# **Car Evaluation using Decision Tree**

# 

# 

# **Table of Content:**

| **Section No** | **Title** |
| --- | --- |
| 1 | Project Objective |
| 2 | Understanding Decision Tree |
| 3 | Dataset Description |
| 4 | Workflow |
| 5 | Code Explanations |
| 6 | Project Setup |
| 7 | Future Enhancements |
| 8 | Conclusion |

## 

## 

## 

## 

## 

## 

## 

## **1. Objective**

The primary objective of this project is to develop a **Decision Tree classifier** that can systematically categorize cars based on various attributes, including **price, maintenance cost, number of doors, seating capacity, luggage boot size, and safety features**. The model will classify vehicles into four distinct categories: **unacceptable, acceptable, good, or very good**, providing a structured and data-driven approach to evaluating car acceptability.

By leveraging **Decision Tree classification techniques**, this project aims to create an interpretable and efficient model capable of making well-defined decisions based on given input features. The Decision Tree algorithm is particularly well-suited for this problem due to its ability to handle both categorical and numerical data while providing clear decision-making rules.

This project serves as a practical demonstration of **supervised machine learning techniques** in the automotive industry, showcasing how predictive modeling can be utilized for consumer recommendations, vehicle assessment, and decision-making in car evaluations. The insights derived from this model can be valuable for **automobile manufacturers, dealerships, and consumers**, aiding in informed decision-making based on key car attributes.

### **Key Objectives of the Project**

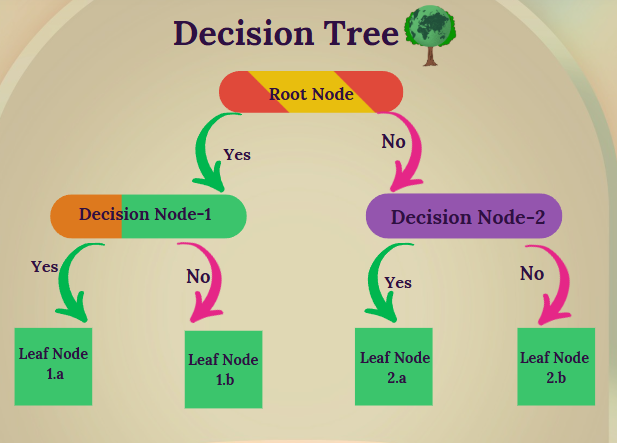
* Gain a deep understanding of the **mathematical foundation and intuition** behind Decision Trees.
* Implement both **Decision Tree Classifier** and **Decision Tree Regressor** models.
* Apply **Pre-pruning and Post-pruning techniques** to prevent overfitting and improve model generalization.
* Evaluate model performance using **relevant metrics** such as accuracy, precision, recall, and F1-score.
* Interpret and visualize the **decision-making process** of the tree using graphical tools.
* Demonstrate how **Decision Trees** can be effectively used in **consumer car evaluation scenarios**, helping buyers make informed choices based on vehicle attributes.

## UnderStanding Decision Tree:

### **1. What is a Decision Tree?**

A **Decision Tree** is a supervised machine learning algorithm used for both **classification** and **regression** tasks. It follows a tree-like structure to make decisions by recursively splitting the dataset based on feature values.

* **Internal Nodes** represent decision points based on specific feature values.
* **Branches** denote possible outcomes of the decision at each node.
* **Leaf Nodes** provide the final prediction, either a class label (for classification) or a numerical value (for regression).



The Decision Tree model mimics human decision-making, making it easy to understand and interpret.

### **2. Key Use Cases of Decision Trees**

Decision Trees are widely applied across various industries due to their interpretability and effectiveness.

