Exercice 1 (Classification):

Classification des données des Iris de Fisher en utilisant le modèle d'arbre de décision C4.5

In [44]:
```
from math import log
import math
import treeplotter as tpl
import numpy as np
import operator
from collections import Counter
import csv
```

In [45]:
```
def load_iris_data(): #Cette fonction lit les données des iris à partir d'un fichier CSV
    with open('Iris.csv', 'r') as f:
        reader = csv.reader(f, delimiter=',')
        iris_data = list(reader)
    return iris_data[1:], iris_data[0] #renvoie deux listes : iris_data, qui contient les enregistrements de données,
                                       #et _, qui contient l'en-tête(columns) du fichier CSV.
```

In [46]:
```
iris_data, _ = load_iris_data()  # Charge les données de l'Iris
print('Ensemble de données des Iris de Fisher: \n', iris_data[:5])  # Affiche les données de l'Iris
```

Ensemble de données des Iris de Fisher:
[['5.1', '3.5', '1.4', '0.2', 'Iris-setosa'], ['4.9', '3', '1.4', '0.2', 'Iris-setosa'], ['4.7', '3.2', '1.3', '0.2', 'Iris-setosa'], ['4.6', '3.1', '1.5', '0.2', 'Iris-setosa'], ['5', '3.6', '1.4', '0.2', 'Iris-setosa']]

Fonction *split_iris_data(iris_data, axis, value)* :

Cette fonction prend en entrée les données des iris, un axe (indice de colonne) et une valeur. Elle divise les données des iris en fonction de l'axe et de la valeur donnés, en excluant l'axe utilisé. Elle renvoie les données des iris divisées.

In [47]:
```
def split_iris_data(iris_data, axis, value):
    ret_iris_data = []
    for featVec in iris_data:
        if featVec[axis] == value:
            reducedFeatVec = featVec[:axis] + featVec[axis+1:]  # Exclut l'axe utilisé
            ret_iris_data.append(reducedFeatVec)
    return ret_iris_data
```

Fonction *calculate_shannon_entropy(iris_data)* :

Cette fonction calcule l'entropie de Shannon des données des iris. Elle utilise la formule de l'entropie : $\sum(-p * \log2(p))$, où p est la probabilité de chaque étiquette de classe dans les données. Elle renvoie l'entropie calculée.

In [48]:
```
def calculate_shannon_entropy(iris_data):
    num_entries = len(iris_data)
    if num_entries == 0:
        return 0 # Returning 0 if no data is present
    # Calculating label counts using Counter class
    label_counts = Counter([featVec[-1] for featVec in iris_data if len(featVec) > 0])
    shannon_entropy = 0.0
    for key in label_counts:
        prob = float(label_counts[key]) / num_entries
        shannon_entropy -= prob * math.log(prob, 2)  # Calculating Shannon Entropy formula
    return shannon_entropy
```

In [49]:
```
# Calcul de l'entropie de Shannon de l'ensemble de données initial
entropy = calculate_shannon_entropy(iris_data)
print("Shannon Entropy:", entropy)
```

Shannon Entropy: 1.584962500721156

Fonction *calculate_gain_ratio(iris_data, attribute_index)* :

Cette fonction calcule le ratio de gain d'un attribut donné dans les données des iris. Elle utilise le concept de gain d'information et d'information de division. Le gain d'information mesure la réduction de l'entropie après la division des données sur un attribut particulier, et l'information de division mesure l'information potentielle générée par la division des données sur cet attribut. Le ratio de gain est le rapport entre le gain d'information et l'information de division. Elle renvoie le ratio de gain calculé.

In [50]:
```
def calculate_gain_ratio(iris_data, attribute_index):
    base_entropy = calculate_shannon_entropy(iris_data)  # Getting base entropy
    num_entries = len(iris_data)
    feat_list = [example[attribute_index] for example in iris_data]  # Extracting feature list
    unique_vals = set(feat_list)  # Finding unique values

    new_entropy = 0.0
    split_info = 0.0

    for value in unique_vals:
        sub_iris_data = [example for example in iris_data if example[attribute_index] == value] # Subsetting data
        prob = len(sub_iris_data) / num_entries
        split_info -= prob * math.log(prob, 2)
        new_entropy += prob * calculate_shannon_entropy(sub_iris_data)

    if split_info == 0:
        return 0  # Avoiding division by zero

    information_gain = base_entropy - new_entropy
    gain_ratio = information_gain / split_info
    return gain_ratio  # Returning calculated Gain Ratio
```

In [51]:
```
# Indices des caractéristiques dans le jeu de données Iris
num_features = len(iris_data[0]) - 1 # -1 car la dernière colonne est la classe
print("Ces valeurs de gain ratio indiquent l'utilité de chaque attribut pour la classification dans votre ensemble de données Iris. Plus le gain ratio est élevé, plus l'attribut est informatif pour la classification. Voici les résultats :")
for i in range(num_features):
    gain_ratio_result = calculate_gain_ratio(iris_data, i)
    print(f"Gain Ratio for attribute at index {i}: {gain_ratio_result}")
```

Ces valeurs de gain ratio indiquent l'utilité de chaque attribut pour la classification dans votre ensemble de données Iris. Plus le gain ratio est élevé, plus l'attribut est informatif pour la classification. A partir des résultats :

Gain Ratio for attribute at index 0: 0.1818611222143824602
Gain Ratio for attribute at index 1: 0.12734469703221557
Gain Ratio for attribute at index 2: 0.28731933777918335
Gain Ratio for attribute at index 3: 0.3531768034035186

Alors Le meilleur index à choisir pour la classification sera l'index 3 (la largeur du pétale), suivi de près par l'index 2 (la longueur du pétale). Ces attributs semblent être les plus pertinents pour séparer les différentes classes d'Iris dans votre ensemble de données.

