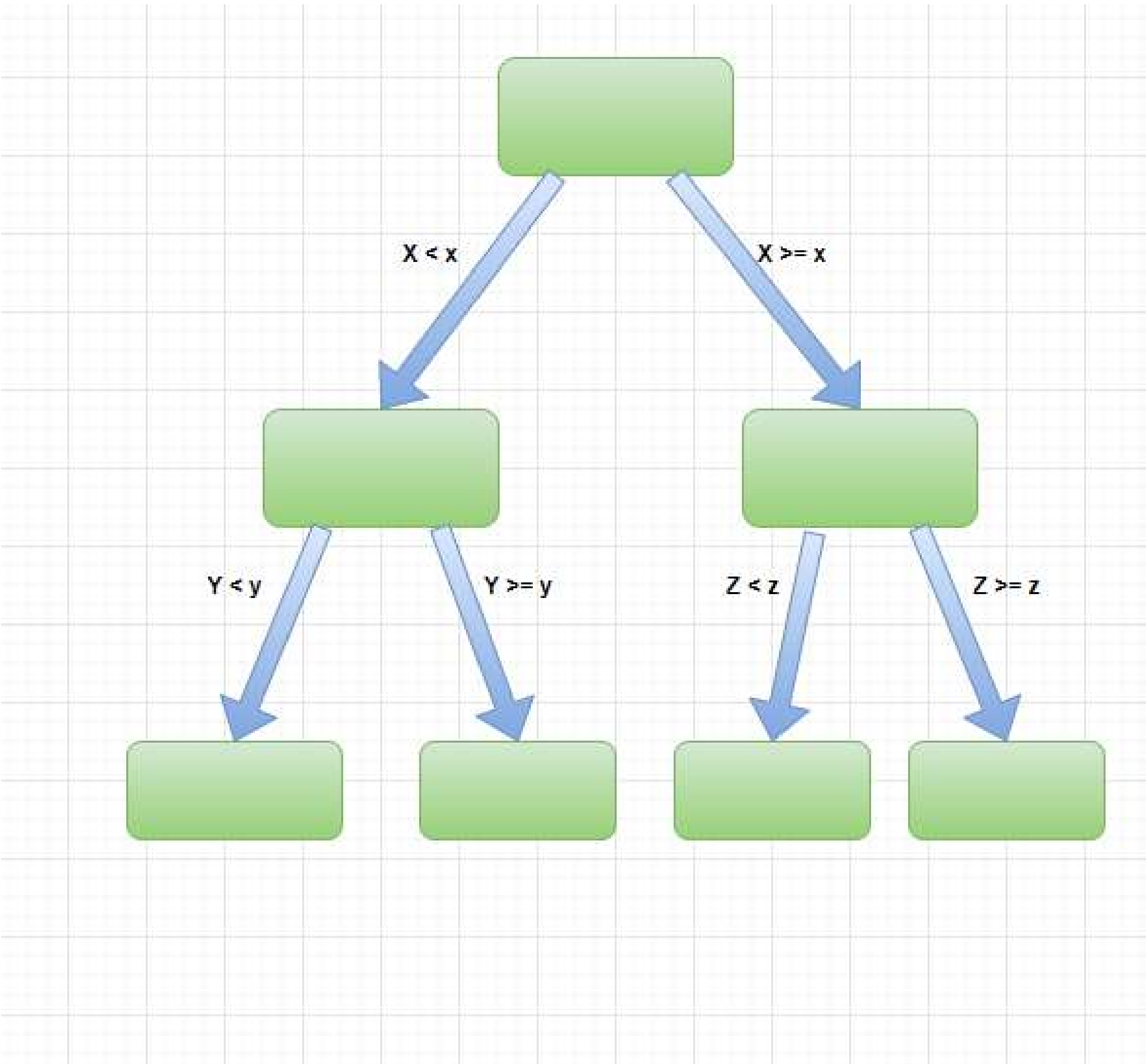
# Decision Tree Algorithm in Python From Scratch

Coding the popular algorithm using just NumPy and Pandas in Python and explaining what's under the hood

Eligijus Bujokas Apr 14 · 8 min read




Decision tree schema; graph by author

· · ·

The aim of this article is to make all the parts of a decision tree classifier clear by walking through the code that implements the algorithm. The code uses only NumPy, Pandas and the standard python libraries.

The full code can be accessed via [https://github.com/Eligijus112/decision-tree-python](https://github.com/Eligijus112/decision-tree-python)

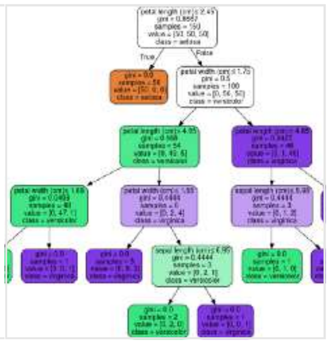
As of now, the code creates a decision tree when the target variable is binary and the features are numeric. This is completely sufficient to understand the algorithm.

