département
Informatique

Université Claude Bernard Lyon 1

Année universitaire	**2025-2026**		
Filière	**Data Science**	**Année**	**M2**
Matière	**Machine Learning**		
Enseignant	**Haytham Elghazel & Ichrak Ennaceur**		
Intitulé TD/TP :	**Atelier 2 : Mise en production et déploiement continu**		
Contenu	• **Conteneurisation des processus** • **Mise en production d'algorithmes de Machine learning** • **Reporting** • **Entraînement et déploiement continu** • **Agent IA**		

Dans cet atelier pratique, vous allez travailler sur la mise en production d'un modèle de prédiction et sur la création d'un véritable pipeline MLOps moderne. Vous développerez une **API de serving** pour votre modèle, une **interface web utilisateur**, un **système de monitoring des performances** et du **drift**, ainsi qu'un **mécanisme de réentraînement et de déploiement continu**. L'ensemble de l'architecture sera conteneurisé avec **Docker** et orchestré avec Docker Compose (ou Kubernetes pour aller plus loin).

Ce projet peut être appliqué à différents types de données selon votre cas d'étude :

- données tabulaires (ex. *transactions bancaires pour détection de fraude* ou *churn client*),
- données multimédia (*images*, *audio* ou *vidéo*),
- ou tout autre type de données pertinentes.

Pour enrichir ce workflow, vous intégrerez également une **brique Agent IA** basée sur un orchestrateur open-source (tel que *n8n*) associé à un **grand modèle de langage** (LLM). Cet agent permettra d'automatiser des actions suite aux prédictions du modèle et de mettre en place une boucle de **validation humaine** (*Human-in-the-Loop*).

Le LLM complète ainsi le modèle ML en interprétant ses résultats, en expliquant la décision au format compréhensible et en automatisant l'interaction avec l'utilisateur (notification, email, recommandation d'action). Il joue donc un rôle d'assistant intelligent autour du modèle prédictif.

Par exemple, dans un scénario de *fraude bancaire*, si une transaction est jugée suspecte, l'agent IA contactera automatiquement l'utilisateur par email (par exemple), collectera sa confirmation, puis renverra l'information à votre système pour alimenter le modèle en nouvelles données annotées.

De même, dans un contexte de *churn client*, si un utilisateur présente un risque élevé de départ, l'agent IA pourra lui envoyer automatiquement un message personnalisé (par exemple une offre ou un questionnaire de satisfaction), récupérer sa réponse, puis l'intégrer comme donnée d'entraînement dans le pipeline afin d'améliorer le modèle.

Enfin, pour un cas appliqué à des données multimédia, par exemple en contrôle qualité industrielle sur image, si le modèle détecte un défaut potentiel sur un produit, l'agent IA pourra notifier un opérateur humain, recueillir sa validation (défaut réel ou faux positif), puis enregistrer ce feedback pour améliorer continuellement le modèle.

Votre pipeline devra enfin surveiller le **maintien des performances en production**, détecter les changements dans les données (*data drift*), et déclencher un réentraînement du modèle lorsque :

- un seuil de nouvelles données annotées est atteint, et/ou
- un drift significatif est détecté entre les données de référence et les données réelles.

Ce TP vous permettra ainsi de mettre *en œuvre un cycle complet d'IA moderne : entraînement, déploiement, supervision, interaction* automatisée avec l'utilisateur, *collecte de feedback*, et *mise à jour continue du* modèle.

