Université
Paris Cité

PROJET

INGENIERIE LOGICIELLE APPLIQUEE A UN PROJET DATA

TRISTAN COADOU
PIERRE GAVREL

Titanic Survival Prediction - Ingénierie logicielle appliquée à un Projet Data

Rapport de Projet : Titanic Survival Prediction – Mise en Pratique des Bonnes Pratiques d'Ingénierie Logicielle

---

1. Introduction

Ce rapport présente le travail réalisé dans le cadre du projet *Titanic Survival Prediction*. L'objectif était de récupérer un notebook Kaggle existant et le transformer en plusieurs scripts Python modulaires tout en mettant en place les bonnes pratiques d'ingénierie logicielle : gestion de version (Git/GitHub), respect de la norme PEP 8, tests unitaires (Pytest), pipeline CI/CD (GitHub Actions), etc.

Équipe (2 personnes) :

- **Pierre GAVREL**
- **Tristan COADOU**

---

2. Présentation du Projet

Le projet *Titanic Survival Prediction* vise à prédire la survie des passagers du Titanic à partir de leurs caractéristiques (âge, sexe, classe, etc.). Les données, disponibles sur Kaggle, sont constituées de deux fichiers à utiliser:

- train.csv pour l'entraînement et la validation du modèle
- test.csv pour la soumission finale (optionnel)

Objectifs

1. **Diviser** le code initial (un notebook) en scripts Python bien structurés et réutilisables.
2. **Appliquer** les bonnes pratiques d'ingénierie logicielle (PEP 8, tests, documentation).
3. **Utiliser** Git et GitHub pour la collaboration, mise en place d'une stratégie de branches et revue de code.

4. **Mettre en place** un pipeline d'intégration continue (CI) pour automatiser le linting et les tests.
5. **(Optionnel)** Simuler le déploiement avec Docker.

---

3. Organisation du Travail

Comme nous sommes deux, nous avons défini les rôles suivants :

- **Membre 1** : Responsable de la partie data/preprocessing et de l'entraînement du modèle.
- **Membre 2** : Responsable de l'évaluation et de la rédaction

Nous avons utilisé :

- **Git** en local pour le versionnement du code.
- **GitHub** pour l'hébergement distant, les pull requests et la revue de code.

Stratégie de Branches

- **Workflow** : Contient la version finale et stable du projet.
    - **donnees** : Regroupe les bases de données utilisé.
- **Docs** : Contient la documentation du projet

---

4. Refactorisation du Code

4.1 Découpage en modules

À partir du notebook initial, nous avons créé trois scripts principaux dans le dossier src :

1. **data_preprocessing.py**
    - **load_data(file_path)** : Chargement du jeu de données à partir d'un fichier CSV.
    - **preprocess_data(df)** : Nettoyage et transformation (drop de colonnes inutiles, encodage one-hot, imputation de valeurs manquantes).
    - **split_data(df, target_column)** : Séparation en features (X) et target (y), puis découpage en jeux d'entraînement et de test.

2. **model_training.py**
    - **train_model(X_train, y_train)** : Entraîne un modèle (RandomForestClassifier par défaut).
    - **save_model(model, file_path)** : Sauvegarde du modèle entraîné dans un fichier .pkl via joblib.
3. **model_evaluation.py**
    - **evaluate_model(model, X_test, y_test)** : Évalue le modèle (accuracy_score, classification_report, etc.).
    - **load_model(file_path)** : Charge le modèle préalablement sauvegardé pour une utilisation ultérieure.

Exemple des codes :

1.data_preprocessing.py

```
import pandas as pd

from sklearn.model_selection import train_test_split


def load_data(file_path="Donnees/Donnees.csv"):

    df = pd.read_csv(file_path)

    return df


def preprocess_data(df):

    df = df.drop(columns=["Name", "Ticket", "Cabin"])  # Drop colonnes superflues

    df = pd.get_dummies(df, columns=["Sex", "Embarked"], drop_first=True)

    df = df.fillna(df.mean())  # Remplace NaN par la moyenne

    return df


def split_data(df, target_column="Survived"):

    X = df.drop(columns=[target_column])

    y = df[target_column]

    return train_test_split(X, y, test_size=0.2, random_state=42)
```