The golden standard of building decision trees in python is the scikit-learn implementation:

**1.10. Decision Trees - scikit-learn 0.24.1 documentation**

Decision Trees (DTs) are a non-parametric supervised learning method used for classification and regression . The goal...

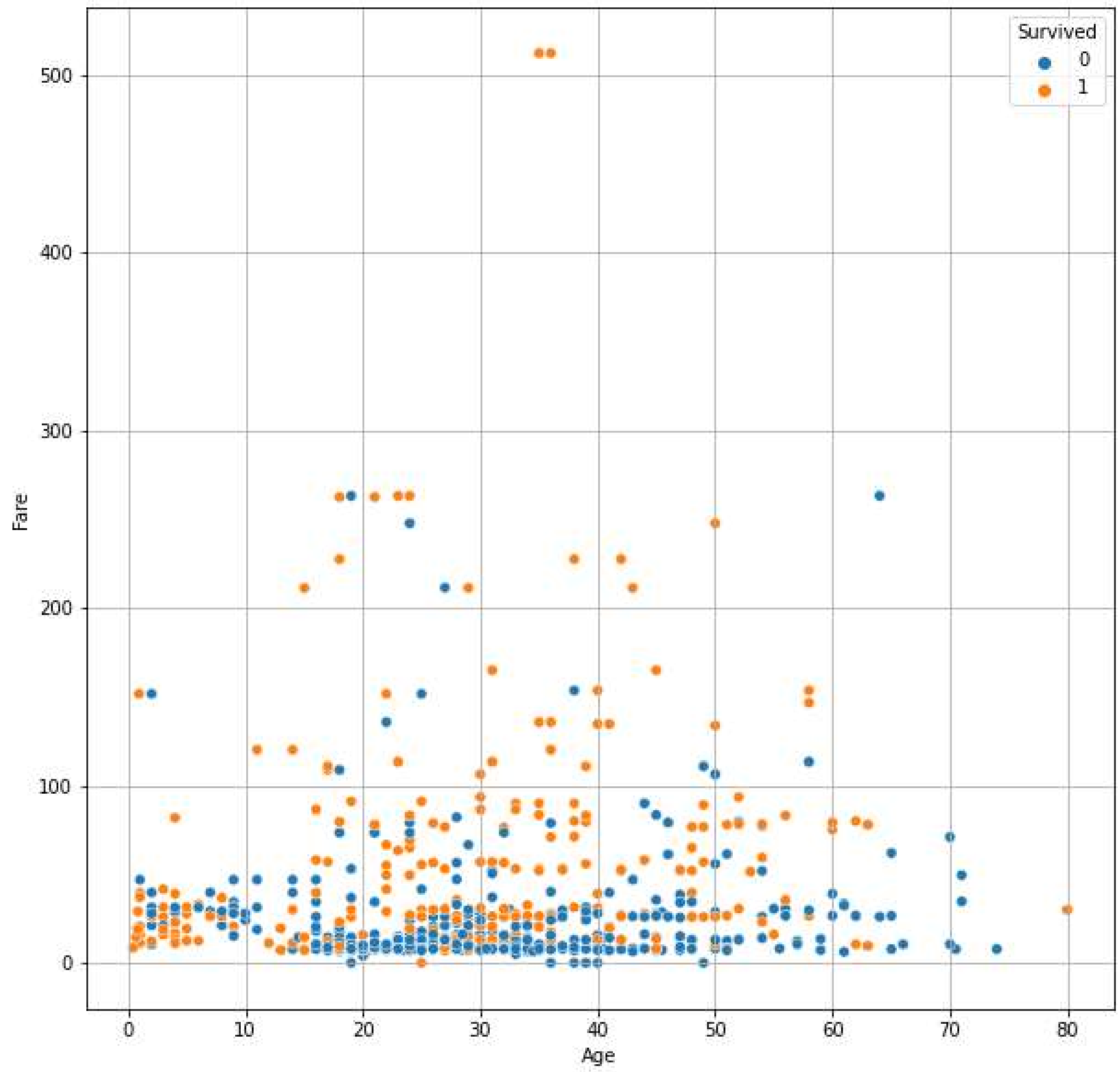
scikit-learn.org



When I tested out my code I wanted to make sure that the results are identical to the scikit-learn implementation.