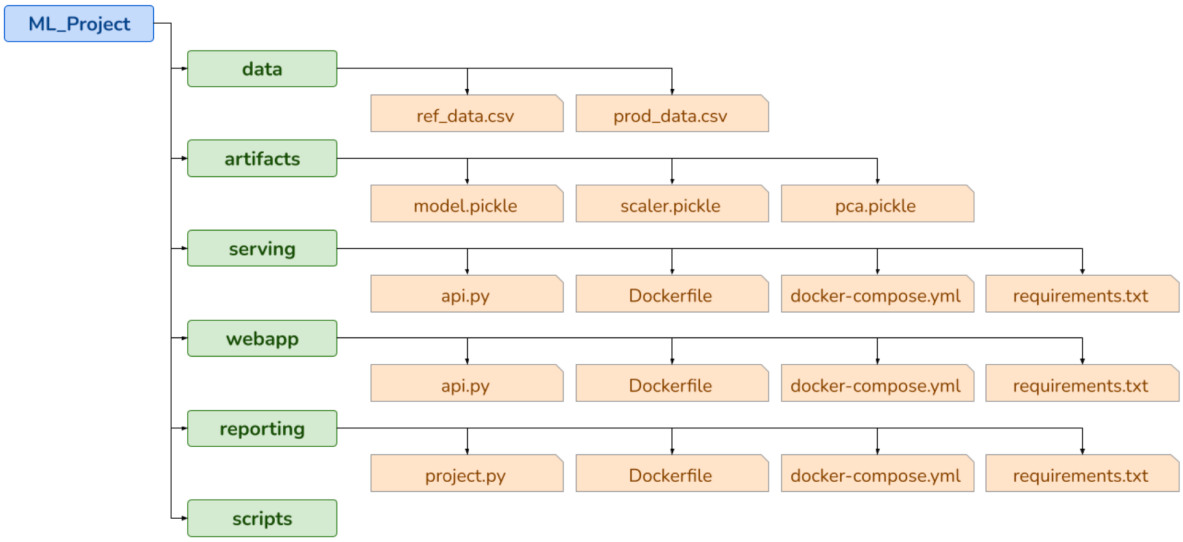
I. Préparation du projet

Tous les composants implémentés lors de ce TP seront conteneurisés à l'aide de Docker. Docker permet de créer une boîte dans laquelle on peut faire fonctionner une application, appelée conteneur. Ce conteneur regroupe l'ensemble des dépendances nécessaires, garantissant que votre application s'exécute de manière identique sur n'importe quelle machine. Cela facilite le partage, la collaboration et le déploiement sur différentes plateformes.

L'orchestration de ces conteneurs se fera via Docker Compose (ou Kubernetes pour ceux souhaitant une version plus avancée). Docker Compose permet de gérer plusieurs services (API, interface web, monitoring, agent IA, etc.) communiquant ensemble dans un environnement reproductible.

Vous allez installer Docker et Docker Compose sur votre machine, nous vous conseillons d'installer Docker Desktop qui contient l'engine Docker, ainsi que Compose : https://docs.docker.com/desktop/.

Au cours de votre progression dans le TP vous allez ajouter des fichiers à votre projet, veuillez-vous référer à l'architecture de fichiers suivante afin d'organiser votre projet :



Le dossier *scripts* comprend tous les scripts/notebooks avec lesquels vous avez développé votre modèle, ainsi qu'un fichier python contenant votre code factorisé.

Le dossier data contient un fichier appelé *ref_data.csv* qui servira de référence pour le reporting futur et que vous pouvez utiliser pour entraîner votre modèle. Créez un script qui transforme les données de votre dataset en vecteurs via le modèle d'embedding choisi et les sauvegarde en un fichier nommé ref_data.csv. Les données doivent être organisées de la manière suivante au sein de votre fichier csv (le modèle d'embedding de l'exemple est une PCA, le votre sera peut-être diffèrent) :

PCA_1	PCA_2	...	PCA_n	target
0.59	-1.04	...	1.32	True
-0.25	0.68	...	-0.89	False
...	...	...	...	...

Utilisez ce fichier *ref_data.csv* afin d'entraîner votre modèle, enregistrez sous forme de fichier pickle votre modèle entrainé, le modèle d'embedding, et le scaler (si applicable) au sein du dossier artifacts.

Ce fichier servira également à détecter le **data drift** au fil du temps lorsque de nouvelles données réelles arriveront via l'application.

