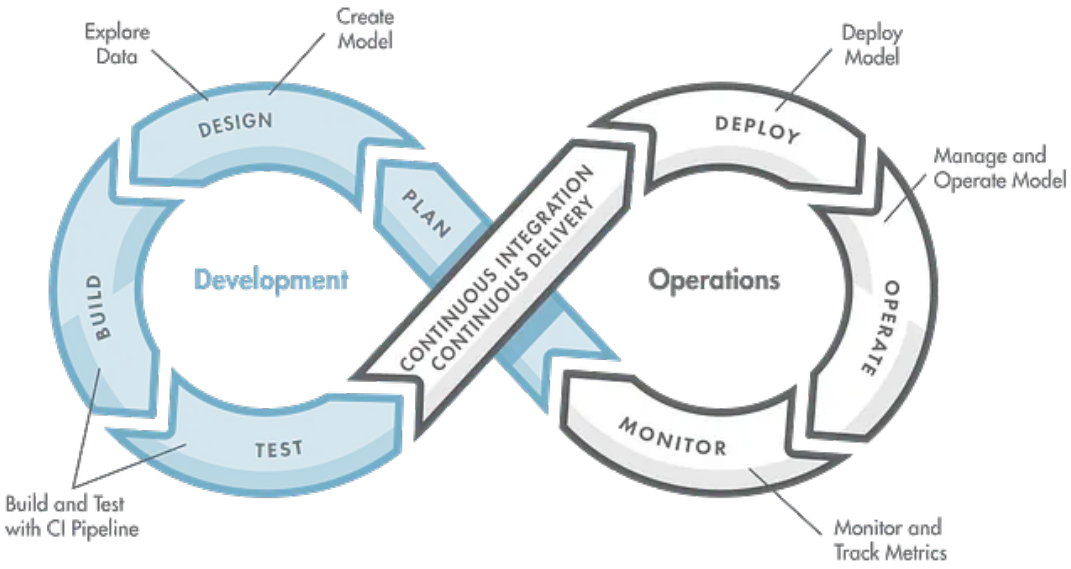
ENSET / Université Hassan II

Classification de texte avec pipeline CI/CD complet

Rapport technique



Réalisé par

Akram Benhammou
Oussama Khouya

Encadré par

Professeur Soufiane HAMIDA

13 décembre 2025

Table des matières

1 Introduction

Ce projet met en place une chaîne MLOps complète pour un problème de classification de texte sur le jeu de données *20 Newsgroups*. L'objectif est de couvrir tout le cycle de vie du modèle : prétraitement des données, entraînement et suivi expérimental, packaging dans une image Docker, mise à disposition via une API FastAPI, et automatisation par des pipelines CI/CD GitHub Actions incluant la génération de rapports CML et un flux de déploiement staging → production.

2 Architecture globale

2.1 Organisation des répertoires

La structure générale du dépôt est la suivante :

- `src/` : code source Python principal
  - `preprocess.py` : téléchargement et préparation des données.
  - `train.py` : entraînement, évaluation, logging MLflow et sauvegarde des artefacts.
  - `app.py` : API FastAPI exposant les endpoints `/health` et `/predict`.
  - `predict.py` : script de prédiction en ligne de commande.
- `data/processed/` : données prétraitées (`train.csv`, `validation.csv`, `test.csv`).
- `models/` : artefacts de modèle (fichiers `model.joblib` et `tfidf_vectorizer.joblib`).
- `reports/` : `metrics.json`, `classification_report.txt` et la matrice de confusion (`confusion_matrix.p`
- `tests/` : tests unitaires et d'intégration.
- `.github/workflows/` : pipelines CI/CD GitHub Actions (`docker.yaml`, `cml.yaml`, `deploy.yaml`).
- `Dockerfile` : définition de l'image de service pour l'API.

2.2 Pipeline de données

Le script `src/preprocess.py` implémente un pipeline de préparation standard :

1. Téléchargement du dataset *20 Newsgroups* via `sklearn.datasets.fetch_20newsgroups`, avec suppression des en-têtes, pieds de page et citations pour limiter les biais.
2. Construction d'un `DataFrame` pandas contenant les colonnes `text` (texte brut) et `target` (label).
3. Nettoyage simple via `clean_text` : mise en minuscules, suppression de la ponctuation, des chiffres et des espaces superflus.
4. Traitement linguistique via `process_text` : tokenisation NLTK, suppression des stopwords anglais, lemmatisation `WordNetLemmatizer`.
5. Filtrage des exemples dont le texte prétraité est vide.
6. Découpage en ensembles d'apprentissage, validation et test avec `train_test_split` en respectant une stratification sur `target` (70% train, 15% validation, 15% test).
7. Sauvegarde des trois ensembles au format CSV dans `data/processed/`.

