Problème : Conception d'un système de prédiction de données avec Python avancé

Interface `IModel`

> L'interface IModel est une classe abstraite qui définit deux méthodes principales que tout modèle de machine learning doit implémenter :

- `train` : Pour entraîner le modèle avec des données d'entraînement.
- `predict` : Pour faire des prédictions sur de nouvelles données.

In [2]:
```python
from abc import ABC, abstractmethod
from typing import List
import pandas as pd

class IModel(ABC):
    @abstractmethod
    #Méthode d'entraînement du modèle.
    def train(self, X: pd.DataFrame, y: pd.Series) -> None:
        pass

    #Méthode de prédiction sur des données.
    @abstractmethod
    def predict(self, X: pd.DataFrame) -> List[float]:
        pass
```

Décorateur `@log_decorator`

> La fonction log_decorator est un décorateur Python qui permet de :

- Enregistrer un message de log avant l'exécution d'une fonction avec ses arguments.

- Enregistrer un message de succès si la fonction s'exécute correctement.
- Enregistrer un message d'erreur avec les détails si une exception survient dans la fonction.

In [3]:
```python
import logging

# Configuration de base du logging
logging.basicConfig(level=logging.INFO, format='%(asctime)s - %(levelname)s - %(message)s', filename='system.log')

def log_decorator(func):
    def wrapper(*args, **kwargs):
        try:
            logging.info(f"Running {func.__name__} with arguments {args} {kwargs}")
            result = func(*args, **kwargs)
            logging.info(f"{func.__name__} completed successfully.")
            return result
        except Exception as e:
            logging.error(f"Error in {func.__name__}: {e}")
            raise e
    return wrapper
```

Classe `DataPipeline`

> La classe DataPipeline gère le chargement, le prétraitement, et la division des données. Elle utilise un générateur pour parcourir les données ligne par ligne.

- Attributs :
  - `_filename` : Nom du fichier CSV contenant les données.
  - `_data` : Données chargées sous forme de DataFrame de pandas.
- Méthodes :
  - `load_data` : Charge les données depuis un fichier CSV.
  - `preprocess_data` : Nettoie les données en supprimant les valeurs manquantes.
  - `split_data` : Divise les données en ensembles d'entraînement et de test.
  - `data_generator` : Générateur qui permet de parcourir les données ligne par ligne.

In [9]:

```python
import pandas as pd
from sklearn.model_selection import train_test_split
from typing import Tuple, Generator

class DataPipeline:
    def __init__(self, filename: str):
        self._filename = filename
        self._data: pd.DataFrame = None

    @property
    #Getter pour les données.
    def data(self) -> pd.DataFrame:
        return self._data

    @data.setter
    #Setter pour les données.
    def data(self, data: pd.DataFrame) -> None:
        self._data = data

    @log_decorator
    #Charge les données depuis un fichier CSV.
    def load_data(self) -> None:
        self._data = pd.read_csv(self._filename)
        print(f"Données chargées depuis 😄 {self._filename}")

    @log_decorator
    def preprocess_data(self) -> None:
        self._data = self._data.dropna(inplace=True)

    @log_decorator
    #Diviser les données en ensemble d'entraînement et de test.
    def split_data(self, test_size: float = 0.2) -> Tuple[pd.DataFrame, pd.DataFrame, pd.Series, pd.Series]:
        X = self._data.drop('target', axis=1)
        y = self._data['target']
        return train_test_split(X, y, test_size=test_size, random_state=42)

    #Générateur pour parcourir les données.
    def data_generator(self, batch_size: int) -> Generator[pd.DataFrame, None, None]:
        for start in range(0, len(self._data), batch_size):
            yield self._data.iloc[start:start + batch_size]
```

Décorateur `@timing`

> Le décorateur @timing mesure et affiche le temps d'exécution des méthodes qu'il décore. Il est utilisé pour suivre la performance en temps réel.

In [5]:
```python
import time

def timing(func):
    def wrapper(*args, **kwargs):
        start_time = time.time()
        result = func(*args, **kwargs)
        end_time = time.time()
        print(f"Temps d'exécution de {func.__name__}: {end_time - start_time:.4f} secondes")
        return result
    return wrapper
```

Classe `RandomForestModel`

> Implémente le modèle de Random Forest à l'aide de l'interface IModel. Ce modèle utilise RandomForestClassifier de scikit-learn.

- Méthodes :
  - `train` : Entraîne le modèle avec des données d'entraînement.
  - `predict` : Fait des prédictions sur de nouvelles données.
  - `evaluate` : Calcule la précision du modèle sur des données de test.

In [10]:
```python
from sklearn.ensemble import RandomForestClassifier
from sklearn.metrics import accuracy_score

class RandomForestModel(IModel):
    def __init__(self):
        self.model = RandomForestClassifier()

    @log_decorator
```

```
    @timing
    #Entraîner le modèle RandomForest.
    def train(self, X: pd.DataFrame, y: pd.Series) -> None:
        self.model.fit(X, y)

    @log_decorator
    @timing
    #Prédiction avec le modèle RandomForest.
    def predict(self, X: pd.DataFrame) -> List[float]:
        return self.model.predict(X).tolist()

    @log_decorator
    #Évaluer le modèle sur un ensemble de test.
    def evaluate(self, X_test: pd.DataFrame, y_test: pd.Series) -> float:
        y_pred = self.predict(X_test)
        return accuracy_score(y_test, y_pred)

    # Évaluer le modèle sans sklearn accuracy_score.
    # def evaluate(self, X_test: pd.DataFrame, y_test: pd.Series) -> float:
    #     y_pred = self.predict(X_test)
    #     correct = sum(y_test == y_pred)
    #     return correct / len(y_test)
```

Classe `SVMModel`

> Implémente le modèle SVM en utilisant SVC de scikit-learn. Ce modèle implémente également l'interface IModel.

