Decision Tree

Agenda

- Introduction to Decision Tree
  - Classification
  - Decision Trees
- Why to choose Decision Tree over Logistic Regression?
  - Challenges with Logistic Regression
  - Motivation for Decision Trees
- Key concepts in Decision Tree construction
  - Root Node
  - Decision Nodes
  - Leaf Nodes
  - Sub-Tree
  - Pruning
  - Branch/Sub-Tree
  - Parent and Child Node
- How do Decision Tree algorithms work?
- Impurity Metrics
  - impurity Measurement
    - Entropy
    - Gini Index
    - Comparison between Entropy and Gini Index
    - Gini Impurity
    - Choice between Entropy and Gini Impurity
- Use Case
  - Problem Statement
  - Exploring Data
    - Data Preprocessing
      - Remove Irrelevant Variables
    - Encoding
      - Label Encoding for Binary Variables
      - One-Hot Encoding for Categorical Variables with Multiple Levels

- Model Prepration
  - Target Encoding for high cardinality variables
  - Applying SMOTE for handling class imbalance
- Implementation of Decision Tree as a classifier
  - Changes with Decision Trees
    - Overfitting
    - Greedy Nature
  - Pruning Techniques
    - Post-Pruning
    - Pre-Pruning
    - Pre-Pruning Implementation
  - Variable Importance

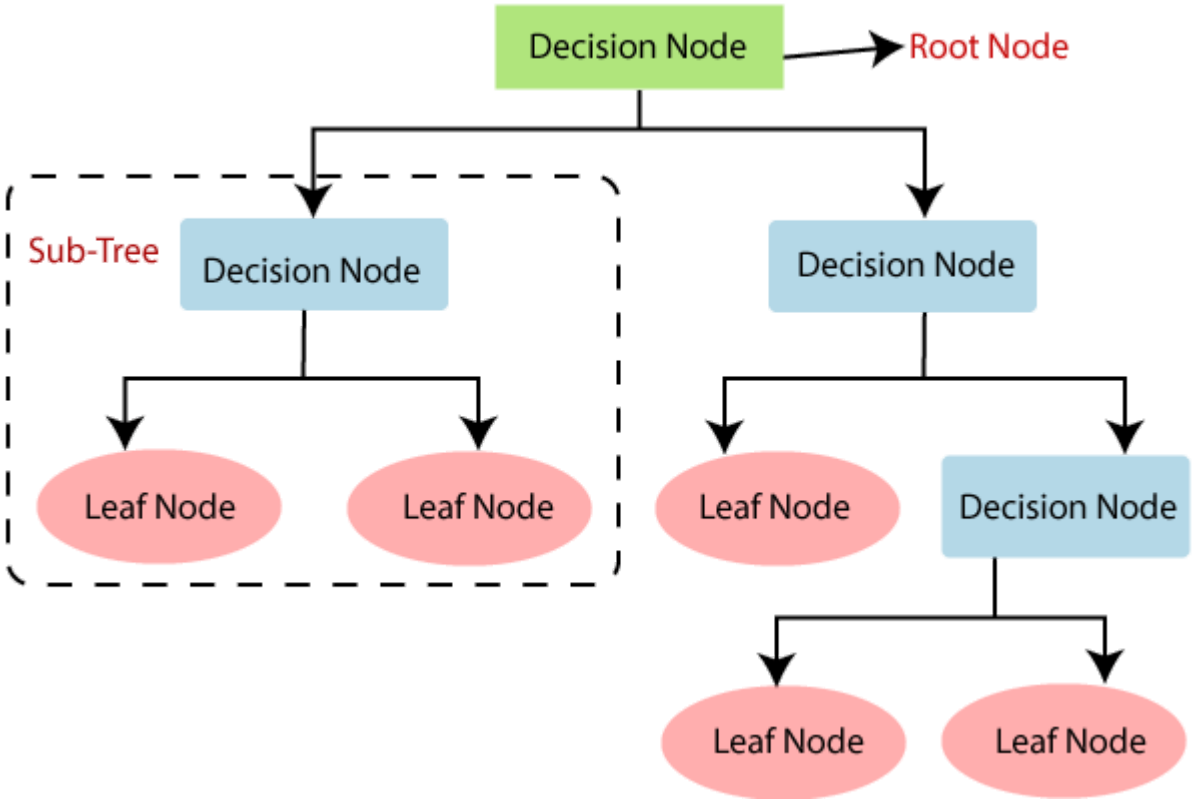
Introduction to Decision Tree

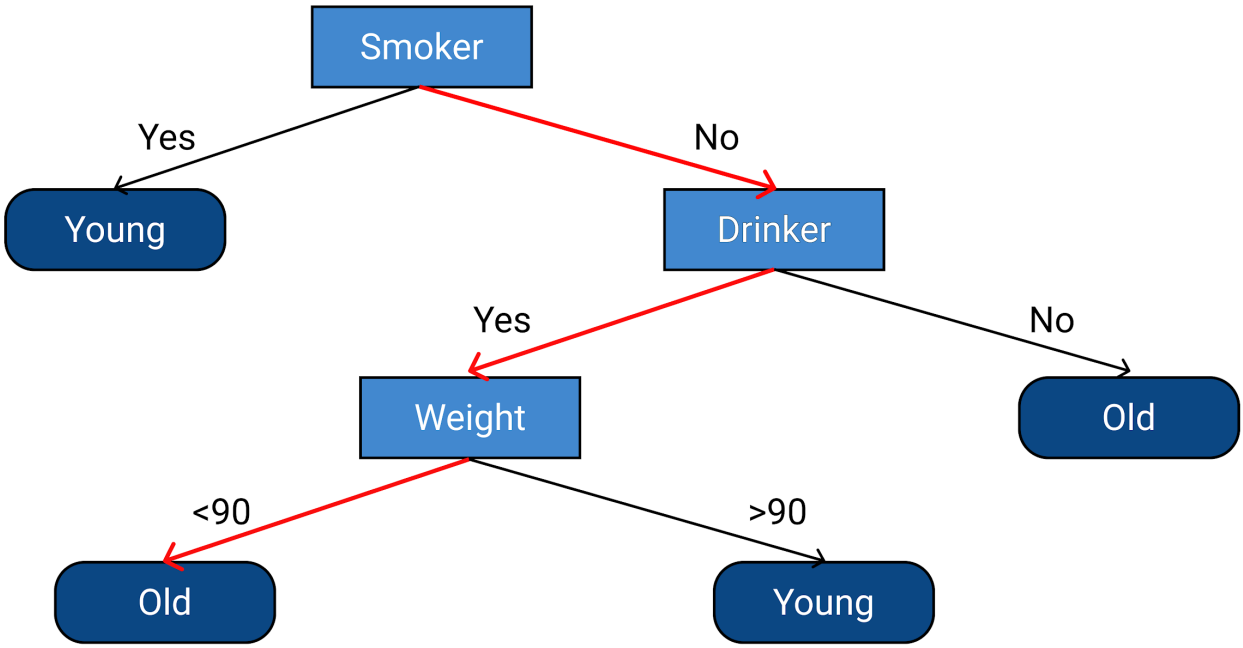
Classification

- Classification is a fundamental task in machine learning where the goal is to assign predefined categories or classes to input data based on certain features or attributes.
- This process is essential for various applications such as image recognition, spam detection, medical diagnosis, and more.
- In classification, the algorithm learns from labeled data, where each data point is associated with a class label, and then makes predictions on new, unseen data.

Decision Trees

- Decision trees are a classical and intuitive approach to classification problems.
- They are particularly effective when the relationship between input features and the target class is non-linear or complex.
- A decision tree is a hierarchical structure consisting of nodes that represent decisions or splits based on feature values, and branches that represent the possible outcomes of these decisions.
- At each node, the algorithm chooses the feature that best separates the data into different classes, leading to a tree-like structure where each leaf node represents a class label.





Jonas' Classification

- How Decision Trees Work
  - The algorithm works by selecting the feature that best splits the data. It evaluates each feature's effectiveness in reducing uncertainty or impurity in the dataset.
    - Classification: Entropy or Gini Index is used to measure the impurity in classification tasks.

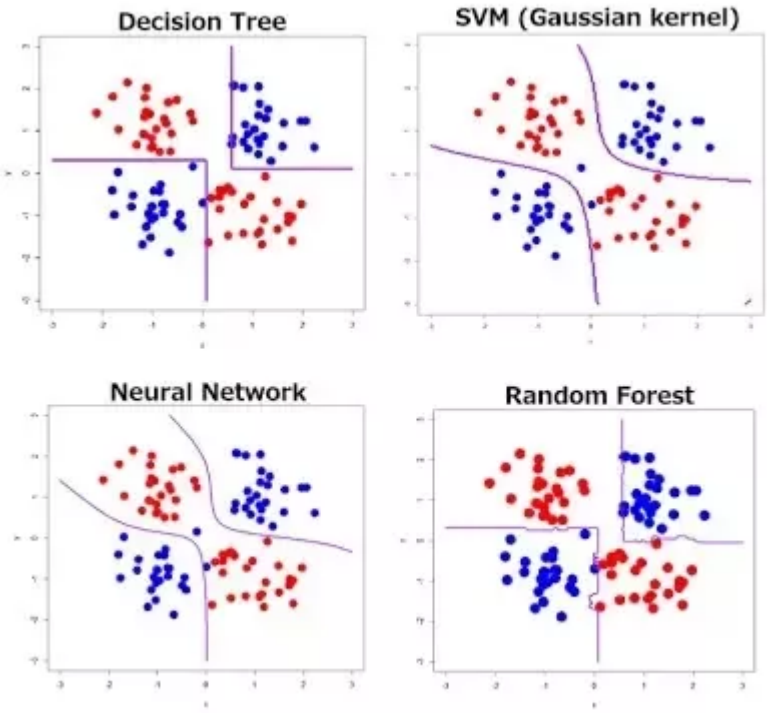
- Regression: Mean Squared Error (MSE) is used for regression tasks.