Fonction *choose_best_feature_to_split(iris_data)* :

Cette fonction choisit le meilleur attribut pour diviser les données des iris en se basant sur le ratio de gain. Elle itère sur tous les attributs des données et calcule le ratio de gain pour chaque attribut. Elle renvoie l'indice du meilleur attribut.

In [52]:
```
def choose_best_feature_to_split(iris_data):
    num_features = len(iris_data[0]) - 1
    best_gain_ratio = 0.0
    best_feature = -1

    for i in range(num_features):
        gain_ratio = calculate_gain_ratio(iris_data, i)
        if gain_ratio > best_gain_ratio:
            best_gain_ratio = gain_ratio
            best_feature = i

    return best_feature
```

Fonction *majority_count(class_list)* :

Cette fonction compte la classe majoritaire dans une liste d'étiquettes de classe. Elle utilise un dictionnaire pour compter les occurrences de chaque étiquette de classe et renvoie l'étiquette de classe avec le compte le plus élevé. **createTree()** utilise cette fonction pour attribuer une classe à une feuille de l'arbre lorsque tous les exemples appartiennent à la même classe.

In [53]:
```
def majority_count(class_list):
    class_count = {}
    for vote in class_list:
        if vote not in class_count.keys():
            class_count[vote] = 0
        class_count[vote] += 1
    sorted_class_count = sorted(class_count.items(), key=lambda item: item[1], reverse=True)
    return sorted_class_count[0][0] # Returning the most frequent class
```

Fonction *createTree(iris_data, labels)* :

En combinant ces fonctions, **createTree()** construit récursivement l'arbre de décision en choisissant les meilleurs attributs pour diviser les données, en créant des branches dans l'arbre et en attribuant des classes aux feuilles. Elle s'appuie sur les autres fonctions pour effectuer les calculs nécessaires et prendre les décisions appropriées lors de la construction de l'arbre.

Ainsi, la fonction **createTree()** dépend des autres fonctions pour accomplir différentes tâches clés dans le processus de création de l'arbre de décision à partir des données des iris.

In [54]:
```
def createTree(iris_data, labels):
    class_list = [example[-1] for example in iris_data] # Extracting class labels
    if class_list.count(class_list[0]) == len(class_list): # Checking if all classes are the same
        return class_list[0] # Returning the class if all are the same
    if len(iris_data[0]) == 1:
        return majority_count(class_list)  # Returning the majority class if no more features to split
    best_feat = choose_best_feature_to_split(iris_data) # Choosing the best feature to split
    best_feat_label = labels[best_feat]
    print(best_feat_label)
    myTree = {best_feat_label: {}}  # Creating a tree structure
    del(labels[best_feat])  # Removing the used label

    feat_values = [example[best_feat] for example in iris_data]
    unique_vals = set(feat_values)
    for value in unique_vals:
        sub_labels = labels.copy()
        myTree[best_feat_label][value] = createTree(split_iris_data(iris_data, best_feat, value), sub_labels)
    return myTree  # Returning the created decision tree
```

Construction de l'Arbre de Décision

In [55]:
```
iris_data, labels = load_iris_data()
tr = createTree(iris_data, labels)
print(tr)
```

```
PetalWidthCm
PetalLengthCm
SepalWidthCm
SepalLengthCm
PetalLengthCm
PetalLengthCm
SepalLengthCm
{'PetalWidthCm': {'1': 'Iris-versicolor', '2.2': 'Iris-virginica', '0.5': 'Iris-setosa', '1.4': {'PetalLengthCm': {'3.9': 'Iris-versicolor', '4.6': 'Iris-versicolor', '5.6': 'Iris-virginica', '4.8': 'Iris-versicolor', '4.7': 'Iris-versicolor', '4.4': 'Iris-versicolor'}}, '1.1': 'Iris-versicolor', '1.9': 'Iris-virginica', '0.1': 'Iris-setosa', '1.8': {'SepalWidthCm': {'3.1': 'Iris-virginica', '2.9': 'Iris-virginica', '3.2': {'SepalLengthCm': {'5.9': 'Iris-versicolor', '7.2': 'Iris-virginica'}}, '2.5': 'Iris-virginica', '2.8': 'Iris-virginica', '2.7': 'Iris-virginica', '3': 'Iris-virginica'}}, '0.3': 'Iris-setosa', '2.4': 'Iris-virginica', '0.4': 'Iris-setosa', '1.6': {'SepalLengthCm': {'6': 'Iris-versicolor', '7.2': 'Iris-virginica', '6.3': 'Iris-versicolor', '5': 'Iris-versicolor', '2.3': 'Iris-virginica', '1.5': {'PetalLengthCm': {'4.6': 'Iris-versicolor', '4.2': 'Iris-versicolor', '4.5': 'Iris-virginica', '4.7': 'Iris-versicolor', '5.1': 'Iris-virginica', '4.9': 'Iris-versicolor'}}, '0.6': 'Iris-setosa', '0.2': 'Iris-setosa', '1.2': 'Iris-versicolor', '2.5': 'Iris-virginica', '1.7': {'SepalLengthCm': {'4.9': 'Iris-virginica', '6.7': 'Iris-versicolor'}}, '2': 'Iris-virginica', '2.1': 'Iris-virginica'}}
```

