In [ ]:
```python
import numpy as np

import pandas as pd

import math

from sklearn.model_selection import train_test_split
```

In [ ]:
```python
# ===== IMPLÉMENTATION D'UN ARBRE DE DÉCISION PERSONNALISÉ =====

# Classe Node : représente un nœud de l'arbre de décision
class Node:
    def __init__(self, feature=None, threshold=None, left=None, right=None, value=None):
        # feature : l'indice de la colonne utilisée pour la division
        # threshold : la valeur seuil pour diviser les données
        # left : nœud enfant gauche (X <= threshold)
        # right : nœud enfant droit (X > threshold)
        # value : la classe prédite si c'est un nœud feuille (terminal)
        self.feature = feature
        self.threshold = threshold
        self.left = left
        self.right = right
        self.value = value

# Classe DecisionTree : implémente un arbre de décision utilisant l'indice de Gini
class DecisionTree:
    def __init__(self, max_depth=None):
        # max_depth : la profondeur maximale de l'arbre (pour éviter l'surapprentissage)
        self.max_depth = max_depth
        # root : le nœud racine de l'arbre
        self.root = None

    def fit(self, X, y):
        # Entraîne l'arbre en construisant la structure à partir des données
        # X : données d'entrée (features), y : labels/classes cibles
        self.root = self.grow_tree(X, y)

    def gini_impurity(self, y):
        # Calcule l'impureté de Gini pour mesurer le mélange de classes
        # Plus proche de 0 = données pures (une seule classe)
        # Proche de 0.5 = données très mélangées
        _, counts = np.unique(y, return_counts=True)
        prob = counts / len(y)
        return 1 - np.sum(prob**2)

    def information_gain(self, parent, left_child, right_child):
        # Calcule le gain d'information d'une division
        # Mesure combien on a réduit l'impureté en divisant les données
        weight_left = len(left_child) / len(parent)
        weight_right = len(right_child) / len(parent)

        return (self.gini_impurity(parent) - (weight_left * self.gini_impurity(left_child) +
                                              weight_right * self.gini_impurity(right_child)))

    def best_split(self, X, y):
        # Trouve la meilleure division (feature et threshold) pour réduire l'impureté
        best_gain = -1
        best_feature, best_threshold = None, None

        # Essaie toutes les features (colonnes)
        for feature in range(X.shape[1]):
            # Essaie tous les seuils possibles pour cette feature
            thresholds = np.unique(X[:, feature])
            for threshold in thresholds:
                # Divise les données : gauche (<=threshold), droite (>threshold)
                left_mask = X[:, feature] <= threshold
                right_mask = ~left_mask

                # Calcule le gain d'information pour cette division
                gain = self.information_gain(y, y[left_mask], y[right_mask])

                # Garde la division avec le meilleur gain
                if gain > best_gain:
                    best_gain = gain
                    best_feature = feature
                    best_threshold = threshold
        return best_feature, best_threshold

    def grow_tree(self, X, y, depth=0):
        # Construit l'arbre de manière récursive
        n_samples, n_features = X.shape

        # Vérification de sécurité : y ne doit pas être vide
        if len(y) == 0:
            print("y vide à la profondeur", depth)
            return None

        # Compte le nombre de classes différentes
        n_classes = len(np.unique(y))

        # CONDITION D'ARRÊT : on crée une feuille (nœud terminal) si :
        # 1. On a atteint la profondeur max, OU
```

```
        # 2. Il y a moins de 2 échantillons, OU
        # 3. Toutes les données appartiennent à la même classe
        if (depth == self.max_depth or n_samples < 2 or n_classes == 1):
            # Retourne la classe la plus fréquente
            return Node(value=np.argmax(np.bincount(y)))

        # Trouve la meilleure division pour ce nœud
        feature, threshold = self.best_split(X, y)

        # Crée les masques pour diviser les données
        left_mask = X[:, feature] <= threshold
        right_mask = ~left_mask

        # Si l'un des côtés est vide, on s'arrête et on crée une feuille
        if len(y[left_mask]) == 0 or len(y[right_mask]) == 0:
            return Node(value=np.argmax(np.bincount(y)))

        # Récursivement, on crée les sous-arbres gauche et droit
        left = self.grow_tree(X[left_mask], y[left_mask], depth + 1)
        right = self.grow_tree(X[right_mask], y[right_mask], depth + 1)

        # Retourne le nœud interne avec la division
        return Node(feature=feature, threshold=threshold, left=left, right=right)

    def traverse_tree(self, x, node):
        # Parcourt l'arbre pour une seule observation et retourne la prédiction
        # Si on atteint une feuille (node.value n'est pas None), on retourne la prédiction
        if node.value is not None:
            return node.value
        # Sinon, on continue à descendre dans l'arbre selon la valeur de la feature
        if x[node.feature] <= node.threshold:
            return self.traverse_tree(x, node.left)
        else:
            return self.traverse_tree(x, node.right)

    def predict(self, X):
        # Prédit la classe pour toutes les observations
        # Applique traverse_tree pour chaque ligne de X
        return np.array([self.traverse_tree(x, self.root) for x in X])
```

