5. Decision Trees

- Decision trees are a non-parametric model used for both regression and classification tasks
- This notebook demonstrates how to use decision trees for classification
- Decision trees are constructed from only two elements - nodes and branches
- Decision trees are based on recursion
    - A function that calls itself until some exit condition is met
    - The algorithm is built by recursively evaluating different features and using the best split feature and criteria at each node
    - You'll learn how the best split feature and criteria are calculated in the math section
- There are multiple types of node a decision tree could have, and all are presented in the following figure:

- **Root node** - node at the top of the tree, contains a feature that best splits the data (a single feature that alone classifies the target variable most accurately)
- **Decision nodes** - nodes where the variables are evaluated. These nodes have arrows pointing to them and away from them
- **Leaf nodes** - final node at which the prediction is made

How to determine the root node

- Check how every input feature classifies the target variable independently
- If neither is 100% correct, we can consider them as *impure*
- The **Entropy** metric can be used to calculate impuritz
    - Formula discussed later
    - Values range from 0 (best) to 1 (worst)
- The variable with the lowest entropy (impurity) is used as a root node

Training process

- Determine the root node (discussed earlier)
- Calculate the **Information gain** for a single split
    - Formula discussed later
    - The higher the gain the better the split
- Do a greedy search
    - Go over all input feature and their unique values (thresholds)
    - Calculate information gain for every feature/threshold combination
    - Save the best split feature and best split threshold for every node
    - Build the tree recursively
    - Some stopping criteria should be applied when doing so
        - Think of it as an exit condition of a recursive function
        - This could be maximum depth, minimum samples at node...
    - If at the leaf node, return the prediction (most common value)
        - You'll know you're at a leaf node if a stopping criteria has been met or if the split is pure

Prediction process

- Recursively traverse the tree
- At each node check if the direction of the traversal (left or right), based on the input data
- When the leaf node is reached, the most common value is returned

Math behind

- Essentially, you only need to implement two formulas
    - Entropy
    - Information gain
- **Entropy**
    - Measures the purity of the split
    - Calculated at the node level
    - Ranges between 0 (pure) and 1 (impure)

- Example:

- Example in Python:

In [1]:

```
import numpy as np
from collections import Counter
```

In [2]:

```
def entropy(s):
    counts = np.bincount(s)
    percentages = counts / len(s)

    entropy = 0
    for pct in percentages:
        if pct > 0:
            entropy += pct * np.log2(pct)
    return -entropy
```

In [3]:

```
s = [0, 0, 0, 0, 0, 0, 0, 1, 1, 1]
print(f'Entropy: {np.round(entropy(s), 5)}')
```

```
Entropy: 0.88129
```

- **Information gain**:
    - Simply an average of all entropy based on a specific split
    - The higher the information gain, the better the decision split is

- Example:

- Example in Python:

In [4]:

```
def information_gain(parent, left_child, right_child):
    num_left = len(left_child) / len(parent)
    num_right = len(right_child) / len(parent)

    gain = entropy(parent) - (num_left * entropy(left_child) + num_right * entropy(right_child))
    return gain
```

In [5]:

```
parent = [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 1, 1, 1, 1, 1, 1, 1]
left_child = [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 1]
right_child = [0, 0, 0, 0, 1, 1, 1, 1]

print(f'Information gain: {np.round(information_gain(parent, left_child, right_child), 5)}')
```

Information gain: 0.18094

Recursion Crash Course

- A lot of decision trees implementation boils down to recursion
- Put simply, a recursive function is a function that calls itself
- Some "exit condition" is required if a function will call itself multiple times
    - It's common to write it at the top of the function
- Let's take a look at the simplest example - a recursive function that returns a factorial of an integer

In [16]:

```
def factorial(x):
    # Exit condition
    if x == 1:
        return 1
    return x * factorial(x - 1)

print(f'Factorial of 5 is {factorial(5)}')
```

