GL-M326 · LECTURE ACADÉMIQUE

MLOps et Industrialisation des modèles IA

Du modèle au système ML traçable, validé et exploitable

SÉANCE 2 / 6 Rendre un entraînement reproductible

ENSEIGNANT
Pr. Youness BOUTYOUR
youness.boutyour@um5.ac.ma

PUBLIC CIBLE
Génie Logiciel
Cycle Ingénieur — 3A

INSTITUTION
Université Mohammed V de Rabat
École Nationale Supérieure d'Informatique et d'Analyse des Systèmes

GL-M326 · LECTURE ACADÉMIQUE

MLOps et Industrialisation des modèles IA

Du modèle au système ML traçable, validé et exploitable

SÉANCE 2 / 6 Rendre un entraînement reproductible

ENSEIGNANT
Pr. Youness BOUTYOUR
youness.boutyour@um5.ac.ma

PUBLIC CIBLE
Génie Logiciel
Cycle Ingénieur — 3A

INSTITUTION
Université Mohammed V de Rabat
École Nationale Supérieure d'Informatique et d'Analyse des Systèmes

Plan de la séance

SÉANCE 2 / 6 — RENDRE UN ENTRAÎNEMENT REPRODUCTIBLE

01 Cadrage

02 Construire l'identité d'un run

03 Évaluer sans contaminer la preuve

04 Versionner les données

05 Construire un pipeline reproductible

06 Synthèse

Objectifs pédagogiques

OBJECTIFS PÉDAGOGIQUES — 5 OBJECTIFS

01 Reconstruire l'identité complète d'un run à partir du code, des données et de l'environnement

02 Prévenir une data leakage en séparant correctement les jeux de données avant le preprocessing

03 Externaliser les paramètres d'entraînement dans une configuration versionnée

04 Identifier une donnée par son contenu, son contrat et son empreinte

05 Décomposer un entraînement en étapes recalculables avec DVC et en conserver les preuves avec MLflow

CHAPITRE

Rendre un entraînement reproductible

01 De la question au protocole

La question directrice

?

PROBLÈME DE LA SÉANCE

Comment reconstruire exactement un entraînement ML lorsque les données, le code et les artefacts évoluent ?

La séance transforme cette question en une chaîne de preuves : **identifier**, **sécuriser**, **versionner**, **recalculer** et **tracer**.

À L'ISSUE DE LA SÉANCE, VOUS SAUREZ

- reconstruire l'identité complète d'un **run** : code, données, configuration et environnement ;
- repérer et corriger une **data leakage** en séparant les jeux avant le preprocessing ;
- identifier une donnée par son **contrat**, sa version et son empreinte ;
- découper un entraînement en étapes explicites et recalculables avec **DVC** ;
- relier les preuves du pipeline aux runs **MLflow**.

Principes MLOps ciblés

La séance ne cherche pas seulement à « rejouer » un script : elle construit un entraînement dont les décisions et les résultats peuvent être vérifiés.

PRINCIPE	CE QUE L'ON MET EN PLACE	PREUVE OBTENUE
P1 — Tracer	Relier code, données, configuration, environnement et artefacts à chaque **run**.	Identité complète du run
P2 — Versionner	Identifier les données par leur contrat, leur historique et leur empreinte.	Snapshot retrouvable
P3 — Valider	Séparer avant le preprocessing et contrôler l'absence de **data leakage**.	Évaluation fiable
P5 — Automatiser	Déclarer un graphe DVC et ne recalculer que les étapes modifiées.	Pipeline incrémental

FIL CONDUCTEUR **Une expérience devient reproductible lorsque ses preuves sont liées et que son pipeline peut être rejoué.**

Reproduire : passer du résultat à la preuve

Enregistrer une métrique ne suffit pas. Un run est reproductible lorsque l'on peut **reconstruire ses entrées, ses transformations et ses conditions d'exécution**, puis obtenir des sorties comparables.

ÉLÉMENT DU RUN	QUESTION À RÉSOUDRE	PREUVE MINIMALE
Données	Sur quel contenu le modèle a-t-il appris ?	version, schéma, empreinte
Code et configuration	Quelle procédure et quels paramètres ont été exécutés ?	commit, fichier de config, seed
Environnement	Avec quelles dépendances et quelles ressources ?	lockfile, versions, image
Pipeline	Quelles étapes ont transformé les données ?	graphe DVC, logs, paramètres
Sorties	Quels objets et quels résultats ont été produits ?	modèle, métriques, manifeste, `run_id`

À RETENIR **La métrique décrit le résultat ; le manifeste et le run décrivent l'expérience.**

Le contrat de la séance

- **Fil rouge :** nous reprenons le cas de classification du **churn client** introduit en séance 1.

TROIS QUESTIONS À POUVOIR JUSTIFIER

01 **Données et configuration :**

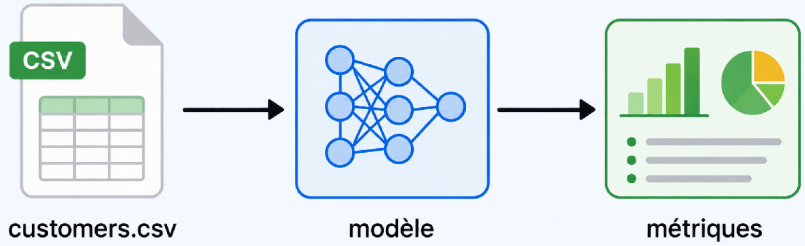
- Avec quelles versions et quels paramètres le modèle a-t-il été produit ?

02 **Préprocessing :**

- Quelle information a été utilisée pour apprendre chaque transformation, et sur quel jeu de données ?

03 **Recalcul :**

- Que faut-il rejouer après une modification, et pourquoi ?



LIVRABLE **Un pipeline d'entraînement rejouable et auditable.**

État initial - un script qui fait tout

- **Point de départ :** à la fin de la séance 1, le projet produit déjà plusieurs artefacts utiles : modèle, métriques, manifeste et run MLflow.
- **Limite actuelle :** `train.py` enchaîne dans un même flux la lecture des données, le split, le preprocessing, l'entraînement, l'évaluation et le suivi.
- Chaque décision est donc difficile à isoler, tester et rejouer séparément.

RISQUE **Une métrique et un nom de fichier ne suffisent pas à reconstruire un run.**
Il faut relier chaque sortie aux données, au code, aux paramètres et à l'environnement.

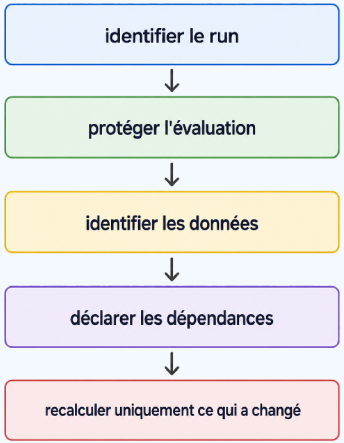
Les quatre idées du chapitre

La séance suit une progression concrète : identifier un run, vérifier la validité de son évaluation, maîtriser ses données, puis automatiser son recalcul.