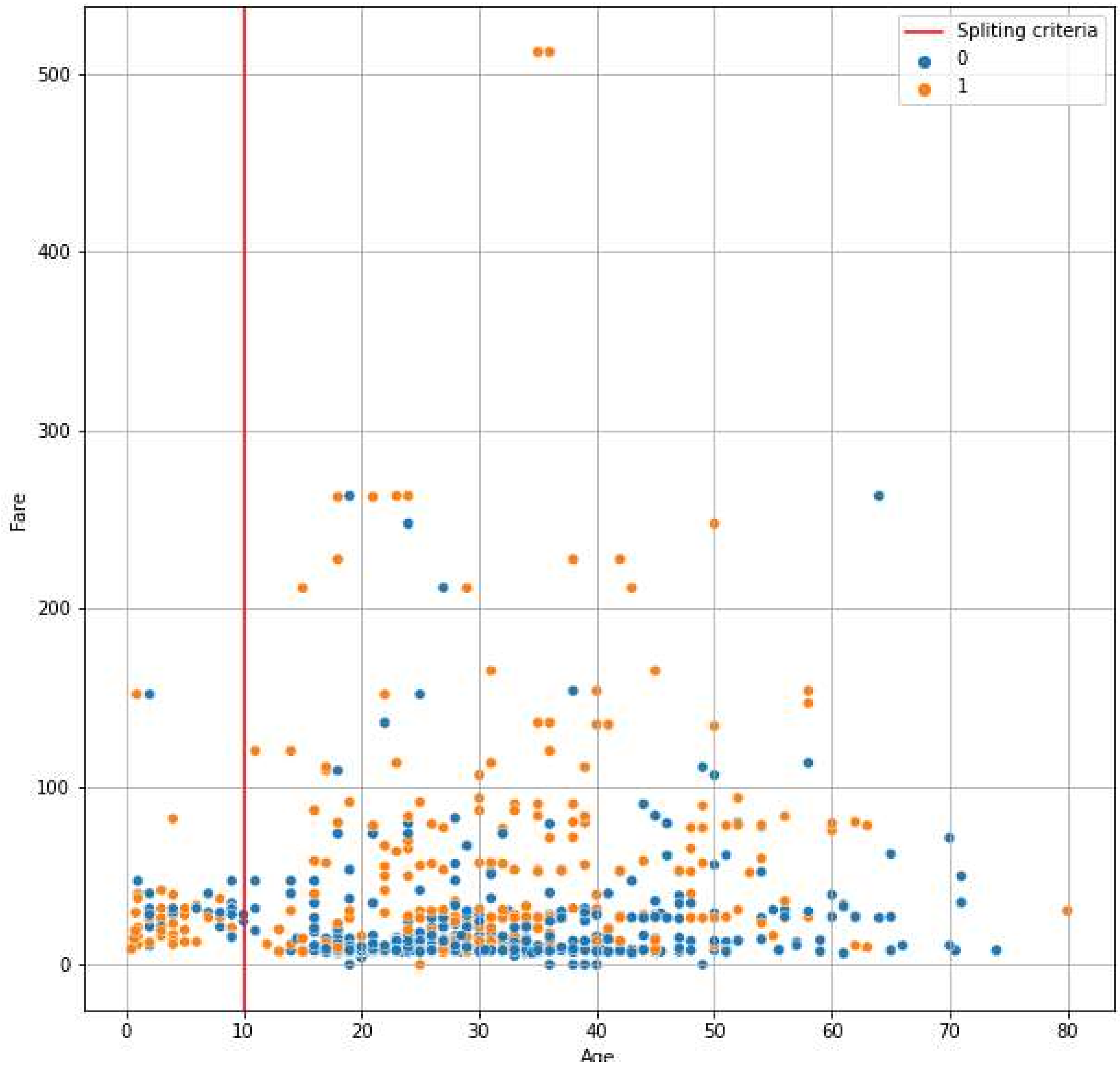
. . .

The data used in this article is the famous Titanic survivor dataset. We will use two numeric variables — Age of the passenger and the Fare of the ticket — to predicting whether a passenger survived or not.




Age + Fare ~ Survival; Graph by author

The goal is to create the **"best"** splits of the numeric variables. Just eyeballing the data, we could guess that one good split is to split the data into two parts: observations that have Age $< 10$ and observations that have Age $\geq 10$:




Splitting the dataset; Graph by author

Now some immediate questions may rise:

*Is this a good split?*

*Maybe a split at the Fare = 200 is a better one?*

*How do we quantify the "goodness" of a split?*

*How does a computer search for the best split?*

All of these questions will be answered by the end of this article.

. . .

A decision tree algorithm (DT for short) is a machine learning algorithm that is used in classifying an observation given a set of input features. The algorithm creates a set of rules at various decision levels such that a certain metric is optimized.

The target variable will be denoted as **Y = {0, 1}** and the feature matrix will be denoted as **X.**

Keywords to expand on:

**Node**

**Gini impurity** (a metric which we are optimizing)

**Level**

**Splitting**

. . .

A **node** is the building block in the decision tree. When viewing a typical schema of a decision tree (like the one in the title picture) the nodes are the rectangles or bubbles that have a downward connection to other nodes.

