nan=False)` permet d'éviter les NaN.

Fonctions à compléter

Fonction	Entrée	Retour et obligation
<code>validate(config)</code>	configuration	DataFrame lu depuis <code>data/raw/machines.csv</code> ; colonnes et valeurs contrôlées
<code>prepare(frame, config)</code>	données brutes	<code>X_train, X_test, y_train, y_test</code> ; retirer <code>machine_id</code> et <code>machine_failure</code> de X
<code>train(X_train, y_train, config)</code>	train uniquement	Pipeline ajusté : ColumnTransformer puis LogisticRegression
<code>evaluate(model, X_test, y_test, config)</code>	modèle et test	dictionnaire metrics, dictionnaire confusion, liste thresholds

Validation minimale : colonnes du CSV présentes, aucune valeur manquante, `machine_id` unique, `machine_type` dans A/B/C, cible contenant les deux classes 0/1, capteurs numériques finis. Pour ce TP, lisez les noms des colonnes dans le CSV et définissez ce schéma explicitement dans le code. Les noms de colonnes sont un contrat de données; seed, C, seuils et coûts restent dans le YAML. Les validations métier avancées et les datasets alternatifs appartiennent au TP2.

Découpez avant tout fit :

PYTHON

```
X = frame.drop(columns=["machine_failure", "machine_id"])
y = frame["machine_failure"]
X_train, X_test, y_train, y_test = train_test_split(
    X, y, test_size=config["test_size"],
    random_state=config["seed"], stratify=y,
)
```

Reprenez le ColumnTransformer du notebook. Appliquez `pipe.fit(X_train, y_train)` et jamais `fit` sur le test. Pour prouver ce point, affichez `len(X_train)` et `pipe.named_steps["preprocess"].named_transformers_["num"].n_samples_seen_`. Les deux valent 3 750 dans la configuration fournie.

Sorties de l'évaluation

`reports/metrics.json` : `accuracy`, `precision`, `recall`, `f1`, calculés au `decision_threshold` configuré (0.50 pour les quatre expériences obligatoires). Utilisez `zero_division=0` pour precision, recall et F1, et expliquez les zéros.

PYTHON

```
positive = list(model.classes_).index(1)
probabilities = model.predict_proba(X_test)[:, positive]
predictions = (probabilities >= config["decision_threshold"]).astype(int)
```

`reports/confusion_matrix.json` : TN, FP, FN, TP entiers. Utilisez `confusion_matrix(y_test, predictions, labels=[0, 1]).ravel()`.

`reports/threshold_analysis.json` : six objets, dans l'ordre de `analysis_thresholds`. Chaque objet contient `threshold`, `accuracy`, `precision`, `recall`, `f1`, TN, FP, FN, TP, `cost`, `alert_rate`.

- `cost = business_costs.false_negative * FN + business_costs.false_positive * FP;`
- `alert_rate = (TP + FP) / nombre_de_lignes_test;`
- recalculer les prédictions et métriques pour chaque seuil;
- conserver le modèle et le split : aucun réentraînement dans cette boucle.

`reports/class_distribution.json` : objets `train` et `test`, chacun contenant `"0"`, `"1"` (effectifs), `total` et `failure_rate` (proportion de classe 1). Présentez les proportions en pourcentages dans le rapport.

Modèle et manifeste

`models/model.joblib` contient le **Pipeline complet** sauvegardé par joblib, pas seulement le classifieur. Le rechargement doit accepter les features brutes.

`models/run_manifest.json` contient obligatoirement :

Clé	Valeur
<code>run_id</code>	ID du run MLflow actif
<code>data_sha256</code>	SHA-256 des octets de <code>data/raw/machines.csv</code>
<code>config_sha256</code>	SHA-256 des octets de <code>configs/train.yaml</code>
<code>code_sha256</code>	SHA-256 des octets de <code>maintenance_ml/modeling/train.py</code>
<code>lock_sha256</code>	SHA-256 des octets de <code>uv.lock</code>
<code>model_sha256</code>	SHA-256 des octets de <code>models/model.joblib</code>
<code>git_commit</code>	sortie de <code>git rev-parse HEAD</code>
<code>git_dirty</code>	false : code et configuration committés avant le run
<code>python_version</code>	<code>platform.python_version()</code>
<code>seed, test_size, model, decision_threshold</code>	valeurs exactes de la configuration
<code>metrics</code>	copie des quatre métriques de <code>reports/metrics.json</code>
<code>artifact</code>	chaîne <code>models/model.joblib</code>
<code>fit_rows</code>	nombre de lignes effectivement vues à l'ajustement

