MEDNARODNA PODIPLOMSKA ŠOLA JOŽEFA STEFANA

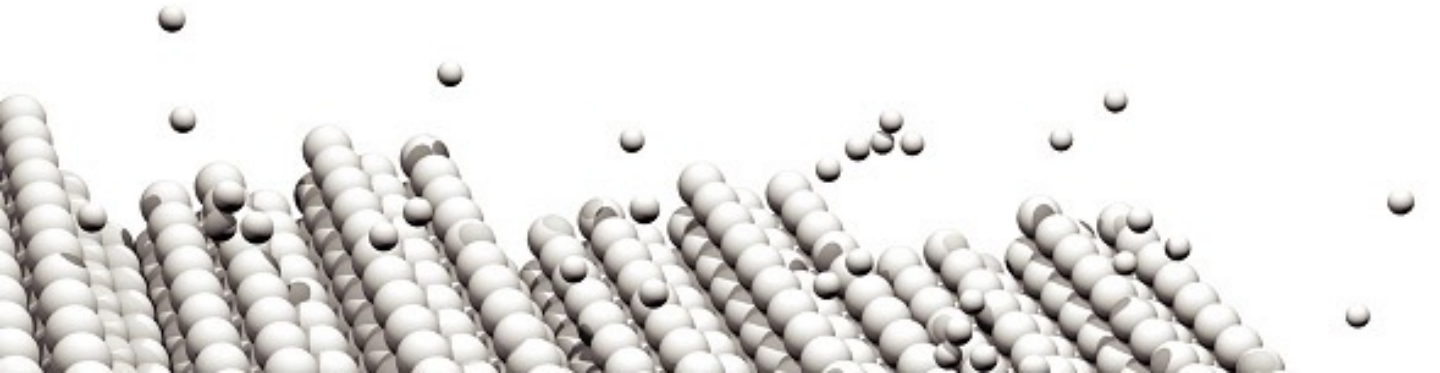
JOŽEF STEFAN INTERNATIONAL POSTGRADUATE SCHOOL

Programming, Data Structures and Algorithms
Seminar presentation

Binary Decision Tree Classifier

Student: Davud Topalović

Professor: doc. dr. Anton Biasizzo

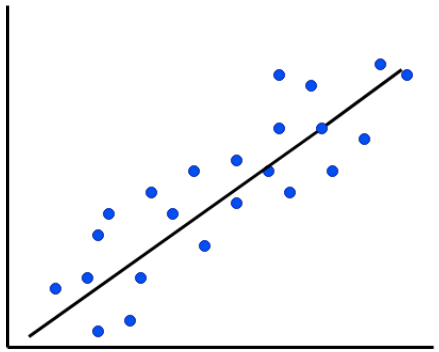


Introduction

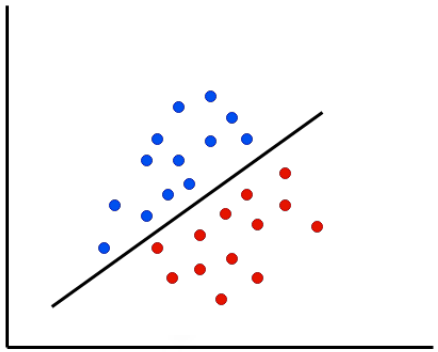
- *A Decision Tree Classifier (DTC) is a non-parametric, supervised learning algorithm that is used for both classification and regression tasks.*
- Classification - categorizing or assigning labels to input data based on their characteristics or features.
- Regression – statistical approach of predicting a continuous or discrete output variable based on the input data.

Supervised Learning

(modeling the relationship between the independent variables and dependent variable (label))



Regression



Classification

Classification

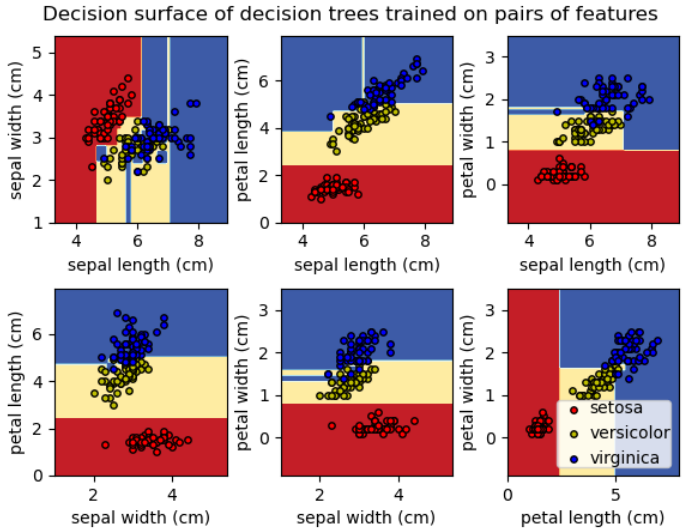
- Definition 2.1 (Classification). Classification is the task of learning a target function f that maps each attribute set X to one of the predefined class labels y.

