Decision Tree classifier

Today your task is to get familiar with decision tree classifier - simple, but powerful case of discrete math usage.

General idea

You are expected to write a quite simple, yet good core logic of decision tree classifier class. Additionaly, you need to test your results and write down a report on what you've done, which principles used and explain the general process.

Hopefully, you have already learned what is decision tree classifier and how it work. For better understanding, and in case if something is still unclear for you, here are some useful links on basics of DTC:

- https://towardsdatascience.com/decision-tree-from-scratch-in-python-46e99dfea775
- https://towardsdatascience.com/decision-tree-algorithm-in-python-from-scratch-8c43f0e40173
- https://machinelearningmastery.com/implement-decision-tree-algorithm-scratch-python/
- https://anderfernandez.com/en/blog/code-decision-tree-python-from-scratch/

Also, for those interested to learn more about machine learning and particulary Desicion Trees - here is a great course on Coursera (you may be interested in the whole course or just this particular week):

- https://www.coursera.org/learn/advanced-learning-algorithms/home/week/4

Dataset

You can use Iris dataset for this task. It is a very popular dataset for machine learning and data science. It contains 150 samples of 3 different species of Iris flowers (Iris setosa, Iris virginica and Iris versicolor). Four features were measured from each sample: the length and the width of the sepals and petals, in centimeters.

Read more on this:
https://scikit-learn.org/stable/auto_examples/datasets/plot_iris_dataset.html
https://en.wikipedia.org/wiki/Iris_flower_data_set

However, using more interesting and intricate datasets is much appreciated. You can use any dataset you want, but it should be a classification one. For example you can use breast cancer or wine datasets, which are also available in sklearn.datasets. Or you can use any other dataset you find interesting.

P.S. In case you are not sure if your dataset is suitable, feel free to ask assistants :).

```
# install the required packages

# !pip install pandas
# !pip install numpy
# !pip install matplotlib
# !pip install graphviz
# !pip install scikit-learn

import pandas as pd
import numpy as np
import matplotlib.pyplot as plt

# scikit-learn package
from sklearn.datasets import load_iris
from sklearn.model_selection import cross_val_score
from sklearn.tree import DecisionTreeClassifier
from sklearn import tree
from sklearn.model_selection import train_test_split

iris = load_iris()
dir(iris)
```

```
['DESCR',
 'data',
 'data_module',
 'feature_names',
 'filename',
 'frame',
 'target',
 'target_names']
```

```
iris.data.shape
```

```
(150, 4)
```

This means that we have 150 entries (samples, infos about a flower). The columns being: Sepal Length, Sepal Width, Petal Length and Petal Width(features). Let's look at first two entries:

```
iris.data[0:2]
```

```
array([[5.1, 3.5, 1.4, 0.2],
       [4.9, 3. , 1.4, 0.2]])
```

