```
This is a class for visualizing the Decision Tree from the given datasets.

    __init__(self, name='root', classifier = None):
        Constructor for the class Tree.
        The default name is 'root' if nothing else is given,
        and a classifier is 'None' by default.

        Every object of class Tree has a name and a classifier.
        They also have a list that holds on other object Tree,
            (Empty if the object Tree has no children)
        and a data variable which holds on the classifier data
            (E.g cold --> yes {'yes': 2}, where {'yes': 2} is the data)

    __repr__(self):
        Returns

    set_data(self, data):
        Set a data to an object

    set_name(self, name):
        Set a name for an object (attribute)

    set_classifier(self, classifier):
        Set a classifier for an object (leaf)

    add_child(self, tree):
        Add a tree object (child) to another tree in a list

    remove_child(self, child):
        Remove a tree object of a tree

    clear_children(self, tree):
        Remove all children of a tree

    isLeaf(self):
        Check if the tree object is a leaf (No children)

    show(self, ind = 0):
        Method for printing the tree. Parameter 'ind' is indent to make the tree
        more readable
```

In [63]:

```
class Tree:

    def __init__(self, name='root', classifier = None):
        self.children = []
        self.name = name
        self.classifier = classifier
        self.data = {}

    def set_data(self, data):
        self.data = data

    def set_name(self, name):
        self.name = name

    def set_classifier(self, classifier):
        self.classifier = classifier

    def __repr__(self):
        return str(self.name)

    def add_child(self, tree):
        self.children.append(tree)

    def remove_child(self, child):
        self.children.remove(child)

    def clear_children(self, tree):
        self.children = []

    def isLeaf(self):
        return self.children.count(None) == len(self.children)

    def show(self, ind = 0):
        indent = ''
        for i in range(ind):
            indent = indent + ' | '

        if self.isLeaf():
            print(indent, self.name, '-->', self.classifier, self.data)
        else:
            print(indent, self.name, 'is a parent with children:',self.children , self.data

            for child in self.children:
                child.show(ind + 1)
```

```
This function learns a decision tree classifier from data X and y.
The learn()-function takes in four parameters:
    X:
        The dataset which includes all the data labels

    y:
        Data which holds the classification
        (not included in data X)

    impurity_measure:
        Chooses a formula to measure the information gain.
        Can choose between 'entropy' or 'gini'.
        By default, the learning function will use 'entropy' as
        an impurity measure for information gain.

    pruning:
        This parameter is used to prune the decision tree created by
        the learning function. Pruning will remove branches that
        causes overfitting.
        Can choose 'True' or 'False' to prune the tree or not.
        By default, the pruning is set to 'False'.

    return Tree:
        The function returns a decision tree with children and leaves.
```

In [64]:

```
import numpy as np
from sklearn.model_selection import train_test_split
from Tree import Tree
import operator

def learn(X, y, impurity_measure = 'entropy', pruning = False):
    if len(y) == 0:
        return 0

    (X,y) = update_data(X, y)
    print('Impurity_measure:', impurity_measure)

    #X = X_train, y = y_train
    #if pruning:
    #    tree = makeTree(X_train, y_train, impurity_measure)

    return makeTree(X, y, impurity_measure)
```

```
Updates the datasets. If a question mark '?' appears in a row in the data X, remove the
whole row. Also remove the corresponding classifier in data y.
    X:
        Dataset X (attributes)
    y:
        Dataset y (classifier)

    return (newX, newy):
        Returns a new X and y without rows containing a question mark '?'
```

In [65]:

```
def update_data(X, y):
    newX = []
    newy = []
    for row in range(len(y)):
        if '?' not in X[row]:
            newX.append(X[row])
            newy.append(y[row])
    return (newX, newy)
```

```
This function is used to predict class label of some new data point x.
Takes in two parameters:
    x:
        Some data point in form of a list which is used to predict
        the classifier.

    tree:
        The decision tree where the predict is used on.

    return tree.classifier:
        The function will return a classifier that matches the
        data point x.
```

In [66]:

```
def predict(x, tree):
    if tree.isLeaf():
        return tree.classifier
    else:
        for child in tree.children:
            child.name = child.name.strip()
            if child.name in x:
                list(x).remove(child.name)
                return predict(x, child)
```

```
Function makeTree() creates the decision tree with data X and y with an impurity measure
which is used to find the best feature to split the decision tree.
The function takes three parameters:
    X:
        The dataset X to build the decision tree, also known as
        attributes. Usually the parent of a classifier or another
        attribute.
    y:
        Data which holds the classifier. This is the leaves in the
        tree.

    impurity_measure:
        Used to measure information gain with a given formula.
```

```
calculateInformationGain() returns a index where the best feature occurs.
```