Why to choose Decision Tree over Logistic Regression

Challenges with Logistic Regression

1. Non-linear Boundaries
    - Logistic regression is a linear classification algorithm that models the relationship between input features and the probability of belonging to a particular class.
    - It assumes a linear decision boundary, which means it can only separate classes using straight lines or hyperplanes in the feature space.
    - In datasets where the true decision boundary is non-linear, logistic regression may struggle to capture complex relationships between features and classes.
2. Linear Separators
    - Logistic regression seeks to find the optimal linear separator that best separates the classes in the feature space.
    - However, in datasets with intricate patterns and non-linear relationships, a linear separator may not adequately capture the underlying structure of the data.
    - This limitation becomes more pronounced when classes overlap or exhibit complex, non-linear interactions.
3. Performance on Complex Datasets
    - Complex datasets with overlapping classes, intricate boundaries, or non-linear relationships pose significant challenges for logistic regression
    - When classes overlap, logistic regression may misclassify data points that lie near the decision boundary, leading to lower classification accuracy.
    - Logistic regression's inability to capture non-linear relationships effectively can result in suboptimal performance on datasets with complex structures.
4. Limitations in Representation
    - Logistic regression assumes a parametric form for the relationship between input features and the log-odds of the target class.
    - This parametric form imposes constraints on the complexity of the decision boundary, limiting the algorithm's ability to represent highly non-linear relationships in the data.
5. Remedies and Alternatives
    - To address the limitations of logistic regression on complex datasets, practitioners often explore alternative classification algorithms.

- Non-linear classifiers such as decision trees, random forests, support vector machines (SVMs), and neural networks are better suited for capturing complex patterns and non-linear relationships in the data.
- Ensemble methods like random forests combine multiple decision trees to improve classification performance on challenging datasets with non-linear boundaries and overlapping classes.



We will discuss SVM, Neural Network etc. in upcoming lectures

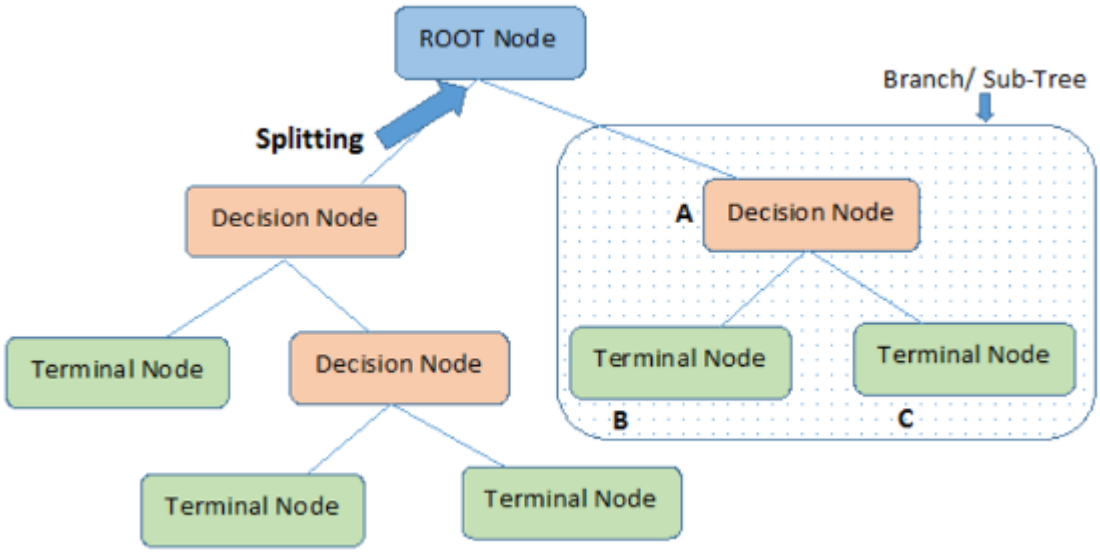
Motivation for Decision Trees

1. Interpretability:
   - Decision trees are inherently interpretable, as their structure resembles a flowchart.
   - This makes it easier to understand the decision-making process and identify the most important features influencing the outcome.
   - Interpretability is particularly valuable in domains where understanding the reasoning behind a prediction is crucial, such as medical diagnosis or financial decision-making.
2. Non-parametric Nature:
   - Decision trees are non-parametric models, meaning they don't make assumptions about the underlying data distribution.
   - This makes them more flexible and adaptable to different types of data, including categorical and numerical features.

- Decision trees can handle non-linear relationships between features and the target variable.

3. Ease of Implementation:
   - Decision tree algorithms are relatively simple to implement and understand.
   - This makes them accessible to a wide range of users, including those with limited machine learning experience.
   - There are many libraries and tools available that provide easy-to-use implementations of decision trees.
4. Ability to Handle Missing Values:
   - Decision trees can handle missing values in the data.
   - They can create branches for missing values or use imputation techniques to fill in missing data.
   - This makes them robust to real-world datasets that often contain missing data.
5. Feature Importance:
   - Decision trees can provide information about the importance of different features in the decision-making process.
   - This can be useful for feature selection and understanding the underlying relationships in the data.
   - Feature importance can be calculated based on the frequency with which a feature is used in the tree or the reduction in impurity achieved by splitting on that feature.
6. Scalability:
   - Decision trees can be scaled to handle large datasets.
   - They can be parallelized to improve performance on distributed systems.

Key Concepts in Decision Tree Construction



Root Node

- The root node is the starting point of a decision tree. It represents the entire dataset that the tree is trying to classify or regress. When the algorithm begins, all instances are initially placed at the root node.
- Key characteristics of the root node:
  - Contains the entire dataset: All data points are initially considered at this node.
  - No parent node: It's the topmost node in the tree, so it doesn't have a parent.
  - Determines the initial split: The attribute selected at the root node determines the first division of the dataset.
- Importance of the root node:
  - Foundation of the tree: The choice of the root node attribute significantly impacts the structure and performance of the tree.
  - Initial decision: It sets the direction for the subsequent decision-making process.
  - Impact on accuracy: A well-chosen root node can lead to more accurate and efficient models.

Decision Nodes

- Decision nodes are the internal nodes of a decision tree. They represent tests or conditions on attributes (or features) of the data. Each decision node has one or more outgoing branches, each corresponding to a possible outcome of the test.
- Key characteristics of decision nodes:
  - Test on an attribute: Each decision node performs a test on a specific attribute of the data.
  - Outgoing branches: The number of branches depends on the possible values of the attribute being tested.
  - Data partitioning: Decision nodes split the dataset into subsets based on the test outcomes.
- Importance of decision nodes:
  - Decision-making: Decision nodes guide the flow of data through the tree, determining the classification or prediction for each instance.
  - Feature importance: The attributes used in decision nodes can provide insights into the most influential factors in the classification or regression task.
  - Structure of the tree: The arrangement of decision nodes determines the shape and complexity of the tree.

- Example:
    - Consider a decision tree for predicting customer churn. A decision node might test the attribute "tenure" and have two branches: "tenure <= 12 months" and "tenure > 12 months". Instances that satisfy the first condition would follow the left branch, while those that satisfy the second condition would follow the right branch.

Leaf Nodes

- Leaf nodes are the terminal nodes of a decision tree. They represent the final outcomes or predictions of the model. Once an instance reaches a leaf node, the classification or regression result is determined.
- Key characteristics of leaf nodes:
    - No outgoing branches: Leaf nodes are at the end of the tree, so they don't have any children.
    - Class label or continuous value: Leaf nodes contain the predicted class label (in classification) or the predicted continuous value (in regression).
    - Final decision: The value stored in a leaf node represents the model's decision for instances that reach that node.
- Importance of leaf nodes:
    - Final outcome: Leaf nodes are the ultimate goal of the decision tree, providing the final classification or prediction for each instance.
    - Accuracy: The accuracy of the model depends on the correctness of the class labels or predicted values stored in the leaf nodes.
    - Tree structure: The number and arrangement of leaf nodes determine the size and complexity of the tree.
- Leaf node creation:
    - Leaf nodes are typically created when a decision node can no longer be split due to one of the following conditions:
        - All instances in the subset belong to the same class: If all instances at a decision node have the same target variable value, it becomes a leaf node with that value.
        - Maximum depth reached: The tree may have a predefined maximum depth, and if a decision node reaches that depth, it becomes a leaf node.
        - Minimum number of instances: A decision node may require a minimum number of instances before it can be split. If the number of instances falls below this threshold, it becomes a leaf node.
- Example:

- In a decision tree for predicting customer churn, a leaf node might contain the label "Churn" or "No Churn". If an instance reaches this leaf node, the model predicts that the customer will churn or not churn, respectively.