2.model_evaluation.py

```
import joblib
from sklearn.metrics import accuracy_score, classification_report

def evaluate_model(model, X_test, y_test):
    y_pred = model.predict(X_test)
    acc = accuracy_score(y_test, y_pred)
    report = classification_report(y_test, y_pred)
    return acc, report

def load_model(file_path="Donnees/model.pkl"):
    return joblib.load(file_path)
```

3.model_training.py

```
import joblib
from sklearn.ensemble import RandomForestClassifier

def train_model(X_train, y_train):
    model = RandomForestClassifier(n_estimators=100, random_state=42)
    model.fit(X_train, y_train)
    return model

def save_model(model, file_path="Donnees/model.pkl"):
    joblib.dump(model, file_path)
```

4.2 Conformité PEP 8

- Utilisation de **noms de variables explicites** (X_train, y_train, etc.).
- Vérification du code automatique avec **flake8** et formatage avec **black** à chaque commit.

4.3 Gestion des dépendances

Pour assurer la reproductibilité, nous avons listé nos bibliothèques dans un fichier requirements.txt ou un environment.yml (selon la méthode choisie). Exemple :

pandas

sklearn

pytest

black

isort

flake8

---

5. Ajout de Tests Unitaires

Nous avons créé un dossier tests contenant plusieurs fichiers Pytest. Par exemple :

- **test_data_preprocessing.py** :
    - Vérifie que load_data charge bien le DataFrame.
    - Vérifie que preprocess_data supprime les colonnes souhaitées et traite les valeurs manquantes.
- **test_model_training.py** :
    - Vérifie que train_model retourne un objet modèle entraîné.
- **test_model_evaluation.py** :
    - Vérifie que evaluate_model retourne l'accuracy et le rapport de classification correctement.

5.1 Exemple de test unitaire

```
# test_data_preprocessing.py

import pandas as pd

from src.data_preprocessing import preprocess_data


def test_preprocess_data():

    df = pd.DataFrame({

        "Name": ["John", "Alice"],

        "Ticket": ["1234", "5678"],

        "Cabin": ["C85", "B28"],

        "Sex": ["male", "female"],

        "Embarked": ["S", "C"],

        "Survived": [1, 0]

    })

    df_processed = preprocess_data(df)

    assert "Name" not in df_processed.columns

    assert "Sex_female" in df_processed.columns

    assert df_processed.isna().sum().sum() == 0
```

---

6. Documentation

README.md

Le fichier **README.md** (à la racine du dépôt) contient :

- **Objectifs du projet**.
- **Méthode d'installation** (cloner le repo, installer les dépendances).
- **Usage** (exemple de commande pour exécuter l'entraînement, les tests, etc.).
- **Contributions de chaque membre**.

---

7. Git et GitHub

7.1 Stratégie de Branches

- **main** : version stable.
- **develop** : intègre les nouvelles fonctionnalités.

Nous créons des branches de fonctionnalité (ex. feature/preprocessing) pour développer chaque partie. Une fois les tests passés, on merge vers develop, puis vers main lorsque tout est validé.

7.2 Pull Requests et Revues de Code

Chaque changement fait l'objet d'une pull request, relue par l'autre membre de l'équipe. Les commentaires et modifications sont discutés avant la fusion.

7.3 Historique des Commits

Des messages de commit explicites sont utilisés, par exemple :

- *"Add data preprocessing script and function to drop irrelevant columns"*
- *"Implement Random Forest training in model_training.py"*

---

9. Conclusion

Bilan

En respectant les bonnes pratiques d'ingénierie logicielle, nous avons :

1. **Réorganisé** le notebook initial en trois modules : data_preprocessing.py, model_training.py, model_evaluation.py.
2. **Assuré** la qualité du code (PEP 8, linting, formatage, docstrings).
3. **Automatisé** la validation et la livraison du code (CI avec GitHub Actions).
4. **Facilité** la collaboration et la gestion des versions via Git/GitHub.

Résultats

- **Autres métriques** : F1-score, classification report, etc.

Ce projet démontre l'importance du respect des standards de l'ingénierie logicielle pour obtenir un code plus maintenable, évolutif et collaboratif, même pour des projets Data/ML.

Annexes

- **Lien GitHub du projet**