#### **A. Classification Tasks**

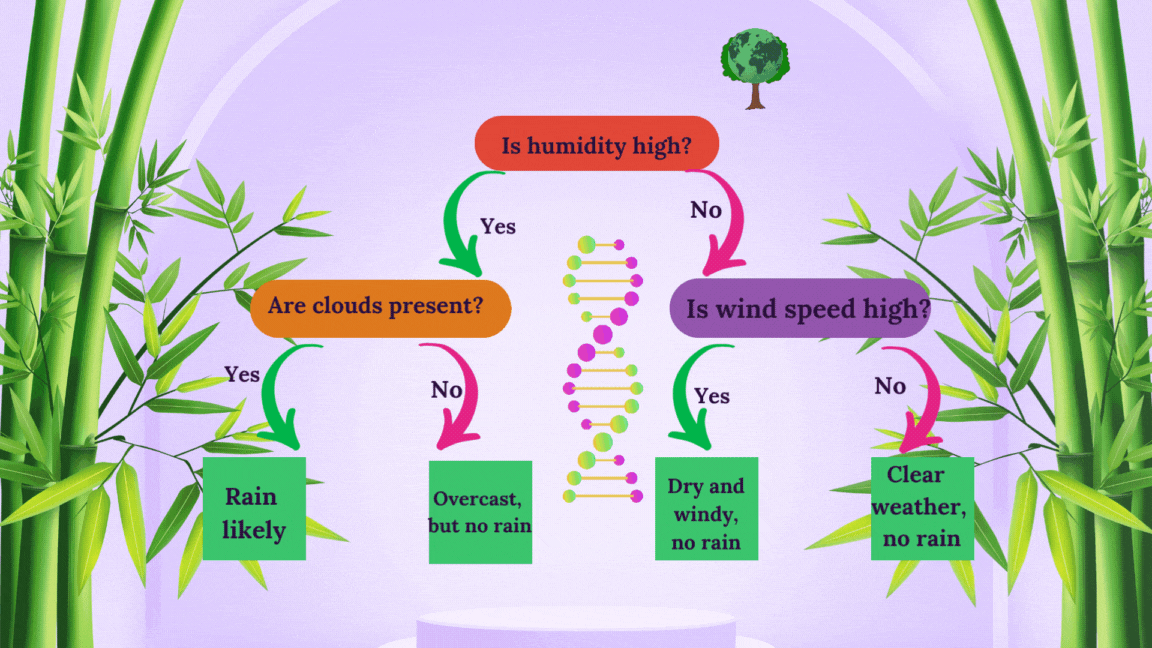
* **Spam Detection:** Identifies whether an email is spam or not based on text features.
* **Medical Diagnosis:** Predicts diseases (e.g., cancer) based on patient symptoms and test results.
* **Credit Risk Assessment:** Determines the likelihood of a loan applicant defaulting based on financial history.

#### **B. Regression Tasks**

* **House Price Prediction:** Estimates property prices using attributes like location, size, and amenities.
* **Stock Market Forecasting:** Predicts future stock prices based on historical trends and market indicators.
* **Energy Consumption Prediction:** Forecasts future energy demand based on past consumption patterns.

#### **C. Feature Selection**

* Decision Trees inherently perform **feature selection** by identifying the most important attributes during training, reducing dimensionality and improving model efficiency.



### **3. Pre-Pruning (Early Stopping)**

Pre-Pruning is a technique used to **prevent overfitting** by stopping the growth of a Decision Tree before it fully expands. This ensures the tree remains generalizable to new data.

#### **Common Pre-Pruning Techniques**

* **Depth-Based Pruning:** Restricts the maximum depth of the tree to control complexity.
* **Size-Based Pruning:** Stops the tree from growing beyond a predefined number of nodes.
* **Minimum Samples per Split:** Ensures that a split occurs only when a minimum number of samples are available.

### **4. Post-Pruning (Pruning After Tree Growth)**

Post-Pruning involves **simplifying** a fully grown tree by removing less significant branches to **enhance generalization** and reduce overfitting.

#### **Common Post-Pruning Techniques**

* **Reduced Error Pruning (REP):** Removes branches that do not contribute to increased accuracy on validation data.
* **Cost Complexity Pruning (CCP):** Uses a cost-complexity parameter to remove branches that add unnecessary complexity to the model.

### **5. Hyperparameters in Decision Trees**

Hyperparameters are user-defined settings that control the model’s structure before training. The key hyperparameters in Decision Trees include:

* **max\_depth** – Limits the tree’s depth to prevent overfitting.
* **min\_samples\_split** – Specifies the minimum number of samples required to split an internal node.
* **min\_samples\_leaf** – Defines the minimum number of samples required to be at a leaf node.
* **max\_features** – Restricts the number of features considered at each split to prevent overfitting.
* **criterion** – Defines the function for measuring the quality of a split (e.g., **Gini impurity** or **Entropy** for classification, **Mean Squared Error (MSE)** for regression).

