Binary Decision Tree Classifier

Davud Topalović (32220055)

Abstract

This paper delves into the Decision Tree Classifier (DTC), specifically binary DTC, a supervised learning algorithm tailored for both classification and regression tasks. By emphasizing the underlying intuition of decision trees and showcasing practical examples, the paper explains how decision trees can be used for classification tasks. The recursive nature of Hunt's algorithm for tree growth and different types of splitting criteria are examined. Additionally, the paper provides an in-depth look at the DTC's practical implementation and discusses the algorithm's time complexity.

1 Introduction

A Decision Tree Classifier (DTC) is a non-parametric, supervised learning algorithm that is used for both classification and regression tasks [3]. Classification refers to the task of categorizing or assigning labels to input data based on their characteristics or features. On the other hand, regression refers to the statistical approach (task) of predicting a continuous or discrete output variable based on input variables or features. Both methods fall under the type of Machine Learning called Supervised Learning, which involves modeling the relationship between the independent variables and dependent, labeled variable.

2 Classification

The main focus of this paper is on analyzing the classification algorithm within decision trees.

Definition 2.1 (Classification). Classification is the task of learning a target function f that maps each attribute set X to one of the predefined class labels y.

The target function is also referred to as a classification model [1].

A classification model is useful for two main purposes:

- **Descriptive Modeling** - A classification model can serve as an explanatory tool that provides an insight in feature importance. For example, it would be useful for medical researchers to have a descriptive model that summarizes the diabetes data shown in Table 1 and explains what features have the most impact on diabetes.
- **Predictive Modeling** - A classification model can also be used to predict a class label of an unknown record. We can use a classification model built from the first 766 participants (input or training data) of the diabetes dataset and use it to predict whether a 767th participant (unseen data or test data) would be tested positive or negative for diabetes.

	Pregnancies	Glucose	Blood Pressure	Skin Thickness	Insulin	BMI	Diabetes Pedigree Function	Age	Outcome
0	6	148	72	35	0	33.6	0.627	5	1
1	1	85	66	29	0	26.6	0.351	31	0
2	8	183	64	0	0	23.3	0.672	32	1
3	1	89	66	23	94	28.1	0.167	21	0
4	0	137	40	35	168	43.1	2.288	33	1
...	...	...	...	...	...	...	...	...	...
766	1	126	60	0	0	30.1	0.349	47	1
767	1	93	70	31	0	30.4	0.315	23	**?**

Table 1: This dataset is originally from the National Institute of Diabetes and Digestive and Kidney Diseases. The dataset is available on Kaggle at https://www.kaggle.com/datasets/mathchi/diabetes-data-set.

So, in order to classify data, we aim to construct a classification model that is based on a specific algorithm. This is illustrated in Figure 1.

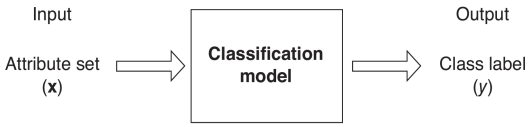

Figure 1: Classification as the task of mapping an input attribute space X to its class label y.

2.1 General Approach to Solving a Classification Problem

A classification technique, also known as *classifier*, is a systematic approach to constructing a classification model using an input data. Various examples of classifiers include decision trees, logistic regression, neural networks, support vector machines (SVM), and k-Nearest Neighbors (k-NN). Each technique utilizes a learning algorithm to identify a model that effectively captures the relationship between the attribute set and class label of the input data. The goal of the learning algorithm is to generate models that not only can fit the input data well but also can accurately predict the class labels of unseen records. Thus, an important objective of the learning algorithm is to create models with strong generalization capability, meaning that they can accurately predict the class labels of previously unseen records. The general approach depicted in Figure 2 outlines the steps for addressing classification problems. Training set, which must be provided with class labels, is used for building (training)[1] a model. The model is subsequently applied to the test set, which is a dataset that consists of unseen records[2].

[1] Building, training, learning, identifying or fitting a model are used interchangeably, but all refer to absolutely the same thing.

[2] Data records that are of the same type as training dataset but were not used during the model training.

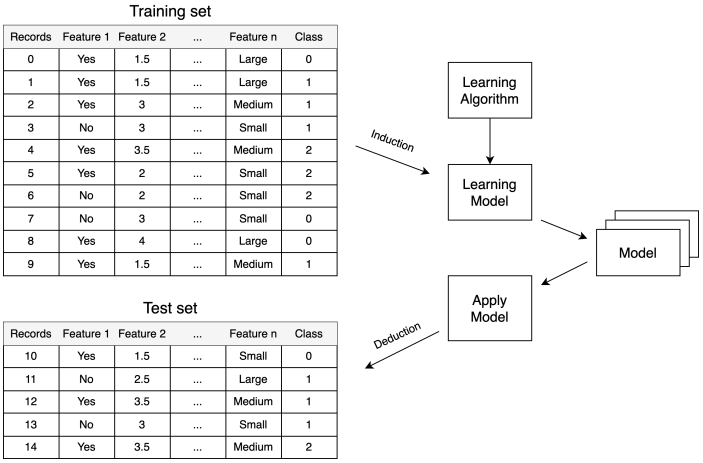

Figure 2: General approach for building a classification model. Learning a model means applying a learning algorithm on training dataset.

3 Decision Trees

One of the most intuitive tools for data classification is the decision tree. It is a greedy algorithm that hierarchically partitions the input space until it reaches a sub-space associated with a class label. Why is it greedy will be clear once we cover its working principle.