Factorial of 5 is 120

- Recursion is needed in decision tree classifiers to build additional nodes of a tree until some condition (exit) is met

Implementation

- We'll need two classes
    - Node - implements a single node of a decision tree
    - DecisionTree - implements the algorithm
- The Node class is here to store the data about the feature, threshold, data going left and right, information gain, and the leaf node value
    - All are initially set to None
    - The leaf node value is available only for leaf nodes

In [6]:

```
class Node:
    '''
    Helper class which implements a single tree node.
    '''
    def __init__(self, feature=None, threshold=None, data_left=None, data_right=None, gain=None, value=None):
        self.feature = feature
        self.threshold = threshold
        self.data_left = data_left
        self.data_right = data_right
        self.gain = gain
        self.value = value
```

- The DecisionTree class contains a bunch of methods
- The constructor holds values for min_samples_split and max_depth. These are hyperparameters. The first one is used to specify a minimum number of samples required to split a node, and the second one specifies a maximum depth of a tree. Both are used in recursive functions as exit conditions
- The _entropy(s) function calculates the impurity of an input vector s
- The _information_gain(parent, left_child, right_child) calculates the information gain value of a split between a parent and two children
- The _best_split(X, y) function calculates the best splitting parameters for input features X and a target variable y

        - It does so by iterating over every column in X and every threshold value in every column to find the optimal split using information gain
    - The _build(X, y, depth) function recursively builds a decision tree until stopping criterias are met (hyperparameters in the constructor)
    - The fit(X, y) function calls the _build() function and stores the built tree to the constructor
    - The _predict(x) function traverses the tree to classify a single instance
    - The predict(X) function applies the _predict() function to every instance in matrix X

