📝 Standards de code et pratiques Machine Learning

Ce document décrit les conventions et bonnes pratiques appliquées dans ce projet.

🏷️ Conventions de branches Git

- **main** : branche de production.
- **dev** : branche de développement.
- **feature/** : branches pour le développement de nouvelles fonctionnalités.
  - Exemple : feature/pipeline-refactor
- **hotfix/** : branches pour les corrections urgentes.
  - Exemple : hotfix/fix-missing-values

Exemple de création de branche fonctionnelle :

```
git checkout -b feature/ajout-endpoint
git push origin feature/ajout-endpoint
```

📝 Conventions de commits

- Utiliser des messages clairs et explicites.
- Structure recommandée :

```
<type>: <description>
```

- Possibilité de lier un numéro d’issue si nécessaire : fix: corrige l'erreur de prédiction (#42)

Types proposés :

- feat: ajout d'une nouvelle fonctionnalité
- fix: correction d'un bug
- docs: modifications de documentation
- test: ajout ou modification de tests
- refactor: amélioration du code sans modification fonctionnelle
- chore: tâches diverses (mise à jour dépendances, etc.)

Exemple :

```
☑ test(pipeline): ajout d’un test minimal de prédiction
🔨 fix(model): suppression d’une feature mal encodée
```

⚙ Standards de codage Python

- Respecter PEP8.
- Utiliser black pour le formatage automatique. black .
- (Optionnel) Utiliser isort pour trier les imports. isort .
- Documenter les fonctions principales avec des docstrings.
- Organiser le code en modules clairs :
  - app/api/ : endpoints FastAPI
  - app/models/ : chargement du pipeline ML
  - app/utils/ : fonctions utilitaires
  - sql/ : scripts SQL éventuels

🧪 Pratiques de test

- Les tests sont écrits avec Pytest.
- Chaque fonctionnalité importante doit avoir au moins un test unitaire.
- Un test minimal est requis pour vérifier :
  - Le chargement du pipeline ML.
  - Une prédiction est possible via un DataFrame conforme.
  - Utilisation d'un fichier feature_names.py pour centraliser la liste des features attendues.

Exemple de test minimal :

```
def test_pipeline_predict():
    pipeline = load_pipeline()
    X = pd.DataFrame([np.zeros(len(FEATURE_NAMES))], columns=FEATURE_NAMES)
    y_pred = pipeline.predict(X)
    assert y_pred.shape == (1,)
```

⚙ Workflow CI/CD

- Utilisation de GitHub Actions.
- Fichier : .github/workflows/Workflow_CI_CD.yml
- Etapes actuelles :
- Installation de Python et Poetry
- Installation des dépendances
- Lancement des tests via pytest
- Création du répertoire build avec app/, pyproject.toml, README.md
- Upload de l'artefact de build
- Déclencheurs :
  - Push sur dev : tests et build.
  - Pull Request vers main : tests, build et validation manuelle avant déploiement.
- Le déploiement cible est Hugging Face Spaces.
- Fichier YAML principal : .github/workflows/ci.yml

Gestion des environnements

- development : environnement de test.
- production : environnement de déploiement.
- Les secrets (API keys, credentials) sont gérés via GitHub Secrets.

Mise à jour du présent document (17/07/2025)

Ce document est vivant et sera mis à jour à mesure que :

- Les cas de test seront précisés
- Le pipeline CI/CD sera complété (ex: déploiement auto)
- Les environnements seront pleinement configurés