Sub-Tree

- A subtree is a portion of a decision tree that can be considered as a separate, smaller tree. It consists of a node (either a decision node or a leaf node) and all of its descendants.
- Key characteristics of subtrees:
  - Self-contained: Subtrees can be analyzed independently of the rest of the tree.
  - Subset of data: Each subtree represents a subset of the original dataset, corresponding to the instances that follow the path from the root to the subtree's root node.
  - Recursive structure: Subtrees can themselves contain smaller subtrees, creating a hierarchical structure.
- Importance of subtrees:
  - Understanding decision-making: Subtrees can help to visualize and understand the decision-making process of the tree for different subsets of data.
  - Pruning: Pruning involves removing subtrees that are not contributing significantly to the model's performance.
  - Ensemble methods: Some ensemble methods, like Random Forest, create multiple decision trees and combine their predictions. Each tree in the ensemble can be considered a subtree of a larger hypothetical tree.
- Example:
  - Consider a decision tree for predicting customer churn. A subtree might consist of a decision node testing the attribute "tenure" and its two descendant nodes: "tenure <= 12 months" and "tenure > 12 months". This subtree represents the portion of the tree that deals with customers based on their tenure.

Pruning

- Pruning is a technique used to reduce the size of a decision tree by removing nodes, typically leaf nodes or branches, without significantly affecting its performance. This helps to prevent overfitting, which occurs when a model becomes too complex and performs poorly on new, unseen data.

Types of Pruning:

1. Post-Pruning:

- The tree is built to its full size first.
- Then, nodes or branches are systematically removed based on certain criteria.
- Common methods include:
  - Cost-complexity pruning: A cost parameter is introduced to balance the tree's size and accuracy.
  - Error-based pruning: Nodes are removed if their removal doesn't significantly increase the error rate.

2. Pre-Pruning:

- The tree is built with predefined stopping criteria that prevent it from growing too deep or wide.
- This can be done by setting limits on:
  - Maximum depth
  - Minimum number of samples per leaf
  - Minimum impurity decrease

- Benefits of Pruning:
  - Reduced complexity: Smaller trees are easier to understand and interpret.
  - Improved generalization: Pruning can help the model perform better on new, unseen data.
  - Faster inference: Smaller trees require less computation time for predictions.

We will discuss about Pruning in detail in this colab only.

Branch / Sub-Tree

- Branch
  - A branch is a single path from a decision node to a leaf node.
  - It represents a specific sequence of decisions or conditions that lead to a particular outcome.
  - Each branch corresponds to a subset of the data that satisfies the conditions along that path.
- Subtree
  - A subtree is a portion of a decision tree that can be considered as a separate, smaller tree.
  - It consists of a node (either a decision node or a leaf node) and all of its descendants.
  - A subtree can be thought of as a collection of branches that share a common root node.

- Relationship between Branches and Subtrees:
  - Branches form subtrees: A subtree can be composed of multiple branches.
  - Subtrees can be branches: A single branch can also be considered a subtree if it doesn't have any further branching.
- Example:
  - Consider a decision tree for predicting customer churn. A branch might represent the path from the root node to a leaf node that predicts "Churn." This branch would involve decisions based on attributes like tenure, monthly charge, and contract type. The subtree rooted at the decision node "tenure <= 12 months" would include all branches that start from that node and lead to leaf nodes.

Parent and Child Node

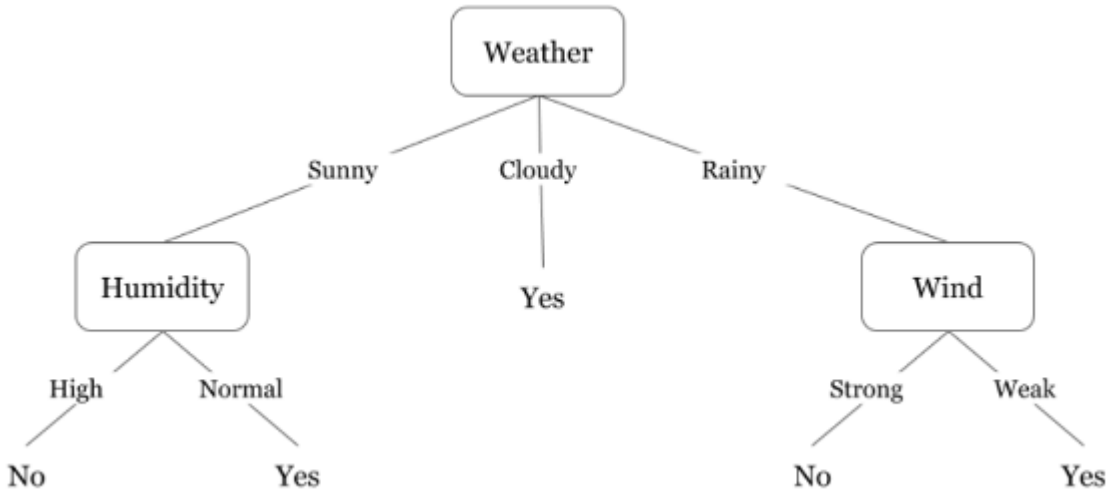
- Parent Node
  - A parent node is a node that has one or more outgoing branches.
  - It represents a decision or condition that splits the data into subsets.
  - The children of a parent node are the nodes that are connected to it by its outgoing branches.
- Child Node
  - A child node is a node that is connected to a parent node by an outgoing branch.
  - It represents a subset of the data that satisfies the condition of the parent node.
  - A child node can be either a decision node or a leaf node.
- Relationship between Parent and Child Nodes:
  - Hierarchical structure: Decision trees are organized in a hierarchical manner, with the root node at the top and leaf nodes at the bottom.
  - Parent-child relationships: Each node, except the root node, has a parent node.
  - Branching: The branches connecting parent and child nodes represent the decisions or conditions that determine the classification or prediction.
- Example:
  - Consider a decision tree for predicting customer churn. The root node might be the parent node, and its children could be decision nodes representing conditions like "tenure <= 12 months" and "tenure > 12 months". These child nodes would then have their own children, forming a hierarchical structure.

Decision Trees with an example:

- Decision trees are structured with the root node at the top, branching downwards into various nodes.
- They function similarly to a series of if-else statements, where conditions are evaluated at each node to determine the next step.
- The root node represents the initial decision point, often based on a feature or attribute.
- In a decision tree for weather-related decisions, the root node might ask, "What is the weather?" with options like sunny, cloudy, or rainy.

Day	Weather	Temperature	Humidity	Wind	Play?
1	Sunny	Hot	High	Weak	No
2	Cloudy	Hot	High	Weak	Yes
3	Sunny	Mild	Normal	Strong	Yes
4	Cloudy	Mild	High	Strong	Yes
5	Rainy	Mild	High	Strong	No
6	Rainy	Cool	Normal	Strong	No
7	Rainy	Mild	High	Weak	Yes
8	Sunny	Hot	High	Strong	No
9	Cloudy	Hot	Normal	Weak	Yes
10	Rainy	Mild	High	Strong	No

- If the condition at the root node is true (e.g., if it's sunny), the tree moves to the next node, which might evaluate additional features like humidity and wind.
- For example, if it's determined that the weather is rainy, the tree might then assess the strength of the wind.
- If the conditions at subsequent nodes are met (e.g., weak wind during rain), the tree reaches a leaf node, which represents the final decision or outcome.
- In this example, if it's determined that there's weak wind during rain, the decision might be that the person will go and play.



How do decision tree algorithms work?

- Decision tree algorithms are machine learning algorithms used for both classification and regression tasks. They create a flowchart-like structure where each internal node represents a test on an attribute (or feature), each branch represents a possible outcome of the test, and each leaf node represents a class label or a continuous value (in case of regression).

Here's a detailed breakdown of how decision tree algorithms work:

1. **Root Node Selection:**
    - The algorithm starts with a root node, which contains the entire dataset.
    - This node represents the initial decision point.
2. **Attribute Selection:**
    - The algorithm selects the best attribute to split the dataset at the root node.
    - This is typically done using metrics like Gini impurity, entropy, or information gain. These metrics measure the homogeneity of the target variable (class labels or continuous values) within the subsets created by splitting on the attribute.
    - The goal is to choose an attribute that maximizes the separation between classes or minimizes the variance of the target variable.
3. **Splitting:**
    - The dataset is divided into subsets based on the chosen attribute's values.
    - Each subset becomes a child node of the root node.
4. ** Recursive Process:**
    - The algorithm recursively repeats steps 2 and 3 for each child node until a stopping criterion is met.
    - Common stopping criteria include:

- All instances in a node belong to the same class: If all instances in a node have the same target variable value, it becomes a leaf node.
- Maximum depth reached: The tree may have a predefined maximum depth, and if a node reaches that depth, it becomes a leaf node.
- Minimum number of instances: A node may require a minimum number of instances before it can be split. If the number of instances falls below this threshold, it becomes a leaf node.

5. **Leaf Nodes:**
    - Once a node reaches a stopping criterion, it becomes a leaf node.
    - Leaf nodes contain the predicted class label (in classification) or the predicted continuous value (in regression).

Example:

Suppose we have a dataset containing information about whether customers purchased a product based on two features: age and income. The target variable is whether they made a purchase (yes or no).

Here's a simplified version of our dataset:

Customer	Age	Income	Purchase
1	30	50k	Yes
2	25	30k	No
3	35	70k	Yes
4	20	25k	No
5	40	60k	Yes

Step 1: Start at the Root Node

The algorithm begins with all the data points at the root node.