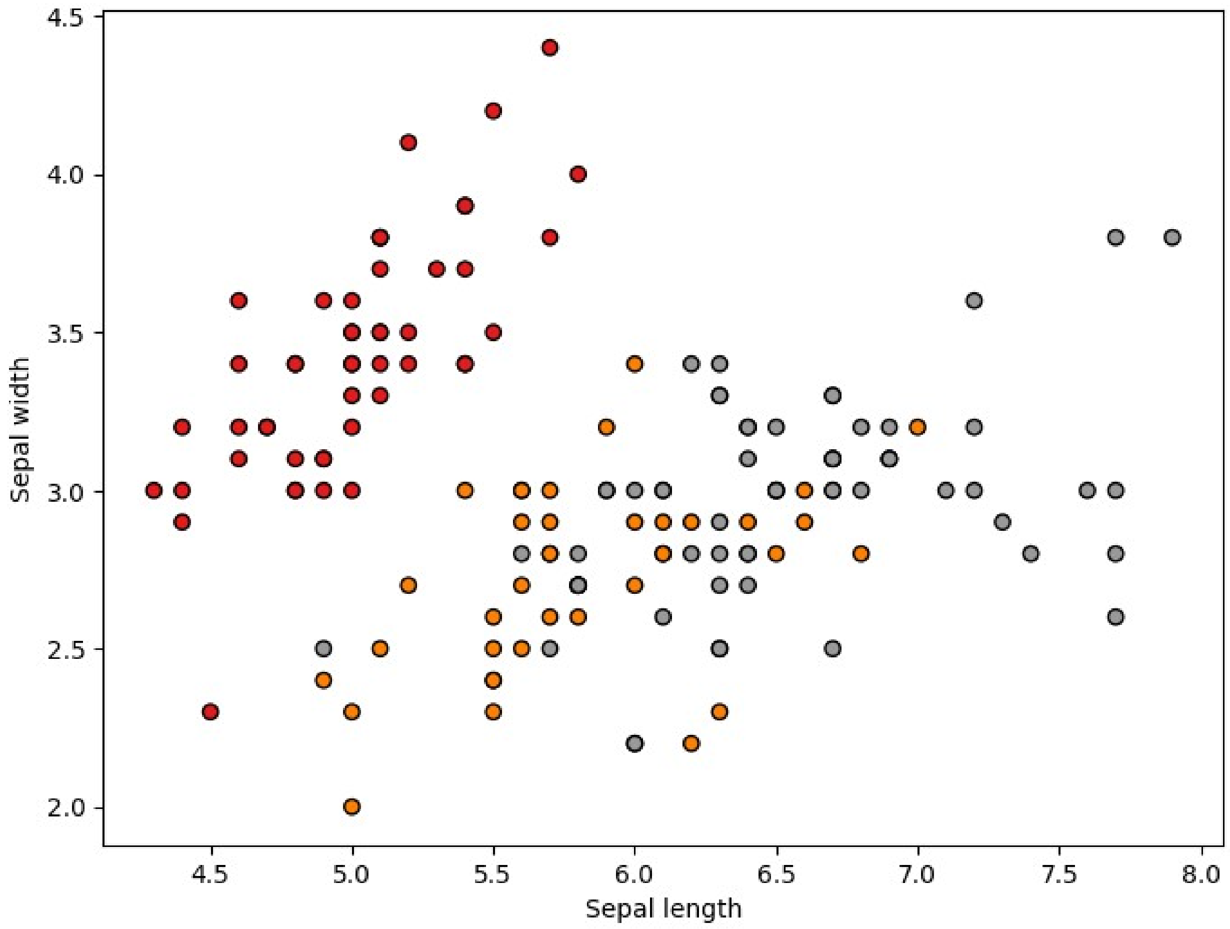
To undestand data little bit better, let's plot some features

```
X = iris.data[:, :2]  # we only take the first two features.
y = iris.target

plt.figure(2, figsize=(8, 6))
plt.clf()

# Plot the training points
plt.scatter(X[:, 0], X[:, 1], c=y, cmap=plt.cm.Set1, edgecolor="k")
plt.xlabel("Sepal length")
plt.ylabel("Sepal width")
```

```
Text(0, 0.5, 'Sepal width')
```



```
X = iris.data[:, :2]
print(X)
```

```
[[5.1 3.5]
 [4.9 3. ]
 [4.7 3.2]
 [4.6 3.1]
 [5.  3.6]
 [5.4 3.9]
 [4.6 3.4]
 [5.  3.4]
 [4.4 2.9]
 [4.9 3.1]
 [5.4 3.7]
 [4.8 3.4]
 [4.8 3. ]
 [4.3 3. ]
 [5.8 4. ]
 [5.7 4.4]
 [5.4 3.9]
 [5.1 3.5]
 [5.7 3.8]
 [5.1 3.8]
 [5.4 3.4]
 [5.1 3.7]
 [4.6 3.6]
 [5.1 3.3]
 [4.8 3.4]
 [5.  3. ]
 [5.  3.4]
 [5.2 3.5]
 [5.2 3.4]
 [4.7 3.2]
 [4.8 3.1]
 [5.4 3.4]
 [5.2 4.1]
 [5.5 4.2]
 [4.9 3.1]
 [5.  3.2]
 [5.5 3.5]
 [4.9 3.6]
 [4.4 3. ]
 [5.1 3.4]
 [5.  3.5]
 [4.5 2.3]
 [4.4 3.2]
 [5.  3.5]
 [5.1 3.8]
 [4.8 3. ]
 [5.1 3.8]
 [4.6 3.2]
 [5.3 3.7]
 [5.  3.3]
 [7.  3.2]
 [6.4 3.2]
 [6.9 3.1]
 [5.5 2.3]
 [6.5 2.8]
 [5.7 2.8]
 [6.3 3.3]
 [4.9 2.4]
 [6.6 2.9]
 [5.2 2.7]
 [5.  2. ]
 [5.9 3. ]
 [6.  2.2]
 [6.1 2.9]
 [5.6 2.9]
 [6.7 3.1]
 [5.6 3. ]
 [5.8 2.7]
 [6.2 2.2]
 [5.6 2.5]
 [5.9 3.2]
 [6.1 2.8]
 [6.3 2.5]
 [6.1 2.8]
 [6.4 2.9]
 [6.6 3. ]
 [6.8 2.8]
 [6.7 3. ]
 [6.  2.9]
 [5.7 2.6]
 [5.5 2.4]
 [5.5 2.4]
 [5.8 2.7]
 [6.  2.7]
 [5.4 3. ]
 [6.  3.4]
 [6.7 3.1]
 [6.3 2.3]
 [5.6 3. ]
 [5.5 2.5]
 [5.5 2.6]
 [6.1 3. ]
 [5.8 2.6]
 [5.  2.3]
 [5.6 2.7]
 [5.7 3. ]
 [5.7 2.9]
 [6.2 2.9]
 [5.1 2.5]
 [5.7 2.8]
 [6.3 3.3]
 [5.8 2.7]
 [7.1 3. ]
 [6.3 2.9]
 [6.5 3. ]
 [7.6 3. ]
 [4.9 2.5]
 [7.3 2.9]
 [6.7 2.5]
 [7.2 3.6]
 [6.5 3.2]
 [6.4 2.7]
 [6.8 3. ]
 [5.7 2.5]
 [5.8 2.8]
 [6.4 3.2]
 [6.5 3. ]
 [7.7 3.8]
 [7.7 2.6]
 [6.  2.2]
 [6.9 3.2]
 [5.6 2.8]
 [7.7 2.8]
 [6.3 2.7]
 [6.7 3.3]
 [7.2 3.2]
 [6.2 2.8]
 [6.1 3. ]
 [6.4 2.8]
 [7.2 3. ]
 [7.4 2.8]
 [7.9 3.8]
 [6.4 2.8]
 [6.3 2.8]
 [6.1 2.6]
 [7.7 3. ]
 [6.3 3.4]
 [6.4 3.1]
 [6.  3. ]
 [6.9 3.1]
 [6.7 3.1]
 [6.9 3.1]
 [5.8 2.7]
 [6.8 3.2]
 [6.7 3.3]
 [6.7 3. ]
 [6.3 2.5]
 [6.5 3. ]
 [6.2 3.4]
 [5.9 3. ]]
```

```
y = iris.target
print(y)
```

```
[0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0
 0 0 0 0 0 0 0 0 0 0 0 0 0 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
1 1
 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 2 2 2 2 2 2 2 2 2
2 2
 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2
2 2
 2 2]
```