IDÉE	DURÉE	QUESTION À RÉSOUDRE
MC1 — Identité d'un run	20 min	Quelles conditions ont produit ce modèle ?
MC2 — Intégrité de l'évaluation	25 min	Le score mesure-t-il vraiment la généralisation ?
MC3 — Données comme dépendance	25 min	Comment prouver que les données sont les mêmes ?
MC4 — Pipeline reproductible	30 min	Que faut-il rejouer après un changement ?

FIL CONDUCTEUR **Chaque idée ajoute une preuve au même projet de classification du churn.**

Une progression du run au recalcul



- La séance avance en cinq étapes : **identifier**, **protéger**, **versionner**, **déclarer**, puis **recalculer**.
- À chaque étape, on relie trois éléments :
  - une notion à comprendre ;
  - une évolution concrète du projet ;
  - une preuve vérifiable.

PREUVES POSSIBLES

Sortie console, fichier, hash, métrique ou run MLflow.

Chaque décision doit laisser une trace et chaque modification doit permettre de savoir quoi rejouer.

Feuille de route — supports intermédiaires

ÉTAPE	SUPPORT	SORTIE ATTENDUE
1 · Split	`steps/01-split/`	`train.csv`, `test.csv`
2 · Validation	`steps/02-validation/`	rapport `PASS/FAIL`
3 · Graphe	`steps/03-dvc/`	stages DVC
4 · Finish	`finish/`	métriques, manifeste, MLflow

- Les fragments intermédiaires sont volontairement incomplets : on les complète pendant la démonstration.

CHAPITRE

Rendre un entraînement reproductible

02

Construire l'identité d'un run

Reconstruire un run

- **Situation :** on vous transmet uniquement un extrait de résultats et un chemin vers le modèle.

```
{
  "accuracy": 0.91,
  "model_path": "model.joblib",
  "seed": 42
}
```

- Ce fichier permet de connaître une partie du résultat, mais pas les conditions exactes de l'entraînement.

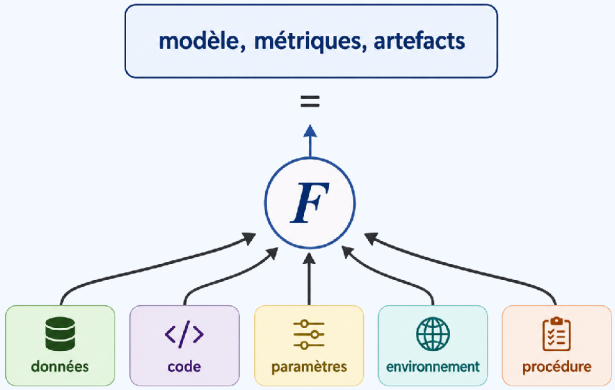
QUESTION DE DÉPART

Quelles informations faut-il ajouter pour relancer exactement l'expérience ?

- Il manque notamment :
  - l'empreinte et la version des données ;
  - le split train/test et le preprocessing ;
  - le commit, la configuration et l'environnement ;
  - les autres métriques et les artefacts produits.

DÉFI **Un résultat enregistré n'est pas encore une expérience reconstructible.**

Un run dépend de ses conditions



F (données, code, paramètres, environnement, procédure)

- Un entraînement peut se lire comme une fonction :
  - **données + code + paramètres + environnement + procédure → modèle, métriques et artefacts**.
- Si une seule entrée change sans être enregistrée, deux exécutions peuvent produire des résultats différents tout en semblant identiques.

> DÉFINITION **L'identité d'un run regroupe les informations qui permettent de répondre à la question :**
> « Qu'est-ce qui a produit cette sortie ? »

- **Conséquence :** un modèle ne doit jamais être conservé sans les preuves de ses conditions de production.

L’identité complète d’une expérience

- Une expérience ne se résume pas au modèle obtenu ni à sa métrique finale. Elle regroupe les **conditions de production** qui permettent de comprendre, comparer et rejouer le run.
- Pour chaque source, on doit pouvoir répondre à une question et fournir une preuve vérifiable :

SOURCE	QUESTION À RÉSOUDRE	PREUVE ATTENDUE
Données	Sur quel contenu le modèle a-t-il appris ?	chemin, révision DVC, SHA-256, schéma et taille
Code	Quelle logique a été exécutée ?	commit Git, branche, diff et point d’entrée
Configuration	Avec quels choix a-t-il été entraîné ?	fichier YAML, hyperparamètres, seed et seuils
Procédure	Quelles étapes et dépendances ?	`dvc.yaml`, graphe DVC, commandes et sorties
Environnement	Avec quelles dépendances ?	versions Python, scikit-learn, lockfile ou image
Résultat	Qu’est-ce qui a été produit ?	métriques, modèle, manifeste, checksum et run ID

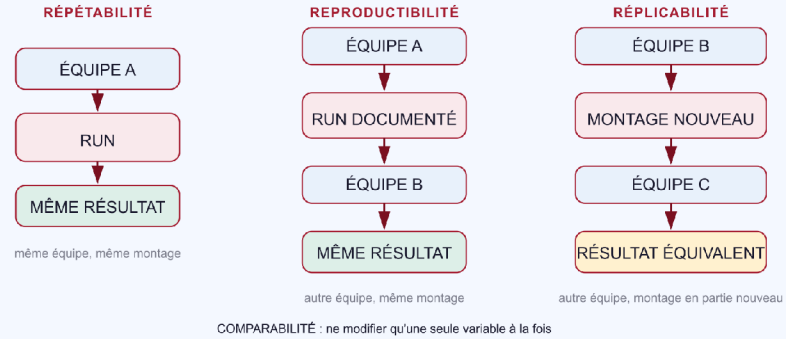
Répétabilité reproductibilité et traçabilité

- **Répétabilité** — la même équipe rejoue le même protocole dans le même environnement et obtient le même résultat.
- **Reproductibilité** — une autre équipe peut reconstruire l'expérience à partir des conditions documentées.
- **Réplicabilité** — une autre équipe peut obtenir un résultat équivalent avec un montage ou une implémentation en partie différents.

POINT DE VIGILANCE

La seed réduit une source d'aléa, mais elle ne remplace pas la provenance.

- **Transition :** ces notions deviennent concrètes dès que l'on doit compléter le manifeste d'un run.



Compléter le manifeste d'un run

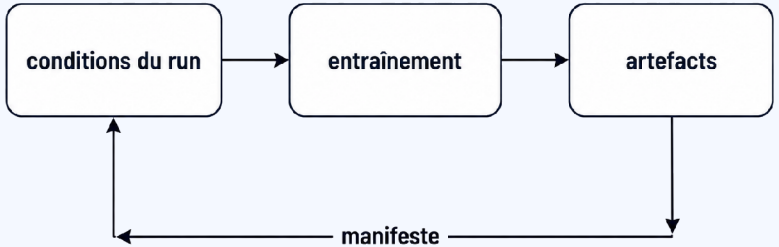
- Le manifeste transforme les notions précédentes en preuves exploitables. Il doit relier les entrées, la procédure, l'environnement et les sorties du run.
- Il doit notamment conserver :
  - les empreintes des jeux de données ;
  - la version du code et la configuration ;
  - les versions de l'environnement ;
  - les métriques et les artefacts produits.