II. API de serving du modèle avec FastAPI

Maintenant que vous avez setup le projet et entrainé un modèle sur votre dataset, vous allez implémenter une API de serving pour votre modèle. Une API de serving permet à une application externe d'appeler votre modèle pour effectuer une prédiction sur une donnée. L'intérêt d'implémenter une telle API est de définir une interface standardisée avec laquelle une application souhaitant utiliser le modèle pourra communiquer et recevoir des prédictions. Ainsi, l'API de serving de votre modèle sera indépendante et permettra à une application cliente (ici Streamlit, ou l'agent IA n8n) d'envoyer une donnée et d'obtenir une prédiction.

Dans un scénario de détection de fraude, par exemple, cette API est appelée lorsqu'une nouvelle transaction est saisie via l'interface utilisateur ou reçue depuis un flux externe. C'est elle qui constitue la *brique de décision du système.*

Les fichiers générés à cette étape doivent être placés au sein du dossier serving. Vous allez utiliser FastAPI pour le développement de votre API de serving. Créez le fichier api.py, renseignez-vous sur l'utilisation du package de FastAPI sous python et implémentez un Endpoint nommé predict contactable via la méthode POST. L'endpoint de prédiction recevra une donnée sous format JSON, votre code devra utiliser votre modèle de prédiction (*la pipeline i.e.* le modèle d'embedding, scaler et modèle de prédiction) pour traiter la donnée et obtenir une prédiction. Cette prédiction est retournée par votre endpoint. Le modèle de prédiction utilisé par l'API est stocké dans le dossier artifacts. Chargez-les dans des variables globales au lancement de l'API afin qu'ils soient disponibles instantanément pour les prédictions.

Gênerez un fichier *requirements.txt* et ajoutez-y les dépendances python nécessaires au fonctionnement de votre API, un nom de package par ligne. Initialement votre fichier contiendra les deux lignes suivantes, vous le mettrez à jour lorsque vous ajouterez des fonctionnalités et dépendances à l'API :

```
fastapi uvicorn
```

Gênerez un fichier *Dockerfile* et ajoutez-y le code suivant :

FROM python:3.10-slim	*Indique au conteneur de baser son environnement sur l'image Docker de Python 3.10, trouvée à l'adresse suivante : https://hub.docker.com/_/python*
WORKDIR /app	*Défini le directory de travail*
COPY api.py . **COPY requirements.txt .**	*Copie le code de votre API et le fichier de requirements*
RUN pip3 install --upgrade pip **RUN pip3 install -r requirements.txt**	*Met à jour pip et installe les packages indiquées dans le fichier de requirements*
CMD uvicorn api:app --host 0.0.0.0 --port 8080	*Héberge l'API sur le port 8080 de l'IP externe du conteneur*

Utilisez la commande **docker build ...**, puis **docker run -p 8080:8080 ...** pour exécuter le conteneur de l'API de serving, l'option **-p 8080:8080** permet de lier le port 8080 du conteneur au port 8080 de votre machine. Ainsi vous pouvez vérifier la bonne exécution de votre API en accédant à la page : http://localhost:8080/docs.

Maintenant que votre API est accessible, vous allez implémenter une simple interface web qui communiquera avec l'API.

III. Interface web avec Streamlit

Vous allez maintenant implémenter une interface utilisateur simple avec **Streamlit**, qui jouera le rôle de portail d'observation pour la prédiction du modèle en production.

L'interface Streamlit permettra de simuler l'utilisation du modèle en production. L'utilisateur pourra saisir ou uploader une donnée (par exemple une transaction bancaire dans le cas de la fraude, un profil client pour le churn ou une image produit pour un contrôle qualité visuel), puis cliquer sur un bouton "Prédire". L'interface enverra alors cette donnée à l'API FastAPI pour obtenir la prédiction correspondante (score de fraude, probabilité de churn ou détection d'un défaut produit), qui sera affichée à l'écran. Vous pouvez utiliser le package python **requests** afin d'envoyer une requête à l'API de serving. Après avoir suivi la suite des explications de cette section vous pourrez requêter l'endpoint de prédiction de votre API de serving via l'URL http://serving-api:8080/predict dans le code de votre interface web.