Step 2: Selecting the Best Feature

The algorithm evaluates each feature (age and income) to find the best split. Let's say it determines that age is the best feature to split on first, based on the information gain.

Step 3: Splitting the Data

The dataset is split into subsets based on the values of the selected feature (age).

- Subset 1: Customers with age <= 30

- Subset 2: Customers with age > 30

Step 4: Recursive Process

For each subset, the algorithm repeats the process:

Subset 1 (Customers with age <= 30):

The algorithm evaluates the remaining features (only income in this case) to find the best split.

Let's say it determines that income <= 40k is the best split. The dataset is further split into:

- Subset 1a: Customers with age <= 30 and income <= 40k
- Subset 1b: Customers with age <= 30 and income > 40k

Subset 2 (Customers with age > 30):

Similarly, the algorithm evaluates the remaining features to find the best split.

Let's say it determines that income <= 55k is the best split. The dataset is further split into:

- Subset 2a: Customers with age > 30 and income <= 55k
- Subset 2b: Customers with age > 30 and income > 55k

Step 5: Leaf Nodes and Predictions

The process continues until certain stopping criteria are met, such as reaching a maximum tree depth. Once the tree is fully grown, each leaf node represents a final outcome or classification.

For example:

- Leaf node 1a: Predict "No" (Customers with age <= 30 and income <= 40k)
- Leaf node 1b: Predict "Yes" (Customers with age <= 30 and income > 40k)
- Leaf node 2a: Predict "No" (Customers with age > 30 and income <= 55k)
- Leaf node 2b: Predict "Yes" (Customers with age > 30 and income > 55k)

Now, when a new customer with age 32 and income 50k is presented to the decision tree, it follows the path from the root node:

- Since age > 30, it goes to the right branch.
- Since income > 40k, it goes to the right branch again.

It reaches the leaf node predicting "Yes", indicating that this customer is likely to make a purchase.

Impurity Metrics

Impurity Measurement

- In decision tree algorithms, the impurity of the data at each node plays a crucial role in determining the quality of splits and ultimately the effectiveness of the tree in classifying

instances accurately. Impurity refers to the degree of uncertainty or mix of classes within a node.

- A node with high impurity contains a diverse mixture of classes, while a node with low impurity is more homogeneous, containing instances predominantly from one class.
- By evaluating impurity, decision trees aim to partition the data in a way that maximizes the homogeneity of child nodes, leading to clearer separation between classes.

Entropy

- Measures the uncertainty or randomness in a dataset.
- A higher entropy value indicates greater uncertainty.
- An entropy of 0 indicates perfect purity, while an entropy of 1 indicates maximum uncertainty (equal distribution of classes).

Impurity Criterion

Gini Index

$$I_G = 1 - \sum_{j=1}^{c} p_j^2$$

p_j: proportion of the samples that belongs to class c for a particular node

Entropy

$$I_H = -\sum_{j=1}^{c} p_j log_2(p_j)$$

p_j: proportion of the samples that belongs to class c for a particular node.

*This is the the definition of entropy for all non-empty classes ($p \neq 0$). The entropy is 0 if all samples at a node belong to the same class.

In the above formula of Entropy and Gini Index, p_j represents the proportion of instances belonging to class j in the node, and c is the total number of classes.

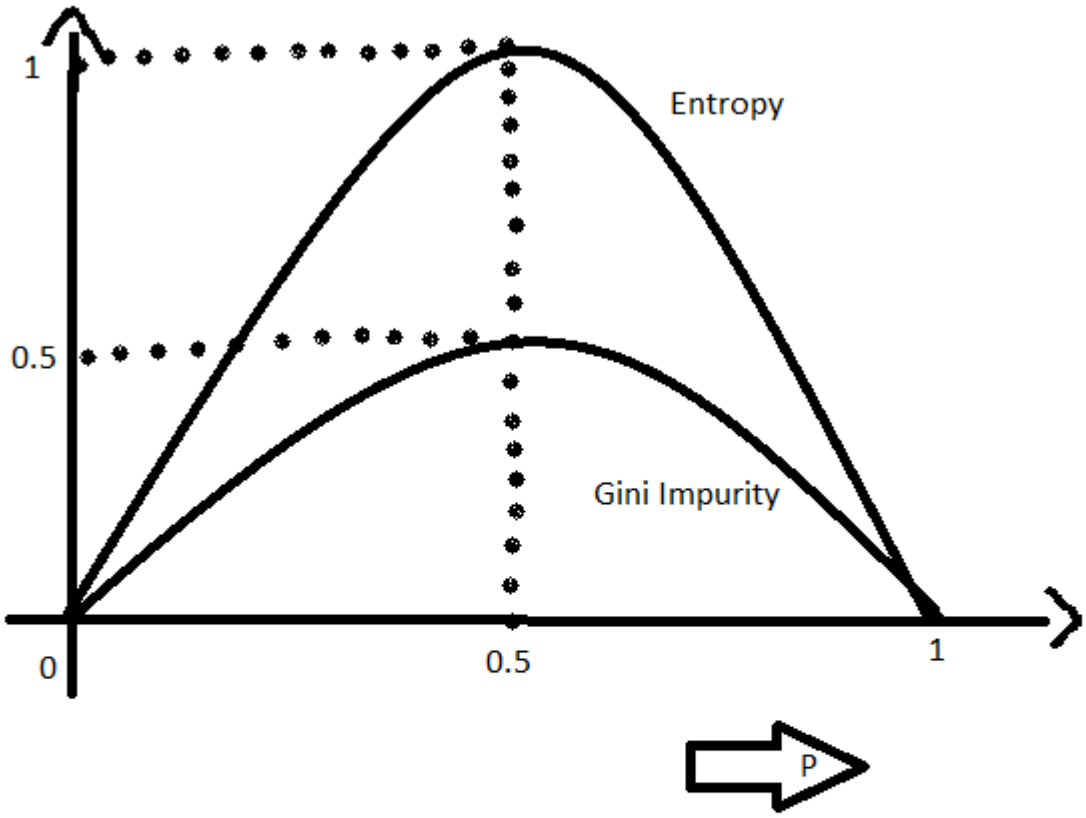
Gini Index

- The Gini index quantifies the probability of misclassification if a randomly chosen instance from a node were to be incorrectly classified according to the distribution of classes in

that node.

- It reflects how often a randomly chosen instance would be incorrectly classified.
- Interpretation
  - The Gini index ranges from 0 to 0.5.
  - A value closer to 0 indicates low impurity (the node is pure), while a value closer to 0.5 indicates high impurity (the node has an equal distribution of instances across classes).
- Similar to entropy, the Gini index is used in decision tree algorithms to assess node impurity and guide the splitting process by selecting splits that minimize impurity.

Comparison



$$E(S) = \sum_{i=1}^{c} - p_i \log_2 p_i$$

$$Gini(E) = 1 - \sum_{j=1}^{c} p_j^2$$

- Both entropy and the Gini index serve similar purposes in decision tree construction, helping to evaluate node impurity and guide splitting decisions.

- While entropy tends to be more sensitive to changes in class probabilities and may result in more balanced splits, the Gini index tends to be less computationally intensive to calculate.
- The choice between entropy and the Gini index often depends on the specific dataset and problem context, as well as the preferences of the practitioner.

Gini Impurity

- Gini impurity is a measure of how often a randomly chosen element from the set would be incorrectly labeled if it was randomly labeled according to the distribution of labels in the subset. In other words, it measures the degree of impurity or uncertainty in a dataset.
- Properties:
    - Range: Gini impurity lies between 0 and 1, where 0 indicates perfect purity (all elements belong to the same class) and 1 indicates maximum impurity (all classes are equally likely).
    - Balanced vs. Unbalanced Classes: Gini impurity tends to favor splits that result in balanced partitions because it penalizes imbalanced distributions. However, it doesn't require balanced classes to work effectively.
    - Computationally Efficient: Gini impurity is computationally less intensive compared to entropy (used in information gain), as it doesn't involve logarithmic calculations.
- Application in Decision Trees:
    - In decision tree algorithms like CART (Classification and Regression Trees), Gini impurity is commonly used as a criterion for selecting the best attribute to split a node. The attribute with the lowest Gini impurity after the split is chosen as the splitting criterion.
- Steps for Splitting using Gini Impurity:
    - Calculate Gini impurity for each candidate split: For each attribute, calculate the Gini impurity for every possible split point.
    - Select the attribute with the lowest Gini impurity: Choose the attribute and split point that result in the lowest Gini impurity.
    - Split the node: Split the node based on the selected attribute and split point.
    - Repeat the process recursively: Repeat the process recursively for each child node until a stopping criterion is met (e.g., maximum depth reached, minimum number of samples per node).
- Advantages:
    - Gini impurity is intuitive and easy to understand.
    - It tends to favor splits that result in balanced partitions.
- Limitations:

- Gini impurity may not always lead to the most informative splits, especially when dealing with categorical variables with many levels.
- It doesn't provide a measure of the magnitude of the difference between classes, only the relative proportions.

Formula of calculating Information Gain:

$$Gini = 1 - \sum_{i=1}^{n} p^2(c_i)$$

$$Entropy = \sum_{i=1}^{n} -p(c_i)log_2(p(c_i))$$

where $p(c_i)$ is the probability/percentage of class c_i in a node.

