Déploiement d’une solution IA avec FastAPI et Docker

1. Contexte du projet

Ce document présente le déploiement d’un modèle de classification du diabète dans le cadre du module UA2 – Optimisation des modèles IA. L’objectif est de :

- exposer le modèle via une API REST (FastAPI),
- préparer un environnement reproductible avec Docker,- vérifier le bon fonctionnement local de l’API et du conteneur,
- documenter les principales étapes dans un journal de projet.

Le modèle utilisé est un Gradient Boosting entraîné sur le dataset diabetes.csv (Pima Indians Diabetes), puis sérialisé au format joblib sous le nom : best_model_GB_opt.joblib.

2. Préparation de la solution IA

2.1. Modèle et données

Le modèle a été entraîné à partir du dataset diabetes.csv, qui contient les colonnes suivantes : Pregnancies, Glucose, BloodPressure, SkinThickness, Insulin, BMI, DiabetesPedigreeFunction, Age et Outcome (variable cible). Un pipeline scikit-learn a été utilisé, avec prétraitement (imputation, standardisation, encodage), ingénierie de variables et un classifieur GradientBoostingClassifier optimisé. Le meilleur pipeline a ensuite été sauvegardé dans app/models/best_model_GB_opt.joblib.

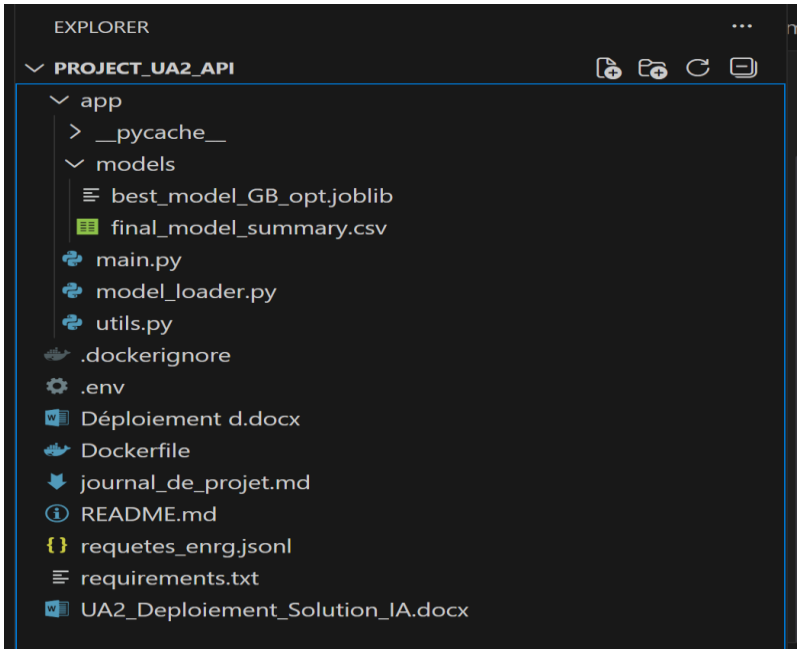
2.2. Features dérivées utilisées par le modèle

Le modèle ne consomme pas uniquement les 8 colonnes brutes du CSV, mais aussi 4 colonnes dérivées : Glucose_over_BMI (rapport Glucose / BMI), BMI_category (catégories d’IMC : underweight, normal, overweight, obese), HighPreg (indicateur 0/1 si Pregnancies ≥ 5) et Age_bin (tranches d’âge : young, middle, old). Dans l’API, ces features sont recalculées à partir des entrées utilisateur avant d’appeler model.predict.

3. Mise en place de l’API (FastAPI)

3.1. Structure du projet côté code

Le projet est organisé dans le dossier project_ua2_api, avec un sous-dossier app/ contenant main.py, utils.py et le modèle best_model_GB_opt.joblib dans app/models/. À la racine se trouvent également requirements.txt, .env, Dockerfile, .dockerignore, journal_de_projet.md et requetes_enrg.jsonl.



3.2. Endpoints de l'API

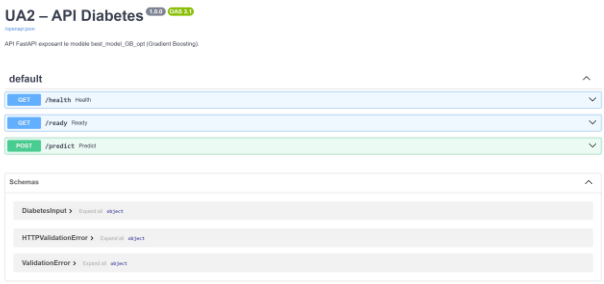
L'API est définie dans app/main.py avec FastAPI. Elle expose les endpoints suivants :

- GET /health : vérifie que l'API est démarrée.
- GET /ready : vérifie que le modèle est chargé.
- POST /predict : reçoit les caractéristiques d'un patient et renvoie une prediction (0/1) et uneprobability_positive.