APPLICATION **On ne reproduit pas seulement un modèle : on reproduit le processus complet qui l'a produit.**

```
{
    "raw_data_sha256": "...",
    "train_data_sha256": "...",
    "test_data_sha256": "...",
    "code_version": "...",
    "environment": {
        "python": "...",
        "sklearn": "..."
    },
    "metrics": {
        "accuracy": 0.90,
        "f1": 0.82
    }
}
```

Extrait d'un manifeste de run

Conserver les causes du résultat



- Une métrique décrit **ce qui s'est passé** ; elle ne suffit pas à expliquer pourquoi le modèle a obtenu ce résultat.
- Le manifeste relie la chaîne complète : **conditions du run → entraînement → artefacts**.
- Il permet de retrouver les données, le code, les paramètres, la procédure et l'environnement associés au modèle.

SYNTHÈSE MC1

La preuve utile n'est pas seulement le résultat : ce sont les causes qui permettent de le reconstruire.

CHAPITRE

Rendre un entraînement reproductible

03

Évaluer sans contaminer la preuve

De l’identité à l’intégrité

- Dans la section précédente, nous avons appris à conserver les conditions qui produisent un run.
- Nous devons maintenant vérifier que la métrique repose sur une information réellement indépendante et représente la **généralisation** du modèle.
- **Généraliser**, c’est produire des prédictions fiables sur des observations qui n’ont pas servi à apprendre le modèle ni à régler ses paramètres.
- La question n’est donc pas seulement « **quel score ?** », mais aussi « **obtenu sur quelles données, après quelles transformations ?** ».

QUESTION DE LA SECTION

Le protocole d'évaluation mesure-t-il la généralisation, ou une information déjà vue ?

Un bon score prouve-t-il la généralisation

LE RISQUE D'UN SCORE TROMPEUR

- Un modèle obtient un score élevé sur le jeu de test.
- Ce score peut être **optimiste** si le test a influencé l'apprentissage, même indirectement.
- Le score ne révèle pas, à lui seul, quelles informations ont été utilisées pour construire la représentation des données.
- Une **data leakage** peut être discrète : elle ne provoque pas toujours une accuracy spectaculaire, mais elle rend la comparaison entre modèles peu fiable.

LA CONDITION D'UNE MESURE FIABLE

- Le test doit rester **inconnu jusqu'à l'évaluation finale**.
- Aucune statistique du test ne doit servir à ajuster le preprocessing, les variables ou les hyperparamètres.
- Les mêmes règles doivent être appliquées à chaque nouvelle exécution.
- Le jeu de test doit rester réservé à la mesure finale ; les choix de modèle peuvent utiliser un jeu de **validation** séparé.

INTÉGRITÉ DE L'ÉVALUATION

Un score n'est crédible que si le jeu évalué est resté indépendant pendant tout l'apprentissage.

Pr. Youness BOUTYOUR
youness.boutyour@um5.ac.ma

Trois rôles pour une évaluation intègre

APPRENDRE SUR `TRAIN`

- ajuster les paramètres du preprocessing ;
- entraîner le modèle ;
- calculer les premières métriques d’apprentissage.
- Exemples : moyenne et écart-type d’un scaler, catégories d’un encodeur, quantiles d’une transformation.

CHOISIR SUR `VALIDATION`

- comparer les modèles, seuils et hyperparamètres ;
- sélectionner la configuration qui sera ensuite évaluée sur le test.

MESURER SUR `TEST`

- rester en dehors de toutes les décisions d’apprentissage ;
- recevoir uniquement les transformations déjà apprises ;
- servir à mesurer la performance finale sur des données inconnues.
- Le test ne doit pas servir à choisir un seuil, une variable, un hyperparamètre ou une version du modèle.

FRONTIÈRE **Le test représente l'information disponible après toutes les décisions de conception.**

Ordre recommandé : `fit(train)` → comparaison sur `validation` → mesure finale sur `test`.

Data leakage : définir le risque

DÉFINITION

Une *data leakage* se produit lorsqu'une information qui ne serait pas disponible au moment de la prédiction est utilisée pour construire le preprocessing, choisir le modèle ou ajuster ses paramètres.

POURQUOI EST-CE GRAVE ?

- l’évaluation devient trop optimiste ou difficile à interpréter ;
- le modèle peut perdre ses performances sur de nouvelles données ;
- la comparaison entre deux runs ne porte plus sur des protocoles équitables.

EXEMPLES

- calculer une moyenne, des quantiles ou des catégories sur `train + test` ;
- ajuster un scaler, un encodeur ou un imputer avant le split ;
- sélectionner des variables avec `y_test` ;
- utiliser le score de test pour choisir un hyperparamètre.

RÈGLE GÉNÉRALE

Toute information utilisée pour apprendre ou décider doit provenir du train ou de la validation, jamais du test.

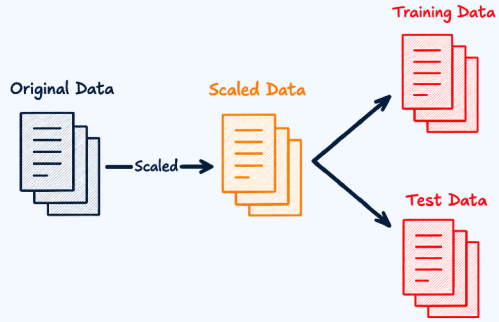
Référence : scikit-learn — Common pitfalls and recommended practices.

Diagnostiquer une fuite de données

LA QUESTION À POSER

Pour chaque étape, vérifier : **quelles données étaient accessibles au moment du `fit` ou de la décision ?**

- `X_train` uniquement : apprentissage autorisé ;
- `X_test` dans `transform` : application autorisée ;
- `X_test` dans `fit` ou dans un choix : fuite.



La séparation doit précéder toute transformation qui apprend des paramètres.

DIAGNOSTIC EN TROIS ÉTAPES

01 Repérer chaque appel à `fit` et chaque décision de sélection.

02 Identifier les lignes et les cibles visibles à cet instant.

03 Relier l'information vue aux paramètres, aux prédictions et aux métriques.

La chaîne correcte est : `split → fit(train) → transform(train, test)`.

La règle avant le code

ORDRE CORRECT

```
split
  → fit(train)
  → transform(train, test)
  → train(model)
  → predict(test)
  → metrics
```

TROIS RESPONSABILITÉS

- **Train** : apprendre les paramètres et ajuster le modèle.
- **Validation** : comparer les configurations et choisir une version.
- **Test** : mesurer la performance finale sur une information restée inconnue.

RÈGLE D'OR

Une étape qui apprend des paramètres doit utiliser le train uniquement.