Choice Between Entropy and Gini Impurity:

- Entropy for Balanced Trees: Entropy tends to create more balanced trees because it seeks to minimize the uncertainty or disorder in each split. This often leads to more equally distributed classes in the resulting subsets.
- Gini Impurity for Computational Efficiency: On the other hand, Gini impurity is computationally faster compared to entropy. It involves simpler calculations, making it more efficient, especially for large datasets or when building multiple decision trees.
- Gini Impurity as Default Choice: Many decision tree algorithms, including CART (Classification and Regression Trees), use Gini impurity as the default impurity measure. This is primarily because of its computational efficiency.

Use Case

Problem Statement

- An organization aims to predict employee attrition using a machine learning model.
- The dataset provided contains various employee attributes, including demographic information, job-related factors, and indicators of attrition.
- The goal is to preprocess the dataset, address any data quality issues, perform feature encoding, and split the data into training and testing sets.
- Additionally, strategies to handle class imbalance in the target variable ("Attrition") need to be implemented to ensure the model's predictive accuracy and reliability.
- The objective is to develop a robust machine learning model capable of accurately predicting employee attrition, thereby assisting the organization in identifying potential

turnover risks and implementing proactive retention strategies.

Exploring Data

Download Dataset From Here

```
import pandas as pd
import numpy as np

import matplotlib.pyplot as plt

import io


# Importing the necessary function from the google.colab library
from google.colab import drive

# Mounting Google Drive to the Colab environment
drive.mount('/content/drive')

# - This code imports the `drive` function from the google.colab library.
# - The `drive.mount()` function is then called with the argument '/content/drive
# - This allows access to files stored in Google Drive within the Colab notebook
```

```
Mounted at /content/drive
```

```
df = pd.read_csv('/content/drive/MyDrive/WA_Fn-UseC_-HR-Employee-Attrition.csv')


df.head()
```

	Age	Attrition	BusinessTravel	DailyRate	Department	DistanceFromHome	Edu
0	41	Yes	Travel_Rarely	1102	Sales	1	
1	49	No	Travel_Frequently	279	Research & Development	8	
2	37	Yes	Travel_Rarely	1373	Research & Development	2	
3	33	No	Travel_Frequently	1392	Research & Development	3	
4	27	No	Travel_Rarely	591	Research & Development	2	

5 rows × 35 columns

```
df.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 1470 entries, 0 to 1469
Data columns (total 35 columns):
 #   Column                    Non-Null Count  Dtype 
---  ------                    --------------  ----- 
 0   Age                       1470 non-null   int64 
 1   Attrition                 1470 non-null   object
 2   BusinessTravel            1470 non-null   object
 3   DailyRate                 1470 non-null   int64 
 4   Department                1470 non-null   object
 5   DistanceFromHome          1470 non-null   int64 
 6   Education                 1470 non-null   int64 
 7   EducationField            1470 non-null   object
 8   EmployeeCount             1470 non-null   int64 
 9   EmployeeNumber            1470 non-null   int64 
 10  EnvironmentSatisfaction   1470 non-null   int64 
 11  Gender                    1470 non-null   object
 12  HourlyRate                1470 non-null   int64 
 13  JobInvolvement            1470 non-null   int64 
 14  JobLevel                  1470 non-null   int64 
 15  JobRole                   1470 non-null   object
 16  JobSatisfaction           1470 non-null   int64 
 17  MaritalStatus             1470 non-null   object
 18  MonthlyIncome             1470 non-null   int64 
 19  MonthlyRate               1470 non-null   int64 
 20  NumCompaniesWorked        1470 non-null   int64 
 21  Over18                    1470 non-null   object
 22  OverTime                  1470 non-null   object
 23  PercentSalaryHike         1470 non-null   int64 
 24  PerformanceRating         1470 non-null   int64 
 25  RelationshipSatisfaction  1470 non-null   int64 
 26  StandardHours             1470 non-null   int64 
 27  StockOptionLevel          1470 non-null   int64 
 28  TotalWorkingYears         1470 non-null   int64 
 29  TrainingTimesLastYear     1470 non-null   int64 
 30  WorkLifeBalance           1470 non-null   int64 
 31  YearsAtCompany            1470 non-null   int64 
 32  YearsInCurrentRole        1470 non-null   int64 
 33  YearsSinceLastPromotion   1470 non-null   int64 
 34  YearsWithCurrManager      1470 non-null   int64 
dtypes: int64(26), object(9)
memory usage: 402.1+ KB
```

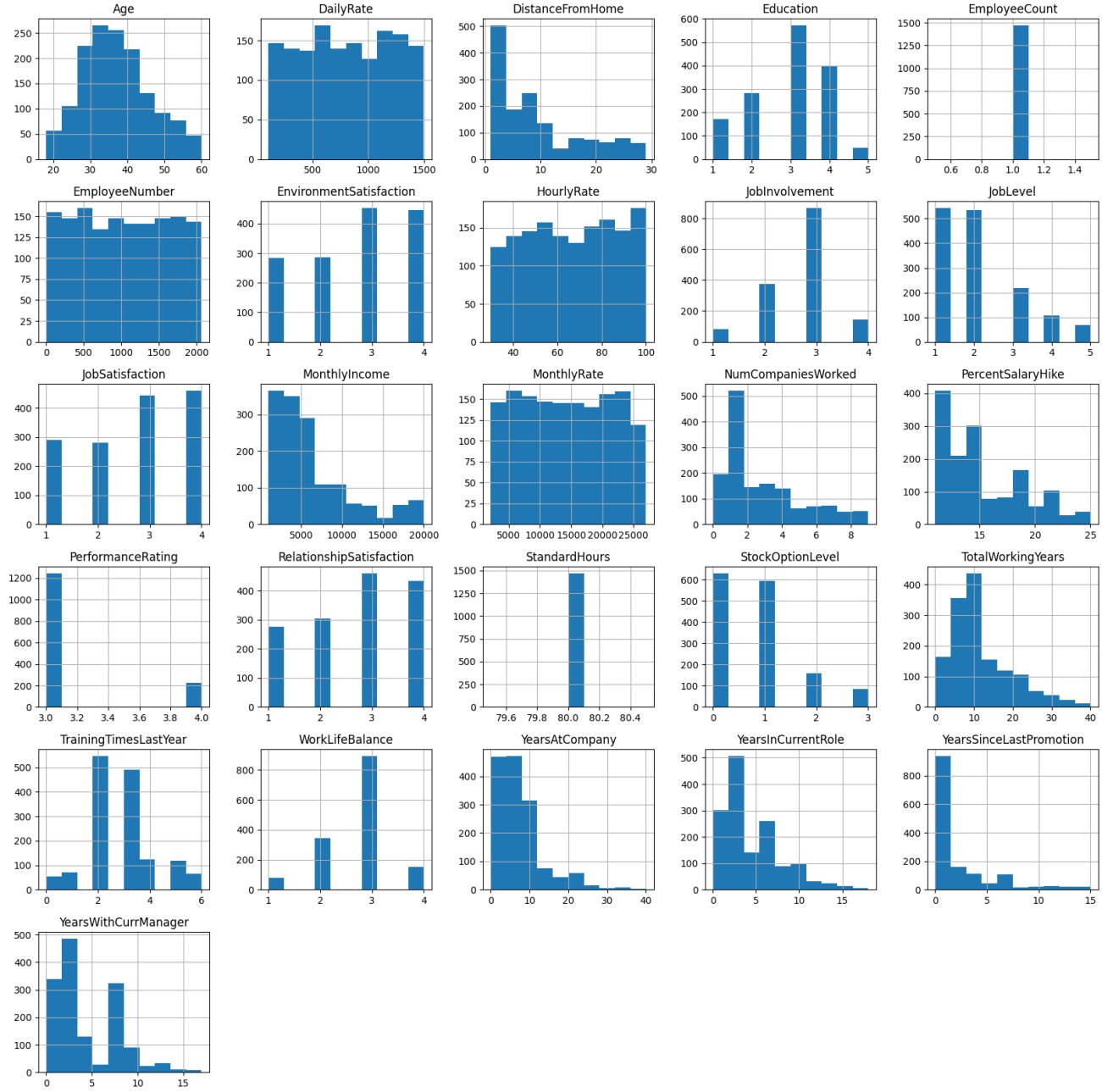
Data Preprocessing

```
df.nunique()
```

```
Age                         43
Attrition                    2
BusinessTravel               3
DailyRate                  886
Department                   3
DistanceFromHome            29
Education                    5
EducationField               6
EmployeeCount                1
EmployeeNumber            1470
```

```
EnvironmentSatisfaction        4
Gender                         2
HourlyRate                    71
JobInvolvement                 4
JobLevel                       5
JobRole                        9
JobSatisfaction                4
MaritalStatus                  3
MonthlyIncome               1349
MonthlyRate                 1427
NumCompaniesWorked            10
Over18                         1
OverTime                       2
PercentSalaryHike             15
PerformanceRating              2
RelationshipSatisfaction       4
StandardHours                  1
StockOptionLevel               4
TotalWorkingYears             40
TrainingTimesLastYear          7
WorkLifeBalance                4
YearsAtCompany                37
YearsInCurrentRole            19
YearsSinceLastPromotion       16
YearsWithCurrManager          18
dtype: int64
```

```
# Display histograms for each numerical variable in the DataFrame
df.hist(figsize=(20, 20))  # Setting the figure size to (20, 20) for better visua
plt.show()  # Display the histograms
```