- target function f = classification model

- Descriptive Modeling - explanatory tool that provides an insight in feature importance
- Predictive Modeling – predicting class labels of new unseen data points

Decision Tree for Classification

- One of the most intuitive method for data classification
- “Greedy” algorithm that hierarchically partitions the input space a sub-spaces associated with a class labels.



Partitioning input space, source: https://scikit-learn.org/stable/auto_examples/tree/plot_iris_dtc.html

- But what’s the intuition behind decision trees and how do they work?

Intuition and working principle

- *Based on the attribute values such as Number of days or Weather forecast, how can we determine the location of the vacation?*

- Start with posing a question about specific attribute:
  - Was the weather always hot when the vacation took place on the beach?
  - what's the personal budget? If it's enough is the location beach or countryside?
  - Is the family joining? Beach is better with famiy?

Posing carefully selected questions can progressively refine the prediction of the location.

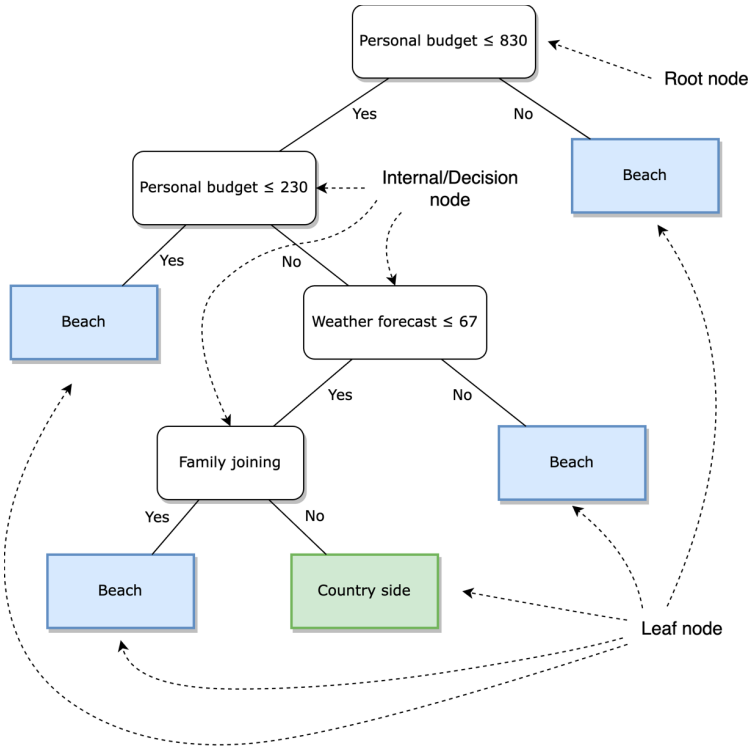
The series of questions and their potential answers can be organized in the form of a tree, *decision tree.*

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

Intuition and working principle

- The binary tree has 3 types of nodes:
  - **root node** that has no incoming edges and zero or two outgoing edges.
  - **Internal or decision nodes,** each of which has exactly one incoming edge and two outgoing edges.
  - **Leaf or terminal nodes**, each of which has exactly one incoming edge and no outgoing edges.



- Each leaf node is assigned a class label. The non terminal (impure) 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.
- **Binary Decision Tree**: each decision node has two child nodes (left and right).