Each node has the following main attributes:

***Gini impurity score***

***Number of observations***

***The number of observations belonging to each of the binary target classes.***

***The feature matrix X representing the observations that fall into the node.***

The custom **Node** class in python (written by me):

```
# Data wrangling
import pandas as pd

# Array math
import numpy as np

# Quick value count calculator
from collections import Counter


class Node:
    """
    Class for creating the nodes for a decision tree
    """
    def __init__(
        self,
        Y: list,
        X: pd.DataFrame,
        min_samples_split=None,
        max_depth=None,
        depth=None,
        node_type=None,
        rule=None
    ):
        # Saving the data to the node
        self.Y = Y
        self.X = X

        # Saving the hyper parameters
```

```
        self.min_samples_split = min_samples_split if min_samples_split else 20
        self.max_depth = max_depth if max_depth else 5

        # Default current depth of node
        self.depth = depth if depth else 0

        # Extracting all the features
        self.features = list(self.X.columns)

        # Type of node
        self.node_type = node_type if node_type else 'root'

        # Rule for spliting
        self.rule = rule if rule else ""

        # Calculating the counts of Y in the node
        self.counts = Counter(Y)

        # Getting the GINI impurity based on the Y distribution
        self.gini_impurity = self.get_GINI()

        # Sorting the counts and saving the final prediction of the node
        counts_sorted = list(sorted(self.counts.items(), key=lambda item: item[1]))

        # Getting the last item
        yhat = None
        if len(counts_sorted) > 0:
            yhat = counts_sorted[-1][0]

        # Saving to object attribute. This node will predict the class with the most frequent c
        self.yhat = yhat

        # Saving the number of observations in the node
        self.n = len(Y)

        # Initiating the left and right nodes as empty nodes
        self.left = None
        self.right = None

        # Default values for splits
        self.best_feature = None
        self.best_value = None

    @staticmethod
    def GINI_impurity(y1_count: int, y2_count: int) -> float:
        """
        Given the observations of a binary class calculate the GINI impurity
        """
        # Ensuring the correct types
        if y1_count is None:
            y1_count = 0

        if y2_count is None:
            y2_count = 0

        # Getting the total observations
        n = y1_count + y2_count

        # If n is 0 then we return the lowest possible gini impurity
        if n == 0:
            return 0.0

        # Getting the probability to see each of the classes
        p1 = y1_count / n
        p2 = y2_count / n

        # Calculating GINI
        gini = 1 - (p1 ** 2 + p2 ** 2)