Le test reçoit ensuite la transformation, sans modifier les paramètres ni les décisions du modèle.

?

QUESTION DE CONTRÔLE

Le test peut-il être supprimé de la phase de `fit` ?

Exemples : autorisé ou interdit ?

OPÉRATIONS POSSIBLES

- séparer `train` et `test` avant le preprocessing ;
- exécuter `fit_transform(X_train)` ;
- exécuter `transform(X_test)` avec les paramètres appris sur le train ;
- choisir un hyperparamètre à partir d'un jeu de validation séparé.

POURQUOI ? **Le test est utilisé pour appliquer et mesurer, jamais pour apprendre.**

OPÉRATIONS INTERDITES

- exécuter `fit_transform(X)` avant le split ;
- calculer une moyenne ou des quantiles sur `train + test` ;
- choisir les variables avec `y_test` ;
- sélectionner le meilleur modèle après consultation du score de test.

POURQUOI ? **Une information du test remonte dans l'apprentissage ou dans une décision.**

Étude de cas — formuler une hypothèse

CADRE DE COMPARAISON

- même dataset ;
- même modèle ;
- deux ordres différents entre preprocessing et split ;
- mêmes métriques pour rendre les résultats comparables.
- Le protocole A applique `fit_transform` aux 500 lignes avant le split ; le protocole B réserve 100 lignes pour le test avant tout apprentissage.

QUESTIONS DE LECTURE

01 combien de lignes chaque `fit` verra-t-il : 500 ou 400 ?

02 les quantiles appris seront-ils identiques ?

03 les distances et les métriques changeront-elles ?

AVANT L'EXÉCUTION **Formulez une hypothèse causale : quelle donnée chaque protocole autorise-t-il à voir avant l'évaluation ?**

Observation : exécuter `python src/compare_protocols.py` depuis `starter/`.

```
PROTOCOL_A  fit_rows=<observé>  accuracy=<observée>  F1=<observé>
PROTOCOL_B  fit_rows=<observé>  accuracy=<observée>  F1=<observé>
```

`fit_rows` indique combien de lignes le preprocessing a vues ; les métriques doivent être comparées sur le même test.

Relier le code aux métriques

PROTOCOLE A — FUITE

```
transformed = preprocessor.fit_transform(X)
train_pos, test_pos = train_test_split(positions, ...)
```

- La transformation voit les lignes de test avant la séparation : ses paramètres incluent donc une information qui devait rester inconnue.
- Dans ce cas, `fit_rows` vaut 500 : les quantiles sont calculés avec les données qui serviront ensuite à l'évaluation.

PROTOCOLE B — RÉFÉRENCE

```
X_train, X_test, y_train, y_test = train_test_split(X,
y, ...)
train_t = preprocessor.fit_transform(X_train)
test_t = preprocessor.transform(X_test)
```

- Le test est isolé avant que `fit` n'apprenne : il est transformé avec les paramètres calculés sur `X_train` uniquement.
- Ici, `fit_rows` vaut 400 : les 100 lignes de test restent absentes de l'apprentissage de la représentation.

PREUVE ATTENDUE

Relier chaque différence de métrique à une différence de données vue par `fit`.

La chaîne causale est : données vues → paramètres appris → représentation → prédictions → métriques.

Pr. Youness BOUTYOUR
youness.boutyour@um5.ac.ma

Corriger puis vérifier

```
# La séparation précède toute transformation apprise.
train_df, test_df = train_test_split(
    df, test_size=0.20, random_state=42, stratify=df["cl
)
train_df.to_csv("data/processed/train.csv", index=False
test_df.to_csv("data/processed/test.csv", index=False)
```

PIPELINE CORRIGÉ

`split → fit(train) → transform(train, test) → train(model) → predict(test) → metrics`

- La frontière devient visible et vérifiable dans les fichiers produits. Le test ne participe ni au preprocessing, ni à l'entraînement, ni au choix du modèle.
- La correction ne consiste donc pas à rechercher un meilleur score, mais à restaurer la validité de la mesure.

Synthèse MC2

- **Affirmation :** une évaluation indépendante exige que le test reste inconnu pendant toutes les décisions d'apprentissage.
- **Preuve :** comparer les lignes vues par `fit`, les paramètres appris et les métriques des deux protocoles.
- **Règle :** toute opération qui apprend des paramètres est ajustée sur `X_train`, puis appliquée à `X_train` et `X_test`.
- **Conséquence :** les métriques ne sont comparables que si les protocoles respectent la même frontière d'information.

TRANSITION VERS LES DONNÉES

Une évaluation peut être intègre, mais elle reste incomplète si l'on ne sait pas exactement quel contenu de données a été utilisé.

Pr. Youness BOUTYOUR
youness.boutyour@um5.ac.ma

CHAPITRE

Rendre un entraînement reproductible

04

Versionner les données

Une donnée ne se résume pas à son nom

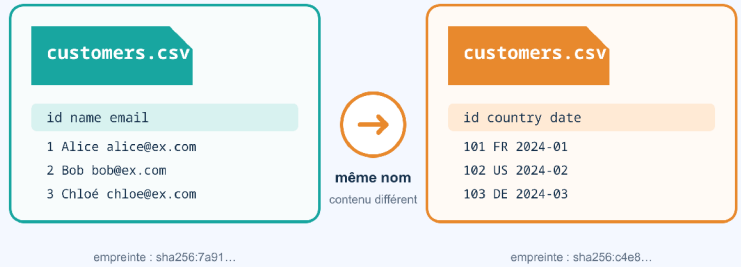
MÊME RÉFÉRENCE APPARENTE

- même chemin : `data/raw/customers.csv` ;
- même code, configuration et seed ;
- métriques pourtant différentes.

Le nom du fichier est identique dans les deux exécutions.

CE QU'IL FAUT VÉRIFIER

Ce que l'on voit	Ce qui peut avoir changé
même nom	contenu du fichier
même chemin	version des données
même pipeline	état réel du dataset



Le nom identifie une référence ; l'empreinte identifie le contenu.

Étape suivante : calculer l'empreinte du dataset avant de l'associer au run.

Une empreinte identifie le contenu

1. IDENTIFIER LE CONTENU

- Une empreinte est un résumé déterministe des octets d'un fichier.
- Le même contenu produit la même valeur ; une modification produit normalement une valeur différente.
- Elle permet de vérifier que le dataset utilisé correspond bien au dataset déclaré.

2. NE PAS LA CONFONDRE AVEC UNE VERSION

- Elle ne conserve pas l'historique des versions.
- Elle ne remplace donc ni **DVC**, ni Git, ni un registre de données.
- Elle complète le versionnement en fournissant une preuve du contenu exact.

3. LA CHAÎNE DE RÉFÉRENCE

CHEMIN	EMPREINTE	VERSION
retrouver le fichier	vérifier le contenu	retrouver l'historique

Chemin indique où lire ; **empreinte** permet de vérifier ; **version** permet de retrouver un état antérieur.