From this we can clearly see, that even basing on those two parameters, we can clearly divide (classify) out data into several groups. For this, we will use decision tree classifier: https://scikit-learn.org/stable/modules/tree.html#tree

Example of usage

Decision Trees (DTs) are a non-parametric supervised learning method used for classification and regression. The goal is to create a model that predicts the value of a target variable by learning simple decision rules inferred from the data features. A tree can be seen as a piecewise constant approximation.

```
clf = DecisionTreeClassifier()

X, y = iris.data, iris.target
X.shape, y.shape
```

```
((150, 4), (150,))
```

Train / test split

We train our model using training dataset and evaluate its performance basing on the test dataset. Reason to use two separate datasets is that our model learns its parameters from data, thus test set allows us to check its possibilities on completely new data.

```
X, X_test, y, y_test = train_test_split(X, y, test_size= 0.20)
```

Model learning

It learns its parameters (where it should split data and for what threshold value) basing on the training dataset. It is done by minimizing some cost function (e.g. Gini impurity or entropy).

```
clf = clf.fit(X, y)
```

Visualization of produced tree

You do not need to understand this piece of code :)

```
import graphviz
dot_data = tree.export_graphviz(clf, out_file=None)
graph = graphviz.Source(dot_data)
graph.render("iris")
```

```
'iris.pdf'
```

```
dot_data = tree.export_graphviz(clf, out_file=None,
                     feature_names=iris.feature_names,
                     class_names=iris.target_names,
                     filled=True, rounded=True,
                     special_characters=True)
graph = graphviz.Source(dot_data)
graph
```

```
X_test.shape
```

```
(30, 4)
```

Prediction step

Now we can use our model to predict which type has a flower, basing on its parameters.

This is conducted basically via traversing the tree that you can see above.

```
predictions = clf.predict(X_test)
```

We can also measure the accuracy of our model

```
sum(predictions == y_test) / len(y_test)
```

```
0.9
```

To get clearer intuition about predicion, let's look at those X, that should be labeled to some flower

```
y_test

array([2, 2, 1, 1, 1, 1, 2, 2, 1, 1, 2, 0, 0, 0, 1, 0, 0, 1, 1, 1, 0,
2,
       1, 0, 2, 1, 0, 0, 0, 1])
```

Here you can traverse the tree above by yourself and make sure that prediction works

```
X_test[1]

array([6.3, 2.9, 5.6, 1.8])

clf.predict([X_test[1]])

array([2])
```

Finally, it is your turn to write such classifier by yourself!

Gini impurity

Decision trees use the concept of Gini impurity to describe how “pure” a node is. A node is pure (G = 0) if all its samples belong to the same class, while a node with many samples from many different classes will have a Gini closer to 1.

$$G=1-\sum_{k=1}^{n} p_k^2$$

For example, if a node contains five samples, with two belonging to the first class (first flower), two of class 2, one of class 3 and none of class 4, then

$$G=1-\left(\frac{2}{5}\right)^2-\left(\frac{2}{5}\right)^2-\left(\frac{1}{5}\right)^2=0.64$$

