Text

Description automatically generated

When running the finding the accuracy of the tree trained with the function importance with the condition set as “random”, the program adds the result to a list, which is done a total of 10.000 times and then averaged we find that if reaches an accuracy of 0.764. The “calculating importance from information gain (see Section 19.3.3. on page 679 in the book)” gives an accuracy of 0.928. As “information\_gain” always trains and returns the same set, its precision does not change and it is thus unnecessary to average.

It is obvious that following the basic maximum utility principle we would find that “information\_gain” with a better result on average is the better choice. The reason for its improved score over a random selection is because it is better “fitted”. As “information\_gain” consistently selects from the lowest uncertainty. The structure of the tree generated is such that it maximizes the decisions it can take with the least amount of uncertainty which is what makes it “fit” better with the presented pattern (in this case “test”).

Code:

import numpy as np

from pathlib import Path

import random

import math

class Node:

    """ Node class used to build the decision tree"""

    def \_\_init\_\_(self):

        self.children = {}

        self.parent = None

        self.attribute = None

        self.value = None

    def classify(self, example):

        if self.value is not None:

            return self.value

        her = example[self.attribute]

        try:

            return self.children[her].classify(example)

        except KeyError:

            if her == 1:

                her = 2

            elif her == 2:

                her = 1

            return self.children[her].classify(example)

def plurality\_value(examples):

    """Implements the PLURALITY-VALUE (Figure 19.5)"""

    labels = examples[:, -1]

    value, count = 0, 0

    for label in np.unique(labels):

        label\_count = np.count\_nonzero(labels == label)

        if label\_count > count:

            value = label

            count = label\_count

    return value

def calc(prob):

    if(prob != 1 and prob!=0):

        return -(prob\*math.log2(prob)+(1-prob)\*math.log2(1-prob))

    else:

        return 0

def importance(attributes, examples, measure):

    """

    This function should compute the importance of each attribute and choose the one with highest importance,

    A ← argmax a ∈ attributes IMPORTANCE (a, examples) (Figure 19.5)

    Parameters:

        attributes (np.ndarray): The set of attributes from which the attribute with highest importance is to be chosen

        examples (np.ndarray): The set of examples to calculate attribute importance from

        measure (str): Measure is either "random" for calculating random importance, or "information\_gain" for

caulculating importance from information gain (see Section 19.3.3. on page 679 in the book)

    Returns:

        (int): The index of the attribute chosen as the test

    """

    # TODO implement the importance function for both measure = "random" and measure = "information\_gain"

    if(measure == "random"):

        return attributes[random.randint(0, len(attributes)-1   )]

    else:

        liste = []

        for i in attributes:

            temp\_list = []

            for j in range(len(examples)):

                temp\_list.append(examples[j][i])

            tot = calc(2-(np.sum(temp\_list)/len(temp\_list)))

            liste.append([tot, i])

        liste = sorted(liste,key=lambda x: x[0])

        return liste[0][1]

def learn\_decision\_tree(examples, attributes, parent\_examples, parent, branch\_value, measure):

    """

    This is the decision tree learning algorithm. The pseudocode for the algorithm can be

    found in Figure 19.5 on Page 678 in the book.

    Parameters:

        examples (np.ndarray): The set data examples to consider at the current node

        attributes (np.ndarray): The set of attributes that can be chosen as the test at the current node

        parent\_examples (np.ndarray): The set of examples that were used in constructing the current node’s parent.

                                        If at the root of the tree, parent\_examples = None

        parent (Node): The parent node of the current node. If at the root of the tree, parent = None

        branch\_value (int): The attribute value corresponding to reaching the current node from its parent.

                        If at the root of the tree, branch\_value = None

        measure (str): The measure to use for the Importance-function. measure is either "random" or "information\_gain"

    Returns:

        (Node): The subtree with the current node as its root

    """

    #print("att:", attributes)

    # Creates a node and links the node to its parent if the parent exists

    node = Node()

    if parent is not None:

        parent.children[branch\_value] = node

        node.parent = parent

    # TODO implement the steps of the pseudocode in Figure 19.5 on page 678

    if len(examples) == 0:

        node.value = plurality\_value(parent\_examples)

    # If all examples have the same classification, return the classification

    elif np.unique(examples[:, -1]).size == 1:

        node.value = examples[0, -1]

    # If attributes is empty, return the plurality value of examples

    elif len(attributes) == 0:

        node.value = plurality\_value(examples)

    else:

        # Choose the attribute with highest importance

        A = importance(attributes, examples, measure)#Value of attributes

        B = np.where(attributes == A)[0][0] #Index of value of attributes

        node.attribute = attributes[B]

        # Create a new decision tree with root test A

        for v in np.unique(examples[:, A]):

            exs = examples[examples[:, A] == v]

            subtree = learn\_decision\_tree(exs, np.delete(attributes, B), examples, node, v, measure)

            node.children[v] = subtree

    return node

def accuracy(tree, examples):

    """ Calculates accuracy of tree on examples """

    correct = 0

    for example in examples:

        pred = tree.classify(example[:-1])

        correct += pred == example[-1]

    return correct / examples.shape[0]

def load\_data():

    """ Load the data for the assignment,

    Assumes that the data files is in the same folder as the script"""

    with (Path.cwd() / "train.csv").open("r") as f:

        train = np.genfromtxt(f, delimiter=",", dtype=int)

    with (Path.cwd() / "test.csv").open("r") as f:

        test = np.genfromtxt(f, delimiter=",", dtype=int)

    return train, test

if \_\_name\_\_ == '\_\_main\_\_':

    train, test = load\_data()

    liste = []

    for i in ["information\_gain", "random"]:

        measure = i

        if(i=="random"):

            for x in range(10000):

                tree = learn\_decision\_tree(examples=train,

                                attributes=np.arange(0, train.shape[1] - 1, 1, dtype=int),

                                parent\_examples=None,

                                parent=None,

                                branch\_value=None,

                                measure=measure)

                liste.append(accuracy(tree, test))

            print(measure)

            print(f"Training Accuracy {accuracy(tree, train)}")

            print(f"Test Accuracy Avg {np.sum(liste)/len(liste)}")

            print("")

        else:

            tree = learn\_decision\_tree(examples=train,

                            attributes=np.arange(0, train.shape[1] - 1, 1, dtype=int),

                            parent\_examples=None,

                            parent=None,

                            branch\_value=None,

                            measure=measure)

            print(measure)

            print(f"Training Accuracy {accuracy(tree, train)}")

            print(f"Test Accuracy     {accuracy(tree, test)}")

            print("")