3.1 Intuition and working principle of a Decision Tree

To illustrate how classification with a decision tree works, let's consider a classification problem using the Vacation dataset, shown in Table 2. Based on feature values such as: Number of days, Weather forecast, etc., our goal is to predict whether a person prefers a countryside or beach location for their vacation.

Record	Number of days	Family joining	Personal budget	Weather forecast	Explore new places	Target
0	10	Yes	950	75	Yes	Countryside
1	10	Yes	250	78	Yes	Beach
2	7	Yes	600	80	No	Beach
3	8	Yes	750	67	Yes	Countryside
4	10	Yes	800	73	Yes	Beach
5	8	Yes	850	64	Yes	Countryside
6	15	No	350	78	No	Beach
7	8	Yes	850	81	Yes	Countryside
8	6	No	750	59	Yes	Beach
9	12	Yes	1050	54	Yes	Beach
10	10	No	230	74	No	Countryside
11	3	No	630	58	Yes	Countryside
12	10	Yes	830	74	No	Beach
13	12	No	730	52	Yes	Beach

Table 2: Vacation dataset. Borrowed from: towardsdatascience.com

Suppose we have a new individual whose information about Number of days, Family joining, etc. is available for analysis. How can we determine their preferred vacation location?

We can start by asking a question about a specific attribute, such as the personal budget for the vacation. For instance, we could ask if personal budget is above a certain threshold, and if yes, we can conclude that the person is more likely to prefer a beach as a location. However, if the personal budget is below the threshold, we proceed to the next question. We might then ask about the weather forecast to further inform our decision-making process and get closer to the conclusion.

If the weather is hot enough, we might determine that the person is still more likely to prefer a beach. On the other hand, if the weather is not hot enough, we continue with additional questions. For example, we might inquire about family joining or number of days for vacation, or interest in exploring new places. By posing a series of carefully selected questions about these attributes, we can progressively refine our prediction until we reach a final conclusion about the person's preferred vacation location.

This series of questions and their potential answers can be organized in the form of a tree, which consists of nodes and directed edges. We'll call this tree a decision tree. In the vacation dataset example, the decision tree would have a root node representing the initial question about the personal budget, internal nodes representing the subsequent attribute-based questions, and leaf nodes representing the final outcomes (countryside or beach).

Figure 3 shows the decision tree for the vacation problem. The tree has three types of nodes:

- A **root node** that has no incoming edges and zero or more outgoing edges.
- **Internal or decision nodes**, each of which has exactly one incoming edge and two or more outgoing edges.
- **Leaf or terminal nodes**, each of which has exactly one incoming edge and no outgoing edges.

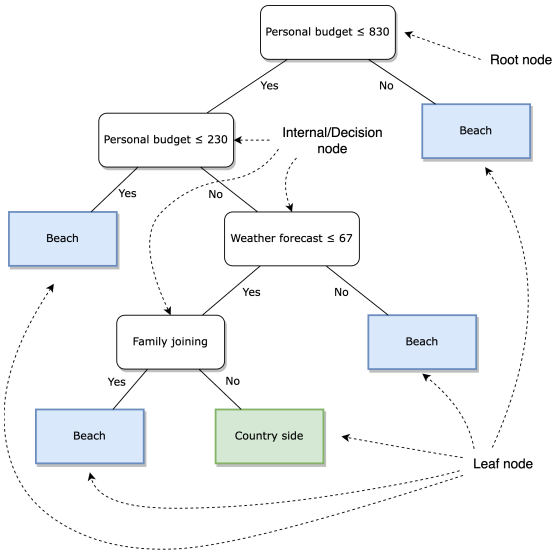

Figure 3: A decision tree for the vacation location problem. The decision node is trained on the first 11 data points (starting from zero). Remaining 3 data points are used for testing.

In a decision tree, each leaf node is assigned a class label. The non terminal nodes, also known as impure nodes, which include the root and other decision nodes, contain attribute test conditions to separate records that have different characteristics. Additionally, each node is assigned a number of records that it received from previous decision (or root) node.

Now, let's examine the decision tree for vacation problem, depicted in Figure 3. Root node contains all the vacation training samples and uses the attribute `Personal budget` to separate vacations according to whether the personal budget is lower/equal or higher than 830$. For the higher ones we come to the right node which is a pure leaf node with class `Beach` since all vacations for the budget higher than 830$ took place on beach. The left child of the root node represents the decision node or another condition, and that is whether, again, the budget is less or equal to some threshold, in this case, 230$. In this case all vacations cheaper than 230$ are also beach vacations, so the left child is again the pure leaf node of class `Beach`. For the vacation with a personal budget higher than 230$ (but remember lower than

830$), we come to the right child, which is a decision node with a condition on weather.

Classifying a test record is straightforward once a decision tree has been constructed. Starting from the root node, we apply the test condition to the record and follow the appropriate branch based on the outcome of the test. This will lead us either to another internal node, for which a new test condition is applied, or to a leaf node. The class label associated with the leaf node is then assigned to the record. As an illustration, Figure 4 traces the path in the decision tree that is used to predict the class label of a new vacation. The path terminates at a leaf node labeled as `Country side`.