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

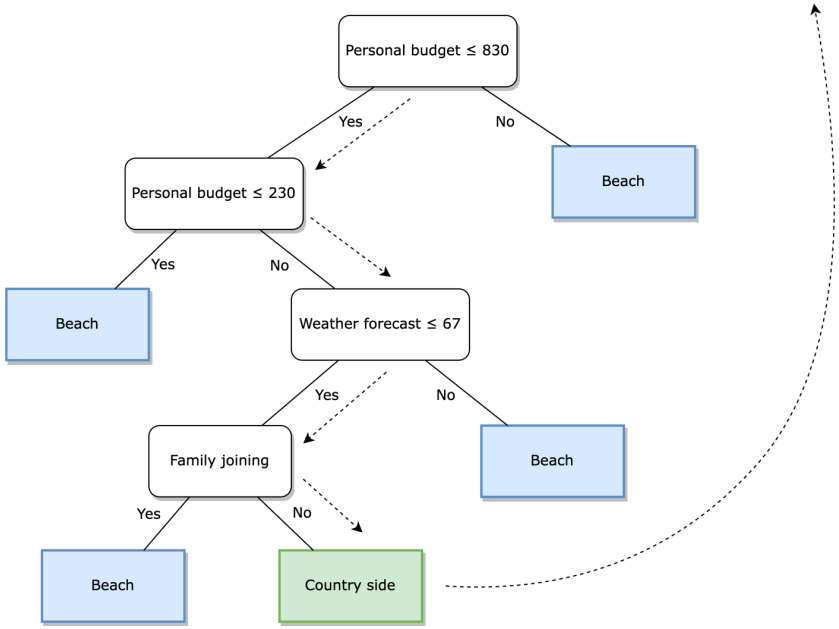
Intuition and working principle

- **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.

Unseen data

Record	Number of days	Family Joining	Personal budget	Weather forecast	Explore new places	Location
11	3	No	630	58	Yes	?



Building a Decision Tree

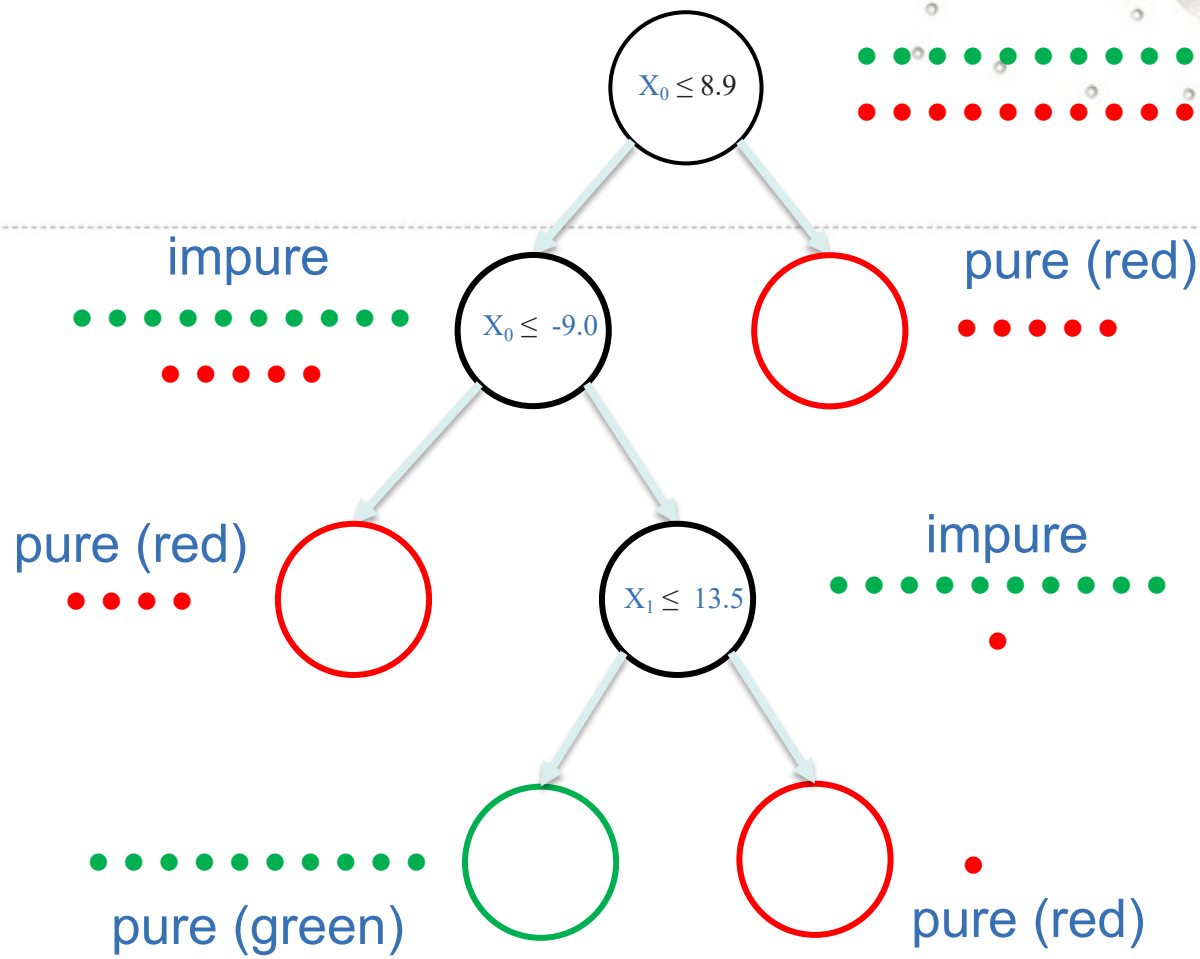
- The process of learning the structure of a decision tree is called **decision tree induction**.
- Exponentially many decision trees that can be constructed from a given set of attributes.
- Nevertheless, efficient algorithms have been developed to induce a reasonably accurate decision tree in a reasonable 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