In [7]:

```
class DecisionTree:
    '''
    Class which implements a decision tree classifier algorithm.
    '''
    def __init__(self, min_samples_split=2, max_depth=5):
        self.min_samples_split = min_samples_split
        self.max_depth = max_depth
        self.root = None

    @staticmethod
    def _entropy(s):
        '''
        Helper function, calculates entropy from an array of integer values.

        :param s: list
        :return: float, entropy value
        '''
        # Convert to integers to avoid runtime errors
        counts = np.bincount(np.array(s, dtype=np.int64))
        # Probabilities of each class label
        percentages = counts / len(s)

        # Caclulate entropy
        entropy = 0
        for pct in percentages:
            if pct > 0:
                entropy += pct * np.log2(pct)
        return -entropy

    def _information_gain(self, parent, left_child, right_child):
        '''
        Helper function, calculates information gain from a parent and two child nodes.

        :param parent: list, the parent node
```

```
        :param left_child: list, left child of a parent
        :param right_child: list, right child of a parent
        :return: float, information gain
        '''
        num_left = len(left_child) / len(parent)
        num_right = len(right_child) / len(parent)

        # One-liner which implements the previously discussed formula
        return self._entropy(parent) - (num_left * self._entropy(left_child) + num_right *
self._entropy(right_child))

    def _best_split(self, X, y):
        '''
        Helper function, calculates the best split for given features and target

        :param X: np.array, features
        :param y: np.array or list, target
        :return: dict
        '''
        best_split = {}
        best_info_gain = -1
        n_rows, n_cols = X.shape

        # For every dataset feature
        for f_idx in range(n_cols):
            X_curr = X[:, f_idx]
            # For every unique value of that feature
            for threshold in np.unique(X_curr):
                # Construct a dataset and split it to the left and right parts
                # Left part includes records lower or equal to the threshold
                # Right part includes records higher than the threshold
                df = np.concatenate((X, y.reshape(1, -1).T), axis=1)
                df_left = np.array([row for row in df if row[f_idx] <= threshold])
                df_right = np.array([row for row in df if row[f_idx] > threshold])

                # Do the calculation only if there's data in both subsets
                if len(df_left) > 0 and len(df_right) > 0:
                    # Obtain the value of the target variable for subsets
                    y = df[:, -1]
                    y_left = df_left[:, -1]
                    y_right = df_right[:, -1]

                    # Caclulate the information gain and save the split parameters
                    # if the current split if better then the previous best
                    gain = self._information_gain(y, y_left, y_right)
                    if gain > best_info_gain:
```

```
                    best_split = {
                        'feature_index': f_idx,
                        'threshold': threshold,
                        'df_left': df_left,
                        'df_right': df_right,
                        'gain': gain
                    }
                    best_info_gain = gain
        return best_split

    def _build(self, X, y, depth=0):
        '''
        Helper recursive function, used to build a decision tree from the input data.

        :param X: np.array, features
        :param y: np.array or list, target
        :param depth: current depth of a tree, used as a stopping criteria
        :return: Node
        '''
        n_rows, n_cols = X.shape

        # Check to see if a node should be leaf node
        if n_rows >= self.min_samples_split and depth <= self.max_depth:
            # Get the best split
            best = self._best_split(X, y)
            # If the split isn't pure
            if best['gain'] > 0:
                # Build a tree on the left
                left = self._build(
                    X=best['df_left'][:, :-1],
                    y=best['df_left'][:, -1],
                    depth=depth + 1
                )
                right = self._build(
                    X=best['df_right'][:, :-1],
                    y=best['df_right'][:, -1],
                    depth=depth + 1
                )
                return Node(
                    feature=best['feature_index'],
                    threshold=best['threshold'],
                    data_left=left,
                    data_right=right,
                    gain=best['gain']
                )
        # Leaf node - value is the most common target value
```

```
        return Node(
            value=Counter(y).most_common(1)[0][0]
        )

    def fit(self, X, y):
        '''
        Function used to train a decision tree classifier model.

        :param X: np.array, features
        :param y: np.array or list, target
        :return: None
        '''
        # Call a recursive function to build the tree
        self.root = self._build(X, y)

    def _predict(self, x, tree):
        '''
        Helper recursive function, used to predict a single instance (tree traversal).

        :param x: single observation
        :param tree: built tree
        :return: float, predicted class
        '''
        # Leaf node
        if tree.value != None:
            return tree.value
        feature_value = x[tree.feature]

        # Go to the left
        if feature_value <= tree.threshold:
            return self._predict(x=x, tree=tree.data_left)

        # Go to the right
        if feature_value > tree.threshold:
            return self._predict(x=x, tree=tree.data_right)

    def predict(self, X):
        '''
        Function used to classify new instances.

        :param X: np.array, features
        :return: np.array, predicted classes
        '''
        # Call the _predict() function for every observation
        return [self._predict(x, self.root) for x in X]
```

Testing

- We'll use the Iris dataset from Scikit-Learn

In [8]:

```
from sklearn.datasets import load_iris

iris = load_iris()

X = iris['data']
y = iris['target']
```

- The below code applies train/test split to the dataset:

In [9]:

```
from sklearn.model_selection import train_test_split

X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=42)
```

- You can now initialize and train the model, and make the predictions afterwards:

In [10]:

```
model = DecisionTree()
model.fit(X_train, y_train)
preds = model.predict(X_test)
```

In [11]:

```
np.array(preds, dtype=np.int64)
```

In [12]:

```
y_test
```

- As you can see, the arrays are identical
- Let's calculate the accuracy to confirm this:

In [13]:

```
from sklearn.metrics import accuracy_score

accuracy_score(y_test, preds)
```

- As expected, the perfect score was obtained on the test set

Comparison with Scikit-Learn

- We already know our model works good, but let's compare it to the DecisionTreeClassifier from Scikit-Learn

In [14]:

```
from sklearn.tree import DecisionTreeClassifier

sk_model = DecisionTreeClassifier()
sk_model.fit(X_train, y_train)
sk_preds = sk_model.predict(X_test)
```

In [15]:

```
accuracy_score(y_test, sk_preds)
```

- Both perform the same, at least accuracy-wise