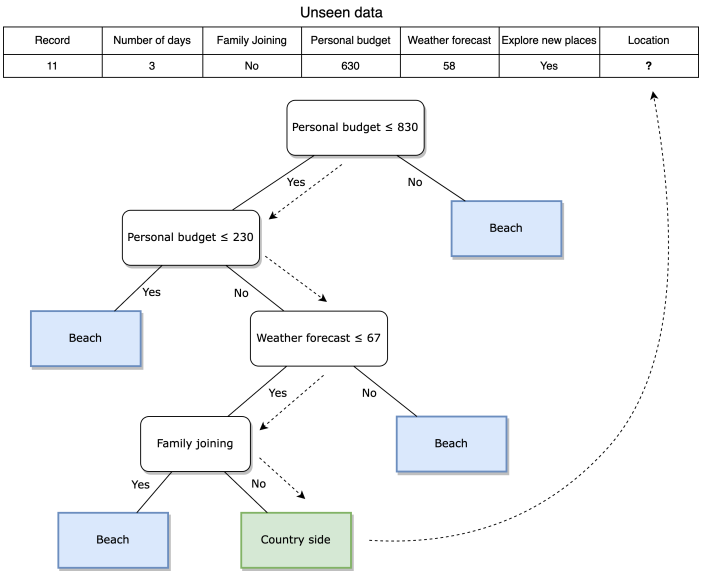

Figure 4: Determining the location of new vacation. The dashed lines represent the outcomes of applying various attribute tests on the unlabeled vacation. The vacation is eventually assigned to the class `Country side`.

3.2 How to Build a Decision Tree - Decision Tree Induction

The process of learning the structure of a decision tree, to construct the classifier, is called decision tree induction [4]. In principle, there are exponentially many decision trees that can be constructed from a given set of attributes. While some of the trees are more accurate than others, finding the optimal tree is computationally infeasible because of the exponential size of the search space. Nevertheless, efficient algorithms have been developed to induce a reasonably accurate decision tree in a reasonable amount of time. These algorithms usually employ a greedy strategy that grows a decision tree by making a series of locally optimum decisions about which attribute to use for partitioning the data. One such algorithm is **Hunt's algorithm**, which is the basis of many existing decision tree induction algorithms, including ID3, C4.5, and CART [2].

Hunt's Algorithm

In Hunt's algorithm, a decision tree is grown in a recursive fashion by partitioning the training records into successively purer subsets. Let D_t be the set of training records that are associated with node t and let the $y = y_1, y_2, ..., Y$ be the class labels. The following is a recursive definition of Hunt's algorith:

- Step 1: If all records in D_t belong to the same class y_i, then t is a leaf node labeled as y_i.
- Step 2: If D_t contains records that belong to more than one class, an **attribute test condition** is selected to partition the records into smaller subsets. A child node is created for each outcome of the test condition and the records in D_t are distributed to the children based on the outcomes. Hence we acquire $D_{t+1,left}$ and $D_{t+1,right}$
- Apply Step 1 and Step 2 to each child node.

To demonstrate the functioning of the algorithm, let's consider the task of predicting whether a loan applicant will repay their loan or default on their payments. Table 3 shows the training set for this problem. Each record in the training set contains the personal information of a borrower and a class label indicating whether they defaulted on their loan.

	Home Owner	Martial Status	Annual Income (k$)	Defaulted Borrower
0	1	single	125	No
1	0	married	100	No
2	0	single	70	No
3	1	married	120	No
4	0	divorced	95	Yes
5	0	married	60	No
6	1	divorced	220	No
7	0	single	85	Yes
8	0	married	75	No
9	0	single	90	Yes

Table 3: Training set for predicting borrowers who will default on loan payments.

Initially, the classification tree for this problem consists of a single node with the class label "Defaulted = No", since the majority of borrowers successfully repaid their loans (see Figure 5). This would imply that every new borrower will also repaid the loan which is a very weak claim, therefore the tree needs further refinement. This means that we skip the Step 1 and proceed to Step 2 and divide the records into smaller subsets based on a test condition related to the attribute `Martial status`. This splitting criterion is chosen based on its effectiveness and justification for choosing this attribute will be discussed later. For now, we will assume that this is the best criterion for splitting the data at this point.

Using Hunt's algorithm, we apply the recursive step to each child of the root node. From the given training set, we observe that all borrowers who are home owners have successfully repaid their loans. Consequently, the left child of the root becomes a leaf node labeled "Defaulted = No." For the right child, we continue applying Hunt's algorithm recursively until all the records in that subset belong to the same class. The resulting trees are shown in Figures 5 (c) and (d).

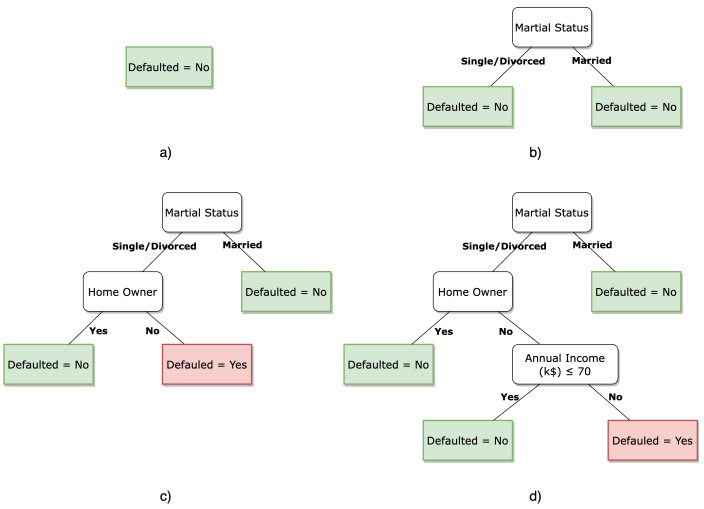

Figure 5: Hunt's algorithm for inducing decision trees

Design Issues of Decision Tree Induction

An effective learning algorithm for inducing decision trees must address the following two key issues:

- **Determining the optimal way to split the training records:** At each recursive step of the tree-growing process, the algorithm needs to select an attribute test condition that can effectively divide the records into smaller subsets. To accomplish this, the algorithm should provide a mechanism for specifying appropriate test conditions for different types of attributes. Additionally, an objective measure is required to evaluate the quality of each test condition and guide the selection of the most informative attribute for splitting.
- **Establishing the stopping criteria for the splitting procedure:** It is necessary to define a stopping condition to terminate the tree-growing process. One common strategy is to continue expanding a node until either all the records belong to the same class or all the records have identical attribute values. While these conditions are sufficient to stop any decision tree induction algorithm, it is also possible to impose additional criteria that allow for earlier termination.