In [ ]:

```
# ===== IMPLÉMENTATION D'UNE FORÊT ALÉATOIRE (RANDOM FOREST) =====

# Counter : pour compter les occurrences et déterminer la classe la plus fréquente
from collections import Counter

class RandomForestClassifier:
    # Une forêt aléatoire = ensemble d'arbres de décision avec agrégation par vote majoritaire

    def __init__(self, n_estimators=10, max_depth=10, max_features=None):
        # n_estimators : nombre d'arbres à entraîner (par défaut 10)
        self.n_estimators = n_estimators
        # max_depth : profondeur maximale de chaque arbre
        self.max_depth = max_depth
        # max_features : nombre de features à utiliser pour chaque arbre
        #                (None = toutes les features)
        self.max_features = max_features
        # trees : liste pour stocker les arbres entraînés
        self.trees = []

    def bootstrap_sample(self, X, y):
        # Bootstrap = rééchantillonnage avec remplacement
        # Crée un sous-ensemble aléatoire de la même taille que l'ensemble original
        # Cela permet à chaque arbre d'avoir des données légèrement différentes
        n_samples = X.shape[0]
        # Tire aléatoirement n_samples indices (avec remplacement)
        indices = np.random.choice(n_samples, size=n_samples, replace=True)
        return X[indices], y[indices]

    def fit(self, X, y):
        # Entraîne tous les arbres de la forêt
        self.trees = []
        n_features = X.shape[1]
        # Utilise max_features si défini, sinon utilise toutes les features
        max_features = self.max_features if self.max_features else n_features

        # Entraîne n_estimators arbres
        for i in range(self.n_estimators):
            # Crée un nouvel arbre
            tree = DecisionTree(max_depth=self.max_depth)

            # Crée un échantillon bootstrap des données
            X_sample, y_sample = self.bootstrap_sample(X, y)

            # Sélectionne aléatoirement max_features features pour cet arbre
            selected_features = np.random.choice(n_features, size=max_features, replace=True)

            # Extrait seulement les features sélectionnées
            X_sample_subset = X_sample[:, selected_features]

            # Entraîne l'arbre sur les données bootstrap avec les features sélectionnées
            tree.fit(X_sample_subset, y_sample)

            # Stocke l'arbre et les features utilisées (utile pour la prédiction)
            self.trees.append((tree, selected_features))
```

```
    def predict(self, X):
        # Prédit la classe pour toutes les observations en utilisant tous les arbres

        # Récupère les prédictions de tous les arbres
        # Pour chaque arbre, on utilise seulement les features qu'il a utilisées
        tree_predictions = np.array([tree.predict(X[:, features]) for tree, features in self.trees])

        # Pour chaque observation, on fait un vote majoritaire parmi les prédictions des arbres
        # Counter.most_common(1)[0][0] retourne la classe la plus fréquente
        majority_votes = [Counter(tree_predictions[:, i]).most_common(1)[0][0] for i in range(X.shape[0])]

        return np.array(majority_votes)
```

In [ ]:
```
# ===== CHARGEMENT DES DONNÉES =====
# Charge le fichier CSV dans un DataFrame pandas
data = pd.read_csv("Employeetest.csv")
```

In [ ]:
```
# ===== EXPLORATION DES DONNÉES : INFORMATIONS =====
# Affiche les infos sur le dataset : types de données, valeurs manquantes, etc.
data.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 4653 entries, 0 to 4652
Data columns (total 9 columns):
 #   Column                     Non-Null Count  Dtype
---  ------                     --------------  -----
 0   Education                  4653 non-null   object
 1   JoiningYear                4653 non-null   int64
 2   City                       4653 non-null   object
 3   PaymentTier                4653 non-null   int64
 4   Age                        4653 non-null   int64
 5   Gender                     4653 non-null   object
 6   EverBenched                4653 non-null   object
 7   ExperienceInCurrentDomain  4653 non-null   int64
 8   LeaveOrNot                 4653 non-null   int64
dtypes: int64(5), object(4)
memory usage: 327.3+ KB
```

In [ ]:
```
# ===== EXPLORATION DES DONNÉES : APERÇU =====
# Affiche les 5 premières lignes du dataset
data.head()
```

Out[ ]:

	Education	JoiningYear	City	PaymentTier	Age	Gender	EverBenched	ExperienceInCurrentDomain	LeaveOrNot
0	Bachelors	2017	Bangalore	3	34	Male	No	0	0
1	Bachelors	2013	Pune	1	28	Female	No	3	1
2	Bachelors	2014	New Delhi	3	38	Female	No	2	0
3	Masters	2016	Bangalore	3	27	Male	No	5	1
4	Masters	2017	Pune	3	24	Male	Yes	2	1

In [ ]:
```
# ===== PRÉTRAITEMENT DES DONNÉES : ENCODAGE =====
# Les modèles ML travaillent avec des nombres, pas des textes
# Il faut transformer les variables catégoriques en nombres

# Importe les encodeurs
from sklearn.preprocessing import LabelEncoder, OneHotEncoder

# LabelEncoder : convertit les catégories en nombres (0, 1, 2, ...)
# Utilisé pour les colonnes binaires ou ordonnelles
encoder = LabelEncoder()

# Encode 'EverBenched' : Yes/No → 0/1
data['EverBenched'] = encoder.fit_transform(data['EverBenched'])

# Encode 'Gender' : Male/Female → 0/1
data['Gender'] = encoder.fit_transform(data['Gender'])

# Encode 'Education' : Bachelor/Master/PhD → 0/1/2
data['Education'] = encoder.fit_transform(data['Education'])

# OneHotEncoder : convertit les colonnes catégoriques multi-classes en colonnes binaires
# Exemple : City [Delhi, London, Paris] → [City_Delhi, City_London, City_Paris]
# Cela évite les relations d'ordre implicites entre les valeurs
ohe = OneHotEncoder(sparse_output=False)

# Applique OneHotEncoder à la colonne 'City'
type_encoded = ohe.fit_transform(data[['City']])

# Crée un DataFrame avec les colonnes encodées
type_encoded_df = pd.DataFrame(type_encoded, columns=ohe.get_feature_names_out(['City']))

# Combine l'ancien DataFrame avec les colonnes One-Hot encodées
data = pd.concat([data.reset_index(drop=True), type_encoded_df.reset_index(drop=True)], axis=1)

# Supprime la colonne originale 'City' (remplacée par les colonnes One-Hot)
data = data.drop(columns="City")

# Affiche les premières lignes pour vérifier
```

```
data.head()
data.head(30)
```

Out[ ]:

	Education	JoiningYear	PaymentTier	Age	Gender	EverBenched	ExperienceInCurrentDomain	LeaveOrNot	City_Bangalore	City_New Delhi	City_Pune
0	0	2017	3	34	1	0	0	0	1.0	0.0	0.0
1	0	2013	1	28	0	0	3	1	0.0	0.0	1.0
2	0	2014	3	38	0	0	2	0	0.0	1.0	0.0
3	1	2016	3	27	1	0	5	1	1.0	0.0	0.0
4	1	2017	3	24	1	1	2	1	0.0	0.0	1.0
5	0	2016	3	22	1	0	0	0	1.0	0.0	0.0
6	0	2015	3	38	1	0	0	0	0.0	1.0	0.0
7	0	2016	3	34	0	0	2	1	1.0	0.0	0.0
8	0	2016	3	23	1	0	1	0	0.0	0.0	1.0
9	1	2017	2	37	1	0	2	0	0.0	1.0	0.0
10	1	2012	3	27	1	0	5	1	1.0	0.0	0.0
11	0	2016	3	34	1	0	3	0	0.0	0.0	1.0
12	0	2018	3	32	1	1	5	1	0.0	0.0	1.0
13	0	2016	3	39	1	0	2	0	1.0	0.0	0.0
14	0	2012	3	37	1	0	4	0	1.0	0.0	0.0
15	0	2017	1	29	1	0	3	0	1.0	0.0	0.0
16	0	2014	3	34	0	0	2	0	1.0	0.0	0.0
17	0	2014	3	34	1	0	4	0	0.0	0.0	1.0
18	0	2015	2	30	0	0	0	1	0.0	0.0	1.0
19	0	2016	2	22	0	0	0	1	0.0	1.0	0.0
20	0	2012	3	37	1	0	0	0	1.0	0.0	0.0
21	1	2017	2	28	1	0	4	0	0.0	1.0	0.0
22	0	2017	2	36	1	0	3	0	0.0	1.0	0.0
23	0	2015	3	27	1	1	5	0	1.0	0.0	0.0
24	0	2017	3	29	1	0	4	0	1.0	0.0	0.0
25	0	2013	3	22	0	1	0	0	1.0	0.0	0.0
26	0	2016	3	37	1	0	2	0	1.0	0.0	0.0
27	0	2015	3	23	1	0	1	0	1.0	0.0	0.0
28	0	2013	2	31	0	0	2	1	0.0	0.0	1.0
29	1	2017	2	30	0	0	2	0	0.0	1.0	0.0