Un timestamp ISO peut être ajouté; il varie normalement entre deux runs. Le hash du code cible ici le seul fichier de pipeline; le commit identifie le reste du projet. Si vous répartissez le pipeline en plusieurs modules, ajoutez leurs hashes dans un champ supplémentaire et expliquez cette extension.



Exemple MLflow à intégrer dans main

Ce fragment illustre l'ordre des opérations. `metrics`, `confusion`, `thresholds` et `class_distribution` sont à construire; complétez tous les champs du manifeste. Les chemins `ROOT` rendent les écritures indépendantes du dossier courant.



PYTHON

```
import hashlib
import json
import platform
import subprocess
import joblib
import mlflow

def sha(path):
    return hashlib.sha256(path.read_bytes()).hexdigest()

def write_json(path, payload):
    path.parent.mkdir(parents=True, exist_ok=True)
    path.write_text(json.dumps(payload, indent=2, allow_nan=False))

# À faire avant l'entraînement final : refuser un code/config non committé.
commit = subprocess.check_output(
    ["git", "rev-parse", "HEAD"], cwd=ROOT, text=True).strip()
dirty = bool(subprocess.check_output(
    ["git", "status", "--porcelain"], cwd=ROOT, text=True).strip())
if dirty:
    raise RuntimeError("Committez le code et la configuration avant le run")

mlflow.set_tracking_uri(config["mlflow"]["tracking_uri"])
mlflow.set_experiment(config["mlflow"]["experiment_name"])
with mlflow.start_run() as run:
    # Exécuter prepare/train/evaluate ici ou avant, sans changer les entrées.
    (ROOT / "models").mkdir(exist_ok=True)
    joblib.dump(model, ROOT / "models/model.joblib")
    write_json(ROOT / "reports/metrics.json", metrics)
    write_json(ROOT / "reports/confusion_matrix.json", confusion)
    write_json(ROOT / "reports/threshold_analysis.json", thresholds)
    write_json(ROOT / "reports/class_distribution.json", class_distribution)
    mlflow.log_params({
        "seed": config["seed"], "test_size": config["test_size"],
        "model_type": config["model"]["type"], "C": config["model"]["C"],
        "max_iter": config["model"]["max_iter"],
        "decision_threshold": config["decision_threshold"],
    })
    mlflow.log_metrics(metrics)
    # Construire manifest selon le tableau, avec run.info.run_id.
    write_json(ROOT / "models/run_manifest.json", manifest)
    for relative in (
        "models/model.joblib", "models/run_manifest.json",
        "reports/metrics.json", "reports/confusion_matrix.json",
        "reports/threshold_analysis.json", "reports/class_distribution.json",
        "configs/train.yaml",
    ):
        mlflow.log_artifact(str(ROOT / relative), artifact_path="evidence")
```

Le modèle est journalisé comme fichier joblib dans `evidence/` ; Model Registry et déploiement ne sont pas demandés. Quitter le contexte termine le run. Ensuite `make snapshot LABEL=A` sauvegarde les fichiers et les métadonnées du serveur. Le snapshot nécessite un serveur accessible; sa lecture ultérieure et le contrôle final fonctionnent hors ligne.

Ce que les contrôles prouvent

`make validate-submission` recharge les modèles locaux, recalcule les métriques, contrôle les hashes et les snapshots, compare A/B et détecte un rapport encore marqué « À compléter ». Il ne certifie ni la qualité des explications, ni l'authenticité des captures, ni l'absence de fuite : le code et la preuve du fit restent à lire. N'utilisez `joblib.load` que sur vos propres modèles de confiance. Aucun score minimal n'est imposé : un recall nul expliqué correctement est recevable.