In [67]:

```
def makeTree(X, y, impurity_measure):
    if is_pure(y):
        return Tree(classifier = y[0])

    elif len(np.transpose(X)) == 0: # no features left
        mcl = most_common_label(y)
        return Tree(classifier = mcl)

    else:
        tree = Tree()
        index = calculateInformationGain(X, y, impurity_measure)

        for attribute_value, [splitted_X, splitted_y] in split(X, y, index).items():
            child = makeTree(splitted_X, splitted_y, impurity_measure)
            child.set_name(attribute_value)
            child.set_data(countLetters(splitted_y))
            tree.add_child(child)

    return tree
```

This function is used to split after finding the best index in data X. X: Data X (may also be a new X, after getting splitted)

```
    y:
        Data y (may be new y, after getting splitted)

    index:
        Index after finding the best feature (index of a column in X)

    return dict:
        Returns a dictionary with attribute and a corresponding classifier.
```

In [3]:

```
def split(X, y, index):
    dict = {}
    for i in range(len(y)):
        if X[i][index] in dict:
            dict[X[i][index]][0].append(X[i][:index] + X[i][index+1:])
            dict[X[i][index]][1].append(y[i])
        else:
            dict[X[i][index]] = [[X[i][:index] + X[i][index+1:]], [y[i]]]
    return dict
```

Function is_pure() checks if a dataset only contains the value y: A dataset

```
return len(set(y)) == 1:
    Returns either True or False depending on the data
```

Function most_common_label() see which attribute is the most common in the dataset y: A dataset

```
return sortedClassifier[0][0]:
    Returns the most common label (attribute) in the dataset
```

In [4]:

```
def is_pure(y):
    return len(set(y)) == 1

def most_common_label(y):
    dict = {}
    for classifier in y:
        if classifier not in dict.keys():
            dict[classifier] = 0
        dict[classifier] += 1
    sortedClassifier = sorted(dict.items(), key = operator.itemgetter(1), reverse=False)
    return sortedClassifier[0][0]
```

```
calculateInformationGain() calculates the information gain with datasets and a given
type of measure.
    X:
        Dataset X

    y:
        Dataset y

    impurity_measure:
        Choice of impurity measure
```

In [5]:

```
def calculateInformationGain(X, y, impurity_measure):
    ig_list = []
    #A dictionary with function mapped to the keys
    impurity_func = {'entropy': calc_entropy, 'gini': calc_gini}
    #measure is a function that matches the impurity_measure
    measure = impurity_func.get(impurity_measure)

    for row in np.transpose(X):
        #Put the probabilities of the values in a list
        probabilities = [counter/len(y) for counter in countLetters(y).values()]

        #Here we calculate the probability
        information_measure = measure(probabilities)
        ig = information_measure
        for attribute_value, occurrence_dict in zip_xy_class(row, y).items():
            s = sum(occurrence_dict.values())
            X_probabilities = [counter/s for counter in occurrence_dict.values()]
            attribute_measure = measure(X_probabilities)
            weight = s/len(y)

            gain -= weight * attribute_measure

        ig_list.append(gain)

    index = np.argmax(ig_list)
    return index
```

```
calc_entropy() takes a list of probabilities, calculates the entropy of each value and
sum them together.
    listprob:
        A list of probabilities

    return entropy:
        Returns the entropy of the calculated values

calc_gini() also takes a list of probabilities, but calculates the gini instead
    return gini:
        Returns the entropy of the calculated values
```

In [6]:

```
def calc_entropy(listprob):
    entropy = 0
    for prob in listprob:
        if prob != 0:
            entropy += -prob * np.log2(prob)
    return entropy

def calc_gini(listprob):
    gini = 0
    for prob in listprob:
        if prob != 0:
            gini += prob * (1 - prob)
    return gini
```

```
Helper function that counts occurrence of an element in a list.
The dictionary holds on an element and a value that is a counter of occurrenes of that
element.
Functions takes an array:
    array:
        An array with elements

    return dict:
        Returns a dictionary with a key (element) and a value (occurrence of an element
in the list)
```