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 = y1, y2, …, 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.



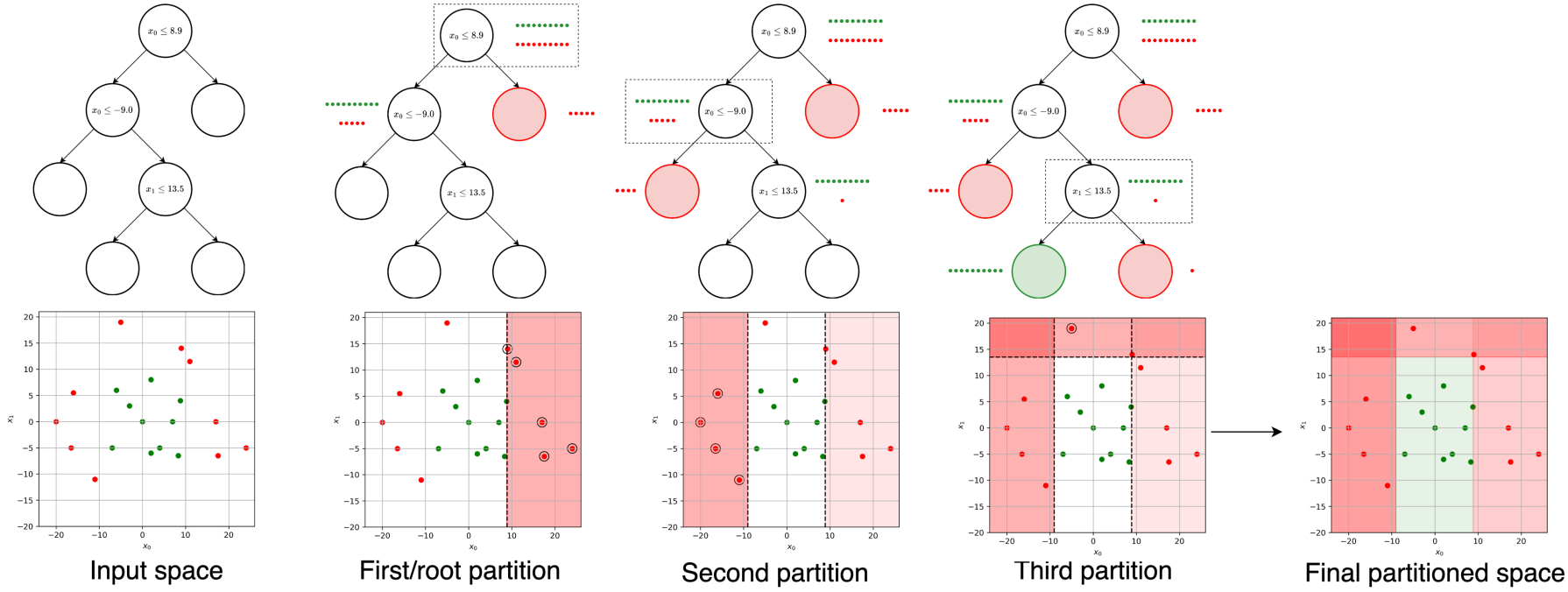
Hunt's Algorithm

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.

