C4.5 - 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

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

Données

```
col_names = ['sepal length', 'sepal width', 'petal length', 'petal width', 'type']
data = seaborn.load_dataset("iris", skiprows=1, header=None, names=col_names)
```

Class noeud

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

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

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

Class Arbre de decision

```
class DecisionTreeClassifier():
    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)

        # 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["info_gain"]>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)
                self.dic[best_split["feature_index"]] = 
best_split["threshold"]
                # retourner le noeud de décision
                return Node(best_split["feature_index"], 
best_split["threshold"], 
                            left_subtree, right_subtree, 
best_split["info_gain"])

        # 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_info_gain = -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 le gain d'information
                    curr_info_gain = self.gain_ratio(y, left_y, 
right_y, "entropy")
                    # mettre à jour la meilleure répartition si 
nécessaire
                    if curr_info_gain>max_info_gain:
                        best_split["feature_index"] = feature_index
                        best_split["threshold"] = threshold
                        best_split["dataset_left"] = dataset_left
                        best_split["dataset_right"] = dataset_right
                        best_split["info_gain"] = curr_info_gain
                        max_info_gain = curr_info_gain
        # retour 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 gain_ratio(self, parent, l_child, r_child, mode="entropy"):
        # fonction permettant de calculer le rapport de gain 
d'information

        weight_l = len(l_child) / len(parent)
        weight_r = len(r_child) / len(parent)
        if mode=="gini":
```

```
            gain = self.gini_index(parent) -
(weight_l*self.gini_index(l_child) +
weight_r*self.gini_index(r_child))
        else:
            gain = self.entropy(parent) -
(weight_l*self.entropy(l_child) + weight_r*self.entropy(r_child))

        if weight_l != 0 and weight_l != 0:
            info_inter = - weight_l*np.log2(weight_l) -
weight_r*np.log2(weight_l)

        return gain/info_inter if info_inter else 0

    def entropy(self, y):
        # fonction pour calculer l'entropie

        class_labels = np.unique(y)
        entropy = 0
        for cls in class_labels:
            p_cls = len(y[y == cls]) / len(y)
            entropy += -p_cls * np.log2(p_cls)
        return entropy

    def gini_index(self, y):
        # Fonction de calcul de l'indice de Gini

        class_labels = np.unique(y)
        gini = 0
        for cls in class_labels:
            p_cls = len(y[y == cls]) / len(y)
            gini += p_cls**2
        return 1 - gini

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

        Y = list(Y)
        return max(Y, key=Y.count)

    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:
            print(col_names[tree.feature_index], "<=", tree.threshold,
```

```
" | gain ratio: ", tree.info_gain)
            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:
            return {feature_name: {f'<={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()

    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

    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)
```

Split en données Train et Test

```
X = data.iloc[:, :-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

```
classifier = DecisionTreeClassifier(min_samples_split=3, max_depth=3)
classifier.fit(X_train,Y_train)
classifier.print_tree()
```

```
petal length <= 5.0  | gain ratio:  1.3662872285584726
 left(true):petal width <= 1.8  | gain ratio:  2.900002940869681
  left(true):petal width <= 1.5  | gain ratio:  2.1339495824489894
    left(true):petal length <= 1.9  | gain ratio:  1.0757147192686414
        left(true):setosa
        right(false):versicolor
    right(false):sepal length <= 6.2  | gain ratio:  1.0
        left(true):virginica
        right(false):versicolor
  right(false):virginica
 right(false):virginica
```

```
print(classifier.dic)
```

```
{'petal length': {'<=5.0': {'left (true)': {'petal width': {'<=1.8':
{'left (true)': {'petal width': {'<=1.5': {'left (true)': {'petal
length': {'<=1.9': {'left (true)': 'setosa', 'right (false)':
'versicolor'}}}, 'right (false)': {'sepal length': {'<=6.2': {'left
(true)': 'virginica', 'right (false)': 'versicolor'}}}}}}, 'right
(false)': 'virginica'}}}, 'right (false)': 'virginica'}}}
```

Test du model pour avoir un metric d'evaluation

```
Y_pred = classifier.predict(X_test)
accuracy_score(Y_test, Y_pred)
```

```
0.8666666666666667
```

Ici j'est utilisé toutes les données pour avoir un arbre plus complet

```
## Fit the model
classifier2 = DecisionTreeClassifier(min_samples_split=3, max_depth=3)
classifier2.fit(X, Y)
classifier2.print_tree()
```

```
petal width <= 1.7  | gain ratio:  1.2979499120245854
 left(true):petal length <= 5.1  | gain ratio:  3.227598619770726
  left(true):petal length <= 4.9  | gain ratio:  1.6198161123722474
    left(true):petal width <= 1.6  | gain ratio:  5.551330718033408
        left(true):setosa
        right(false):virginica
    right(false):petal width <= 1.5  | gain ratio:  1.0
        left(true):virginica
        right(false):versicolor
  right(false):virginica
 right(false):sepal width <= 3.1  | gain ratio:  0.07279600530526306
  left(true):virginica
  right(false):sepal length <= 5.9  | gain ratio:  0.09750399798300154
    left(true):versicolor
    right(false):virginica
```

```
print(classifier2.dic)
```

```
{'petal width': {'<=1.7': {'left (true)': {'petal length': {'<=5.1':
{'left (true)': {'petal length': {'<=4.9': {'left (true)': {'petal
width': {'<=1.6': {'left (true)': 'setosa', 'right (false)':
'virginica'}}}, 'right (false)': {'petal width': {'<=1.5': {'left
(true)': 'virginica', 'right (false)': 'versicolor'}}}}}}, 'right
(false)': 'virginica'}}}, 'right (false)': {'sepal width': {'<=3.1':
{'left (true)': 'virginica', 'right (false)': {'sepal length':
{'<=5.9': {'left (true)': 'versicolor', 'right (false)':
'virginica'}}}}}}}}}
```