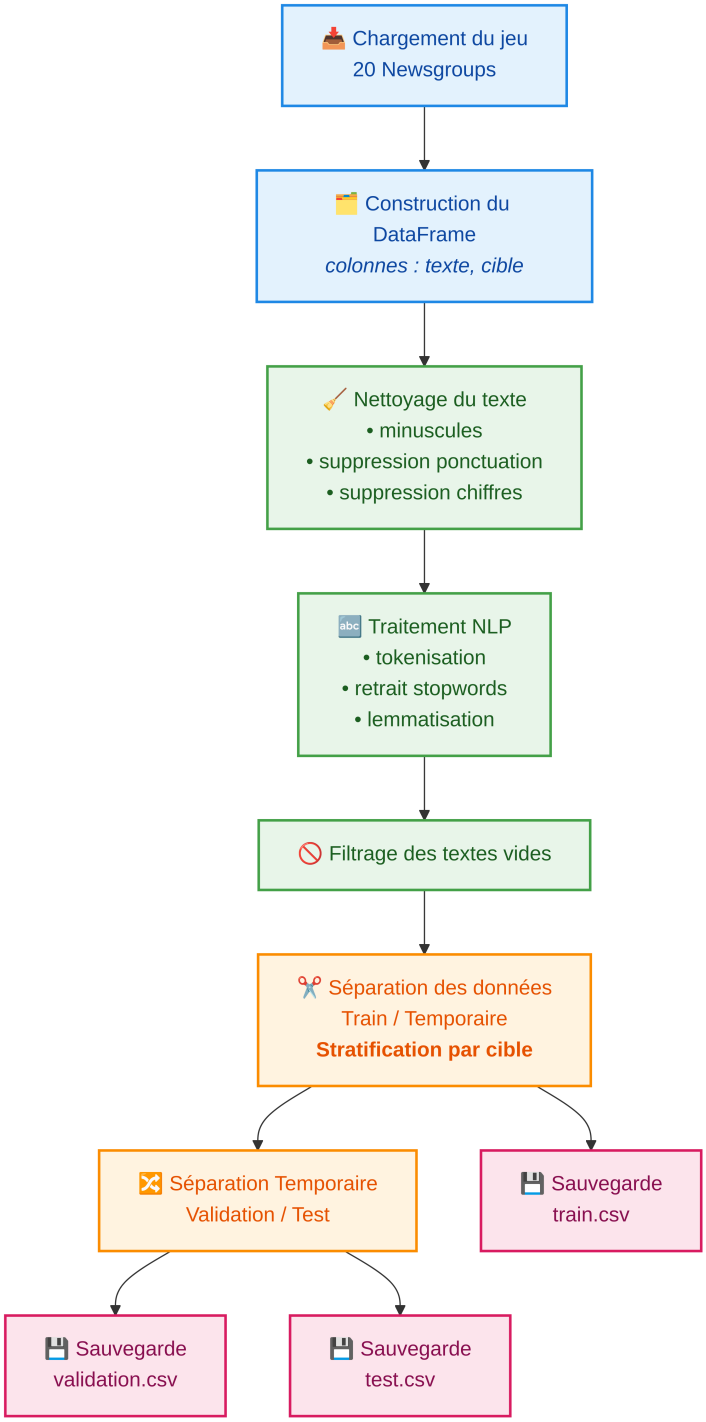


FIGURE 1 – Diagramme du pipeline de données

2.3 Entraînement et évaluation du modèle

Le script `src/train.py` orchestre la partie *training* :

— Chargement des fichiers `train.csv` et `test.csv` préalablement générés.

— Nettoyage des valeurs manquantes sur la colonne `processed_text`.

— Vectorisation TF–IDF des textes (`TfidfVectorizer` de `scikit-learn`) avec un nombre maximal de caractéristiques fixé à 5000.

— Entraînement d'une forêt aléatoire (`RandomForestClassifier`) avec 100 arbres, `random_state=42` et `n_jobs=-1`.

