6. Random Forest

- Random forest is a supervised machine learning algorithm used for both regression and classification
- This notebook demonstrates how to implement the classification algorithm from scratch
- Random forest is built on top of weak learners - Decision trees
    - An analogy of many trees forming a forest
    - The term "random" indicates that each decision tree is built with a random subset of data
- Random forest algorithm is based on the bagging method - a combination of learning models with the aim of increase in accuracy
- In a nutshell:
    - If you understand how a single decision tree works, you'll understand random forest
    - The math for the entire forest is identical as for a single tree, so we don't have to go over it again

The algorithm in a nutshell

- Make N data subsets from the original set (training)
- Build N decision trees (training)
- Make predictions with every trained decision tree, and return a final prediction as a majority vote (prediction)

Implementation

- We'll need three classes
    - Node - implements a single node of a decision tree

    - DecisionTree - implements a single decision tree
    - RandomForest - implements our ensamble 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 [1]:

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

In [63]:

```
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 thresold 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 [62]:

```
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]
```

The RandomForest class is built on top of a single decision tree and has the following methods:

- The __init__() method holds hyperparameter values for the number of trees in the forest, minimum samples split and maximum depth. It will also hold individually trained decision trees once the model is trained
- The _sample(X, y) applies bootstrap sampling to input features and input target
- The fit(X, y) method trains the Random Forest classifier
- The predict(X) method makes predictions with individual decision trees and then applies majority voting for the final prediction

In [48]:

```
class RandomForest:
    '''
    A class that implements Random Forest algorithm from scratch.
    '''
    def __init__(self, num_trees=25, min_samples_split=2, max_depth=5):
        self.num_trees = num_trees
        self.min_samples_split = min_samples_split
        self.max_depth = max_depth
        # Will store individually trained decision trees
        self.decision_trees = []

    @staticmethod
    def _sample(X, y):
        '''
        Helper function used for boostrap sampling.

        :param X: np.array, features
        :param y: np.array, target
        :return: tuple (sample of features, sample of target)
        '''
        n_rows, n_cols = X.shape
        # Sample with replacement
        samples = np.random.choice(a=n_rows, size=n_rows, replace=True)
        return X[samples], y[samples]

    def fit(self, X, y):
        '''
        Trains a Random Forest classifier.

        :param X: np.array, features
        :param y: np.array, target
        :return: None
        '''
        # Reset
        if len(self.decision_trees) > 0:
            self.decision_trees = []
        # Build each tree of the forest
        num_built = 0
        while num_built < self.num_trees:
            try:
                clf = DecisionTree(
                    min_samples_split=self.min_samples_split,
                    max_depth=self.max_depth
                )
                # Obtain data sample
                _X, _y = self._sample(X, y)
                # Train
                clf.fit(_X, _y)
                # Save the classifier
                self.decision_trees.append(clf)
                num_built += 1
            except Exception as e:
                continue

    def predict(self, X):
        '''
        Predicts class labels for new data instances.

        :param X: np.array, new instances to predict
        :return:
        '''
        # Make predictions with every tree in the forest
        y = []
        for tree in self.decision_trees:
            y.append(tree.predict(X))

        # Reshape so we can find the most common value
        y = np.swapaxes(a=y, axis1=0, axis2=1)

        # Use majority voting for the final prediction
        predictions = []
        for preds in y:
            counter = Counter(x)
            predictions.append(counter.most_common(1)[0][0])
        return predictions
```

Testing

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

In [49]:

```
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 [54]:

```
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)
```

In [55]:

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

In [57]:

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

In [58]:

```
Y_test
```

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

In [59]:

```
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 RandomForestClassifier from Scikit-Learn

In [60]:

```python
from sklearn.ensemble import RandomForestClassifier

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

In [61]:

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