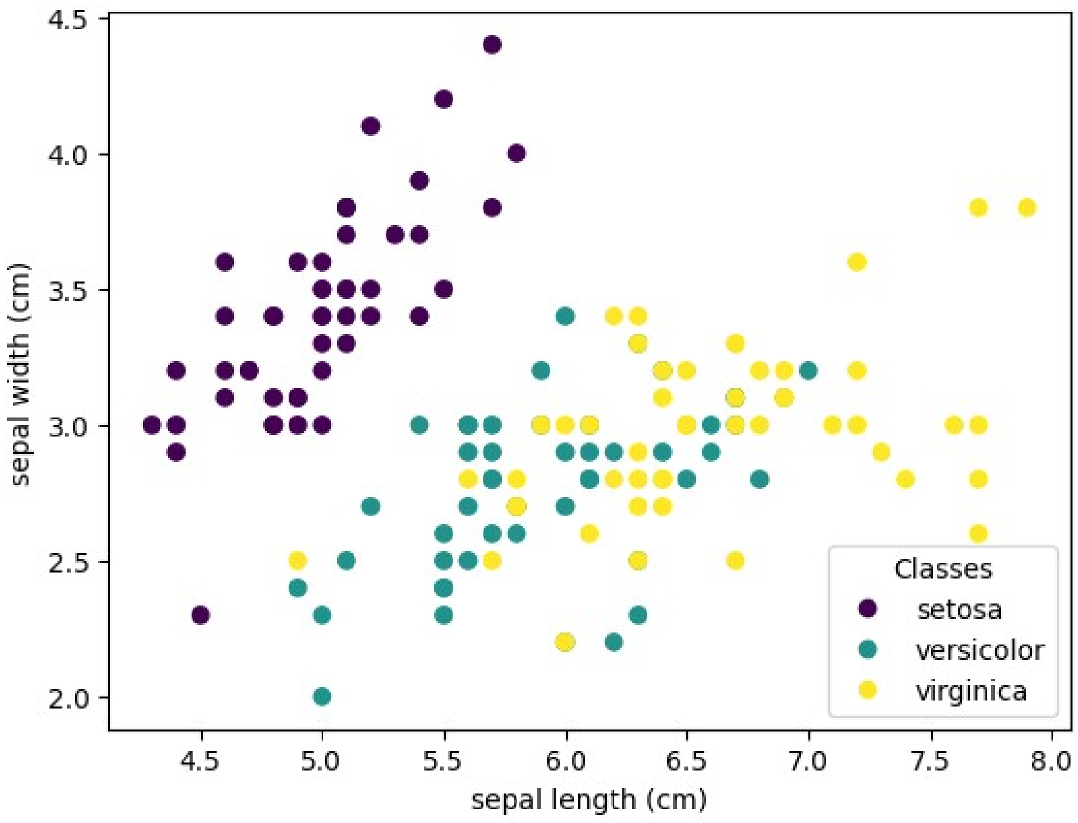
```
from sklearn import datasets

iris = datasets.load_iris()

import matplotlib.pyplot as plt

_, ax = plt.subplots()
scatter = ax.scatter(iris.data[:, 0], iris.data[:, 1], c=iris.target)
ax.set(xlabel=iris.feature_names[0], ylabel=iris.feature_names[1])
```

```
_ = ax.legend(
    scatter.legend_elements()[0], iris.target_names, loc="lower
right", title="Classes"
)
```



```
class Node:

    def __init__(self, X, y, gini):
        self.X = X
        self.y = y
        self.gini = gini
        self.feature_index = 0
        self.predicted_class=y
        self.threshold = 0
        self.left = None
        self.right = None

# Implement a decision tree classifier
class MyDecisionTreeClassifier:

    def __init__(self, max_depth, classes, predictions):
        '''
        This function will initialize the decision tree classifier
        with the maximum depth of the tree.
        '''
        self.max_depth = max_depth
        self.classes = classes
        self.predictions = predictions

    def gini(self, groups, classes):
        '''
        A Gini score gives an idea of how good a split is by how mixed
the

        classes are in the two groups created by the split.

        A perfect separation results in a Gini score of 0,
        whereas the worst case split that results in 50/50
        classes in each group result in a Gini score of 0.5
        (for a 2 class problem).
        '''
        n_instances = float(sum([len(group) for group in groups]))

        gini = 0.0
        for group in groups:
            size = float(len(group))

            if size == 0:
                continue
            score = 0.0

            for class_val in classes:
                p = [row[-1] for row in group].count(class_val) / size
                score += p * p

            gini += (1.0 - score) * (size / n_instances)
        return gini

    def split_data(self, X, y):
        '''
        This function will take the dataset and return the best split.
        '''
        m = y.size
        if m <= 1:
            return None, None

        num_parent = [np.sum(y == c) for c in range(self.n_classes_)]

        best_gini = 1.0 - sum((n / m) ** 2 for n in num_parent)
        best_idx, best_thr = None, None

        for idx in range(self.n_features_):
            thresholds, classes = zip(*sorted(zip(X[:, idx], y)))

            num_left = [0] * self.n_classes_
            num_right = num_parent.copy()
            for i in range(1, m):
                c = classes[i - 1]
                num_left[c] += 1
                num_right[c] -= 1
                gini_left = 1.0 - sum(
                    (num_left[x] / i) ** 2 for x in
range(self.n_classes_))
                gini_right = 1.0 - sum(
                    (num_right[x] / (m - i)) ** 2 for x in