Afin d'améliorer l'expérience utilisateur, Streamlit pourra également afficher quelques statistiques locales et mini-visualisations sur les prédictions réalisées (par exemple histogramme des scores de fraude, répartition des classes, taux d'alerte, aperçu de l'image analysée, etc.). De plus, l'interface pourra proposer une première interprétation des résultats, comme l'explicabilité des prédictions (par exemple les variables ayant le plus influencé la décision ou les régions importantes d'une image).

Contrairement à une simple application de visualisation, la boucle de feedback ne sera pas gérée directement dans Streamlit. Un second bouton, "Notifier l'utilisateur", déclenchera une interaction avec un agent IA (n8n + LLM). En pratique, au clic, Streamlit appellera un endpoint de cet agent qui se chargera : (1) de récupérer la prédiction, (2) de formuler automatiquement un message adapté via un modèle de langage, et (3) d'envoyer un e-mail à l'utilisateur final. Ainsi, dans un scénario de fraude bancaire, un message sera envoyé au client afin de confirmer la transaction ; pour un cas de churn, le système pourra envoyer une offre ou un message de rétention ; et pour un système de contrôle visuel, une alerte pourra être envoyée à un opérateur humain. L'e-mail contiendra un lien de validation ou de correction généré par n8n, permettant de collecter la réponse de l'utilisateur, de la stocker, et ainsi de l'utiliser ultérieurement pour améliorer le modèle via une boucle Human-in-the-Loop automatisée (fichier prod_data.csv).

Créez un fichier *api.py* au sein du dossier *webapp*, il contiendra votre code pour l'interface. Comme pour l'API de serving vous devez créer et compléter un fichier *requirements.txt* et *Dockerfile*, la commande exécutée au sein du Dockerfile devra cette fois utiliser **streamlit run ...** afin d'exécuter votre application Streamlit.

Une particularité de travailler avec Docker est que chaque conteneur possède une IP dynamique qui lui est attribuée à son lancement, rendant complexe la communication entre deux conteneurs. Afin de pallier ce problème vous utiliserez Docker Compose pour définir un réseau permettant de facilement connecter des conteneurs.

Créez un fichier *docker-compose.yml* dans le dossier *serving* et ajoutez-y le code suivant :

```
version: '3.8'
```
Indique la version de Docker Compose

```
networks:
  prod_net:
    driver: bridge
```
Défini le réseau "prod_net" qui permettra à la webapp de communiquer avec l'API de serving

```
services: serving-api:
  container_name:
    serving-api
```
Crée le service serving-api, on utilisera le nom de ce service pour communiquer avec l'API de serving

```
build:
  context: .
    dockerfile: Dockerfile
```
Indique l'emplacement du Dockerfile

```
volumes:
  - ../data:/data
  - ../artifacts:/artifacts
```
Partage le dossier data avec l'API de serving, toute modification de /data depuis l'API sera effectif sur le dossier data de votre projet, le conteneur pourra également accéder aux artifacts pour les utiliser

```
ports:
  - "8080:8080"
```
*Équivalent de l'option **-p 8080:8080** de la commande **Docker run** que vous utilisiez précédemment*

```
networks:
  - prod_net
```
Indique le ou les network(s) auquel le service serving-api peut accéder

Créez un fichier docker-compose.yml dans le dossier webapp et ajoutez-y le code suivant :

```
version: '3.8'
```
Indique la version de Docker Compose

```
networks:
  serving_prod_net:
    external: true
```
Récupère le réseau précédemment défini pour l'API, le préfixe "serving_" s'ajoute automatiquement lors de la création du réseau, c'est le nom du dossier de l'API

```
services:
  webapp:
    container_name: webapp
```
Crée le service webapp

```
build:
  context: .
    dockerfile: Dockerfile
```
Indique l'emplacement du Dockerfile

```
ports:
  - "8081:8081"
```
Lie le port 8081 du conteneur au port 8081 de votre machine

```
networks:
  - serving_prod_net
```
Connecte l'application web au réseau de l'API de serving