Age
DailyRate
DistanceFromHome
Education
EmployeeCount
EmployeeNumber
EnvironmentSatisfaction
HourlyRate
JobInvolvement
JobLevel
JobSatisfaction
MonthlyIncome
MonthlyRate
NumCompaniesWorked
PercentSalaryHike
PerformanceRating
RelationshipSatisfaction
StandardHours
StockOptionLevel
TotalWorkingYears
TrainingTimesLastYear
WorkLifeBalance
YearsAtCompany
YearsInCurrentRole
YearsSinceLastPromotion
YearsWithCurrManager

These histograms offer a visual representation of the distribution and spread of various numerical variables within the dataset, aiding in exploratory data analysis and understanding the characteristics of the workforce.

```
df.columns
```

```
Index(['Age', 'Attrition', 'BusinessTravel', 'DailyRate', 'Department',
       'DistanceFromHome', 'Education', 'EducationField', 'EmployeeCount',
       'EmployeeNumber', 'EnvironmentSatisfaction', 'Gender', 'HourlyRate',
       'JobInvolvement', 'JobLevel', 'JobRole', 'JobSatisfaction',
       'MaritalStatus', 'MonthlyIncome', 'MonthlyRate', 'NumCompaniesWorked',
       'Over18', 'OverTime', 'PercentSalaryHike', 'PerformanceRating',
       'RelationshipSatisfaction', 'StandardHours', 'StockOptionLevel',
       'TotalWorkingYears', 'TrainingTimesLastYear', 'WorkLifeBalance',
       'YearsAtCompany', 'YearsInCurrentRole', 'YearsSinceLastPromotion',
       'YearsWithCurrManager'],
      dtype='object')
```

Identifying Irrelevant Variables:

1. Employee Count: This variable likely contains the same value for all observations, making it irrelevant for analysis as it doesn't provide any variability or useful information.
2. Employee Number: Assuming this is an identification number assigned to each employee, it's not useful for analysis purposes as it doesn't contribute to predicting attrition.
3. Over 18: If this variable indicates whether an employee is over 18 years old, it might not be relevant for predicting attrition unless there's a specific legal requirement related to age.
4. Standard Hours: If this variable represents the standard number of hours worked per week by an employee, it may not vary much across observations and therefore might not be informative for predicting attrition.

Removing Irrelevant Variables

```
df.drop(['EmployeeCount', 'EmployeeNumber', 'Over18', 'StandardHours'], axis = 1,
```

```
df.head()
```

	Age	Attrition	BusinessTravel	DailyRate	Department	DistanceFromHome	Edu
0	41	Yes	Travel_Rarely	1102	Sales	1	
1	49	No	Travel_Frequently	279	Research & Development	8	
2	37	Yes	Travel_Rarely	1373	Research & Development	2	
3	33	No	Travel_Frequently	1392	Research & Development	3	
4	27	No	Travel_Rarely	591	Research & Development	2	

5 rows × 31 columns

Encoding

Encoding refers to the process of converting categorical data into a numerical format that can be used for training machine learning models. Categorical data represents attributes that have a fixed number of unique values or categories. Example include variables like Attrition, BusinessTravel, Department, or any other non-numeric data.

```
# Iterate over each column in the DataFrame starting from the second column
for col in df.columns[1:]:
    # Check if the data type of the column is 'object', indicating it contains ca
    if df[col].dtype == 'object':
        # Print the column name and the number of unique values in that column
        print(f"{col}: {df[col].nunique()}")
```

```
Attrition: 2
BusinessTravel: 3
Department: 3
EducationField: 6
Gender: 2
JobRole: 9
MaritalStatus: 3
OverTime: 2
```

Label Encoding for Binary Variables:

- For binary categorical variables such as "Attrition" and "Gender" (which typically have two unique values), label encoding can be used.
- Label encoding replaces the categorical values with numerical labels, usually 0 and 1.
- For example, in the "Attrition" column, 'Yes' might be encoded as 1 and 'No' as 0. Similarly, 'Male' could be encoded as 1 and 'Female' as 0.

```
# only two level -> Attrition, Gender, OverTime
# lebel encoding

from sklearn.preprocessing import LabelEncoder
lb = LabelEncoder()

for col in df.columns[1: ]:
  if(df[col].dtype == 'object'):
    if len(list(df[col].unique())) == 2:
      df[col] = lb.fit_transform(df[col])


df.head()
```

	Age	Attrition	BusinessTravel	DailyRate	Department	DistanceFromHome	Edu
0	41	1	Travel_Rarely	1102	Sales	1	
1	49	0	Travel_Frequently	279	Research & Development	8	
2	37	1	Travel_Rarely	1373	Research & Development	2	
3	33	0	Travel_Frequently	1392	Research & Development	3	
4	27	0	Travel_Rarely	591	Research & Development	2	

5 rows × 31 columns

One-Hot Encoding for Categorical Variables with Multiple Levels

- For categorical variables with more than two levels (multiple categories), one-hot encoding is a common approach.
- One-hot encoding creates binary columns for each category and assigns a 1 or 0 to indicate the presence or absence of that category.
- For instance, if there is a "BusinessTravel" column with categories like 'Travel_Rarely', 'Travel_Frequently', and 'Non-Travel', one-hot encoding would create three new binary columns, each representing one of these categories.

```
# three level -> BusinessTravel, Department, MaritalStatus
# one hot encoding

df = pd.get_dummies(df, columns = ['BusinessTravel', 'Department', 'MaritalStatus


df.head()
```

	Age	Attrition	DailyRate	DistanceFromHome	Education	EducationField	Envi
0	41	1	1102	1	2	Life Sciences	
1	49	0	279	8	1	Life Sciences	
2	37	1	1373	2	2	Other	
3	33	0	1392	3	4	Life Sciences	
4	27	0	591	2	1	Medical	

5 rows × 34 columns

Model Prepration

```
# Assigning Variables:
target = df["Attrition"]  # Extracting the "Attrition" column as the target varia
X = df.drop(['Attrition'], axis=1)  # Creating a DataFrame X containing features

# - In machine learning, it's common to separate the target variable (dependent v
# - 'target' represents the dependent variable, which is what we want to predict
# - 'X' contains the independent variables (features) used for prediction after r
# - Separating the target variable from the features simplifies data preprocessin
# - The 'df.drop(['Attrition'], axis=1)' function removes the 'Attrition' column
# - After running this code, 'target' will be a Series containing the "Attrition"


# Calculating Normalized Value Counts for the Target Variable:
target.value_counts(normalize=True)

# - This code calculates the normalized value counts of the target variable 'targ
# - 'target.value_counts()' computes the frequency of each unique value in the ta
# - 'normalize=True' parameter normalizes the counts to show proportions instead
# - The output will display the proportion of each unique value in the target var
# - Normalized value counts are useful for understanding the class distribution,
```

```
0    0.838776
1    0.161224
Name: Attrition, dtype: float64
```

- This output represents the normalized value counts of the target variable 'Attrition'.
- The value '0' corresponds to employees who did not leave the company (no attrition), while '1' corresponds to employees who left (attrition).
- The proportion of employees who did not leave the company (no attrition) is approximately 83.88% (0.838776), and the proportion of employees who left the company (attrition) is approximately 16.12% (0.161224).

- This information provides insights into the class distribution within the target variable, indicating that the dataset is imbalanced, with a larger proportion of employees not experiencing attrition compared to those who did.

```
from sklearn.model_selection import train_test_split

# Splitting the dataset into training and testing sets
# X: Features (independent variables)
# target: Target variable (dependent variable)
# test_size: Proportion of the dataset to include in the testing split (20%)
# random_state: Seed for random number generation to ensure reproducibility
# stratify: Ensures that the class distribution of the target variable is preserv
X_train, X_test, y_train, y_test = train_test_split(X, target, test_size=0.2, ran
```

Target Encoding for High-Cardinality Variables

- Target encoding is useful for categorical variables with a high cardinality (a large number of unique categories).
- It replaces each category with the mean of the target variable (e.g., the mean of the target variable for all instances with that category).
- This technique can be effective for preserving information in high-cardinality variables while converting them into a numerical format that can be used by machine learning algorithms.
- However, it's crucial to perform target encoding after splitting the data into training and validation sets to prevent data leakage.

```
!pip install category_encoders
```

```
Collecting category_encoders
  Downloading category_encoders-2.6.3-py2.py3-none-any.whl (81 kB)
     ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 81.9/81.9 kB 2.4 MB/s eta 0:00:(
Requirement already satisfied: numpy>=1.14.0 in /usr/local/lib/python3.10/dist
Requirement already satisfied: scikit-learn>=0.20.0 in /usr/local/lib/python3
Requirement already satisfied: scipy>=1.0.0 in /usr/local/lib/python3.10/dist-
Requirement already satisfied: statsmodels>=0.9.0 in /usr/local/lib/python3.1(
Requirement already satisfied: pandas>=1.0.5 in /usr/local/lib/python3.10/dist
Requirement already satisfied: patsy>=0.5.1 in /usr/local/lib/python3.10/dist-
Requirement already satisfied: python-dateutil>=2.8.1 in /usr/local/lib/pytho
Requirement already satisfied: pytz>=2020.1 in /usr/local/lib/python3.10/dist-
Requirement already satisfied: six in /usr/local/lib/python3.10/dist-packages
Requirement already satisfied: joblib>=1.1.1 in /usr/local/lib/python3.10/dist
Requirement already satisfied: threadpoolctl>=2.0.0 in /usr/local/lib/python3
Requirement already satisfied: packaging>=21.3 in /usr/local/lib/python3.10/d
Installing collected packages: category_encoders
Successfully installed category_encoders-2.6.3
```

```
import category_encoders as ce

# Initializing a TargetEncoder object from the category_encoders module
# specifying the columns to encode ('EducationField' and 'JobRole')
ce_target = ce.TargetEncoder(cols=['EducationField', 'JobRole'])

# Encoding the categorical variables in the training set using target encoding
# fit_transform() method is used to both fit the encoder to the data and transfor
X_train = ce_target.fit_transform(X_train, y_train)

# Transforming the categorical variables in the testing set using the target enco
# transform() method is used here as the encoder has already been fitted on the t
X_test = ce_target.transform(X_test)
```

```
X_train.head()
```