Calculer et tracer une empreinte

CALCULER L'EMPREINTE

```
sha256sum data/raw/customers.csv
```

- La commande lit les octets du fichier et produit une valeur comparable entre deux exécutions. Le résultat doit être associé au run qui a utilisé le fichier.
- SHA-256 est courant, mais il n'est pas obligatoire. L'algorithme doit être stable, documenté et identique entre les exécutions.

CHOISIR L'ALGORITHME

BESOIN	CHOIX POSSIBLE
identification robuste	SHA-256, SHA-512, BLAKE2, BLAKE3
détection rapide d'une erreur accidentelle	CRC32, xxHash
intégrité face à une modification volontaire	algorithme cryptographique

À ENREGISTRER DANS LE MANIFESTE

- `path` : emplacement du fichier utilisé ;
- `algorithm` : méthode de calcul, par exemple `SHA-256` ;
- `digest` : empreinte exacte du contenu ;
- `dataset_version` : révision DVC ou identifiant du registre.

Trois niveaux à distinguer

NIVEAU	QUESTION À POSER	EXEMPLE	RÔLE
Chemin	où lire ?	`data/raw/customers.csv`	localiser
Contenu	quels octets ?	`SHA-256(...)`	vérifier
Version	quel état retrouver ?	révision DVC ou commit Git du fichier `.dvc`	reproduire

- Le chemin est une adresse, pas une preuve d'identité : deux fichiers de même nom peuvent avoir des contenus différents.
- Pour rendre un run retrouvable, enregistrer les trois : chemin pour accéder, empreinte pour identifier, version pour restaurer.

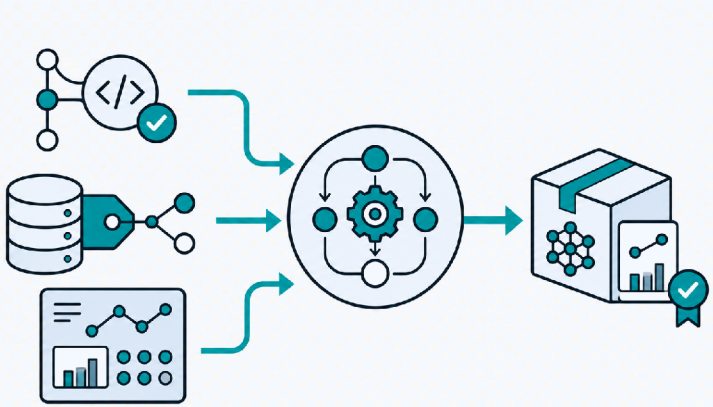
Trois rôles pour Git, DVC et MLflow

Les trois niveaux précédents deviennent exploitables lorsqu'on les rattache à des outils complémentaires : **gérer**, **déclarer**, puis **tracer**.

OUTIL	QUESTION PRINCIPALE
Git	Quelle recette et quelles métadonnées versionner ?
DVC	Quelles données et quelles étapes recalculer ?
MLflow	Quel run, quels paramètres et quelles métriques comparer ?

CHAÎNE DE PREUVE

- Git fixe la recette
- DVC fixe les données et les dépendances
- MLflow trace l'exécution et ses résultats.



Git versionne la recette ; DVC les données ; MLflow trace le run.

Démonstration MC3 — de l'hypothèse à la preuve

- **Point de départ :** comment prouver qu'un dataset utilisé par un run est bien celui que l'on croit ?
- Nous allons construire cette preuve en trois actions exécutables :

01 vérifier le contrat du fichier ;

02 enregistrer son contenu avec DVC ;

03 modifier le fichier et observer la détection.

DÉCISION **À chaque étape, une commande produit une preuve.**
La preuve finale relie contrat, contenu, version et dépendances du modèle.

Pr. Youness BOUTYOUR
youness.boutyour@um5.ac.ma

Étape 1 — Vérifier le contrat

1. FORMALISER L'HYPOTHÈSE

```
REQUIRED = {"age", "contract", "churn"}

def validate(df):
    errors = []
    missing = REQUIRED.difference(df.columns)
    if missing:
        errors.append(f"colonnes manquantes: {sorted(mis
    if not df["age"].between(18, 100).all():
        errors.append("age hors domaine [18, 100]")
    if not df["contract"].isin({"monthly", "yearly"}).a
        errors.append("contract hors vocabulaire")
    if not df["churn"].isin({0, 1}).all():
        errors.append("churn doit valoir 0 ou 1")
    return errors
```

2. EXÉCUTER LA VALIDATION

Introduire plusieurs erreurs dans le même fichier, puis exécuter :

```
make validate
```

Le validateur doit **accumuler** les violations. Corriger ensuite le fichier et relancer la même commande : seul un `PASS` autorise l'étape suivante.

PREUVE ATTENDUE

Sortie de l'étape 1 : `FAIL — 3 violations`, puis `PASS` après correction : le dataset est recevable pour la suite.

Étape 2 — Enregistrer le contenu

1. DÉCLARER LA DÉPENDANCE

```
make init-dvc
git add data/raw/customers.csv.dvc .gitignore
git commit -m "Versionner le dataset"
```

- `make init-dvc` initialise DVC puis exécute `dvc add` si nécessaire.
- `dvc add` calcule l'identité du contenu et crée le pointeur `.dvc`.
- Git versionne ce pointeur ; DVC conserve et restitue le contenu.
- Le commit relie cet état du dataset à l'état du code.

Une fois le commit créé, on dispose d'un état de référence à perturber.

CHAÎNE DE PREUVE

Sortie de l'étape 2 : `customers.csv.dvc` est versionné et contient l'empreinte de référence.

Étape 3 — Provoquer puis détecter un changement

1. MESURER L'ÉTAT DE RÉFÉRENCE

```
sha256sum data/raw/customers.csv
dvc status
```

- Modifier ensuite **une seule valeur** dans `customers.csv`, puis rejouer les commandes de contrôle.

2. LIRE L'ÉCART DÉTECTÉ

```
data/raw/customers.csv.dvc:
    changed outs:
        modified: data/raw/customers.csv
```

- Le chemin reste identique, mais l'empreinte change. DVC détecte que la dépendance brute n'est plus celle déclarée à l'étape 2.

CONCLUSION

Sortie de l'étape 3 :
La modification invalide l'état de référence. Exécuter ensuite `make repro` pour recalculer les étapes dépendantes.

Pr. Youness BOUTYOUR
youness.boutyour@um5.ac.ma

Synthèse MC3 — les données comme dépendance

TROIS NIVEAUX DE PREUVE

NIVEAU	QUESTION	PREUVE PRODUITE
Contrat	Le fichier est-il conforme ?	rapport `PASS/FAIL`
Contenu	Est-ce bien le même dataset ?	empreinte SHA-256
Version	Peut-on retrouver son état ?	fichier `.dvc` + commit

DU DATASET AU MODÈLE

- Un changement en amont doit rendre visibles les étapes et les artefacts à recalculer en aval.

RÈGLE MC3

Règle MC3 : les données sont une entrée du calcul.