range(self.n_classes_))

                gini = (i * gini_left + (m - i) * gini_right) / m

                if thresholds[i] == thresholds[i - 1]:
                    continue

                if gini < best_gini:
                    best_gini = gini
                    best_idx = idx
                    best_thr = (thresholds[i] + thresholds[i - 1]) / 2

        return best_idx, best_thr

    def build_tree(self, X, y, depth=0):
        '''
        Build a decision tree by recursively finding the best split.
        '''
        num_samples_per_class = [np.sum(y == i) for i in
range(self.n_classes_)]
        predicted_class = np.argmax(num_samples_per_class)
        gini = 1.0 - sum((n / y.size) ** 2 for n in
num_samples_per_class)

        node = Node(
            gini=gini,
            X=num_samples_per_class,
            y=predicted_class,)

        if depth < self.max_depth:
            idx, thr = self.split_data(X, y)
            if idx is not None:
                indices_left = X[:, idx] < thr
                X_left, y_left = X[indices_left], y[indices_left]
                X_right, y_right = X[~indices_left], y[~indices_left]
                node.feature_index = idx
                node.threshold = thr
                node.left = self.build_tree(X_left, y_left, depth + 1)
                node.right = self.build_tree(X_right, y_right, depth +
1)
```

```
        return node

    def fit(self, X, y):
        '''
        Wrapper for build tree / train
        '''
        self.n_classes_ = len(set(y))
        self.n_features_ = X.shape[1]
        self.tree_ = self.build_tree(X, y)

    def predict(self, X_test):
        return [self._predict(inputs) for inputs in X_test]

    def _predict(self, inputs):
        '''
        Predict class for a single sample.
        '''
        node = self.tree_
        while node.left:
            if inputs[node.feature_index] < node.threshold:
                node = node.left
            else:
                node = node.right
        return node.predicted_class

    def evaluate(self, X_test, y_test):
        '''
        This function will evaluate the model on
        the test set and return the accuracy of the model.
        '''
        predictions = self.predict(X_test)
        return sum(predictions == y_test) / len(y_test)

cif = MyDecisionTreeClassifier(max_depth = 10, classes = y,
predictions=10)
cif.fit(X,y)
print(cif.predict(X))
print(cif.evaluate(X,y))
```

```
[0, 0, 0, 2, 1, 1, 1, 0, 1, 1, 2, 0, 2, 2, 1, 1, 0, 2, 1, 1, 0, 1, 0,
0, 2, 0, 1, 2, 0, 0, 2, 0, 1, 0, 0, 0, 1, 2, 1, 1, 2, 2, 2, 0, 0, 0,
1, 2, 1, 1, 1, 2, 0, 2, 2, 0, 1, 1, 1, 1, 2, 2, 0, 1, 1, 1, 0, 2, 2,
1, 2, 1, 2, 2, 1, 0, 2, 2, 1, 2, 0, 2, 2, 0, 1, 1, 2, 0, 2, 0, 2, 2,
2, 0, 0, 0, 0, 2, 0, 2, 2, 1, 1, 1, 2, 1, 2, 0, 2, 0, 0, 2, 0, 2, 2,
0, 2, 0, 1, 0]
1.0
```