3.3 Measures for Selecting the Best Split

The method used to define the best split makes different decision tree algorithms. There are many measures that can be used to determine the best way to split the records. These measures are defined in terms of the class distribution of the records before and after splitting. The best splitting is the one that has more purity after the splitting. If we were to split D_t into smaller partitions according to the outcomes of the splitting criterion, ideally each partition after splitting would be pure (i.e., all the records that fall into a given partition would belong to the same class). Instead of defining a split's purity, the impurity of its child node is used. There are a number of commonly used impurity measurements, but the main ones are *Entropy, Gini Index and Classification Error.*

Let $p_{i/t}$ denote the fraction of records belonging to class i at a given node t. We will define each impurity measurement:

- **Entropy:** measures the degree of uncertainty, impurity, or disorder. Entropy at node t is given with the formula:

$$E(t) = -\sum_{i=1}^{n} p_{i/t} \log_2(p_{i/t})$$

- **Gini Index:** also called Gini impurity, measures the degree of probability of a particular variable being incorrectly classified when it is chosen randomly. The degree of the Gini index varies between zero and one, where zero denotes that all elements belong to a certain class or only one class exists, and one denotes that the elements are randomly distributed across various classes. A Gini index of 0.5 denotes equally distributed elements into some classes. Gini index at node t is given with the formula:

$$GINI(t) = 1 - \sum_{i=1}^{n} p_{i/t}^2$$

- **Classification Error** measures the misclassified class labels and it is given with the formula:

$$Classificationerror(t) = 1 - max(p_{i/t})$$

Figure 6 compares the values of the impurity measures for binary classification problems for which class distribution at any node can be written as (p_0, p_1), where $p_1 = 1 - p_0$. We observe that all three measures attain their maximum value when the class distribution is uniform (i.e. when $p = 0.5$). The minimum values for the measures are attained when all the records belong to the same class (i.e., when p equals to 0 or 1).

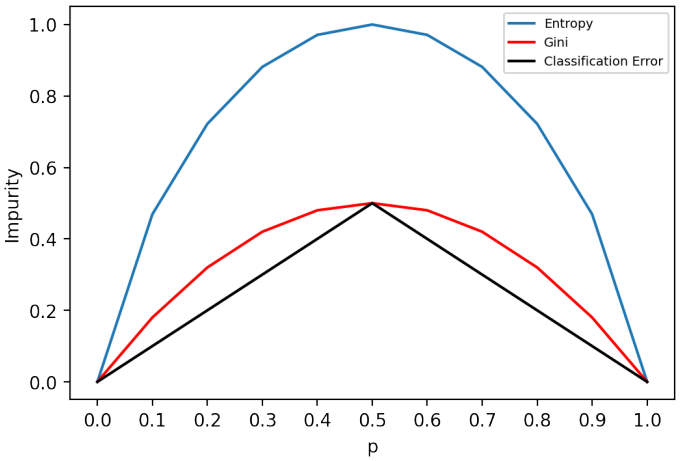

Figure 6: Comparison among the impurity measures for binary classification problems. p refers to the fraction of records that belong to one of the two classes.

To determine how well a test condition performs, we need to compare the degree of impurity of the parent node (before splitting) with the degree of impurity of the child nodes (after splitting). The larger their difference, the better the test condition. The gain, Δ, is a criterion that can be used to determine the goodness of a split:

$$\Delta = I(\text{parent}) - \sum_{j=1}^{k} \frac{N(v_j)}{N} I(v_j)$$

where $I()$ is the impurity measure of a given node, N is the total number of records at the parent node, k is the number of child nodes, and $N(v_j)$ is the number of records associated with the child node v_j. Decision tree induction algorithm choose a test condition that maximizes the gain Δ.

3.4 Algorithm for Decision Tree Induction

Algorithm 1 build(X, y, curr_depth $= 0$)

if unique(y) **then**
  return TreeNode($X, y,$ curr_depth $=$ curr_depth, leaf $=$ True)
end if
if len(X) $\geq$ min_samples **and** curr_depth $<$ max_depth **then**
  split_index, split_threshold, split_info_gain $\leftarrow$ best_split($X, y, candidates$)
  left_idxs, right_idxs $\leftarrow$ split(X[:, split_index], split_threshold)
  left $\leftarrow$ build(X[left_idxs, :], y[left_idxs], curr_depth $+ 1$)
  right $\leftarrow$ build(X[right_idxs, :], y[right_idxs], curr_depth $+ 1$)
  if curr_depth $= 0$ **then**
    return TreeNode($X, y,$ split_index, split_threshold, left, right, split_info_gain, curr_depth=0, root=True, class_to_int $=$ class_to_integer)
  end if
  return TreeNode($X, y,$ split_index, split_threshold, left, right, split_info_gain, curr_depth $=$ curr_depth)
end if
return TreeNode($X, y,$ curr_depth $=$ curr_depth, leaf $=$ True)

The algorithm in form of a `build` function recursively builds a decision tree by finding the best split at each node based on the information gain, until the stopping criteria (minimum number of samples or maximum depth) are reached. It constructs decision nodes and leaf nodes of the tree using the TreeNode class.