Le chemin localise ; le contrat vérifie ; l'empreinte identifie ; DVC versionne et permet de recalculer les sorties dépendantes.

Ouverture vers la section suivante : quelles étapes faut-il recalculer lorsqu'une dépendance change ?

CHAPITRE

Rendre un entraînement reproductible

05

Construire un pipeline reproductible

MC4 — que faut-il recalculer ?

- MC3 nous a appris à identifier une donnée et à détecter son changement.
- Supposons maintenant que seul `model.max_depth` passe de `8` à `4` : les données brutes et les splits sont encore valides, mais le modèle et ses métriques ne le sont plus.
- Il faut donc transformer une modification en une décision : **conserver, restaurer ou recalculer chaque sortie**.

QUESTION DIRECTRICE

Comment déduire automatiquement le sous-ensemble du calcul devenu obsolète ?

La réponse dépend du graphe, pas de l'intuition de l'opérateur.

COMMANDE D'OBSERVATION

```
dvc dag
```

Lire les stages et les flèches avant de prédire ce qui sera recalculé.

Définition — un pipeline reproductible

- Un **pipeline ML** est un graphe exécutable de stages : chaque stage déclare la commande à lancer, les éléments lus et les artefacts produits.
- Une sortie d’un stage peut devenir la dépendance du suivant ; cette relation donne au graphe sa signification causale.
- Le pipeline devient reproductible lorsque son graphe, ses entrées, ses paramètres et son environnement peuvent être retrouvés puis rejoués.

DÉFINITION

Un pipeline reproductible transforme un enchaînement implicite en un contrat exécutable et traçable.

Il explique ce qui a été calculé, à partir de quoi, dans quel ordre et avec quelles sorties.

Du script monolithique au pipeline explicite

AVANT — UNE SEULE FRONTIÈRE

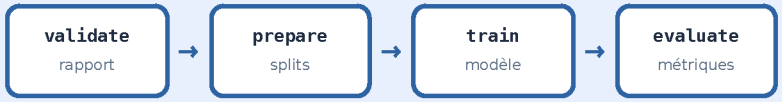
train.py

lire → préparer → entraîner → évaluer

Une erreur oblige à relire tout le script

→

APRÈS — QUATRE CONTRATS



Chaque frontière est observable et recalculable

CHANGEMENT DE MODÈLE MENTAL

Découper ne consiste pas à multiplier les fichiers : il faut rendre les dépendances et les artefacts intermédiaires explicites.

Définition — un stage est une unité de calcul

```
stage = contrat d'entrée + commande déterminée
        → sorties identifiables + preuves d'exécution
```

- Un stage possède une **responsabilité unique** et une frontière observable.
- Son résultat ne dépend que de ses dépendances, de ses paramètres, de son code et de l’environnement d’exécution déclaré.
- Il constitue la plus petite unité que DVC peut invalider, exécuter ou restaurer depuis le cache.

IDÉE CLÉ **Une bonne frontière de stage correspond à une décision que l'on souhaite tester, observer ou recalculer séparément.**

Pr. Youness BOUTYOUR
youness.boutyour@um5.ac.ma

Choisir la bonne granularité

TROP GROSSIER

```
train_all
```

- invalidation de tout le calcul ;
- diagnostic difficile ;
- peu de réutilisation du cache.

TROP FIN

```
read → clean → encode → scale → split → fit
```

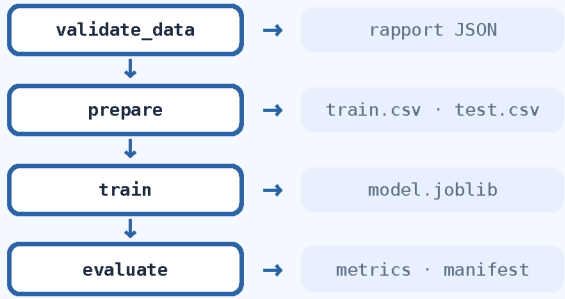
- graphe bruyant ;
- nombreux artefacts techniques ;
- coût d’orchestration inutile.

HEURISTIQUE **Créer un stage lorsqu'une sortie a une valeur pédagogique ou opérationnelle propre.**
Ici : rapport validé, splits figés, modèle entraîné et évaluation traçable.

Pr. Youness BOUTYOUR
youness.boutyour@um5.ac.ma

Les quatre stages et leurs contrats

STAGE	RESPONSABILITÉ	CRITÈRE DE SUCCÈS
`validate_data`	accepter ou refuser l'entrée	rapport sans violation
`prepare`	construire des jeux figés	splits identifiés
`train`	apprendre les paramètres	modèle sérialisé
`evaluate`	mesurer sans réapprendre	métriques + manifeste



CONTRAT D'UN STAGE `cmd + deps + params → outs + metrics`

Une sortie matérialise la frontière entre deux responsabilités.

Définition — DAG et ordre d’exécution

GRAPHE ORIENTÉ ACYCLIQUE

Un **DAG** (*Directed Acyclic Graph*) contient :

- des **nœuds**, qui représentent les stages ;
- des **arêtes**, qui signifient « la sortie du parent est l’entrée de l’enfant » ;
- aucune boucle : un stage ne peut pas dépendre de lui-même, directement ou indirectement.

```
validate_data → prepare → train → evaluate
```

TROIS NOTIONS POUR LIRE LE GRAPHE

- **parent** : stage qui fournit une dépendance directe ;
- **descendant** : stage atteignable en suivant les flèches vers l’aval ;
- **ordre topologique** : ordre qui garantit que tous les parents sont terminés avant leur enfant.

CONSÉQUENCE **Le DAG encode à la fois l'ordre d'exécution et le chemin de propagation d'un changement.**

Définition — dépendance et invalidation

UNE DÉPENDANCE EST UNE ENTRÉE DU CALCUL

Une dépendance peut être :

NATURE	EXEMPLE
donnée	`customers.csv`
code	`src/prepare.py`
paramètre	`test_size`
artefact amont	`data_validation.json`

QUAND UNE SORTIE DEVIENT-ELLE OBSOLÈTE ?

Une sortie est **invalide** lorsque l'état courant d'au moins une cause déclarée ne correspond plus à l'état mémorisé lors du dernier calcul réussi.

```
cause modifiée
      ↓
stage obsolète → nouvelle sortie → descendants obsolètes
```

RÈGLE **DVC ne peut propager que les dépendances déclarées : une entrée cachée rend le graphe faux.**

Lire un contrat DVC — du YAML à la décision

Question de lecture : comment DVC sait-il si le stage `prepare` doit être relancé ?

```yaml
stages:
  prepare:
    cmd: python src/prepare.py
    deps:
      - src/prepare.py
      - src/common.py
      - data/raw/customers.csv
      - reports/data_validation.json
    params:
      - configs/params.yaml:
          - seed
          - test_size
          - data.raw_path
          - data.train_path
          - data.test_path
    outs: [data/processed/train.csv,
data/processed/test.csv]
```

- **Exécuter — `cmd` :** fixe le point d'entrée ; modifier la commande invalide le stage.
- **Dépendre — `deps` :** liste le code, les données et le rapport réellement lus ; DVC compare leurs empreintes.
- **Paramétrer — `params` :** sélectionne les clés qui influencent `prepare`, pas toute la configuration par défaut.
- **Transmettre — `outs` :** matérialise les fichiers produits et consommables par les stages suivants.