```

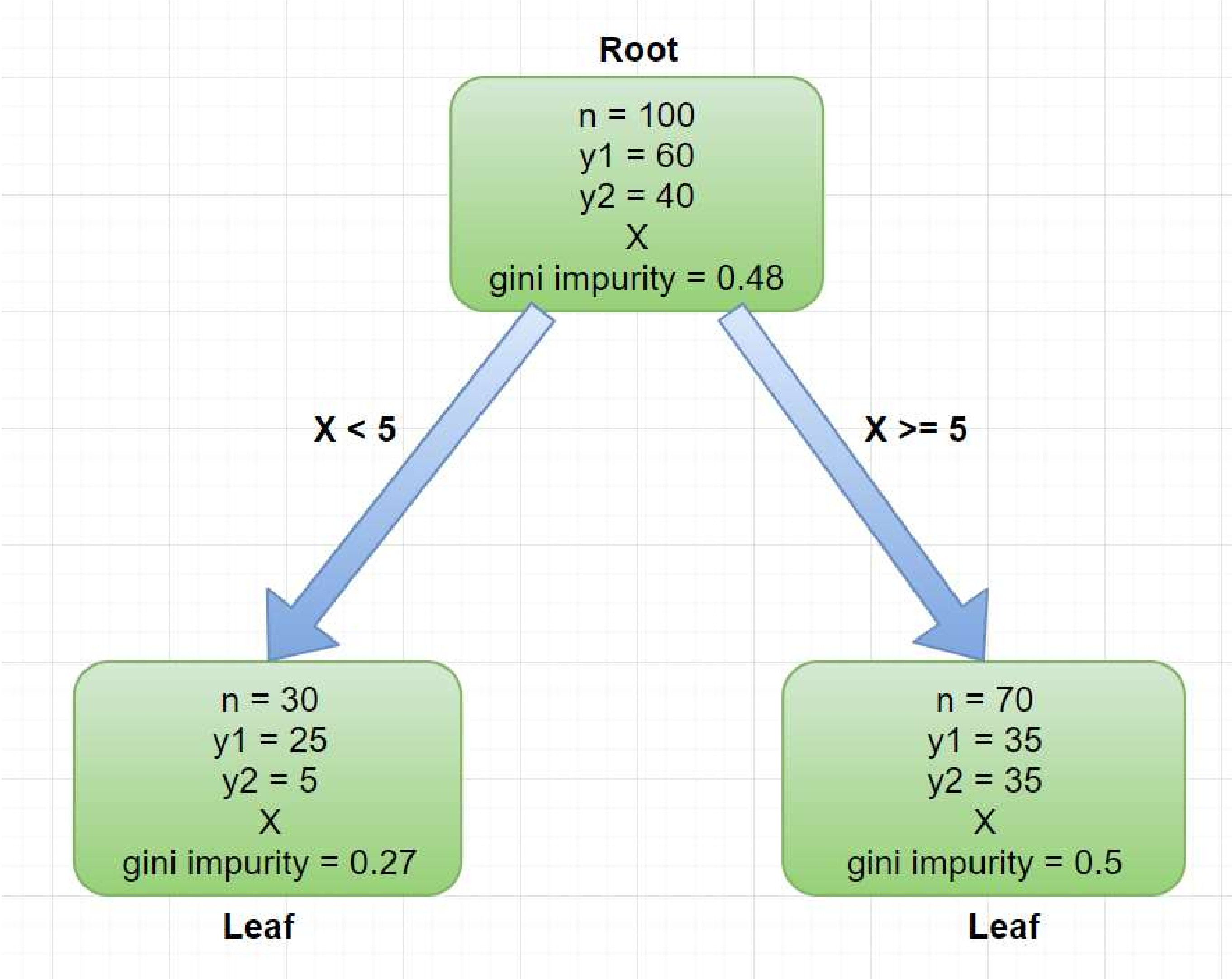
```
        # Returning the gini impurity
        return gini

    @staticmethod
    def ma(x: np.array, window: int) -> np.array:
        """
        Calculates the moving average of the given list.
        """
        return np.convolve(x, np.ones(window), 'valid') / window

    def get_GINI(self):
        """
        Function to calculate the GINI impurity of a node
        """
        # Getting the 0 and 1 counts
        y1_count, y2_count = self.counts.get(0, 0), self.counts.get(1, 0)

        # Getting the GINI impurity
        return self.GINI_impurity(y1_count, y2_count)

    def best_split(self) -> tuple:
        """
        Given the X features and Y targets calculates the best split
        for a decision tree
        """
        # Creating a dataset for spliting
        df = self.X.copy()
        df['Y'] = self.Y

        # Getting the GINI impurity for the base input
        GINI_base = self.get_GINI()

        # Finding which split yields the best GINI gain
        max_gain = 0

        # Default best feature and split
        best_feature = None
        best_value = None

        for feature in self.features:
            # Droping missing values
            Xdf = df.dropna().sort_values(feature)

            # Sorting the values and getting the rolling average
            xmeans = self.ma(Xdf[feature].unique(), 2)

            for value in xmeans:
                # Spliting the dataset
                left_counts = Counter(Xdf[Xdf[feature]<value]['Y'])
                right_counts = Counter(Xdf[Xdf[feature]>=value]['Y'])

                # Getting the Y distribution from the dicts
                y0_left, y1_left, y0_right, y1_right = left_counts.get(0, 0), left_counts.get(1

                # Getting the left and right gini impurities
                gini_left = self.GINI_impurity(y0_left, y1_left)
                gini_right = self.GINI_impurity(y0_right, y1_right)

                # Getting the obs count from the left and the right data splits
                n_left = y0_left + y1_left
                n_right = y0_right + y1_right

                # Calculating the weights for each of the nodes
                w_left = n_left / (n_left + n_right)
                w_right = n_right / (n_left + n_right)

                # Calculating the weighted GINI impurity
                wGINI = w_left * gini_left + w_right * gini_right
```

```
                # Calculating the GINI gain
                GINIgain = GINI_base - wGINI

                # Checking if this is the best split so far
                if GINIgain > max_gain:
                    best_feature = feature
                    best_value = value

                    # Setting the best gain to the current one
                    max_gain = GINIgain

        return (best_feature, best_value)

    def grow_tree(self):
        """
        Recursive method to create the decision tree
        """
        # Making a df from the data
        df = self.X.copy()
        df['Y'] = self.Y

        # If there is GINI to be gained, we split further
        if (self.depth < self.max_depth) and (self.n >= self.min_samples_split):

            # Getting the best split
            best_feature, best_value = self.best_split()

            if best_feature is not None:
                # Saving the best split to the current node
                self.best_feature = best_feature
                self.best_value = best_value

                # Getting the left and right nodes
                left_df, right_df = df[df[best_feature]<=best_value].copy(), df[df[best_feature

                # Creating the left and right nodes
                left = Node(
                    left_df['Y'].values.tolist(),
                    left_df[self.features],
                    depth=self.depth + 1,
                    max_depth=self.max_depth,
                    min_samples_split=self.min_samples_split,
                    node_type='left_node',
                    rule=f"{best_feature} <= {round(best_value, 3)}"
                    )

                self.left = left
                self.left.grow_tree()

                right = Node(
                    right_df['Y'].values.tolist(),
                    right_df[self.features],
                    depth=self.depth + 1,
                    max_depth=self.max_depth,
                    min_samples_split=self.min_samples_split,
                    node_type='right_node',
                    rule=f"{best_feature} > {round(best_value, 3)}"
                    )

                self.right = right
                self.right.grow_tree()

    def print_info(self, width=4):
        """
        Method to print the infromation about the tree
        """
        # Defining the number of spaces
        const = int(self.depth * width ** 1.5)
```

```
        spaces = "-" * const

        if self.node_type == 'root':
            print("Root")
        else:
            print(f"|{spaces} Split rule: {self.rule}")
        print(f"{' ' * const}   | GINI impurity of the node: {round(self.gini_impurity, 2)}")
        print(f"{' ' * const}   | Class distribution in the node: {dict(self.counts)}")
        print(f"{' ' * const}   | Predicted class: {self.yhat}")

    def print_tree(self):
        """
        Prints the whole tree from the current node to the bottom
        """
        self.print_info()

        if self.left is not None:
            self.left.print_tree()

        if self.right is not None:
            self.right.print_tree()

    def predict(self, X:pd.DataFrame):
        """
        Batch prediction method
        """
        predictions = []

        for _, x in X.iterrows():
            values = {}
            for feature in self.features:
                values.update({feature: x[feature]})

            predictions.append(self.predict_obs(values))

        return predictions

    def predict_obs(self, values: dict) -> int:
        """
        Method to predict the class given a set of features
        """
        cur_node = self
        while cur_node.depth < cur_node.max_depth:
            # Traversing the nodes all the way to the bottom
            best_feature = cur_node.best_feature
            best_value = cur_node.best_value

            if cur_node.n < cur_node.min_samples_split:
                break

            if (values.get(best_feature) < best_value):
                if self.left is not None:
                    cur_node = cur_node.left
            else:
                if self.right is not None:
                    cur_node = cur_node.right

        return cur_node.yhat
```