3.5 Implementing Decision Tree Algorithm

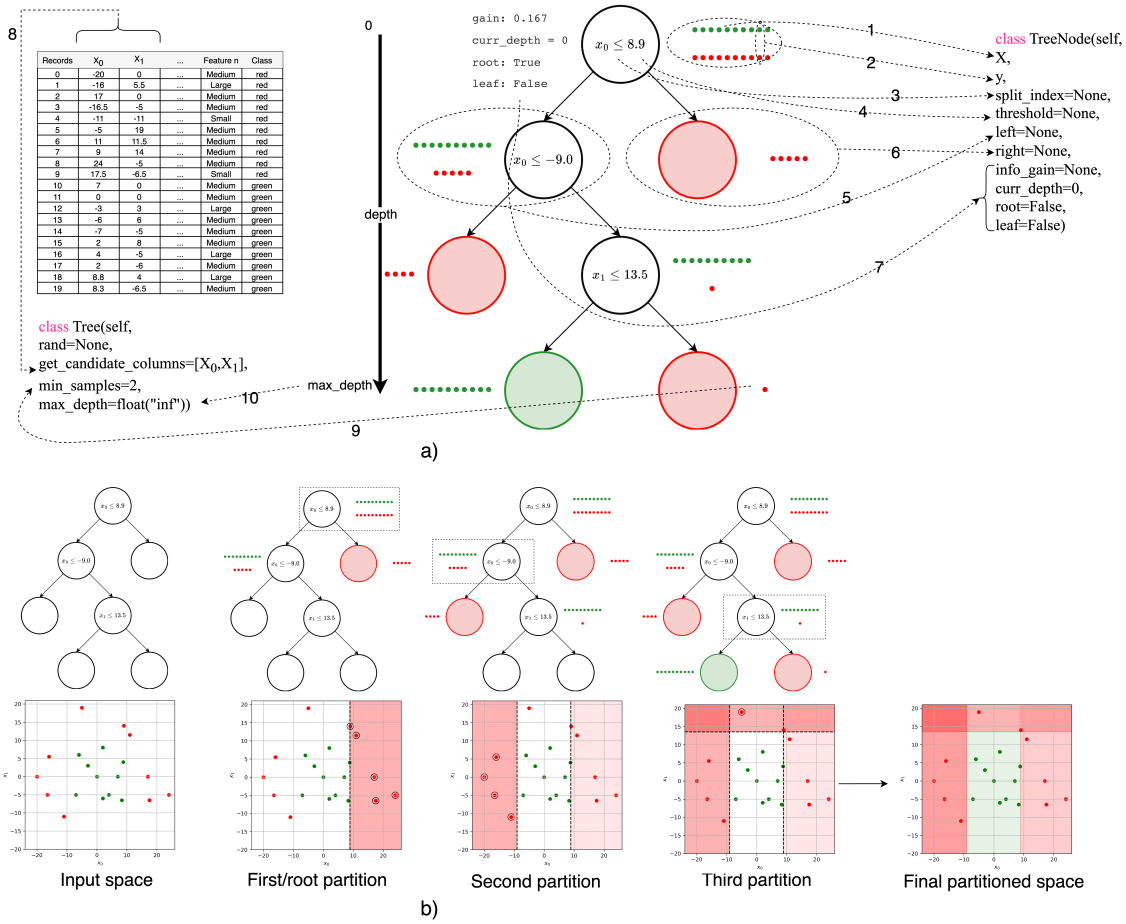

Figure 7: Implementing the Decison Tree Algorithm. Decision Tree algorithm is implemented using two classes: `TreeNode` and `Tree`.

Decision Tree algorithm is implemented using two classes: `TreeNode` and `Tree`. For better understanding of the logic behind the two classes look at the Figure 7 a). Here's an explanation of each class, their purpose, attributes and methods:

- `TreeNode` class:
  - *Purpose*: This class represents a node in the decision tree. It can either be a decision node (internal node) or a leaf node.
  - *Attributes*:
    - `split_index`: The index of the feature used for splitting at this node.
    - `threshold`: The threshold value used for splitting the feature.
    - `left`: The left child node. Takes the records if the condition is met.
    - `right`: The right child node. Takes the records if the condition is not met.
    - `info_gain`: The information gain achieved by the split at this node.
    - `leaf`: A boolean flag indicating if the node is a leaf node.
    - `class_to_int`: A dictionary mapping class labels to integer values.
    - `curr_depth`: The current depth of the node in the tree.
    - `X`: The subset of the input data associated with this node.
    - `y`: The labels associated with the subset of data at this node.
  - *Methods*:
    - `predict(X)`: This method is used to predict the class labels for a given set of input data `X` based on the decision tree. It recursively traverses the tree starting from the root node and makes predictions by following the appropriate path based on the feature values of the input data.
- `Tree` class:
  - *Purpose*: This class represents the decision tree itself and provides methods for building and using the tree.
  - *Attributes*:
    - `rand`: An instance of the `random.Random` class used for randomization.
    - `get_candidate_columns`: A function that determines the candidate columns (features) to consider for splitting.
    - `min_samples`: The minimum number of samples required to perform a split at a node.
    - `max_depth`: The maximum depth allowed for the tree.
  - *Methods*:
    - `build(X, y, curr_depth)`: This method is the main recursive function for building the decision tree. It takes the input data `X`, the labels `y`, and the current depth `curr_depth` as inputs. It recursively splits the data based on the best split at each node until a stopping criterion is reached (e.g., minimum samples or maximum depth). It returns the root node of a subtree.
    - `best_split(X, y, candidates)`: This method finds the best split for a given dataset. It takes the dataset `X`, the labels `y`, and a set of candidate feature indices `candidates` as inputs. It iterates over the candidate features and their possible threshold values to calculate the information gain for each split. It returns the index of the feature, the threshold value, and the information gain for the best split.
    - `information_gain(y, X_column, split_threshold)`: This method calculates the information gain for a given split. It takes the labels `y`, a single column `X_column` from the dataset, and a `split_threshold` as inputs. It uses the `split()` method to obtain the left and right indices based on the feature and threshold. It then calculates the information gain by subtracting the weighted sum of the Gini impurities of the left and right subsets from the Gini impurity of the original set. It returns the information gain.
    - `split(X_column, split_threshold)`: This method is used to split the data based on a given feature and threshold value. It takes a single column `X_column` from the dataset and a `split_threshold` as inputs. It creates two sets of indices: `left_idxs` for the data points where the feature value is less than or equal to the threshold and `right_idxs` for the data points where the feature value is greater than the threshold. It returns these indices.
    - `gini_index(y)`: This method computes the Gini index for a given set of labels `y`. It measures the impurity or disorder of the labels in the set. The Gini index is calculated by summing the squared probabilities of each class in the set. It returns the Gini index.
    - `to_int(y)`: This method converts string class labels to integer class labels. It takes the original labels `y` as input and returns the converted labels and a dictionary `class_to_int` mapping the original class labels to integer values.

