IUT de Paris - Rives de Seine
Université Paris Cité

Rapport de Projet : Industrialisation et Qualité Logicielle
Analyse et Refactorisation du modèle Titanic

Équipe : Léo Jean UNITE, Diego CASAS BARCENAS, Romain SALMERON, Baye-Badou DIENG.
Date : 8 février 2026
Logiciels : Visual Studio Code (VS Code), Git, GitHub.

1. Introduction

Ce projet s'inscrit dans une démarche de professionnalisation des pratiques de Data Science. L'objectif n'est plus seulement de produire un modèle prédictif performant, mais de garantir que le code soit maintenable, testable et industrialisable. Nous avons travaillé sur le dataset "Titanic" pour transformer un notebook exploratoire en une solution logicielle structurée.

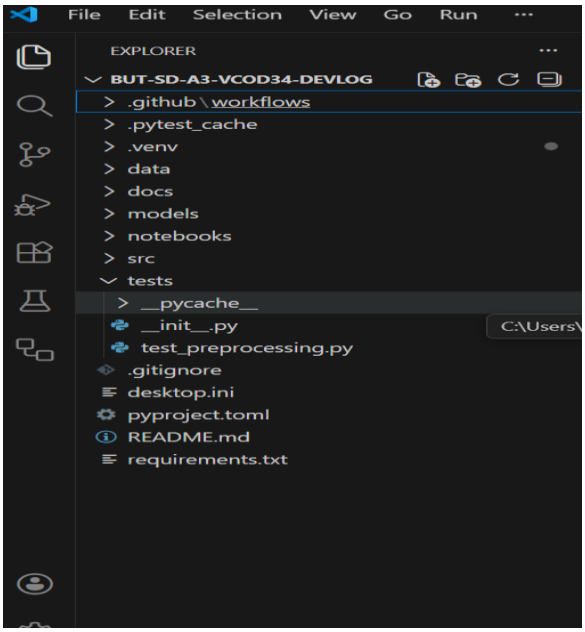
2. Environnement de Travail et Reproductibilité

Dans une première phase du projet, l'ensemble des fichiers a été centralisé sur un espace OneDrive partagé afin de faciliter le partage rapide des informations et le démarrage du travail collaboratif. Cette approche a permis à tous les membres de l'équipe d'accéder aux ressources initiales du projet.

Par la suite, le projet a été migré vers un dépôt GitHub afin de mettre en place un véritable suivi de versions et d'appliquer les bonnes pratiques de collaboration logicielle. La branche main correspond à la version stable du projet, tandis que des branches dédiées ont été utilisées pour le développement de fonctionnalités spécifiques, notamment le prétraitement des données et l'ajout des tests unitaires. Cette organisation a permis de sécuriser les intégrations et de faciliter les revues de code.

Nous avons utilisé l'IDE **Visual Studio Code** pour l'ensemble du développement. Pour garantir que notre code puisse être exécuté par n'importe quel collaborateur, nous avons mis en place :

- **Un environnement virtuel (.venv)** : Pour isoler les dépendances du projet.
- **Le fichier requirements.txt** : Ce fichier liste les librairies strictement nécessaires (Pandas, Scikit-Learn, Pytest, Black, Flake8). Nous avons veillé à supprimer Bibliothèques inutiles pour alléger l'installation.



3. Refactorisation Modulaire (Architecture src/)

Le passage d'un notebook monolithique à une architecture en scripts est la pierre angulaire du projet. Nous avons découpé la logique en quatre modules distincts dans le dossier src :

- **data_loader.py** : Centralise l'accès aux données.
- **data_preprocessing.py** : Contient la fonction preprocess_data. C'est ici que nous gérons le nettoyage des valeurs manquantes et l'encodage des variables Sex et Pclass.
- **model_training.py** : Gère l'entraînement du modèle Random Forest. Nous avons séparé cette logique pour permettre de tester différents hyperparamètres sans relancer tout le nettoyage.
- **model_evaluation.py** : Fournit une fonction d'évaluation standardisée utilisant accuracy_score et classification_report.

[Code de data_processing] :

```
import pandas as pd


def preprocess_data(df):
    """
```

```
    Nettoie les données du Titanic et transforme les variables catégorielles.
    Args:
        df (pd.DataFrame): Le DataFrame brut.
    Returns:
        tuple: (X, y) les features et la cible.
    """

    # 1. Sélection des colonnes utiles (on enlève le nom, le ticket, etc.)
    features = ["Pclass", "Sex", "SibSp", "Parch"]

    # 2. Transformation du texte en chiffres (One-Hot Encoding)
    # Convertit 'Sex' en colonnes numériques (0 ou 1)
    X = pd.get_dummies(df[features])

    # 3. On récupère la cible (Survived) si elle existe dans le DataFrame
    y = df["Survived"] if "Survived" in df.columns else None

    return X, y


if __name__ == "__main__":
    # Petit test rapide
    from data_loader import load_titanic_data

    train, _ = load_titanic_data()
    X, y = preprocess_data(train)
    print("Colonnes après préparation :", X.columns.tolist())
    print("Aperçu des données :\n", X.head())
```

4. Normes de Qualité (PEP 8, Black, Flake8)

Pour que notre code soit "propre" (Clean Code), nous avons imposé des standards stricts :

- **Docstrings** : Chaque fonction est documentée avec ses arguments et son type de retour.
- **Formatage automatique** : Nous avons utilisé l'outil **Black** pour reformater nos fichiers. Cela élimine les débats sur le style et assure une lecture fluide.
- **Linting** : **Flake8** a été configuré pour détecter les erreurs de syntaxe et les variables inutilisées avant chaque commit.