Building a Decision Tree



b)

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 Dt 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 $D_{t+1,\text{left}}$ and $D_{t+1,\text{right}}$

Apply Step 1 and Step 2 to each child node

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)

Decision Tree Induction

- **Design Issues of Decision Tree Induction**

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

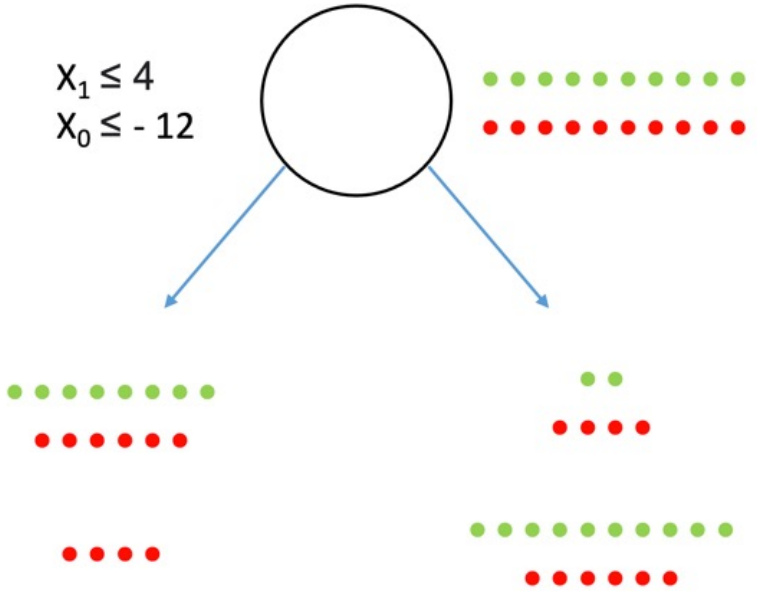
Determining the optimal way to split the training records: At each recursive step of the tree-growing process, the algorithm needs to select an attribute test condition that can effectively divide the records into smaller subsets. To accomplish this, the algorithm should provide a mechanism for specifying 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.

Measure for Selecting the Best Split

- How does an algorithm pick the best split? Answer: It calculates the impurity of the parent node and each possible set of child nodes. Then it takes the split with the highest purity gain.



- How do we measure the impurity of the node?

Measure 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.

- Commonly used impurity measurements are Entropy, Gini Index and Classification Error

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.

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

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

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

Measure for Selecting the Best Split

Gini Index: also 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:

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

- 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:

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

Measure for Selecting the Best Split

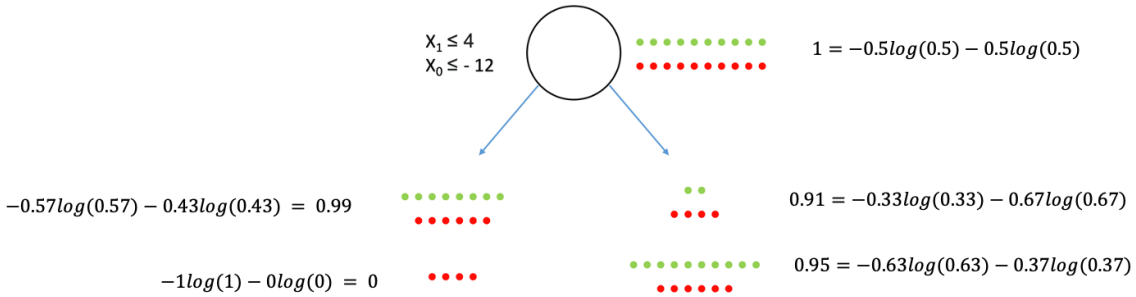
$$GINI(t) = 1 - \sum_{i=1}^{n} p_{i/t}^2$$

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

```python
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
```

```python
def information_gain(self, y, X_column, split_threshold):
    "function to calculate the information gain of the split"

    #generate split, that is get the left and right indices
    left_idxs, right_idxs = self.split(X_column, split_threshold)

    #if either left or right indices are zero, there is no information gain because we end up with
    #one empty node and the same node we perfomed a split on.
    if len(left_idxs) == 0 or len(right_idxs) == 0:
        return 0

    #calculate the information gain of the split with gini index (Gini impurity)
    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 information gain
    return gain
```

Measure for Selecting the Best Split



We can notice that the state with the 4 red points gives the minimum entropy, that's why we call it a pure node. Now to find the information gain corresponding to split we need to subtract the combined entropy of the child nodes from the entropy of the parent node. The formula below describes this action.

$$IG = E(parent) - \sum_i w_i \cdot E(child_i) \tag{2}$$

$$w_i = weight\ of\ child\ i\ (relative\ size\ of\ a\ child\ with\ respect\ to\ the\ parent\ |$$