Python code

```
class TreeNode:
    def __init__(self, X, y, split_index=None, threshold=None,
      left=None, right=None,
                 info_gain=None, curr_depth=0, root=False,
     leaf=False, class_to_int=None):

        #DECISION NODE
        self.split_index = split_index
        self.threshold = threshold
        self.left = left
        self.right = right
        self.info_gain = info_gain
        #LEAF NODE
        self.leaf = leaf
        #for BOTH NODES
        self.curr_depth = curr_depth
        self.X = X
        self.y = y

    def predict(self, X):
        '''Function for predicting classes'''
        if self.split_index is None:
            return np.array([np.argmax(np.bincount(self.y))
     for _ in range(X.shape[0])])
        pred = np.zeros(X.shape[0])
        left_idx = X[:, self.split_index] <= self.threshold
        pred[left_idx] = self.left.predict(X[left_idx])
        pred[~left_idx] = self.right.predict(X[~left_idx])
        return pred.astype(int)


class Tree:
    def __init__(self,
                 rand=None,
                 get_candidate_columns=all_columns,
                 min_samples=2,
                 max_depth=float("inf")):

        if rand is None:
            rand = random.Random(42)
        self.rand = rand
        self.get_candidate_columns = get_candidate_columns
        self.min_samples = min_samples
        self.max_depth = max_depth


    def build(self, X, y, curr_depth=0):
        '''Recursive function to build the tree'''
        if curr_depth == 0:
            y = self.to_int(y)

        candidates = np.array(self.get_candidate_columns(X,
     self.rand))
        self.rand.shuffle(candidates)

        if np.unique(y).shape[0] == 1:
            return TreeNode(X, y, curr_depth=curr_depth, leaf=
     True)

        if len(X) >= self.min_samples and curr_depth < self.
     max_depth:
            split_index, split_threshold, split_info_gain =
     self.best_split(X, y, candidates)

            left_idxs, right_idxs = self.split(X[:,
     split_index], split_threshold)

            #Recur left
            left = self.build(X[left_idxs,:], y[left_idxs],
     curr_depth + 1)
            #Recur right
            right = self.build(X[right_idxs,:], y[right_idxs],
      curr_depth + 1)

            if curr_depth == 0:
                return TreeNode(X, y, split_index,
     split_threshold,
                                left, right, split_info_gain,
     curr_depth=0, root=True,
                                class_to_int=class_to_integer)
            return TreeNode(X, y, split_index, split_threshold
     , left, right, split_info_gain, curr_depth=curr_depth)
        return TreeNode(X, y, curr_depth=curr_depth, leaf=True
     )

    def best_split(self, X, y, candidates):
        '''Function to find the best split'''
        split_index, split_threshold, split_info_gain = None,
     None, None
        max_info_gain = -float("inf")

        for feature_index in candidates:
            X_column = X[:, feature_index]
            possible_thresholds = np.unique(X_column)

            for i in range(len(possible_thresholds) - 1):
                threshold = (possible_thresholds[i] +
     possible_thresholds[i + 1]) / 2
                curr_info_gain = self.information_gain(y,
     X_column, threshold)
                if curr_info_gain > max_info_gain:
                    split_index = feature_index
                    split_threshold = threshold
                    split_info_gain = curr_info_gain
                    max_info_gain = curr_info_gain
        return split_index, split_threshold, split_info_gain

    def information_gain(self, y, X_column, split_threshold):
        '''Function to calculate the information gain of the
     split'''
        left_idxs, right_idxs = self.split(X_column,
     split_threshold)
        if len(left_idxs) == 0 or len(right_idxs) == 0:
            return 0
        n = len(y)
        n_l, n_r = len(left_idxs), len(right_idxs)
        weight_l = n_l / n
        weight_r = n_r / n
        gain = self.gini_index(y) - (weight_l*self.gini_index(
     y[left_idxs]) + weight_r*self.gini_index(y[right_idxs]))
        return gain

    def to_int(self, y):
        '''Function to convert string classes to integer
     classes'''
        y = y.copy()
        unique_classes = np.unique(y)
        for i, cl in enumerate(unique_classes):
            y[y==cl] = i
        return y.astype(int).flatten()

    def split(self, X_column, split_threshold):
        '''Function to split the data'''
        left_idxs = np.argwhere(X_column <= split_threshold).
     flatten()
        right_idxs = np.argwhere(X_column > split_threshold).
     flatten()
        return left_idxs, right_idxs

    def gini_index(self, y):
        '''Function to compute Gini index'''
        class_labels = np.unique(y)
        gini = 0
        for cls in class_labels:
            p_cls = len(y[y == cls]) / len(y)
            gini += p_cls**2
        return 1 - gini
def all_columns(X, rand):
    '''Function returns the range of the features'''
    return range(X.shape[1])
```