In [7]:

```
def countLetters(array):
    dict = {}
    for i in array:
        if i in dict:
            dict[i] += 1
        else:
            dict[i] = 1
    return dict
```

```
Helper function that connects the data X and y together. Also count how many occurrences
of a classifier an attribute has.
E.g {' sunny': {'no': 1, 'yes': 1}, ' cloudy': {'yes': 2}}, where attribute sunny has
two classification and occurrences of the classification for the attribute.
This functions takes:
    X:
        Dataset X, attributes
    y:
        Dataset y, classifications

    return dict:
        Returns a dictionary for an attribute in X with the corresponding
        classifiers and the number of occurrences
```

In [8]:

```
def zip_xy_class(X, y):
    dict = {}
    for attribute, classifier in list(zip(X, y)):
        if attribute in dict:
            if classifier in dict[attribute]:
                dict[attribute][classifier] += 1
            else:
                dict[attribute][classifier] = 1
        else:
            dict[attribute] = {classifier : 1 }
    return dict
```

In [1]:

```
import csv
import numpy as np
from ID3 import learn, makeTree, predict
from Tree import Tree

from sklearn import tree
from sklearn.model_selection import train_test_split
from sklearn.tree import DecisionTreeClassifier
from sklearn.preprocessing import OneHotEncoder, LabelEncoder

X = []
y = []

with open('agaricus-lepiota.data') as csv_file:
        reader = csv.reader(open("agaricus-lepiota.data", "rb"), delimiter=",")
        csv_reader = csv.reader(csv_file, delimiter=',')
        for row in csv_reader:
            y.append(row[0])
            X.append(row[1:])

X_train, X_test, y_train, y_test = train_test_split(X, y, test_size = 0.33, random_state =


#For my ID3 implementation:
myTree = learn(X_train, y_train, 'entropy', True)
myTree.show()

dict = {}
for row in range(len(X_test)):
    pred = predict(X[row], myTree)
    result = pred == y_test[row]
    if result not in dict:
        dict[result] = 1

    if (result):
        dict[result] += 1
    else:
        dict[result] += 1
print(dict)

#print(prune(X, y, tree))

#5. Implementation comparision
X_T = []
le = LabelEncoder()
for i in range(len(np.transpose(X_train))):
    X_T.append(le.fit_transform(np.transpose(X_train)[i]))
X_train = np.transpose(X_T)

X_T = []
for i in range(len(np.transpose(X_train))):
    X_T.append(le.fit_transform(np.transpose(X_test)[i]))
X_test = np.transpose(X_T)

y_train = le.fit_transform(y_train)
y_test = le.fit_transform(y_test)
```

```
dtc = tree.DecisionTreeClassifier(criterion = 'entropy')
dtc.fit(X_train, y_train)
dtc_predict = dtc.predict(X_test)

dict = {}
for i in dtc_predict:
    result = dtc_predict[i] == y_test[i]
    if result not in dict:
        dict[result] = 1

    if (result):
        dict[result] += 1
    else:
        dict[result] += 1

print(dict)
```

```
Impurity_measure: entropy
 branch is a parent with children: [n, c, f, l, a, p, m] {}
 |  n is a parent with children: [k, n, w, r] {'e': 1825, 'p': 59}
 |  |  k --> e {'e': 865}
 |  |  n --> e {'e': 896}
 |  |  w is a parent with children: [p, y, n, c, g, w] {'e': 64, 'p': 12}
 |  |  |  p --> e {'e': 4}
 |  |  |  y --> p {'p': 7}
 |  |  |  n --> e {'e': 31}
 |  |  |  c --> e {'e': 24}
 |  |  |  g --> e {'e': 5}
 |  |  |  w --> p {'p': 5}
 |  |  r --> p {'p': 47}
 |  c --> p {'p': 128}
 |  f --> p {'p': 1053}
 |  l --> e {'e': 253}
 |  a --> e {'e': 264}
 |  p --> p {'p': 162}
 |  m --> p {'p': 26}
{False: 1740, True: 943}
{True: 2682}
```

1. Implement the ID3 algorithm from scratch
The implementation is above

2. Gini Index
The learn() has entropy as a default impurity measure, but can be switched to gini with giving 'gini' as parameter

3. Pruning
Did not implement the pruning because of time remaining.

4. Classify edible and poisonous mushrooms
Entropy and gini gives the same result of splitting even with random state.

5. Implementation comparison:

X_train, X_test, y_train, y_test = train_test_split(X, y, test_size = 0.33, random_state = 42)

Splitted the data into training data and test data with test_size 0.33

For my implementation, my prediction got {False: 1740, True: 943} while the sklearn got {True: 2682}.

My predict() is not completely correct, because I had problems with removing an attribute from the data I want to predict. Did not find another solution for this because of the time remaining.