5. Stratégie de Tests Unitaires

Nous avons utilisé le framework **Pytest** pour valider nos fonctions critiques. Notre test principal, situé dans tests/test_preprocessing.py, vérifie que le traitement des données génère bien les colonnes numériques attendues, comme Sex_female.

Nous avons dû configurer la variable PYTHONPATH pour que les tests puissent "voir" le dossier src lors de l'exécution, ce qui a été une étape clé de la robustesse de notre environnement local.

SORTIE DE PYTEST ET CODE DE TEST_processing

```
(.venv) PS C:\Users\badou\Documents\BUT-SD-A3-VCOD34-DEVLOG> pytest
============================= test session starts =============================
platform win32 -- Python 3.10.11, pytest-9.0.2, pluggy-1.6.0
rootdir: C:\Users\badou\Documents\BUT-SD-A3-VCOD34-DEVLOG
configfile: pyproject.toml
plugins: anyio-4.12.1
collected 1 item

tests\test_preprocessing.py .                                            [100%]

============================== 1 passed in 0.82s ==============================
```

```python
import pytest
import pandas as pd
from src.data_preprocessing import preprocess_data


def test_preprocess_data_columns():
    """Vérifie que le prétraitement génère les bonnes colonnes numériques."""
    # On simule un petit tableau de données Titanic
    df_test = pd.DataFrame(
        {
            "Pclass": [3, 1],
            "Sex": ["male", "female"],
            "SibSp": [1, 0],
            "Parch": [0, 0],
            "Survived": [0, 1],
        }
    )

    X, y = preprocess_data(df_test)

    # Test 1 : Vérifier que Sex a été transformé en colonnes dummies
    assert "Sex_female" in X.columns
    assert "Sex_male" in X.columns

    # Test 2 : Vérifier qu'on a bien récupéré la cible 'y'
    assert y is not None
    assert len(y) == 2
```

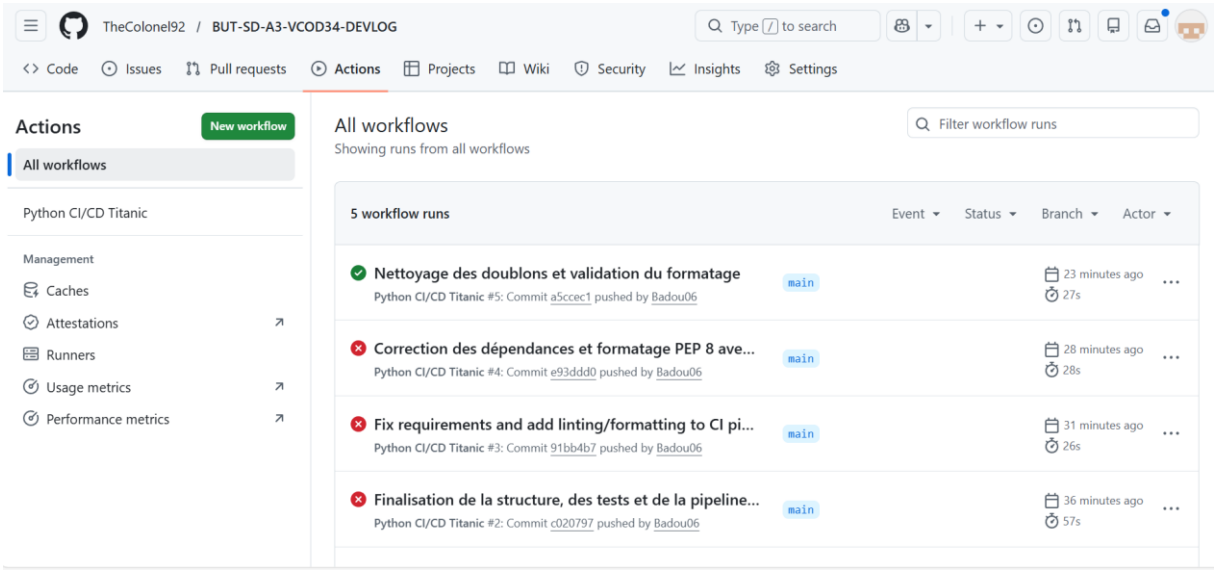
6. Pipeline CI/CD et Gestion des Difficultés Techniques

La mise en place du pipeline via **GitHub Actions** a été la partie la plus complexe. Le fichier main.yml automatise l'installation, le linting, le formatage et les tests à chaque push.

Difficultés rencontrées : Notre principale difficulté est survenue lors de l'exécution de la pipeline sur les serveurs GitHub (Linux). Le build échouait systématiquement à l'étape "Install dependencies" car notre fichier requirements.txt contenait des dépendances spécifiques à Windows, notamment pywinpty.

Pour résoudre ce problème, nous avons dû :

1. Analyser les logs d'erreur de la GitHub Action.
2. Purger le fichier requirements.txt pour ne garder que les librairies multi-plateformes.
3. Relancer la pipeline pour valider que le correctif fonctionnait sur l'environnement de production (Ubuntu).



7. Résultats et Conclusion

Le modèle final présente une précision de **76%**. Au-delà de ce score, la réussite du projet réside dans la mise en place d'une infrastructure solide. Le code est désormais protégé contre les régressions grâce aux tests et à la pipeline CI/CD.

En travaillant en tant qu'entité unique, nous avons pu harmoniser nos pratiques et livrer un projet conforme aux exigences de l'ingénierie logicielle moderne.