Let's now calculate the information gain:

$$IG_1 = 1 - \frac{14}{20} \cdot 0.99 - \frac{6}{20} \cdot 0.91 = 0.034 \tag{3}$$

$$IG_2 = 1 - \frac{4}{20} \cdot 0 - \frac{16}{20} \cdot 0.95 = 0.24 \tag{4}$$

Best Split Search

Algorithm 2 Best Split Search Algorithm

Procedure `BestSplit` $(X, y, candidates)$

```
split_index, split_threshold, split_info_gain ← None, None, None
max_info_gain ← −∞
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
```

Best Split Search

For feature_index in [1,0]:

feature_index = 1

x_1 = [0. , 5.5, 0. , -5. , -11. , 19. , 11.5, 14. , -5. , -6.5, 0. , 0. , 3. , 6. , -5. , 8. , -5. , -6. , 4. , -6.5]

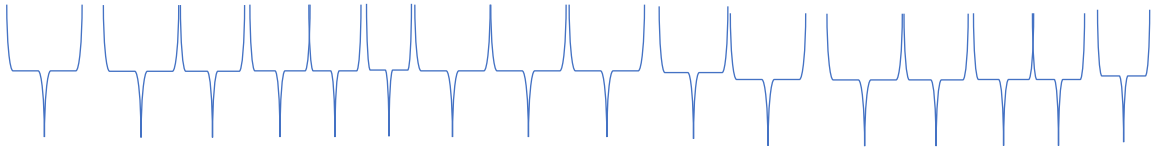
x_1_unique = [-11. , -6.5, -6. , -5. , 0. , 3. , 4. , 5.5, 6. , 8. , 11.5, 14. , 19.]

8.75, -6.25, -5.5, -2.5, 1.5, 3.5, 4.75, 5.75, 7.0, 9.75, 12.75, 16.5 = possible thresholds → calculate information gain

→ max info_gain of 0.088 for 9.75

feature_index = 0

x_0 = [-20. , -16. , 17. , -16.5, -11. , -5. , 11. , 9. , 24. , 17.5, 7. , 0. , -3. , -6. , -7. , 2. , 4. , 2. , 8.8, 8.3]



-18.25, -16.25, -13.5, -9.0, -6.5, -5.5, -4.0, -1.5, 1.0, 3.0, 5.5, 7.65, 8.55, 8.9, 10.0, 14.0, 17.25, 20.75 = possible thresholds