In [ ]:
```
# ===== SÉPARATION FEATURES (X) ET TARGET (y) =====
# X = toutes les colonnes sauf 'LeaveOrNot' (features/prédicteurs)
X = data.drop(columns="LeaveOrNot")

# y = colonne 'LeaveOrNot' (cible/variable à prédire)
y = data["LeaveOrNot"]

# Affiche la distribution de la cible (nombre de 0 et de 1)
print(data['LeaveOrNot'].value_counts())
```

```
LeaveOrNot
0    3053
1    1600
Name: count, dtype: int64
```

Out[ ]:

	Education	JoiningYear	PaymentTier	Age	Gender	EverBenched	ExperienceInCurrentDomain	City_Bangalore	City_New Delhi	City_Pune
0	0	2017	3	34	1	0	0	1.0	0.0	0.0
1	0	2013	1	28	0	0	3	0.0	0.0	1.0
2	0	2014	3	38	0	0	2	0.0	1.0	0.0
3	1	2016	3	27	1	0	5	1.0	0.0	0.0
4	1	2017	3	24	1	1	2	0.0	0.0	1.0

In [ ]:
```
# ===== CONVERSION EN TABLEAUX NUMPY =====
# Convertit X et y de DataFrame/Series pandas en tableaux NumPy
# Les modèles custom implémentés ci-dessus travaillent avec des arrays NumPy
X = X.values
y = y.values
```

In [ ]:
```python
# ===== DIVISION EN ENSEMBLE D'ENTRAÎNEMENT ET TEST =====
# Divise les données : 65% pour l'entraînement, 35% pour le test
# random_state=42 pour la reproductibilité (mêmes résultats à chaque fois)
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.35)
```

In [ ]:
```python
# ===== ENTRAÎNEMENT DE LA FORÊT ALÉATOIRE PERSONNALISÉE =====
# Crée une instance de RandomForestClassifier avec nos paramètres
# n_estimators=500 : 500 arbres de décision
# max_depth=9 : chaque arbre a une profondeur max de 9
# max_features=15 : chaque arbre utilise 15 features aléatoires (au lieu de toutes)
model = RandomForestClassifier(n_estimators=500, max_depth=9, max_features=15)

# Entraîne le modèle sur les données d'entraînement
model.fit(X_train, y_train)

# Fait les prédictions sur l'ensemble de test
predictions = model.predict(X_test)

# Calcule la précision (accuracy) = nombre de bonnes prédictions / total
prob = np.sum(predictions == y_test) / len(y_test)
print(f'Accuracy de notre Random Forest: {prob * 100:.2f}%')
```

```
Sklearn Accuracy: 85.64%
```

In [ ]:
```python
# ===== COMPARAISON AVEC LA FORÊT ALÉATOIRE DE SKLEARN =====
# Importe la Random Forest de la librairie sklearn
from sklearn.ensemble import RandomForestClassifier as SklearnRandomForest
# Importe la métrique de précision
from sklearn.metrics import accuracy_score

# Crée une instance de Random Forest de sklearn avec paramètres par défaut
clf = SklearnRandomForest()

# Entraîne le modèle sklearn
# .ravel() transforme y_train en vecteur 1D pour éviter un warning
clf.fit(X_train, y_train.ravel())

# Fait les prédictions avec le modèle sklearn
y_sklearn_pred = clf.predict(X_test)

# Calcule la précision du modèle sklearn
sklearn_accuracy = accuracy_score(y_test, y_sklearn_pred)
print(f'Accuracy de sklearn Random Forest: {sklearn_accuracy * 100:.2f}%')
```

```
Sklearn Accuracy: 83.12%
```

In [ ]:
```python
# ===== SAUVEGARDE DU MODÈLE ENTRAÎNÉ =====
# Importe le module pickle pour sérialiser/déserialiser les objets Python
import pickle

# Ouvre un fichier en mode écriture binaire ('wb')
# 'with' ferme automatiquement le fichier à la fin
with open('random_forest_model.pkl', 'wb') as f:
    # Sérialise le modèle et le sauvegarde dans le fichier
    pickle.dump(model, f)

# Affiche un message de confirmation
print("Modèle Random Forest sauvegardé dans 'random_forest_model.pkl'")
```

```
✅ Modèle Random Forest sauvegardé dans 'random_forest_model.pkl'
```

In [ ]:
```python
# ===== CHARGEMENT DU MODÈLE SAUVEGARDÉ =====
# Ouvre le fichier du modèle en mode lecture binaire ('rb')
with open('random_forest_model.pkl', 'rb') as f:
    # Désérialise l'objet modèle depuis le fichier
    loaded_model = pickle.load(f)

# Maintenant 'loaded_model' est identique au 'model' d'avant la sauvegarde
# Vous pouvez l'utiliser pour faire de nouvelles prédictions sans réentraîner
```