L'utilisation de fichier 'treePlotter.py' pour dessiner l'arbre

In [56]:
```
import matplotlib.pyplot as plt

decisionNode = dict(boxstyle="sawtooth", fc="1")
leafNode = dict(boxstyle="round4", fc="1")
arrow_args = dict(arrowstyle="<-")

def getNumLeafs(myTree):
    numLeafs = 0
    firstStr = list(myTree.keys())[0] # error: firstStr = myTree.keys()[0]
    secondDict = myTree[firstStr]
    for key in secondDict.keys():
        if type(secondDict[key]).__name__=='dict':#test to see if the nodes are dictonaires, if not they are leaf nodes
            numLeafs += getNumLeafs(secondDict[key])
        else:   numLeafs +=1
    return numLeafs

def getTreeDepth(myTree):
    maxDepth = 0
    firstStr = list(myTree.keys())[0] # error : firstStr = myTree.keys()[0]
    secondDict = myTree[firstStr]
    for key in secondDict.keys():
        if type(secondDict[key]).__name__=='dict':#test to see if the nodes are dictonaires, if not they are leaf nodes
            thisDepth = 1 + getTreeDepth(secondDict[key])
        else:   thisDepth = 1
        if thisDepth > maxDepth: maxDepth = thisDepth
    return maxDepth

def plotNode(nodeTxt, centerPt, parentPt, nodeType):
    createPlot.axi.annotate(nodeTxt, xy=parentPt,  xycoords='axes fraction',
             xytext=centerPt, textcoords='axes fraction',
             va="center", ha="center", bbox=nodeType, arrowprops=arrow_args )

def plotMidText(cntrPt, parentPt, txtString):
    xMid = (parentPt[0]-cntrPt[0])/2.0 + cntrPt[0]
    yMid = (parentPt[1]-cntrPt[1])/2.0 + cntrPt[1]
    createPlot.axi.text(xMid, yMid, txtString, va="center", ha="center", rotation=30)

def plotTree(myTree, parentPt, nodeTxt):#if the first key tells you what feat was split on
    numLeafs = getNumLeafs(myTree)  #this determines the x width of this tree
    depth = getTreeDepth(myTree)
    firstStr = list(myTree.keys())[0]     #the text label for this node should be this # error : firstStr = myTree.keys()[0]
    cntrPt = (plotTree.xOff + (1.0 + float(numLeafs))/2.0/plotTree.totalW, plotTree.yOff)
    plotMidText(cntrPt, parentPt, nodeTxt)
    plotNode(firstStr, cntrPt, parentPt, decisionNode)
    secondDict = myTree[firstStr]
    plotTree.yOff = plotTree.yOff - 1.0/plotTree.totalD
    for key in secondDict.keys():
        if type(secondDict[key]).__name__=='dict':#test to see if the nodes are dictonaires, if not they are leaf nodes
            plotTree(secondDict[key],cntrPt,str(key))        #recursion
        else:   #it's a leaf node print the leaf node
            plotTree.xOff = plotTree.xOff + 1.0/plotTree.totalW
            plotNode(secondDict[key], (plotTree.xOff, plotTree.yOff), cntrPt, leafNode)
            plotMidText((plotTree.xOff, plotTree.yOff), cntrPt, str(key))
    plotTree.yOff = plotTree.yOff + 1.0/plotTree.totalD
#if you do get a dictonary you know it's a tree, and the first element will be another dict

def createPlot(inTree):
    fig = plt.figure(1, facecolor='white')
    fig.clf()
    axprops = dict(xticks=[], yticks=[])
    createPlot.axi = plt.subplot(111, frameon=False, **axprops)    #no ticks
    #createPlot.axi = plt.subplot(111, frameon=False) #ticks for demo puropses
    plotTree.totalW = float(getNumLeafs(inTree))
    plotTree.totalD = float(getTreeDepth(inTree))
    plotTree.xOff = -0.5/plotTree.totalW; plotTree.yOff = 1.0;
    plotTree(inTree, (0.5,1.0), '')
    plt.show()

def grabTree(filename):
    import ast
    fr = open(filename)
    outputTree = fr.read()
    fr.close()
    return ast.literal_eval(outputTree)
```

Visualisation de l'arbre de décision

In [57]:
```
createPlot(tr)
```