— Évaluation sur le jeu de test avec les métriques : accuracy, précision, rappel et F1 pondérés.
— Génération d'un rapport détaillé par classe via `classification_report` et d'une matrice de confusion.
— Sauvegarde locale des métriques (`metrics.json`), du rapport texte (`classification_report.txt`) et de la matrice de confusion (`confusion_matrix.png`).
— Sauvegarde des artefacts de modèle : `model.joblib` et `tfidf_vectorizer.joblib` dans `models/`.

L'ensemble de l'expérience est instrumenté avec `MLflow` :
— `mlflow.set_tracking_uri("file:./mlruns")` pour un backend local.
— `mlflow.set_experiment("Text_Classification_Projet9")` pour regrouper les runs.
— Logging des hyperparamètres (`max_features`, `n_estimators`), des métriques d'évaluation, de la matrice de confusion et des artefacts (`model`, `tfidf_vectorizer.joblib`).

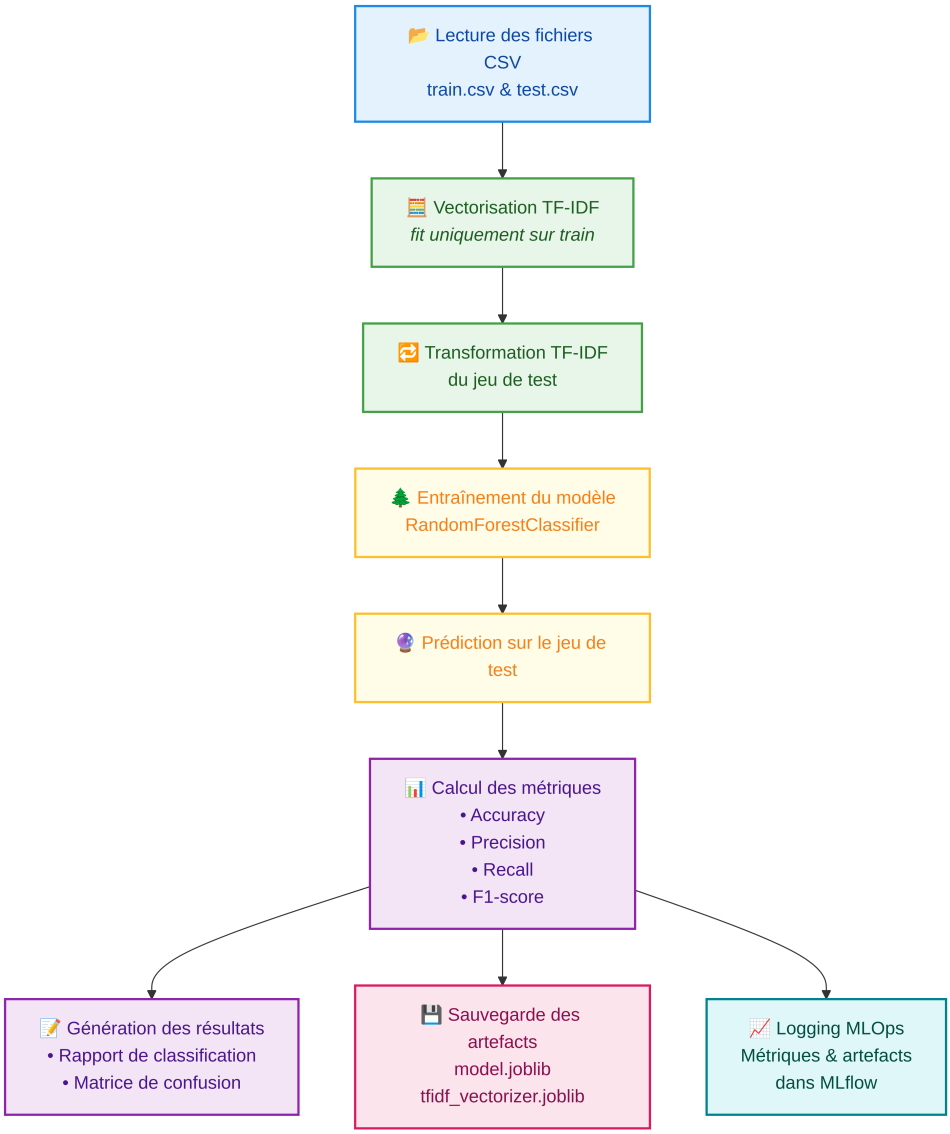


FIGURE 2 – Diagramme du pipeline d'entraînement et d'évaluation du modèle

2.4 Service d'inférence

Le service de prédiction est implémenté dans `src/app.py` avec FastAPI :