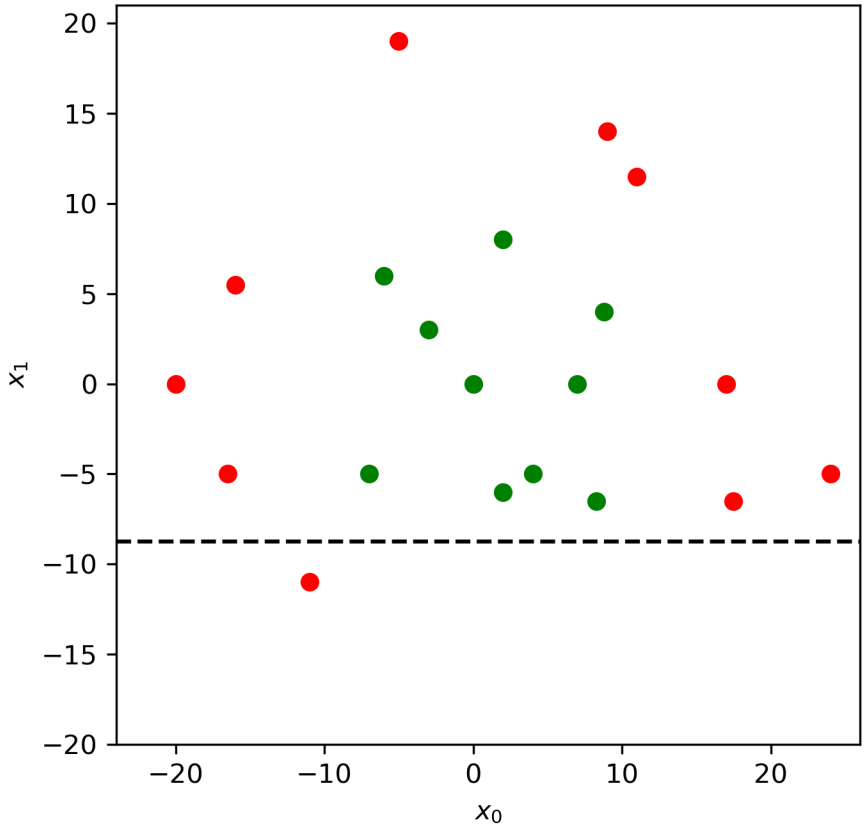
→ calculate information gain

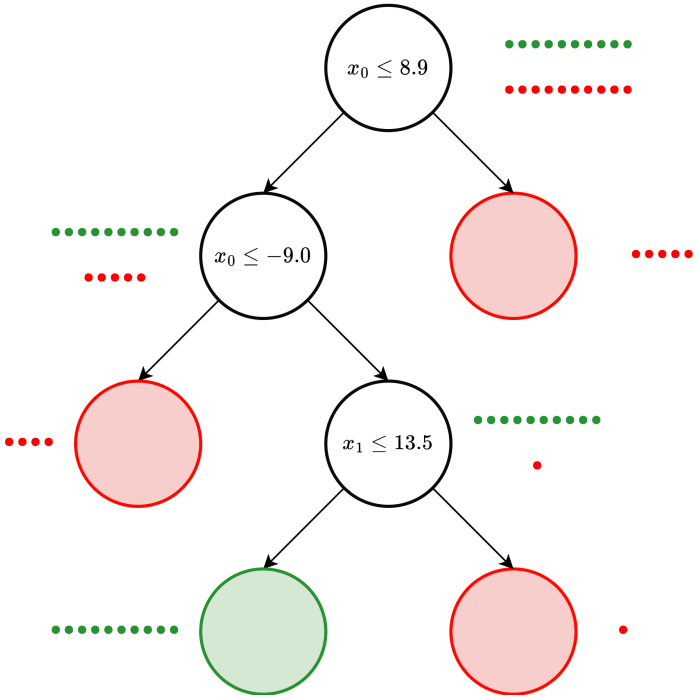
→ max info_gain of 0.167 for 8.9

Return:

- split_index = 0
- split_threshold = 8.9
- split_info_gain = 0.167

Best Split Search





Time complexity

Algorithm 2 Best Split Search Algorithm

```
Procedure BestSplit(X, y, candidates)
    split_index, split_threshold, split_info_gain ← None, None, None
    max_info_gain ← −∞
    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
```

$$O(m) \cdot (O(n \log n + O(n))$$

$$O(m \cdot n \log n)$$

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

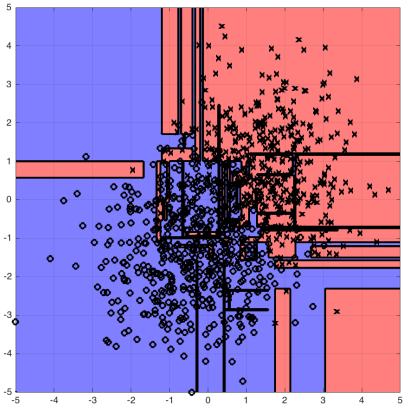
#nodes depends on the data:

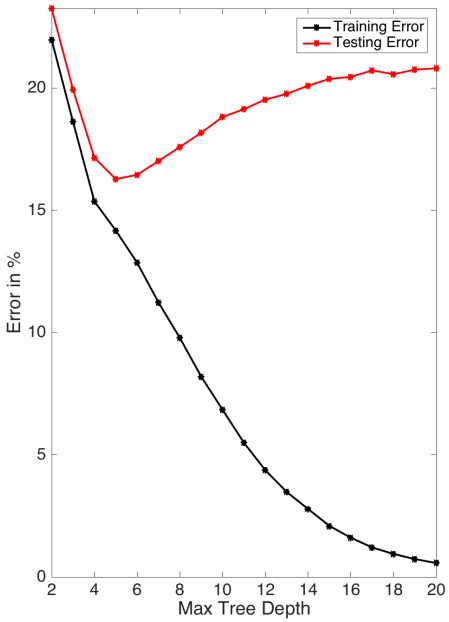
1. Best case – balanced tree #nodes = log(n) → $O(m \cdot n \log^2 n)$
2. Worst case – highly unbalanced tree #nodes = n-1 → $O(m \cdot n^2 \log n)$
3. Most effective way – sort only once → $O(m \cdot n \log n)$

Overfitting

- Overfitting refers to the condition when the model completely fits the training data but fails to generalize the testing unseen data. Overfit condition arises when the model memorizes the noise of the training data and fails to capture important patterns. A perfectly fit decision tree performs well for training data but performs poorly for unseen test data.
- Solution:

  1. Pre-Pruning - Cross validation – grid search for best set of hyperparameters
  2. Post-pruning technique allows the decision tree model to grow to its full depth, then removes the tree branches to prevent the model from overfitting.
- 3. Random Forest. - bootstrapping multiple decision trees

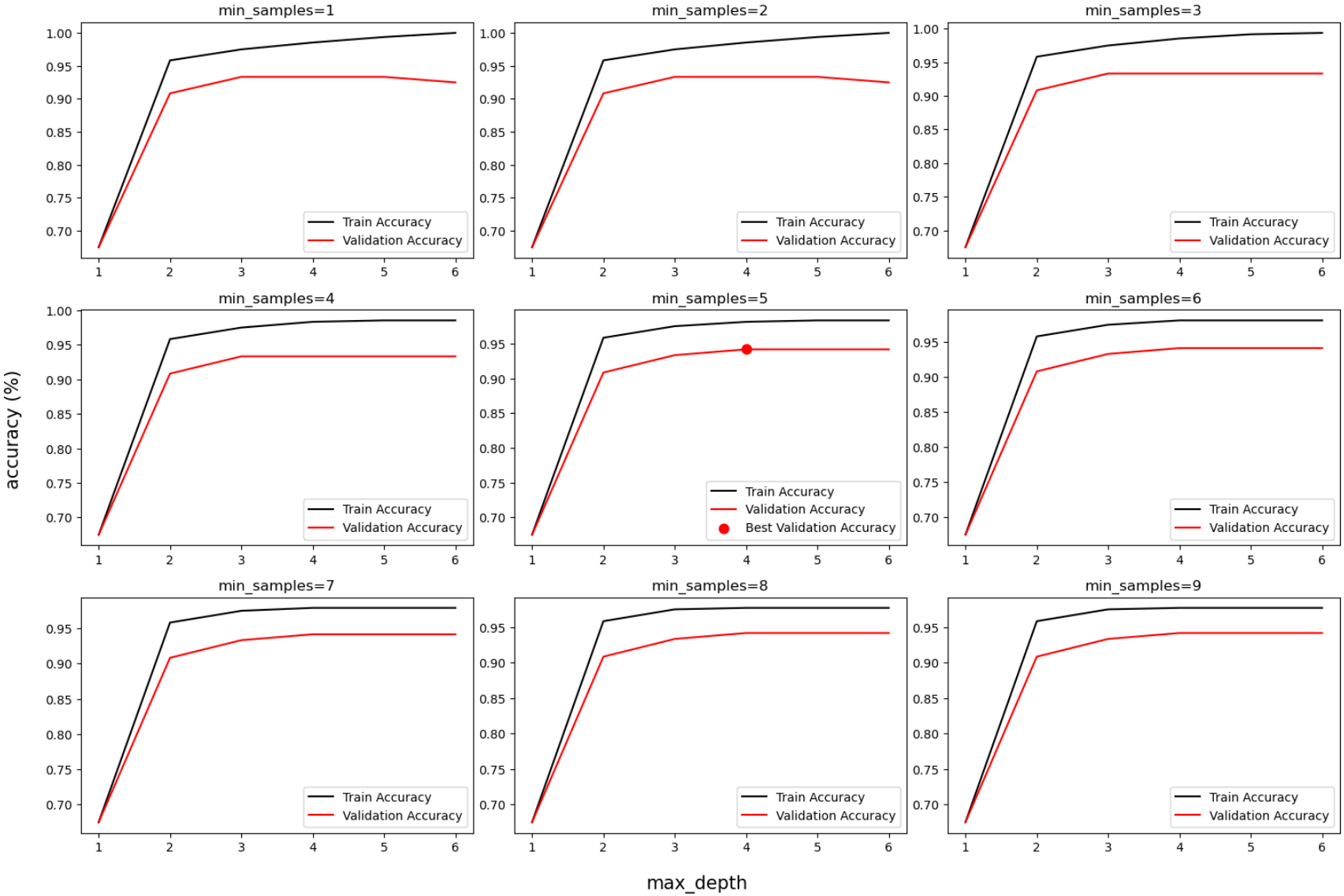




Overfitting

(source: https://towardsdatascience.com/3-techniques-to-avoid-overfitting-of-decision-trees-17d3d985a09)

Cross Validation for Iris dataset classifiaction



Thank you for your attention!