```python
# CART (DECISTION TREE REGRESSION) - Ce model implementé a l'aide des notions de la programmation orientée objet, puisque je vois que c'est plus facile de comprendre le code et de le modifier (la programation fonctionnelle pour implementer ce type de model rend les choses plus compliquées)
```

Import des bibliothèques

```python
import numpy as np
import seaborn
from sklearn.model_selection import train_test_split
from sklearn.metrics import mean_squared_error
```

Données

```python
data = seaborn.load_dataset('tips')
col_names = data.columns.tolist()
col_names.remove('tip')
```

Class noeud

```python
class Node():
    def __init__(self, feature_index=None, threshold=None, left=None, right=None, var_red=None, value=None):
        # constructeur

        # pour les noeuds de décision
        self.feature_index = feature_index
        self.threshold = threshold
        self.left = left
        self.right = right
        self.var_red = var_red

        # pour les noeuds feuilles
        self.value = value
```

Tree class

```python
class DecisionTreeRegressor():
    def __init__(self, min_samples_split=2, max_depth=2):
        # constructeur

        # inisialisation de la racine de l'arbre
        self.root = None
        self.dic = {}

        # conditions d'arret
        self.min_samples_split = min_samples_split
```

```
        self.max_depth = max_depth

    def build_tree(self, dataset, curr_depth=0):
        # fonction récursive pour construire l'arbre de decision

        X, Y = dataset[:,:-1], dataset[:,-1]
        num_samples, num_features = np.shape(X)
        best_split = {}
        # fractionner jusqu'à ce que les conditions d'arrêt soient
vrai
        if num_samples>=self.min_samples_split and
curr_depth<=self.max_depth:
            # trouver la meilleure répartition
            best_split = self.get_best_split(dataset, num_samples,
num_features)
            # vérifier si le gain d'information est positif
            if best_split["var_red"]>0:
                # récurrence gauche
                left_subtree =
self.build_tree(best_split["dataset_left"], curr_depth+1)
                # récurrence droite
                right_subtree =
self.build_tree(best_split["dataset_right"], curr_depth+1)
                # retourner le noeud de décision
                return Node(best_split["feature_index"],
best_split["threshold"],
                            left_subtree, right_subtree,
best_split["var_red"])

        # Calculer le noeud feuille
        leaf_value = self.calculate_leaf_value(Y)
        # retourner le noeud feuille
        return Node(value=leaf_value)

    def get_best_split(self, dataset, num_samples, num_features):
        # pour trouver la meilleure répartition

        # dictionnaire pour stocker la meilleure part
        best_split = {}
        max_var_red = -float("inf")

        # boucle sur l'ensemble des caractéristiques
        for feature_index in range(num_features):
            feature_values = dataset[:, feature_index]
            possible_thresholds = np.unique(feature_values)
            # boucler sur toutes les valeurs des caractéristiques
présentes dans les données
            for threshold in possible_thresholds:
                # obtenir la répartition actuelle
```

```
                dataset_left, dataset_right = self.split(dataset,
feature_index, threshold)
                # vérifier si les enfants ne sont pas nuls
                if len(dataset_left)>0 and len(dataset_right)>0:
                    y, left_y, right_y = dataset[:, -1],
dataset_left[:, -1], dataset_right[:, -1]
                    # calculer la variance intra-groupe
                    curr_var_red = self.variance_reduction(y, left_y,
right_y)
                    # mettre à jour la meilleure répartition si
nécessaire
                    if curr_var_red>max_var_red:
                        best_split["feature_index"] = feature_index
                        best_split["threshold"] = threshold
                        best_split["dataset_left"] = dataset_left
                        best_split["dataset_right"] = dataset_right
                        best_split["var_red"] = curr_var_red
                        max_var_red = curr_var_red

        # retourer meilleure répartition
        return best_split

    def split(self, dataset, feature_index, threshold):
        # pour diviser les données

        dataset_left = np.array([row for row in dataset if
row[feature_index]<=threshold])
        dataset_right = np.array([row for row in dataset if
row[feature_index]>threshold])
        return dataset_left, dataset_right

    def variance_reduction(self, parent, l_child, r_child):
        # fonction permettant de calculer la variance intra-groupe

        weight_l = len(l_child) / len(parent)
        weight_r = len(r_child) / len(parent)
        reduction = np.var(parent) - (weight_l * np.var(l_child) +
weight_r * np.var(r_child))
        return reduction

    def calculate_leaf_value(self, Y):
        # pour calculer le nœud de la feuille

        val = np.mean(Y)
        return val

    def print_tree(self, tree=None, indent=" "):
        # pour imprimer l'arbre
```

```
        if not tree:
            tree = self.root

        if tree.value is not None:
            print(tree.value)

        else:
            if isinstance(tree.threshold, float):
                print(col_names[tree.feature_index], "<=", tree.threshold, " | gain ratio", tree.var_red)
            else:
                print(col_names[tree.feature_index], "->", tree.threshold, " | gain ratio", tree.var_red)
            print("%sleft (true):" % (indent), end="")
            self.print_tree(tree.left, indent + indent)
            print("%sright (false):" % (indent), end="")
            self.print_tree(tree.right, indent + indent)

    def tree_to_dict(self, tree=None):
        # pour transformer l'arbre en dictionnaire

        if tree is None:
            tree = self.root

        if tree.value is not None:
            return tree.value

        feature_name = col_names[tree.feature_index] if tree.feature_index is not None else None

        left_tree = self.tree_to_dict(tree.left)
        right_tree = self.tree_to_dict(tree.right)

        if feature_name is not None:
            if isinstance(tree.threshold, float):
                return {feature_name: {f'<={tree.threshold}': {'left (true)': left_tree, 'right (false)': right_tree}}}
            else:
                return {feature_name: {tree.threshold: {'left (true)': left_tree, 'right (false)': right_tree}}}
        else:
            # Il s'agit du nœud feuille
            return tree.value

    def fit(self, X, Y):
        # pour former l'arbre

        dataset = np.concatenate((X, Y), axis=1)
        self.root = self.build_tree(dataset)
        self.dic = self.tree_to_dict()
```

```python
    def make_prediction(self, x, tree):
        # pour prédire un seul point de données

        if tree.value!=None: return tree.value
        feature_val = x[tree.feature_index]
        if feature_val<=tree.threshold:
            return self.make_prediction(x, tree.left)
        else:
            return self.make_prediction(x, tree.right)

    def predict(self, X):
        # fonction de prédiction d'un nouvel ensemble de données

        preditions = [self.make_prediction(x, self.root) for x in X]
        return preditions
```