L'API utilise un schéma d'entrée Pydantic (DiabetesInput) avec les champs pregnancies, glucose, blood_pressure, skin_thickness, insulin, bmi, diabetes_pedigree_function et age. Les features dérivées Glucose_over_BMI, BMI_category, HighPreg et Age_bin sont reconstruites avant l'appel au modèle.

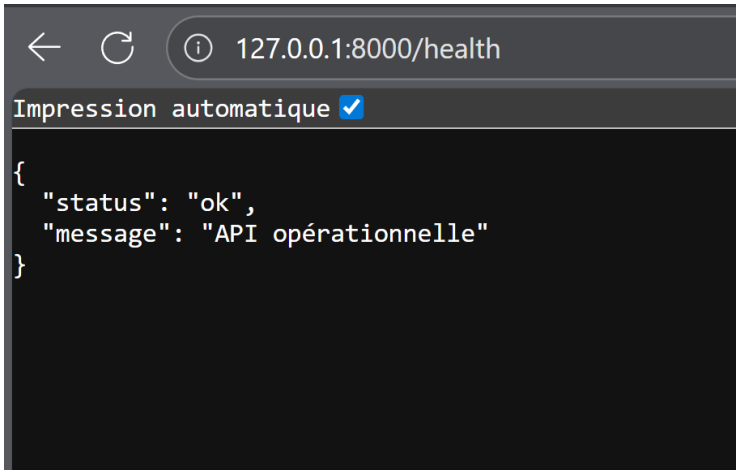
3.3. Tests locaux de l'API

L'API est lancée en local avec la commande : uvicorn app.main:app --reload. Elle est accessible à l'adresse http://127.0.0.1:8000 et la documentation interactive Swagger est disponible sur /docs.



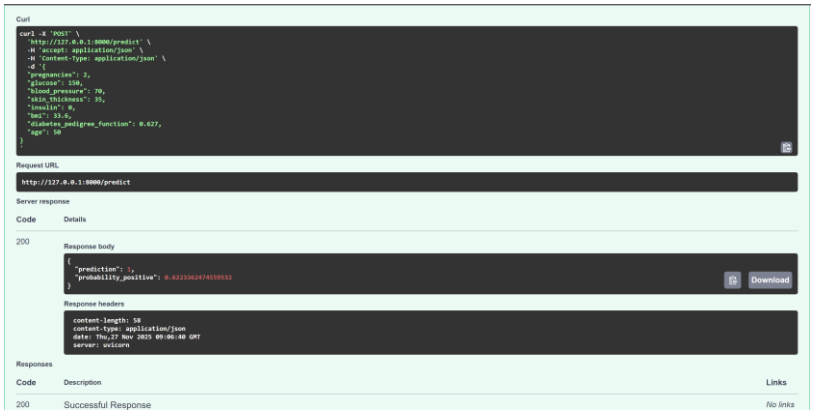
Test de /health :

Une requête GET /health permet de vérifier que l'API fonctionne et renvoie un JSON de confirmation avec un statut HTTP 200.



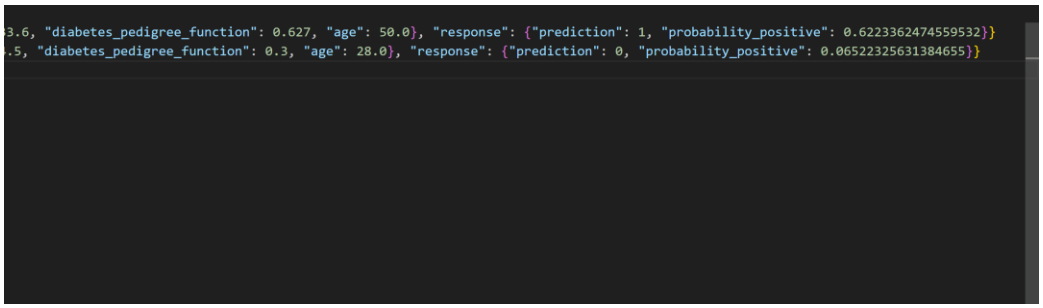
Test de /predict :

Un exemple de requête POST /predict est effectué via /docs avec un patient de test, en envoyant un JSON contenant les 8 features (pregnancies, glucose, etc.). L'API renvoie une prediction (0 ou 1) et une probability_positive. Cela confirme le bon fonctionnement de la chaîne complète : entrée JSON → DataFrame → features dérivées → pipeline scikit-learn → réponse JSON.



4. Journalisation des requêtes

Afin de tracer l'utilisation de l'API, un fichier requetes_enrg.jsonl est utilisé. À chaque appel de /predict, une fonction utilitaire log_prediction (définie dans app/utils.py) ajoute une ligne JSON contenant le timestamp, la source (api_local ou api_docker), l'input et la réponse.