	Age	DailyRate	DistanceFromHome	Education	EducationField	EnvironmentS
1052	30	990	7	3	0.266645	
1267	34	1375	10	3	0.137500	
997	27	135	17	4	0.137500	
882	36	363	1	3	0.266645	
404	28	1300	17	2	0.139785	

5 rows × 33 columns

Applying SMOTE For Handling Class Imbalance

```
from imblearn.over_sampling import SMOTE

# Initializing an SMOTE (Synthetic Minority Over-sampling Technique) object with
# Here, the sampling_strategy parameter is set to 0.3, which means the number of
# will be increased to 30% of the number of samples in the majority class after r
sm = SMOTE()
# sm = SMOTE(sampling_strategy=0.3)

# Resampling the training data using SMOTE to handle class imbalance
# fit_resample() method is used to fit the SMOTE model to the training data (X_tr
# and then resample the data according to the specified strategy
X_sm, y_sm = sm.fit_resample(X_train, y_train)
```

```
print("Distribution of target {}".format(y_sm.value_counts()))
```

```
Distribution of target 0    986
1    986
Name: Attrition, dtype: int64
```

Implementation of Decision Tree as a Classifier

```
from sklearn.tree import DecisionTreeClassifier

# Initializing a DecisionTreeClassifier object for classification
# The random_state parameter is set to 1 to ensure reproducibility of results
tree_clf = DecisionTreeClassifier(random_state=1)
```

```
from sklearn.model_selection import cross_validate, KFold

# Creating a KFold cross-validation splitter object with 5 folds
kFold = KFold(n_splits=5)

# Performing cross-validation using the cross_validate function
# The tree_clf decision tree classifier is evaluated using cross-validation
# X_sm and y_sm are the features and target variables, respectively, for the trai
# cv=kFold specifies the cross-validation strategy using the KFold splitter
# scoring='accuracy' specifies the evaluation metric as accuracy
# return_train_score=True indicates that training scores should be returned in th
cv_acc_results = cross_validate(tree_clf, X_sm, y_sm, cv=kFold, scoring='accuracy
```

```
cv_acc_results['train_score'].mean()
```

```
1.0
```

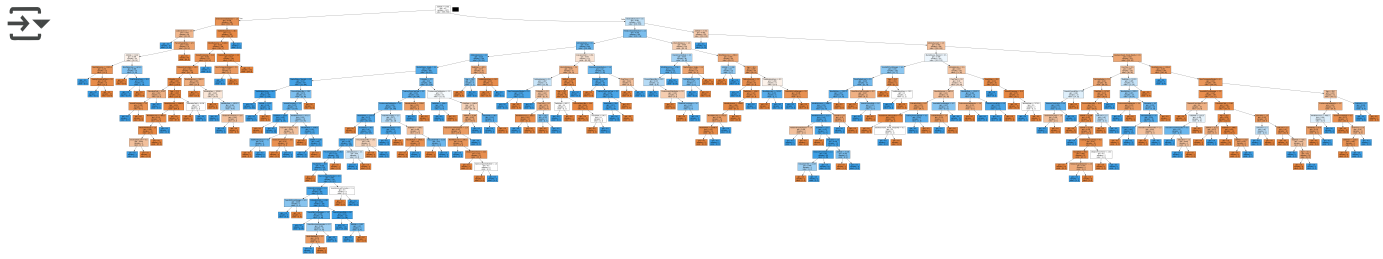
- An average training accuracy of 100% suggests that the model achieves perfect accuracy on the training data across all folds of cross-validation.
- However, this high training accuracy may indicate overfitting, where the model has learned to memorize the training data rather than generalize well to unseen data.
- Overfitting can lead to poor performance on new, unseen data because the model captures noise or random fluctuations in the training data instead of underlying patterns.

```
cv_acc_results['test_score'].mean()
```

```
0.8200359827796697
```

- The mean test accuracy of approximately 79.37% indicates the average accuracy achieved by the model on unseen data.
- A test accuracy of around 79.37% suggests that the model generalizes reasonably well to new, unseen data.
- The test accuracy being lower than the perfect training accuracy (100%) suggests that the model may be slightly overfitting to the training data but still performs well on unseen data, indicating a good balance between bias and variance.
- Overall, this test accuracy provides a reliable estimate of the model's performance on new data, which is essential for assessing its real-world applicability.

```
from IPython.display import Image
from io import StringIO
from sklearn.tree import export_graphviz
import pydot

# Get the list of features
features = list(X_sm.columns)

# Ensure that our target variable "Attrition" is not included in the feature list
# This is to prevent the target variable from being included in the visualization


# Fit the decision tree classifier to the oversampled training data
tree_clf = tree_clf.fit(X_sm, y_sm)

# Initialize a StringIO object to store the graphviz data
dot_data = StringIO()

# Export the decision tree visualization to the StringIO object
export_graphviz(tree_clf, out_file=dot_data, feature_names=features, filled=True)

# Create a graph from the graphviz data stored in the StringIO object
graph = pydot.graph_from_dot_data(dot_data.getvalue())

# Display the decision tree visualization as an image
Image(graph[0].create_png())
```



Challenges with Decision Trees

Overfitting

- Definition: Overfitting occurs when a model learns the training data too well, capturing noise or random fluctuations that are specific to the training set but do not generalize well to unseen data.
- Cause: Decision trees are inherently flexible and have the capacity to partition the feature space into increasingly specific regions to minimize training error. This flexibility can lead to the creation of overly complex trees that capture noise in the training data.

- Consequences: An overfitted decision tree will perform well on the training data but will likely perform poorly on unseen data, as it has essentially memorized the training examples rather than learned the underlying patterns.
- Mitigation Strategies:
    - Pruning: Pruning is a technique used to reduce the size of a decision tree by removing nodes that do not provide significant predictive power. This helps prevent overfitting by simplifying the tree structure.
    - Limiting Tree Depth or Minimum Samples per Leaf: Constraining the maximum depth of the tree or setting a minimum number of samples required to split a node can also prevent overfitting by limiting the complexity of the tree. `- Cross-Validation: Cross-validation techniques, such as k-fold cross-validation, can be used to assess the performance of a decision tree model on unseen data and tune hyperparameters to prevent overfitting.

Greedy Nature

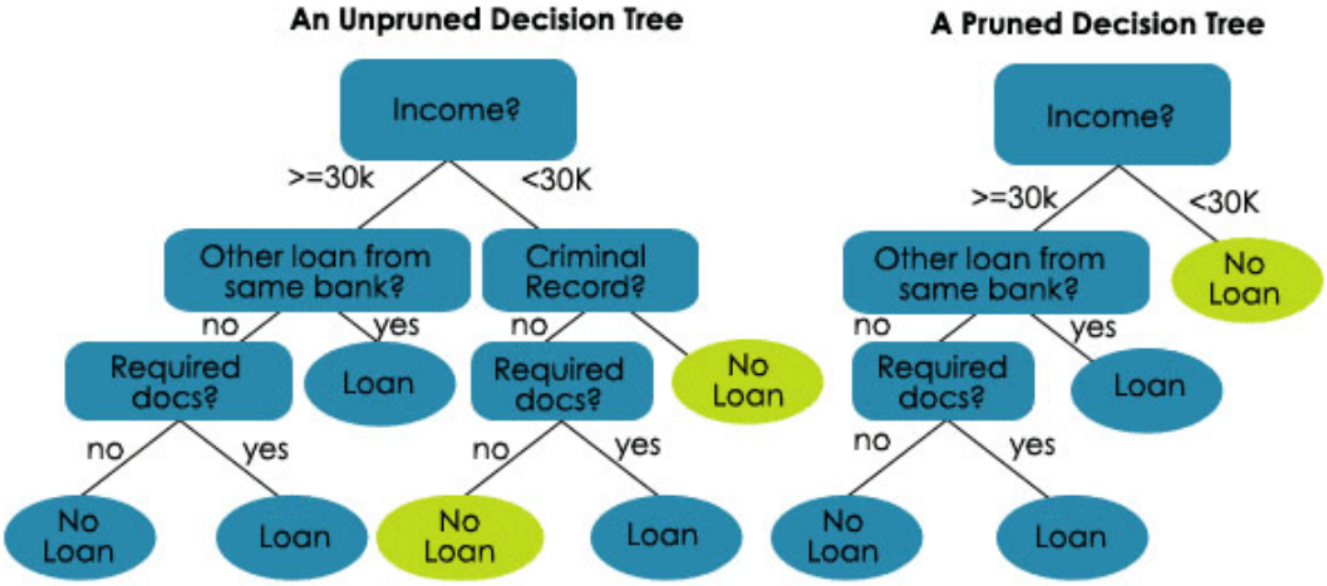
- Definition: Decision trees are constructed using a greedy algorithm that makes locally optimal decisions at each node without considering the global optimal tree structure.
- Cause: At each node of the tree, the algorithm chooses the feature and split that maximizes information gain or another impurity measure. While this approach is computationally efficient, it may not always lead to the best overall tree structure.
- Consequences: The locally optimal decisions made by the greedy algorithm may result in suboptimal splits, leading to a decision tree that is less accurate or interpretable than it could be with a more exhaustive search.
- Mitigation Strategies:
    - Ensemble Methods: Ensemble methods such as Random Forest and Gradient Boosting combine multiple decision trees to mitigate the impact of individual suboptimal trees. By aggregating the predictions of multiple trees, these methods can improve overall predictive performance.
    - Optimization Algorithms: Some advanced decision tree algorithms, such as CART (Classification and Regression Trees) or C4.5, use heuristics or optimization techniques to explore a broader search space and find better split points, reducing the impact of the greedy nature.
    - Feature Engineering: Preprocessing the data and engineering informative features can help improve the quality of splits and guide the decision tree algorithm towards better overall tree structures.