Split en données Train et Test

```python
X = data.iloc[:, [col for col in range(data.shape[1]) if col !=
1]].values
Y = data.iloc[:, 1].values.reshape(-1, 1)
X_train, X_test, Y_train, Y_test = train_test_split(X, Y,
test_size=.2, random_state=41)
```

Fit le model au données Train

```python
regressor = DecisionTreeRegressor(min_samples_split=3, max_depth=3)
regressor.fit(X_train,Y_train)
regressor.print_tree()
```

```
total_bill <= 20.45  | gain ratio 0.6335364363128835
 left (true):total_bill <= 16.27  | gain ratio 0.16350535284516765
  left (true):total_bill <= 13.81  | gain ratio 0.04675673437500033
    left (true):total_bill <= 5.75  | gain ratio 0.033146260683760476
        left (true):1.0
        right (false):1.9640384615384616
    right (false):day -> Sat  | gain ratio 0.12595238095238093
        left (true):2.718571428571429
        right (false):2.006666666666667
  right (false):smoker -> No  | gain ratio 0.04996230670776791
    left (true):total_bill <= 16.29  | gain ratio 0.028722773946360136
        left (true):3.71
        right (false):2.765862068965517
    right (false):total_bill <= 16.32  | gain ratio
0.09963669421487614
        left (true):4.3
        right (false):3.2020000000000004
 right (false):total_bill <= 45.35  | gain ratio 0.881903970548394
```

```
  left (true):size -> 3  | gain ratio 0.22224944056438378
    left (true):sex -> Female  | gain ratio 0.09893555555555622
        left (true):3.9859999999999998
        right (false):3.2474999999999996
    right (false):smoker -> No  | gain ratio 0.2507772282876326
        left (true):4.773888888888889
        right (false):3.741818181818182
  right (false):total_bill <= 48.27  | gain ratio 1.7050888888888882
    left (true):6.73
    right (false):9.5
```

```python
print(regressor.dic)
```

```
{'total_bill': {'<=20.45': {'left (true)': {'total_bill': {'<=16.27':
{'left (true)': {'total_bill': {'<=13.81': {'left (true)':
{'total_bill': {'<=5.75': {'left (true)': 1.0, 'right (false)':
1.9640384615384616}}}, 'right (false)': {'day': {'Sat': {'left
(true)': 2.718571428571429, 'right (false)': 2.006666666666667}}}}}},
'right (false)': {'smoker': {'No': {'left (true)': {'total_bill':
{'<=16.29': {'left (true)': 3.71, 'right (false)':
2.765862068965517}}}, 'right (false)': {'total_bill': {'<=16.32':
{'left (true)': 4.3, 'right (false)': 3.2020000000000004}}}}}}}}},
'right (false)': {'total_bill': {'<=45.35': {'left (true)': {'size':
{3: {'left (true)': {'sex': {'Female': {'left (true)':
3.9859999999999998, 'right (false)': 3.2474999999999996}}}, 'right
(false)': {'smoker': {'No': {'left (true)': 4.773888888888889, 'right
(false)': 3.741818181818182}}}}}}, 'right (false)': {'total_bill':
{'<=48.27': {'left (true)': 6.73, 'right (false)': 9.5}}}}}}}}}
```

Test du model pour avoir un metric d'evaluation

```python
from sklearn.metrics import accuracy_score

Y_pred = regressor.predict(X_test)
print(f"MSE: {np.sqrt(mean_squared_error(Y_test, Y_pred))}")
```

```
MSE: 1.1184175011994184
```