— Utilisation d'un gestionnaire de cycle de vie (`lifespan`) pour charger les artefacts au démarrage de l'application : modèle, vectoriseur TF–IDF, stopwords et lemmatiseur NLTK.
— Endpoint `GET /health` qui retourne un statut `ok` et indique si le modèle est correctement chargé.
— Endpoint `POST /predict` acceptant un payload JSON contenant un champ `text` ; le texte est prétraité avec les mêmes fonctions que celles utilisées pour l'entraînement, vectorisé, puis passé au modèle.
— La réponse contient le texte d'origine, l'identifiant de classe prédit (`prediction_class_id`) et un champ `status`.

Un script `src/predict.py` fournit également une interface en ligne de commande pour charger les artefacts et réaliser une prédiction sur un exemple de texte.

3 Pipeline CI/CD

3.1 Build, tests et image Docker

Le workflow `.github/workflows/docker.yaml` implémente une pipeline de CI centrée sur la construction de l'image de service :

— Déclenchement sur `push` et `pull_request` vers la branche `master`.
— Installation de Python 3.9 et des dépendances à partir de `requirements.txt`.
— Exécution séquentielle de `src/preprocess.py` puis `src/train.py` afin de régénérer les données prétraitées et les artefacts de modèle.
— Lancement de la suite de tests `pytest tests/ -v`.
— Connexion au registre GitHub Container Registry (`ghcr.io`) et construction de l'image Docker à partir du `Dockerfile`, avec publication optionnelle des tags (`latest`, `sha`) sur les branches non-`pull_request`.

Le `Dockerfile` correspondant utilise `python:3.9-slim` comme base, installe les dépendances, télécharge les ressources NLTK nécessaires, copie les sources, les artefacts et les données prétraitées, expose le port 8000 et lance l'application via Uvicorn.

3.2 Rapport automatique avec CML

Le workflow `.github/workflows/cml.yaml` s'appuie sur `iterative/setup-cml` pour générer un rapport expérimental automatique :

— Installation des dépendances, exécution du pipeline de prétraitement et d'entraînement.
— Création d'un fichier `report.md` contenant :
  — un titre,
  — les métriques au format JSON (`metrics.json`),
  — le rapport de classification détaillé (`classification_report.txt`),
  — la matrice de confusion publiée comme image via `cml publish`.
— Publication automatique du rapport en commentaire sur la pull request via `cml comment create`.

3.3 Déploiement staging → production

Le workflow `.github/workflows/deploy.yaml` met en place une logique de déploiement en deux phases :

— Il se déclenche à la complétion du workflow « Docker Build & Push » sur la branche `master`.
— Job **deploy-staging** :
  — Récupère la dernière image Docker depuis le registre.
  — Lance un conteneur en environnement staging sur le port 8000.

  - Effectue des tests d'intégration simples via `curl` sur les endpoints `/health` et `/predict`.
  - Vérifie que le modèle est bien chargé (`model_loaded=true`) et fixe une accuracy simulée (0,64) pour illustrer la validation.
- Job **deploy-production** (conditionné par le succès du staging) :
  - Vérifie que l'accuracy renvoyée dépasse un seuil minimum configurable (`MIN_ACCURACY`, par défaut 0,5), en utilisant `bc` pour la comparaison flottante.
  - « Déploie » l'image en production (placeholder avec traces dans les logs et possibilités futures de déploiement Kubernetes ou SSH).
  - Marque la version comme release de production (étapes de tagging commentées mais prévues).
- Job **rollback** :
  - Se déclenche en cas d'échec du staging ou de la production.
  - Journalise les actions de rollback prévues (restauration d'une version précédente, notifications, sauvegarde des logs).
  - Crée automatiquement un ticket GitHub décrivant le rollback.

4 Choix techniques

4.1 Stack machine learning

- **Dataset** : *20 Newsgroups*, classique pour les tâches de classification de texte multi-classe.
- **Représentation** : TF–IDF (`TfidfVectorizer`) limité à 5000 features pour contrôler la dimensionnalité.
- **Modèle** : `RandomForestClassifier`, modèle robuste et interprétable, moins sensible au scaling des features que d'autres méthodes.
- **Prétraitement** : pipeline NLTK (tokenisation, stopwords, lemmatisation) appliqué de façon cohérente entre entraînement et prédiction.
- **Suivi expérimental** : MLflow pour tracer hyperparamètres, métriques et artefacts en local (`file:./mlruns`).