PRÉDIRE AVANT D'EXÉCUTER

`seed`, `test_size` ou le dataset changent `prepare`.

`max_depth` n'apparaît pas ici : `prepare` reste valide ; `train`, puis `evaluate`, sont recalculés.

Comment DVC décide-t-il de recalculer ?

COMPARER DEUX ÉTATS

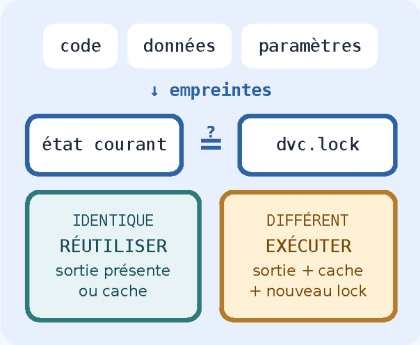
01 `dvc.yaml` déclare les causes possibles du résultat ;

02 DVC calcule l'état courant du code, des données et des paramètres ;

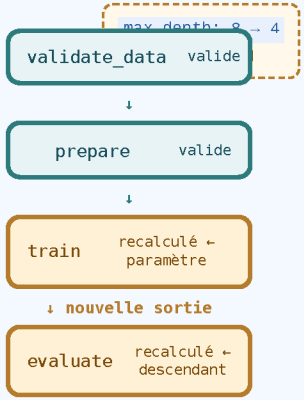
03 `dvc.lock` décrit l'état associé au dernier calcul réussi ;

04 la comparaison détermine si le stage est encore valide.

DÉCISION **État identique : conserver ou restaurer la sortie. État différent : exécuter, indexer la nouvelle sortie, puis actualiser `dvc.lock` .**



Définition — le recalcul sélectif



RÉUTILISER CE QUI RESTE VALIDE

- Un stage dont toutes les causes sont inchangées reste **valide** ; sa sortie présente est conservée ou restaurée depuis le cache si nécessaire.
- Le premier stage qui lit l'élément modifié est **invalidé et réexécuté**.
- L'invalidation se propage transitivement aux descendants qui consomment ses nouvelles sorties.

RECALCUL SÉLECTIF

Le travail minimal est la fermeture transitive des stages invalidés vers l'aval.

Ce que DVC garantit — et ce qu'il ne peut pas deviner

SI LE CONTRAT EST COMPLET

DVC peut :

- ordonner les stages ;
- détecter les changements déclarés ;
- restaurer des sorties connues ;
- recalculer les descendants invalidés ;
- mémoriser l'état réussi dans `dvc.lock`.

SI LE CONTRAT EST INCOMPLET

DVC ne détecte pas :

- une variable d'environnement non déclarée ;
- un service distant lu implicitement ;
- une source d'aléa non maîtrisée ;
- une erreur métier dans les données ;
- une métrique scientifiquement trompeuse.

LIMITE DU MÉCANISME

DVC garantit la cohérence du graphe déclaré, pas la validité scientifique du calcul.

Les contrats de données, les tests et la quality gate complètent cette garantie.

Pr. Youness BOUTYOUR
youness.boutyour@um5.ac.ma

Démonstration 1 — établir la référence

CRÉER L'ÉTAT DE RÉFÉRENCE

```
make init-dvc
make repro-offline
dvc dag
dvc metrics show
```

`repro-offline` exécute le DAG avec un suivi MLflow SQLite local, sans serveur.

Preuves attendues : quatre stages exécutés, `dvc.lock` créé, artefacts et métriques présents.

REJOUER SANS MODIFICATION

```
make repro-offline
```

```
Data and pipelines are up to date.
```

La seconde exécution constitue le **test témoin** : aucun stage ne doit être relancé lorsque causes et sorties correspondent encore à l'état verrouillé.

PREUVE DE RÉFÉRENCE **`dvc.lock` décrit l'état calculé et le cache contient les sorties réutilisables.**

Démonstration 2 — modifier et prédire

CAS A — HYPERPARAMÈTRE

```
model:
  max_depth: 4
```

```
make repro-offline
```

Avant d'exécuter, suivre la clé `model.max_depth` dans `dvc.yaml`.

Prédiction : `train` puis `evaluate`.

CAS B — DATASET BRUT

```
# après une modification de customers.csv
dvc status
make repro-offline
```

Le dataset est une dépendance de `validate_data` et de `prepare`.

Prédiction : les quatre stages sont recalculés.

À OBSERVER **Formuler la prédiction à partir du graphe, puis la confronter au journal de `dvc repro`.**

Le pipeline devient un modèle explicatif, pas seulement une commande d'automatisation.

Démo 3 — bloquer une donnée invalide

INTRODUIRE UNE DONNÉE INVALIDE

Modifier une valeur de `age` ou de `contract`, puis lancer le pipeline :

```
make repro-offline
echo $?
```

LIRE LE GARDE-FOU

```
ERROR: failed to reproduce 'validate_data'
python src/validate_data.py, exited with 1
```

Le pipeline s'arrête avant `prepare` : aucun split ou modèle ne doit être produit à partir d'une entrée non conforme.

PRINCIPE **Un échec contrôlé est une sortie correcte du pipeline : il empêche la fabrication d'artefacts non fiables.**

Corriger la donnée, puis relancer `make repro-offline` .

Synthèse conceptuelle — le vocabulaire du pipeline

CONCEPT	QUESTION ASSOCIÉE
stage	quelle unité de calcul exécuter ?
dépendance	quelle entrée peut invalider le stage ?
paramètre	quel choix contrôle le calcul ?
sortie	quel artefact vérifiable est produit ?

CONCEPT	RÔLE
DAG	ordonner les stages
`dvc.lock`	mémoriser l'état calculé
cache	réutiliser une sortie connue
invalidation	désigner ce qui devient obsolète

DU CONCEPT À LA DÉCISION

Identifier la dépendance modifiée, localiser son stage, puis suivre les descendants dans le DAG.

Synthèse MC4 — décider quoi recalculer

MODIFICATION	STAGES RECALCULÉS
`README.md`	aucun
`model.max_depth`	`train`, `evaluate`
dataset brut	les quatre stages
`evaluate.py`	`evaluate`

LA MÉTHODE

1. identifier l'entrée modifiée
2. trouver chaque stage qui la déclare
3. ajouter tous ses descendants
4. confronter la prédiction à dvc repro

Les stages sans dépendance modifiée restent réutilisables.

MESSAGE À RETENIR

Le graphe transforme une modification en décision de recalcul explicable.

Cette décision n'est fiable que si les dépendances sont complètes et les frontières de stages pertinentes.