Vous pouvez désormais créer le réseau "serving_prod_net" et démarrer votre API de serving avec la commande suivante :

$ docker compose -f serving/docker-compose.yml up

Démarrez votre interface web en utilisant la commande **docker compose ...**, une fois lancée vous pourrez y accéder sur votre navigateur à l'adresse http://localhost:8081/. Ajoutez les options **--build --force-recreate** aux commandes **docker compose ...** en cas de modification de votre code, cela permet de forcer la recréation de l'image Docker, et ainsi, prendre en compte les modifications apportées depuis la dernière exécution.

IV. Reporting avec Evidently

Vous allez maintenant utiliser la librairie **Evidently** afin de générer un rapport sur l'état de santé du modèle ainsi que sur l'évolution des données en production. Evidently évaluera à la fois la performance du modèle (métriques de classification) et la dérive des données (data drift). Pour cela, Evidently s'appuiera sur deux fichiers :

- **ref_data.csv** : données de référence issues de votre dataset initial, transformées en vecteurs via votre modèle d'embedding ;
- **prod_data.csv** : données collectées en production avec leurs prédictions et les validations/corrections fournies par les utilisateurs.

Dans ce TP, le fichier **prod_data.csv** ne sera pas rempli directement depuis l'interface Streamlit. À la place, lorsque l'utilisateur clique sur "Notifier l'utilisateur", l'agent IA (orchestré avec n8n et un LLM) envoie un email à l'utilisateur final pour confirmer ou corriger la prédiction. Lorsqu'il clique sur le lien présent dans l'email, le feedback est renvoyé à votre backend (*via un endpoint dédié*) puis enregistré dans prod_data.csv.

Ainsi, chaque ligne du fichier prod_data.csv devra contenir :

- la donnée transformée en vecteur via le modèle d'embedding,

- la prédiction produite par votre modèle,
- la réponse fournie par l'utilisateur (cible réelle ou validation/correction).

Ce mécanisme permet de constituer progressivement un historique réel de production qui servira à :

- *monitorer* la performance du modèle au fil du temps,
- *détecter* toute dérive des données ou de la distribution des prédictions,
- *déclencher* un ré-entraînement automatique lorsque le volume de feedback recueilli dépasse un seuil ou lorsqu'un drift significatif est détecté.

Comme pour le fichier ref_data.csv, veillez à appliquer exactement la même transformation de données (embedding, normalisation, etc.) afin de garantir la comparabilité des features. Les données doivent suivre la structure suivante dans prod_data.csv :

PCA_1	PCA_2	...	PCA_n	target	prediction
0.59	-1.04	...	1.32	True	True
-0.25	0.68	...	-0.89	False	True
...	...	...	...	...	

Créez le fichier *project.py* dans le dossier reporting, suivez le tutoriel de setup pour créer un projet Evidently qui sera utilisé pour le Dashboard : https://docs.evidentlyai.com/tutorials-and-examples, code complet : https://github.com/evidentlyai/evidently/blob/main/examples/sample_notebooks/get_started_monitoring.py.

Créez le *Dockerfile*, *requirements.txt* et *docker-compose.yml*, ajoutez le code suivant au *Dockerfile* en complétant les lignes manquantes :

...

CMD python project.py && evidently ui --host 0.0.0.0 --port 8082

Complétez le fichier *docker-compose.yml* de reporting de manière à partager le volume data avec le Docker et lier le port 8082 du conteneur au port 8082 de votre machine. Lancez le conteneur avec la commande **docker compose** ..., vous pouvez accéder à l'interface web du dashboard à l'adresse : http://localhost:8082/. Vous devriez voir le projet de démo sur un dataset d'exemple.

Modifiez maintenant le projet Evidently pour l'adapter à vos données. Remplacez les données de références et de production d'exemple par les données de vos fichiers ref_data.csv et prod_data.csv respectivement. Explorez la documentation et modifiez le code du fichier project.py afin de générer un rapport avec des métriques de classification (F1_score, Balanced_accuracy, Rappel, Precision) en plus de celles de data drift.