4.2 Stack applicative et DevOps

- **API** : FastAPI, choix moderne pour exposer des services REST performants, avec génération automatique de documentation OpenAPI.
- **Serving** : Uvicorn en mode ASGI dans le conteneur Docker.
- **Containerisation** : image `python:3.9-slim`, installation minimale des dépendances, téléchargement en amont des ressources NLTK pour éviter les téléchargements au runtime.
- **CI/CD** : GitHub Actions pour la construction, les tests, la génération d'image, le déploiement et la génération de rapport CML.
- **Qualité** : tests unitaires et d'intégration avec `pytest`, couvrant la préparation de données, le service d'API et la présence des artefacts.
- **Reporting** : CML pour transformer les résultats expérimentaux (métriques + plots) en rapports Markdown intégrés dans les pull requests.

5 Limites et axes d'amélioration

5.1 Limites du modèle et des données

- **Modèle classique uniquement** : l'utilisation d'un Random Forest sur des features TF–IDF est simple mais ne profite pas des avancées récentes en NLP (transformers, embeddings contextuels, etc.).

— **Pas de tuning systématique** : les hyperparamètres (`max_features`, `n_estimators`) sont fixés en dur, sans recherche automatisée (Grid Search, Random Search ou optimisation bayésienne).
— **Prétraitement basique** : pas de prise en compte d'éléments comme les bigrammes/trigrammes, la détection de langue, la gestion des caractères spéciaux ou des émojis.
— **Gestion partielle des labels** : la correspondance entre identifiant de classe et nom de catégorie n'est pas exposée dans l'API, ce qui complique l'interprétation métier.

5.2 Limites MLOps / CI/CD

— **Tracking local uniquement** : MLflow est configuré en mode fichier local, ce qui limite l'utilisation en environnement distribué ou multi-utilisateurs.
— **Absence de véritable registry de modèles** : il n'existe pas encore de gestion de versions de modèles avec promotion formelle (staging/production) au niveau du *model registry*.
— **Couplage fort pipeline–build** : le workflow de build exécute systématiquement prétraitement et entraînement avant les tests, ce qui rallonge les temps de CI et peut être coûteux.
— **Seuil de performance codé en dur** : l'accuracy utilisée dans le déploiement est actuellement simulée (valeur fixe 0,64), ce qui réduit le réalisme de la validation automatisée.
— **Monitoring runtime absent** : pas de collecte de logs métier, de métriques en production, ni de détection de dérive de données ou de performance.

5.3 Sécurité, robustesse et industrialisation

— **Gestion des ressources NLTK** : le téléchargement des ressources NLTK au démarrage (dans `app.py`) peut être fragile en contexte de production (dépendance réseau) même si le Dockerfile anticipe déjà certains téléchargements.
— **Tests limités** : les tests couvrent les points clés (API, prétraitement, artefacts) mais ne couvrent pas tous les cas d'erreur (entrées invalides, timeouts, modèles manquants, etc.).
— **Configuration statique** : les chemins, hyperparamètres et URI MLflow sont codés dans le code ; une externalisation via des variables d'environnement ou des fichiers de configuration renforcerait la flexibilité.
— **Déploiement simplifié** : le job de déploiement est illustratif (logs, placeholders pour Kubernetes ou SSH) sans intégration à une infrastructure réelle.

6 Conclusion

Le projet illustre un pipeline MLOps cohérent de bout en bout pour la classification de texte : préparation des données, entraînement, suivi expérimental, exposition d'une API, containerisation et automatisation CI/CD avec rapport CML et logique de déploiement staging → production. Les choix techniques privilégient la simplicité et la pédagogie (Random Forest + TF–IDF, MLflow local, GitHub Actions, CML) et offrent une base solide pour explorer les bonnes pratiques MLOps.

Les principales pistes d'amélioration concernent la modernisation du modèle (transformers), la mise en place d'un véritable registry et d'un tracking centralisé, l'enrichissement des tests et du monitoring en production, ainsi que la dé-corrélation des pipelines de data science et des pipelines de build/déploiement.