Pruning Techniques

Pruning techniques are essential for controlling the complexity of decision trees and preventing overfitting. There are two main pruning techniques: post-pruning and pre-pruning.

Post-Pruning

- Definition: Post-pruning, also known as backward pruning or cost-complexity pruning, involves growing the decision tree to its maximum size and then selectively pruning branches or leaf nodes that are deemed unnecessary based on a pruning criterion.
- Process:
    - Tree Growth: Initially, the decision tree is grown to its maximum size, allowing it to capture as much information from the training data as possible.
    - Evaluation: Once the tree is fully grown, each node in the tree is evaluated for its contribution to the overall performance of the tree. This evaluation can be based on metrics such as information gain, Gini impurity, or accuracy.
    - Pruning Decision: Nodes that do not significantly improve the performance of the tree are pruned, either by collapsing them into leaf nodes or removing them entirely from the tree structure.
- Advantages:
    - Post-pruning allows the decision tree to grow without constraints, capturing complex relationships in the data.
    - It can potentially lead to higher accuracy compared to pre-pruning methods if the tree is allowed to grow sufficiently large.
- Disadvantages:
    - Post-pruning requires evaluating the entire tree, which can be computationally expensive, especially for large trees or datasets.
    - Selecting an appropriate pruning criterion and threshold can be challenging, as it may vary depending on the dataset and problem domain.
- Techniques:
    - Cost-Complexity Pruning: This technique assigns a cost to each node in the tree based on a complexity measure (e.g., node impurity or node depth) and a penalty parameter. Nodes with the smallest increase in cost after pruning are pruned first, leading to a sequence of pruned trees that are evaluated using cross-validation to select the optimal tree size.



Pre-Pruning

- Definition: Pre-pruning, also known as forward pruning, involves setting conditions during the tree construction process to halt the growth of the tree based on certain criteria.
- Process:
  - Early Stopping Criteria: During the construction of the decision tree, conditions are imposed on when to stop splitting nodes based on predefined criteria. Common stopping criteria include maximum tree depth, minimum number of samples required to split a node, minimum samples per leaf node, or maximum number of leaf nodes.
  - Tree Construction: The decision tree is grown recursively, splitting nodes based on the selected features and evaluating the stopping criteria at each step. If a stopping criterion is met, the node is not split further, and it becomes a leaf node.
- Advantages:
  - Pre-pruning is computationally efficient since it avoids the need to evaluate the entire tree after construction.
  - It allows for more control over the complexity of the tree and can prevent the tree from growing too large and overfitting the training data.
- Disadvantages:
  - Pre-pruning may lead to underfitting if the stopping criteria are too restrictive, causing the tree to be too shallow or simplistic to capture the underlying patterns in the data.
  - Selecting appropriate stopping criteria requires domain knowledge or experimentation, and it may not always generalize well across different datasets or problem domains.

- Techniques:
  - Max Depth: Setting a maximum depth for the tree limits the number of levels in the tree, preventing it from growing too deep and capturing noise in the data.
  - Min Samples Split: This criterion specifies the minimum number of samples required to split a node, preventing further partitioning of nodes with a small number of samples, which helps avoid overfitting.
  - Min Samples Leaf: Similar to min samples split, this criterion specifies the minimum number of samples required to be at a leaf node, effectively controlling the granularity of the tree.

Pre-Pruning Implementation

```
# Define a DecisionTreeClassifier with specified hyperparameters
tree_clf = DecisionTreeClassifier(random_state=7, max_depth=10, min_samples_leaf=

# Perform cross-validation on the classifier
cv_acc_results = cross_validate(tree_clf, X_sm, y_sm, cv=kFold, scoring='accuracy
```

Above code snippet demonstrates the use of a decision tree classifier with specified hyperparameters and the application of cross-validation to assess its performance on a balanced dataset. The chosen hyperparameters aim to control the complexity of the decision tree and prevent overfitting, while the use of cross-validation helps in estimating the model's generalization performance.

```
cv_acc_results['train_score'].mean()
```

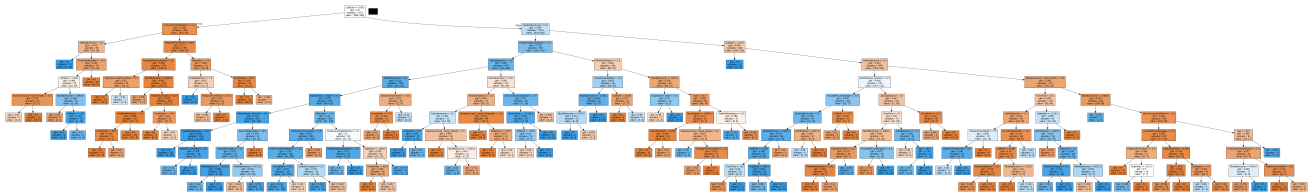
```
0.9234280327232482
```

```
cv_acc_results['test_score'].mean()
```

```
0.8058266401079484
```

```
# Fit the decision tree classifier to the training data
tree_clf = tree_clf.fit(X_sm, y_sm)

# Make predictions on the test data using the trained classifier
pred = tree_clf.predict(X_test)
```

```
# Fit the decision tree classifier to the training data
tree_clf = tree_clf.fit(X_sm, y_sm)

# Create a StringIO object to store the graphviz dot data
dot_data = StringIO()

# Export the decision tree classifier to the dot format using graphviz
export_graphviz(tree_clf, out_file=dot_data, feature_names=features, filled=True)

# Create a graph from the dot data
graph = pydot.graph_from_dot_data(dot_data.getvalue())

# Generate and display the image of the decision tree
Image(graph[0].create_png())
```



- The decrease in the size of the decision tree plot suggests that pre-pruning techniques were employed during the construction of the decision tree classifier.
- Pre-pruning involves setting conditions during the tree growth process to limit the maximum depth of the tree or the minimum number of samples required to split a node or form a leaf.
- By restricting the growth of the tree based on these criteria, pre-pruning helps prevent the model from becoming overly complex and capturing noise in the training data, which can lead to overfitting.
- In above code snippet, the decision tree classifier (tree_clf) is trained using pre-pruning techniques, resulting in a smaller and more interpretable tree structure compared to an unconstrained tree that may grow excessively.

Variable Importance

- Variable Importance identifies which variables are crucial in determining the splits within the tree structure.
- Root Node Importance:
  - Variables that split at the root node are typically deemed more important as they contribute significantly to the initial partitioning of the data.
  - However, other variables might also split at subsequent nodes, influencing the overall tree structure.
- Computation Method:
  - Variable importance is calculated by measuring the mean decrease in impurity (often using the Gini index) at each split.
  - The impurity decrease is weighted by the number of observations impacted by the split, providing a normalized importance score for each variable.

Calculation of Variable Importance:

The process for computing variable importance involves the following steps:

1. Impurity Decrease Calculation:
    - Evaluate the decrease in impurity (Gini index) at each split node.
    - This decrease represents the improvement in purity achieved by splitting based on a particular feature.
2. Observations Impact Assessment:
    - Determine the number of observations influenced by each split.
    - Weight the impurity decrease by the number of impacted observations.
3. Normalization:
    - Normalize the impurity decrease across different splits for each variable.
    - This normalization process ensures that importance scores are comparable across features.

Interpretation of Variable Importance:

Understanding the implications of variable importance:

1. Lack of Directionality:
    - Unlike logistic regression coefficients, variable importance does not specify the direction of impact (i.e., whether a feature increases or decreases the risk).
    - Instead, it emphasizes the significance of features in the context of splitting nodes based on impurity decrease.
2. Significance Assessment:
    - Variables with higher importance scores contribute more substantially to reducing impurity.
    - Important features lead to better splits in the decision tree, resulting in clearer delineations between classes.
3. Limitations:
    - While variable importance aids in identifying influential features, it does not directly reveal the direction or strength of the relationship with the target variable.
    - Additional analysis may be necessary to fully understand the relationships between features and the target.

```
importances = tree_clf.feature_importances_
indices = np.argsort(importances)[::-1]  # Sort feature importances in descending
names = [features[i] for i in indices]   # Rearrange feature names so they match

plt.figure(figsize=(15, 7))  # Create plot
plt.title("Feature Importance")  # Create plot title
```