Vous avez mis en place un système de reporting statique, en pratique il serait plus intéressant de programmer la génération automatique de rapports à intervalles de temps réguliers et de mettre en place un système de monitoring en temps réel. Il serait trop long d'intégrer cette partie à ce TP, pour un bonus vous pouvez vous intéresser à l'utilisation de CronJobs avec Kubernetes pour programmer une exécution régulière des rapport et à la mise en place de la stack prometheus+grafana qui permettra de récupérer les métriques extraites par Evidently et de les afficher en temps réel sur un dashboard web.

Vous avez mis en place un système de reporting statique. En pratique, il serait plus intéressant de programmer la génération automatique de rapports à intervalles réguliers et de mettre en place un monitoring en temps réel. Pour aller plus loin (bonus), vous pouvez étudier :

- l'utilisation de **CronJobs Kubernetes** pour planifier l'exécution régulière des rapports,
- la mise en place de la stack **Prometheus + Grafana** pour collecter les métriques extraites par Evidently et les visualiser en temps réel sur un tableau de bord.

Pour finaliser ce projet, vous allez automatiser l'entraînement de votre modèle sur la concaténation des données de référence et les nouvelles données provenant de votre environnement de production, puis le déploiement de ce nouveau modèle à la place de l'ancien.

Le réentrainement et la mise en production d'un nouveau modèle peut être déclenché automatiquement selon de nombreux triggers : *lorsque les performances du modèle diminuent*, *lorsqu'un drift est détecté entre les données de référence et celles de production*, *lorsqu'assez de nouvelles données de production ont été ajoutées*, etc. Vous allez mettre en place un ré-entraînement et déploiement automatique de votre modèle basé sur la quantité de nouvelles données de production.

Vous allez modifier l'Endpoint "feedback" de votre API de serving pour déclencher la mise à jour du modèle. Définissez un seuil **k** qui sera un trigger, toutes les **k** données de production reçues, la mise à jour du modèle sera lancée. Si le nombre de ligne dans le fichier *prod_data.csv* après avoir ajouté le feedback est un multiple de **k**, appelez une fonction de votre code factorisé qui entraîne un nouveau modèle de prédictions sur la concaténation des données de référence et de production, et met à jour les fichiers pickle du dossier *artifacts*. Une fois le modèle entrainé, remplacez le modèle de prédiction utilisé par l'API par les nouveaux en modifiant les variables globales. Ainsi, les nouveaux appels de prédiction à l'API utiliseront le nouveau modèle de production.

Bonus :

- Par défaut, les données de retour utilisateur seront enregistrées dans le fichier **prod_data.csv** afin d'être exploitées pour le réentraînement du modèle. Cependant, dans un contexte industriel, ce stockage pourrait également être réalisé dans une base de données (PostgreSQL, MySQL, MongoDB, etc.), permettant un suivi structuré, sécurisé et scalable des interactions. De la même manière, les données de référence (**ref_data.csv**) sont ici stockées dans un fichier pour simplifier la manipulation et la visualisation. Dans un système réel, ces données de référence seraient généralement conservées dans une table dédiée en base de données, ou dans un data lake, afin de :
    - *maintenir* un historique versionné des données d'entraînement,
    - *permettre* une gouvernance et une traçabilité complètes,
    - *faciliter* la comparaison entre différentes versions de jeux de données,
    - *offrir* un accès centralisé à Evidently et au pipeline de réentraînement.
- Comme précisé plus haut, Mettre en place un système de monitoring en temps réel avec prometheus+grafana.
- Comparer le modèle nouvellement entrainé avec le modèle actuellement en production afin de vérifier la supériorité du nouveau modèle, ne remplacer le modèle en production que si c'est le cas.
- Mettre en place un processus de fine-tuning du dernier modèle de production plutôt que de ré-entrainer un modèle de 0.
- Utiliser Kubernetes plutôt que Docker Compose pour la gestion des conteneurs, volumes et réseaux.