- Méthodes :
  - `train` : Entraîne le modèle SVM.
  - `predict` : Fait des prédictions sur de nouvelles données.
  - `evaluate` : Calcule la précision du modèle.

In [13]:
```
from sklearn.svm import SVC

class SVMModel(IModel):
    def __init__(self):
        self.model = SVC()
```

```
    @log_decorator
    @timing
    #Entraîner le modèle SVM.
    def train(self, X: pd.DataFrame, y: pd.Series) -> None:
        self.model.fit(X, y)

    @log_decorator
    @timing
    #Prédiction avec le modèle SVM.
    def predict(self, X: pd.DataFrame) -> List[float]:
        return self.model.predict(X).tolist()

    #Évaluer le modèle sur un ensemble de test.
    def evaluate(self, X_test: pd.DataFrame, y_test: pd.Series) -> float:
        y_pred = self.predict(X_test)
        return accuracy_score(y_test, y_pred)

    # Évaluer le modèle sans sklearn accuracy_score.
    # def evaluate(self, X_test: pd.DataFrame, y_test: pd.Series) -> float:
    #     y_pred = self.predict(X_test)
    #     correct = sum(y_test == y_pred)
    #     return correct / len(y_test)
```

In [14]:

```
if __name__ == "__main__":
    # 1. Charger et prétraiter les données
    pipeline = DataPipeline('data.csv')
    pipeline.load_data()
    pipeline.preprocess_data()

    # 2. Diviser les données
    X_train, X_test, y_train, y_test = pipeline.split_data(test_size=0.2)

    # 3. Entraîner et évaluer le modèle RandomForest
    print("\nRandomForestModel")
    rf_model = RandomForestModel()
    rf_model.train(X_train, y_train)
    rf_model.evaluate(X_test, y_test)

    # 4. Entraîner et évaluer le modèle SVM
```

```
    print("\nSVMModel")
    svm_model = SVMModel()
    svm_model.train(X_train, y_train)
    svm_model.evaluate(X_test, y_test)

    # 7. Utiliser le générateur pour parcourir les données
    print("\nData Generator:")
    for row in pipeline.data_generator(batch_size=1000):
        print(row)
        break  #show the first row  😎
```

```
Données chargées depuis 😄 data.csv

RandomForestModel
Temps d'exécution de train: 0.1964 secondes
Temps d'exécution de predict: 0.0089 secondes

SVMModel
Temps d'exécution de train: 0.0020 secondes
Temps d'exécution de predict: 0.0010 secondes

Data Generator:
   feature1  feature2  feature3  feature4  target
0  0.374540  2.378596  1.068571  0.773856       1
1  0.950714  0.405175  0.836367  0.046878       0
2  0.731994  2.123425  0.599406  0.084544       0
3  0.598658  0.299509  1.188251  0.991677       0
4  0.156019  2.862336  0.296391  1.194115       1
5  0.155995  1.220905  1.526876  0.953861       0
6  0.058084  0.346888  0.354275  0.741358       1
7  0.866176  0.173258  0.904170  1.169807       1
```

Output:

1. **Données chargées depuis `data.csv`**:
    - Les données ont été chargées avec succès depuis le fichier `data.csv`.
2. **RandomForestModel**:

- **Temps d'exécution de `train`** : 0.1964 ondes
  Cela indique que la méthode `train` du modèle `RandomForestModel` a pris **0.1964condes** pour s'exécuter.
- **Temps d'exécution de `predict`** : 0.0089econdes
  Cela montre que la méthode `predict` du modèle `RandomForestModel` a pris **0.0089econdes** pour faire des prédictions.

3. **SVMModel**:

- **Temps d'exécution de `train`** : 0.0020 secondes
  La méthode `train` du modèle `SVMModel` a pris **0.0020 secondes** pour s'exécuter.
- **Temps d'exécution de `predict`** : 0.0010 secondes
  La méthode `predict` du modèle `SVMModel` a pris **0.0010 secondes** pour faire des prédictions.

4. **Data Generator**:

- Le générateur de données a renvoyé les premières lignes des données suivantes :

feature1	feature2	feature3	feature4	target
0.374540	2.378596	1.068571	0.773856	1
0.950714	0.405175	0.836367	0.046878	0
0.731994	2.123425	0.599406	0.084544	0
0.598658	0.299509	1.188251	0.991677	0
0.156019	2.862336	0.296391	1.194115	1
0.155995	1.220905	1.526876	0.953861	0
0.058084	0.346888	0.354275	0.741358	1
0.866176	0.173258	0.904170	1.169807	1

Mohamed BELANNAB