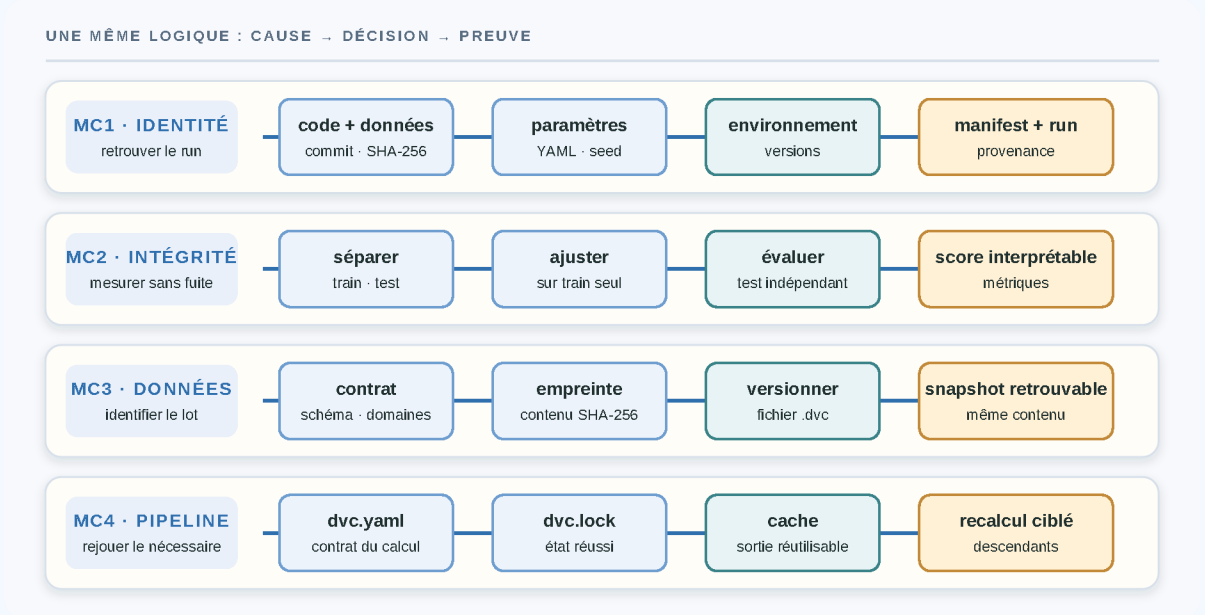
Pr. Youness BOUTYOUR
youness.boutyour@um5.ac.ma

CHAPITRE

Rendre un entraînement reproductible

07 Consolider les acquis

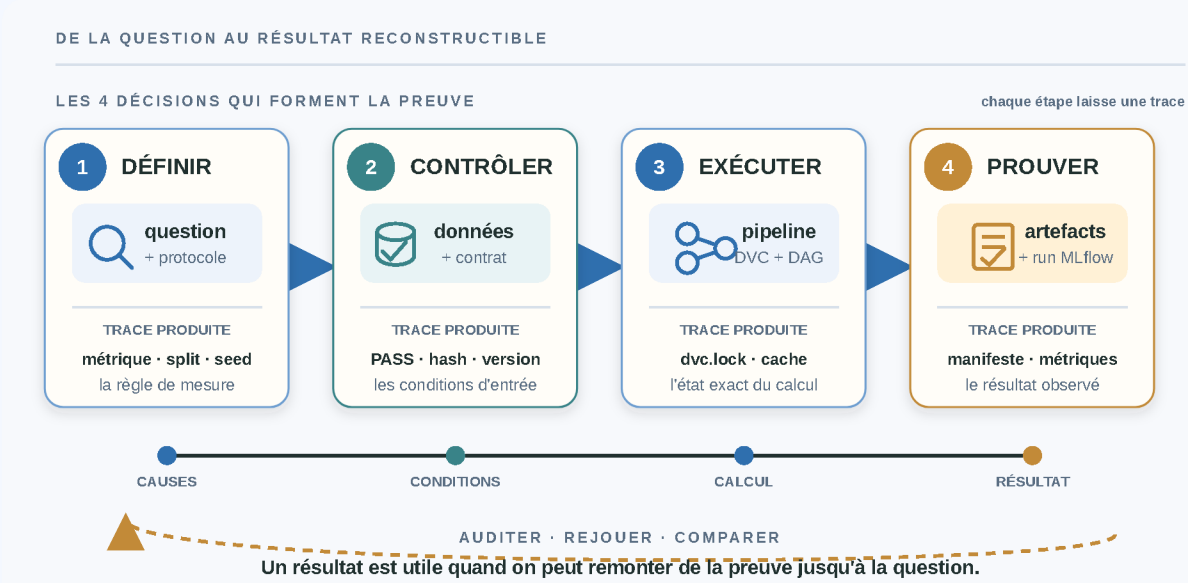
Synthèse — de l’expérience à la preuve



Chaque MC ajoute une preuve vérifiable au même objet scientifique : l'expérience.

THÈSE DU CHAPITRE **Une expérience ML devient vérifiable lorsque ses causes, ses transformations et ses sorties sont identifiables et reliées.**

La chaîne de preuve du chapitre



Le pipeline relie les conditions de mesure aux artefacts et au résultat observé.

LECTURE ACADÉMIQUE **La reproductibilité n'est pas une commande isolée : c'est une chaîne de preuves qui relie hypothèse, données, calcul et résultat.**

Ce que le chapitre établit — et ce qu'il reste à décider

ÉTABLI DANS LE PROJET

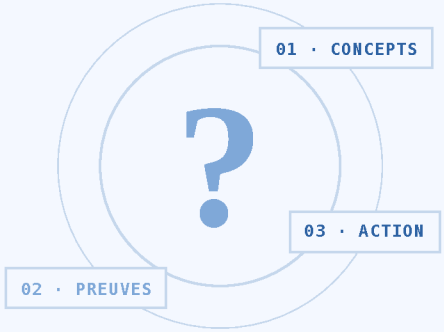
- les données ont un contrat et une empreinte ;
- le split précède l'apprentissage des transformations ;
- les stages, dépendances et sorties sont déclarés ;
- DVC recalcule le sous-graphe nécessaire.

ENCORE À VÉRIFIER

- les données restent-elles conformes dans le temps ?
- les tests du modèle sont-ils suffisants ?
- la métrique respecte-t-elle un seuil acceptable ?
- peut-on accepter, rejeter ou comparer ce résultat ?

OUVERTURE — SÉANCE 3 **Le chapitre 2 rend le calcul traçable et rejouable ; la section suivante ajoutera les tests et la quality gate pour décider si le résultat est acceptable.**

UM5 ENSIAS
LECTURE ACADÉMIQUE



MLOPS ET INDUSTRIALISATION DES MODÈLES IA

Questions et discussion

Qu'est-ce qui mérite d'être vérifié, discuté ou mis en pratique ?

POUR PROLONGER

Pr. Youness BOUTYOUR youness.boutyour@um5.ac.ma