The in-built implementation of the decision tree classifier in scikit-learn is a powerful and widely-used tool for classification tasks. It provides a convenient and efficient way to build decision tree models and make predictions. The implementation allows for customization of various parameters, such as the maximum depth of the tree, the splitting criterion, and the minimum number of samples required to split a node.

Scikit-learn's decision tree classifier also offers useful features, including visualization of the decision tree, evaluation of the model's performance, and the ability to handle both categorical and numerical features. The library is well-documented, making it easy to understand and use.

On the other hand, our own implementation of the decision tree classifier (MyDecisionTreeClassifier) provides an opportunity to have more control over the model's behavior and customize it according to specific requirements. It allows you to define the maximum depth of the tree, the classes, and the predictions. This can be beneficial when you need to experiment with different settings or have specific constraints. It is worth noting that our implementation also has a high accuracy, close to the accuracy of the built-in version.

However, it's important to note that the in-built implementation has been thoroughly tested and optimized, ensuring reliable and accurate predictions.

In conclusion, both the in-built implementation of the decision tree classifier in scikit-learn and our own implementation have their advantages. The in-built implementation provides a solution for most classification tasks, while our own implementation offers more flexibility and control. The choice between the two depends on the specific requirements and constraints of your project.

For those who want to do it a little bit more complicated ;) (**optional**)

Consider also using some techniques to avoid overfitting, like pruning or setting a maximum depth for the tree. You can also try to implement some other metrics, to measure the quality of a split and overall performance. Also, you can try to implement some other algorithms, like proper CART, ID3 or C4.5. You can find more information about them here: https://scikit-learn.org/stable/modules/tree.html#tree