Algorithm 2 Best Split Search Algorithm

Procedure `best_split` ($X, y,$ *candidates*)
- split_index, split_threshold, split_info_gain ← None, None, None
- max_info_gain ← $-\infty$
- **foreach** *feature_index* ***in*** *candidates* **do**
  - X_column ← X[:, feature_index]
  - possible_thresholds ← unique(X_column)
  - **foreach** i ***in*** 0 **to** *len*(*possible_thresholds*) $- 2$ **do**
    - threshold ← (possible_thresholds[i] + possible_thresholds[$i + 1$])/2
    - curr_info_gain ← InformationGain(y, X_column, threshold)
    - **if** *curr_info_gain* $>$ *max_info_gain* **then**
      - split_index ← feature_index
      - split_threshold ← threshold
      - split_info_gain ← curr_info_gain
      - max_info_gain ← curr_info_gain
- **return** split_index, split_threshold, split_info_gain

Function Description

The `best_split` function is used to find the best split for a given dataset when constructing a decision tree. It takes the dataset X, the labels y, and a set of candidate feature indices $candidates$ as inputs. The primary goal is to identify the feature and threshold that result in the highest information gain when splitting the data.

Algorithm Steps

Here's how the algorithm works:

1. **Initialization:**
   - `split_index`, `split_threshold`, and `split_info_gain` are initialized to `None`, indicating that no split has been found yet.
   - `max_info_gain` is initialized to negative infinity, ensuring that any valid information gain will be greater than this initial value.
2. **Iterate Over Candidate Features:**
   - The algorithm iterates through each candidate feature, which are specified by the `candidates` input.
3. **Identify Possible Thresholds:**
   - For each feature, it extracts the values of that feature (`X_column`) from the dataset.
   - It calculates `possible_thresholds` by finding the unique values in `X_column`. These unique values are potential split points.
4. **Iterate Over Possible Thresholds:**
   - For each feature, the algorithm iterates through the `possible_thresholds` to find the best split.
   - It calculates a `threshold` as the midpoint between consecutive unique values in `possible_thresholds`. This step considers midpoints as potential split thresholds.
5. **Calculate Information Gain:**
   - For each candidate feature and threshold combination, the algorithm calculates the information gain (`curr_info_gain`) using the `information_gain` function. Information gain measures the reduction in impurity or entropy after the split.
6. **Update Best Split:**
   - If the calculated `curr_info_gain` is greater than the current `max_info_gain`, it updates the following:
     - `split_index`: The index of the feature that results in the best split so far.
     - `split_threshold`: The threshold value that produces the best split.
     - `split_info_gain`: The information gain associated with the best split.
     - `max_info_gain`: Updated to the new maximum information gain.
7. **Return Best Split:**
   - Once all candidate features and thresholds have been evaluated, the function returns the following:
     - `split_index`: The index of the feature that provides the best split.
     - `split_threshold`: The threshold value associated with the best split.
     - `split_info_gain`: The information gain achieved by the best split.

Splitting Method

The algorithm considers midpoints between consecutive unique values in each candidate feature as potential split thresholds. These midpoints represent possible ways to divide the feature values into two groups.

4 Time Complexity

4.1 Time Complexity of the Prediction Step

Time complexity for the prediciton step is simply equal to the depth of a decision tree.

- Best case - balanced tree
  - Under the assumption that a decision tree is a balanced binary decision tree, the final tree will have a depth of $\log_2 n$, where n is the number of examples in the training set.
  - In a balanced binary tree, we are effectively "halving" the number of remaining samples at each step as we move down the tree, from the root to a leaf.
  - Therefore, the maximum number of decisions that must be made to reach a leaf node (i.e., to make a prediction) from the root is $\log_2 n$.
  - As a result, it is clear that the time complexity for the prediction step is $O(\log n)$.
- Worst case - highly unbalanced tree
  - Each split simply splits data in 1 and $n-1$ examples, see Figure 8. In this case the depth of a tree is n, hence the time complexity is $O(n)$.

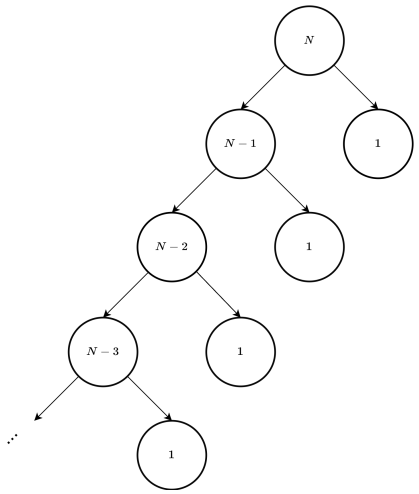

Figure 8: Unbalanced tree

4.2 Time Complexity of the Training Step

- Determining the runtime complexity of decision tree training is less straightforward and can vary significantly based on the algorithm choice and implementation. Specifically for our implementation (where we sort the values for each feature at each node), the time complexity is given with:

$$O\left((\text{sorting time} + \text{best split}) \cdot \#\text{ nodes}\right) \tag{1}$$