5. Dockerisation de l’application

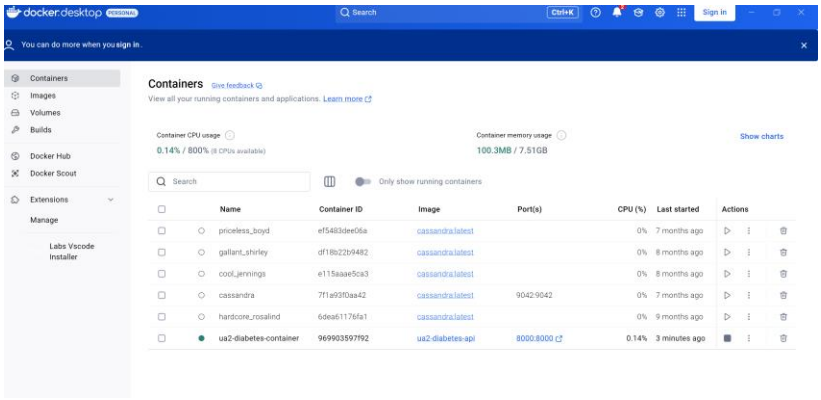
5.1. Fichiers de configuration

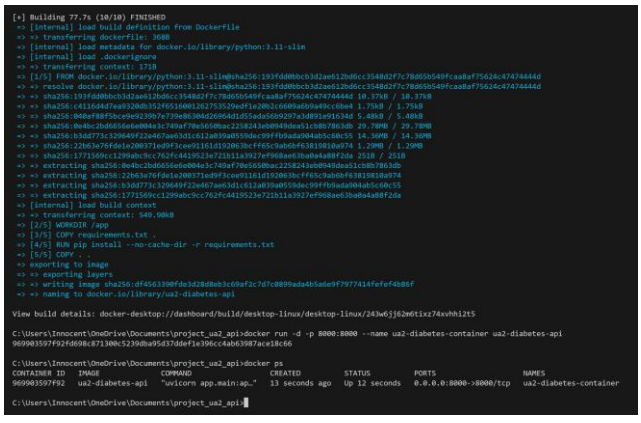
Plusieurs fichiers de configuration sont utilisés pour le déploiement : requirements.txt (liste des dépendances, avec scikit-learn==1.6.1, fastapi, uvicorn, pandas, etc.), .env (pour le chemin du modèle), .dockerignore (pour exclure les fichiers inutiles du build) et Dockerfile (définition de l’image).

5.2. Construction et exécution du conteneur

L’image Docker est construite à partir d’une base python:3.11-slim. Les dépendances sont installées via requirements.txt, puis le code de l’API et le modèle sont copiés dans l’image. La commande de démarrage est : uvicorn app.main:app --host 0.0.0.0 --port 8000.

Les principales commandes utilisées sont : docker build -t ua2-diabetes-api . pour construire l’image, puis docker run -d -p 8000:8000 --name ua2-diabetes-container ua2-diabetes-api pour lancer le conteneur, et docker ps pour vérifier qu’il est en cours d’exécution.





5.3. Tests de l’API dans le conteneur Docker

Une fois le conteneur lancé, l’API est de nouveau testée via les endpoints /health et /predict, cette fois en passant par Docker. Les résultats observés sont identiques à ceux obtenus en local hors conteneur, ce qui confirme la reproductibilité du déploiement.

Principaux problèmes rencontrés et solutions

Plusieurs problèmes techniques ont été rencontrés et résolus lors du déploiement :

- Incompatibilité de version scikit-learn lors du chargement du modèle : le modèle avait été entraîné avecscikit-learn 1.6.1 et rechargé avec une version 1.7.x. Solution : fixer la version scikit-learn==1.6.1 dans requirements.txt.
- Colonnes manquantes dans le ColumnTransformer : le pipeline attendait les features dérivéesBMI_category, HighPreg, Age_bin et Glucose_over_BMI. Solution : les reconstruire dans l'API avant d'appeler model.predict.
- Erreur 'Cannot use median strategy with non-numeric data' : HighPreg était encodé en texte ('yes'/'no') alors qu'un imputer numérique (médiane) était utilisé. Solution : encoder HighPreg en 0/1.
- Erreurs 422 liées à du JSON invalide dans Swagger : corrigées en envoyant un JSON bien formé avecdes guillemets doubles.
- Problèmes de connexion à Docker (daemon non démarré) : résolus en lançant Docker Desktop avant lescommandes docker build et docker run.

6. Conclusion et perspectives

Ce travail m'a permis de passer d'un notebook d'entraînement de modèle à une API de prédiction réutilisable, de comprendre l'importance de sérialiser un pipeline complet (prétraitement + modèle), de gérer l'ingénierie de features côté API et d'utiliser FastAPI avec Docker pour un déploiement reproductible.

Parmi les améliorations possibles, on peut citer l'ajout de tests automatisés des endpoints, l'ajout d'un endpoint /info exposant des métadonnées sur le modèle, la mise en place d'une authentification si l'API est exposée publiquement, ainsi qu'un système de logs et de monitoring plus avancé.