The first node in a decision tree is called ***the root.*** The nodes at the bottom of the tree are called ***leaves***.

If splitting criteria are satisfied, then each node has two linked nodes to it: the left node and the right node. For example, a very simple decision tree with one root and two leaves may look like this:




Example decision tree; Graph by author

```
n — number of observations

y1 — number of first class elements

y2 — number of second class elements

X — numeric feature for the observations
```

An input if the feature X less than 5 would go to the left node. A feature value of more or equal to 5 would go to the right node.

. . .

As mentioned above, each node has a GINI impurity score. To calculate the GINI impurity, all that is needed is the distribution of the target variable in the node or simply, how many Y=1 and Y=0 observations there are in a node.

The formal definition of GINI impurity is as follows:

*Gini impurity is a measure of how often a randomly chosen element from the set would be incorrectly labelled if it was randomly labelled according to the distribution of labels in the subset.*

The algebraic definition:

Suppose we have two classes in the dataset:

$$k_1, k_2$$

Each of the classes have $n_1$ and $n_2$ observations.

The probability of observing something from one of the $k$ classes is:

$$p(i) = P(x_i \in k_i) = \frac{n_i}{n_1 + n_2}, i \in \{1, 2\}$$

The GINI impurity of such a system is calculated with the following formula:

$$G = 1 - \Sigma_{i=1}^{2} p(i)^2$$

GINI impurity formula; Formulas are taken from the author's notebook

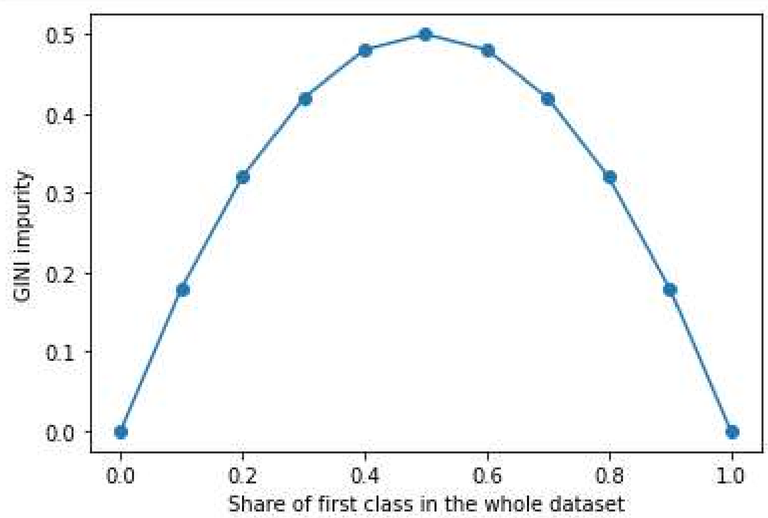
For example, if we have in a node 10 observations that are survivors and 5 observations that are not survivors, then:

```
p(survivors) = 10 /(10 + 5) = 0.66..
p(non-survivors) = 5 /(10 + 5) = 0.33..
```

Thus the GINI impurity can be calculated by squaring the two numbers, adding them up and subtracting from one:

```
gini impurity = 1 - (0.66..^2 + 0.33..^2) = 0.44..
```

In a binary case, the maximum Gini impurity is equal to 0.5 and the lowest is 0. The lower the value, the more "pure" the node is. No matter how many observations we have, we can calculate the share of one of the classes in terms of the whole dataset and draw a relationship:




Gini impurity vs share of class in a dataset; Image by author

***We always search for a split where the GINI impurity will be the lowest.***

*If nothing else, the basic intuition is that the more one class observations there is in a node, the lower its impurity.*

. . .