- **Sorting Features and Best Split Search**
  - Optimal binary splits on continuous features are typically on the boundary between adjacent examples with different class labels. Sorting the values of continuous features is a key step in determining a decision threshold efficiently.[3]
  - If we have n examples, sorting a single feature has a time complexity of $O(n \log n)$.F or m features, this becomes $O(m \cdot n \log n)$.
  - Best split search requires checking every value (possible unique threshold) for every feature. Thus it has time complexity of $O(m \cdot n)$
  - Adding these two we get:

$$O(m{\cdot}n \log n{+}m{\cdot}n) = O(m{\cdot}n(\log n{+}1)) = O(m{\cdot}n \log n)$$

- **Best case - balanced tree** (number of nodes $= \log n$)
  - Assuming that we are performing binary splits in a balanced manner, the runtime of the decision tree construction is $O(m \cdot n \log^2 n)$, where m is the number of features and n is the number of examples in the training set.
  - *Proof*: The average number of nodes for a balanced tree is $\log n$, therefore we get:

$$O(m \cdot n \log n) \cdot O(\log n) = O(m \cdot n \log^2 n)$$

  - Additionally we consider the worst case but for the balanced tree, in which the number of splitting nodes is

$$n - 1 \tag{2}$$

    Therefore the time complexity is given with:

$$m \cdot n \log n \cdot (n-1) = m \cdot n^2 \log n \rightarrow O(m \cdot n^2 \log n)$$

    We can see that worst case for balanced tree is the same as for highly unbalanced tree.

- **Worst case - unbalanced tree** (number of nodes $= n$)
  - Assuming we have an unbalanced tree the runtime of the decision tree construction is: $O(m \cdot n^2 \log n)$, where m is the number of features and n is the number of examples in the training set.
  - *Proof*: The number of splitting nodes is $n-1$, therefore the time complexity is:

$$O(m \cdot n \log n) \cdot O(n) = O(m \cdot n^2 \log n)$$

More Efficient Implementation:

- Many implementations, such as scikit-learn, use efficient caching tricks to keep track of the general order of indices at each node. This means that the features do not need to be re-sorted at each node.
- As a result, the time complexity of these efficient implementations is reduced to

$$O(m \cdot n \log n) \tag{3}$$

Proof for 2

d	m = # of nodes	nodes by depth
0	$1 = 2^1 - 1$	1
1	$3 = 2^2 - 1$	$1 + 2$
2	$7 = 2^3 - 1$	$1 + 2 + 4$
3	$15 = 2^4 - 1$	$1 + 2 + 4 + 8$
. . .		
d	$2^{d+1} - 1$	$1 + 2 + ... + 2^d$

Table 4

[3] Sorting the feature values makes it easier to determine the best split for the data. The sorted values help you identify potential split points quickly and accurately. By efficiently finding the threshold that minimizes impurity or maximizes information gain, you can create more effective and predictive decision trees.

By looking at the table 4 we can see that the total number of nodes is given with:

$$m = 2^{d+1} - 1 \tag{4}$$

As mentioned, in balanced binary tree we assume that we are "halving" the number of remaining samples at each step until we arrive to leaf nodes, which count n. (every element of n elements is assumed to be of different class, thus we end up with n leaves). Therefore, the maximum depth is given with $d = \log_2 n$. By plugging this into 4 and subtracting the number of leaf nodes we get the number of splitting (decision nodes):

$$m_{splitting} = 2^{\log_2 n+1} - 1 - n = 2n - 1 - n = n - 1$$

Figure 9 shows the example for $n = 16$.

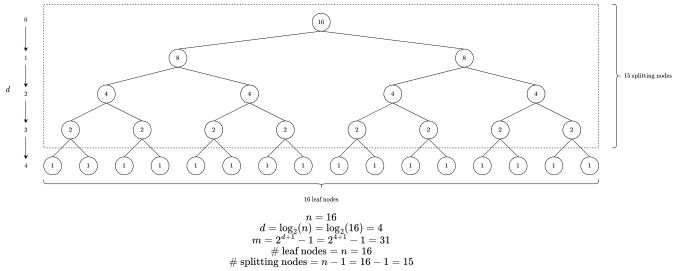

Figure 9: Balanced binary tree for $n = 16$

Proof for 3

Considering that sorting is done only once we have:

$$m \cdot n \log n + \log n = (mn + 1) \cdot \log n = m \cdot n \log n$$

4.3 Summary:

- The time complexity for the **prediction step** of a decision tree is $O(\log n)$, under the assumption of a balanced binary tree.
- The time complexity for the **training step** of a decision tree can vary, but it is generally $O(m \cdot n^2 \log n)$ without optimizations, and can be reduced to $O(m \cdot n \log n)$ with efficient implementations. Our implementation has time complexity of $O(m \cdot n \log^2 n)$.

References

[1] Abhijeet Godase and Vahida Attar. "Classification of data streams with skewed distribution". In: *ACM International Conference Proceeding Series* (May 2012). DOI: 10.1109/EAIS.2012.6232821.

[2] Gangmin Li. *Do A Data Science Project in 10 Days*. https://bookdown.org/gmli64/do_a_data_science_project_in_10_days/. Accessed on August 15, 2023. 2021.

[3] F. Pedregosa et al. "Scikit-learn: Machine Learning in Python". In: *Journal of Machine Learning Research* 12 (2011), pp. 2825–2830. URL: https://scikit-learn.org/stable/modules/tree.html.

[4] Lior Rokach and Oded Maimon. "Decision Trees". In: vol. 6. Jan. 2005, pp. 165–192. DOI: 10.1007/0-387-25465-X_9.