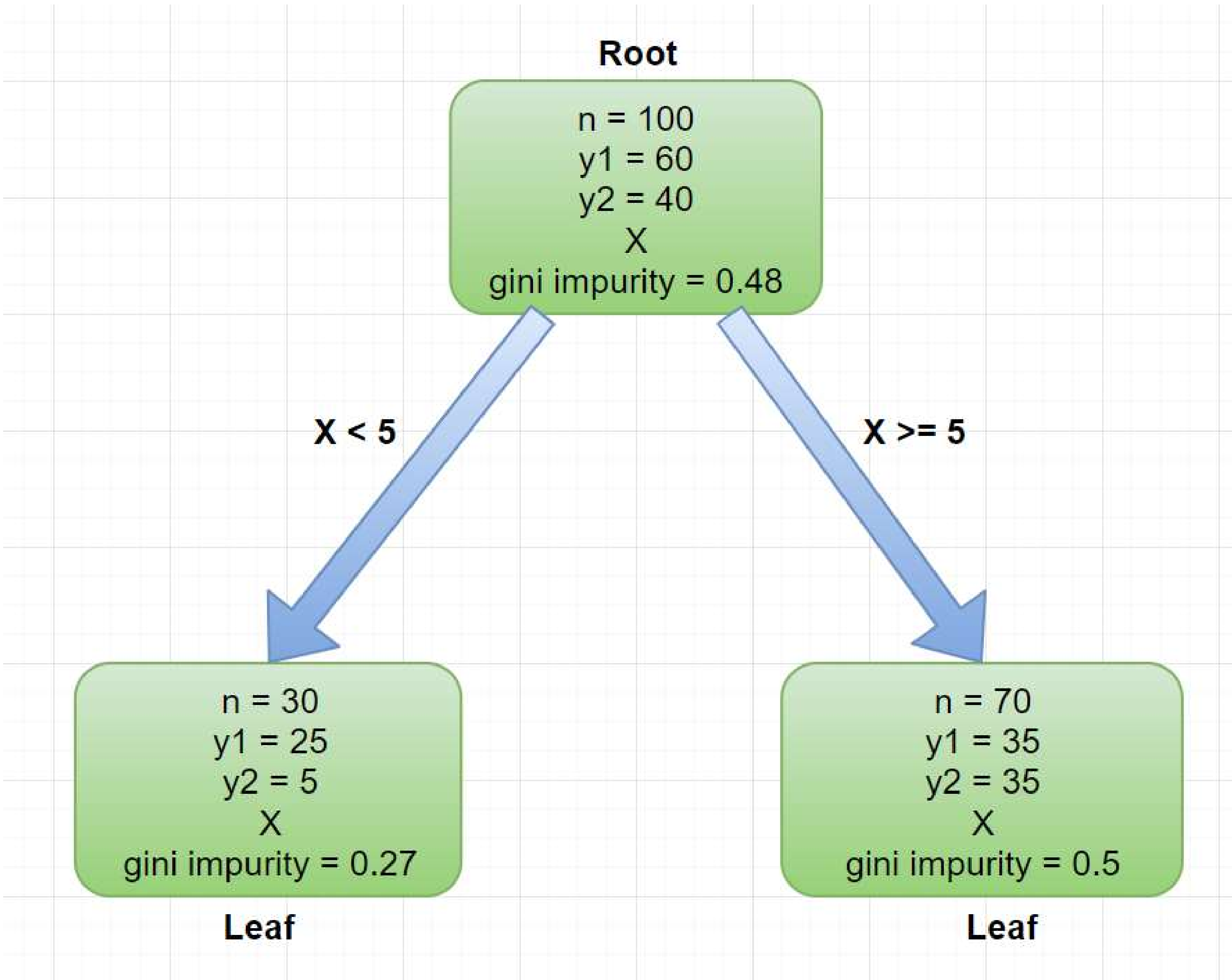
The level attribute defines how many splits were made before the node was created. So for example, the root node's level is 0, then the left and the right nodes have a level of 1, and so on.

In the custom Node class, the maximum depth of the tree can be regulated with the hyperparameter **max_depth.**

. . .

The splitting procedure is a process where we search in each node which feature and which feature value is the best to divide the data into two smaller parts.

In the classification case, we want to maximize the **Gini gain.** Going back to the example before:




Example decision tree; Graph by author

The root node has a Gini score of 0.48. The left node has a score of 0.27 and the right node has 0.5. ***How much Gini did we "gain"?***

The formula to calculate this is:

Gini gain = Parent node Gini impurity subtracted by the weighted average of the Gini impurities of the left and right nodes.

$$GINIgain = \Delta Gini = Gini_{parent} - (Gini_{left}\frac{n_{left}}{n_{right}+n_{left}} + Gini_{right}\frac{n_{right}}{n_{right}+n_{left}})$$

Gini gain formula; From author's notebook

In the example above, plugging in all the values we would get:

```
0.48 — (0.27 * 30 /100 + 0.5 * 70/100) = 0.049
```

The GINI gain is equal to 0.049. **Any positive Gini gain is an improvement. This means that our decision would make the nodes more "pure".**

How does the algorithm search for the best split among numeric columns?

For each of the feature, we sort the feature values and get the means of two neighbouring values.

For example, suppose our feature_1 is the following:

```
feature_1 = [7, 5, 9, 1, 2, 8]
```

Sorted:

```
feature_1 = [1, 2, 5, 7, 8, 9]
```

Means of neighbours:

```
feature_1 = [1.5, 3.5, 6.5, 7.5, 8.5]
```

We then check what is the GINI gain with each of the values from the above vector. And, we do this for all the features in our dataset.

***The final split value and split feature is the one that has the highest GINI gain.***

. . .

The hyperparameters of the custom Node object that grows the tree (more will be added in the future) are the **max_depth: int** and **min_samples_split: int** variables.

The max_depth integer defines how deep should the tree grow. At the depth of max_depth, the searching for the best split feature and split feature values stops.

The min_samples_split integer defines the minimal number of observations in a node for the best split search to start. So for example, if the node has 51 observations but the min_samples_split = 55, then the growth of the tree stops.

. . .

So, how does the code work?

First of all, read the data:

```
# Loading data
d = pd.read_csv('data/train.csv')

# Dropping missing values
dtree = d[['Survived', 'Age', 'Fare']].dropna().copy()

# Defining the X and Y matrices
Y = dtree['Survived'].values
X = dtree[['Age', 'Fare']]

# Saving the feature list
features = list(X.columns)
```

Then we define the dictionary of hyperparameters.

```
hp = {
 'max_depth': 3,
 'min_samples_split': 50
}
```

Then we initiate the root node:

```
root = Node(Y, X, **hp)
```

The main tree building function is the **grow_tree()** function.

```
root.grow_tree()
```

And that's it!

To view the results, we can invoke the **print_tree()** function.

```
root.print_tree()
```

The results:

```
Root
   | GINI impurity of the node: 0.48
   | Class distribution in the node: {0: 424, 1: 290}
   | Predicted class: 0
|-------- Split rule: Fare <= 52.277
           | GINI impurity of the node: 0.44
           | Class distribution in the node: {0: 389, 1: 195}
           | Predicted class: 0
|---------------- Split rule: Fare <= 10.481
                   | GINI impurity of the node: 0.32
                   | Class distribution in the node: {0: 192, 1: 47}
                   | Predicted class: 0
|------------------------ Split rule: Age <= 32.5
                           | GINI impurity of the node: 0.37
                           | Class distribution in the node: {0: 134, 1: 43}
                           | Predicted class: 0
|------------------------ Split rule: Age > 32.5
                           | GINI impurity of the node: 0.12
                           | Class distribution in the node: {0: 58, 1: 4}
                           | Predicted class: 0
|---------------- Split rule: Fare > 10.481
                   | GINI impurity of the node: 0.49
                   | Class distribution in the node: {0: 197, 1: 148}
                   | Predicted class: 0
|------------------------ Split rule: Age <= 6.5
                           | GINI impurity of the node: 0.41
                           | Class distribution in the node: {0: 12, 1: 30}
                           | Predicted class: 1
|------------------------ Split rule: Age > 6.5
                           | GINI impurity of the node: 0.48
                           | Class distribution in the node: {0: 185, 1: 118}
                           | Predicted class: 0
|-------- Split rule: Fare > 52.277
           | GINI impurity of the node: 0.39
           | Class distribution in the node: {1: 95, 0: 35}
           | Predicted class: 1
|---------------- Split rule: Age <= 63.5
                   | GINI impurity of the node: 0.38
                   | Class distribution in the node: {1: 95, 0: 32}
                   | Predicted class: 1
|------------------------ Split rule: Age <= 29.5
                           | GINI impurity of the node: 0.44
                           | Class distribution in the node: {0: 17, 1: 34}
                           | Predicted class: 1
|------------------------ Split rule: Age > 29.5
                           | GINI impurity of the node: 0.32
                           | Class distribution in the node: {1: 61, 0: 15}
                           | Predicted class: 1
|---------------- Split rule: Age > 63.5
                   | GINI impurity of the node: 0.0
                   | Class distribution in the node: {0: 3}
                   | Predicted class: 0
```

Full decision tree; Snippet by author

The decision tree obtained from the scikit-learn implementation is identical:

```
|--- Fare <= 52.28
|   |--- Fare <= 10.48
|   |   |--- Age <= 32.50
|   |   |   |--- class: 0
|   |   |--- Age >  32.50
```

```
|   |   |   |--- class: 0
|   |--- Fare >  10.48
|   |   |--- Age <= 6.50
|   |   |   |--- class: 1
|   |   |--- Age >  6.50
|   |   |   |--- class: 0
|--- Fare >  52.28
|   |--- Age <= 63.50
|   |   |--- Age <= 29.50
|   |   |   |--- class: 1
|   |   |--- Age >  29.50
|   |   |   |--- class: 1
|   |--- Age >  63.50
|   |   |--- class: 0
```

Although, the scikit-learn implementation prints out less information than my implementation.

As it turns out, the best first initial split is the Fare feature at a value of 52.28 and not at the proposed Age feature at value 10.

. . .

The code that I have written builds the same trees as scikit-learn implementation and the predictions are the same. But the training time for the scikit-learn algorithm is much faster. But my goal was not to grow the trees faster. My goal was to write an understandable code for any machine learning enthusiast to have a better grasp of what is happening under the hood.

Feel free to create a pull request in this repo https://github.com/Eligijus112/decision-tree-python if you see any bugs or just want to add functionalities.



Recursion Python Decision Tree Machine Learning Gini Impurity