### **6. Parameters in Decision Trees**

Unlike hyperparameters, parameters are **learned from the data** during model training. Important Decision Tree parameters include:

* **feature\_importances\_** – Indicates the significance of each feature in the decision-making process.
* **tree\_** – Represents the internal structure of the decision tree model.
* **n\_features\_** – Number of features used in the training dataset.
* **n\_classes\_** – Number of unique class labels (only applicable for classification tasks).

### **7. Popular Decision Tree Algorithms**

Several algorithms are used to construct Decision Trees, each with its own approach to splitting data:

* **ID3 (Iterative Dichotomizer 3):** Uses **Entropy** and **Information Gain** to determine the best splits.
* **CART (Classification and Regression Trees):** Uses **Gini impurity** for classification and **Mean Squared Error (MSE)** for regression.
* **C4.5:** An extension of ID3 that can handle both **categorical and continuous features** while also handling missing values effectively.

### **8. Advantages of Decision Trees**

Decision Trees offer several benefits, making them a widely used machine learning technique:

✔ **Easy to Interpret & Visualize** – The tree structure makes decision rules transparent.  
 ✔ **Handles Missing Values** – Can work effectively even when some data points are missing.  
 ✔ **Non-Parametric Model** – Does not assume any prior distribution of data.  
 ✔ **Feature Importance** – Automatically identifies and prioritizes the most important features.

### **9. Disadvantages of Decision Trees**

Despite their advantages, Decision Trees come with some limitations:

❌ **Overfitting** – Can grow excessively deep, making them too specific to the training data.  
 ❌ **Bias Toward Dominant Classes** – When trained on imbalanced datasets, they may favor majority classes.  
 ❌ **Instability** – Small changes in the dataset can lead to significantly different tree structures.  
 ❌ **Not Ideal for Complex Relationships** – Struggles to model non-linear relationships unless combined with techniques like **ensemble methods (Random Forest, Gradient Boosting)**.

## **Dataset Description:**

### **Car Evaluation Database:**

| **Feature** | **Description** |
| --- | --- |
| buying | Buying price |
| maint | Maintenance cost |
| doors | Number of doors |
| persons | Capacity in terms of persons to carry |
| lug\_boot | Luggage boot size |
| safety | Safety of the car |

### **Overview**

The **Car Evaluation Database** is a well-structured dataset designed to assess the acceptability of cars based on multiple attributes. It originates from a hierarchical decision model initially developed to demonstrate **DEX**, a decision-expert methodology. The dataset is widely used for classification tasks in machine learning, particularly for evaluating decision-making systems.

### **Dataset Attributes**

The dataset consists of **six key input attributes**, each representing a crucial factor in determining a car's suitability:

1. **Buying Price (buying)**: The initial purchase price of the car, categorized into four levels:  
   * **vhigh** (Very High)
   * **high** (High)
   * **med** (Medium)
   * **low** (Low)
2. **Maintenance Cost (maint)**: The cost required to maintain the car, also divided into four categories:  
   * **vhigh** (Very High)
   * **high** (High)
   * **med** (Medium)
   * **low** (Low)
3. **Number of Doors (doors)**: The total number of doors in the car, classified as:  
   * **2**
   * **3**
   * **4**
   * **5more** (Five or more doors)
4. **Passenger Capacity (persons)**: The number of passengers the car can accommodate:  
   * **2**
   * **4**
   * **more** (More than four passengers)
5. **Luggage Boot Size (lug\_boot)**: The available storage capacity in the trunk of the car, categorized into:  
   * **small**
   * **med** (Medium)
   * **big**
6. **Safety Level (safety)**: The estimated safety rating of the car, classified into:  
   * **low**
   * **med** (Medium)
   * **high**

### **Target Variable: Car Acceptability (CAR)**

The target variable, **CAR**, determines the overall acceptability of the car based on the given attributes. It is divided into four distinct categories:

1. **unacc** – Unacceptable
2. **acc** – Acceptable
3. **good** – Good
4. **vgood** – Very Good

### **Additional Information**

* The dataset is **complete and does not contain any missing values**.
* It has a well-defined **underlying decision structure**, making it particularly valuable for research in:  
  + **Machine learning classification tasks**
  + **Constructive induction techniques**
  + **Knowledge-based decision models**
  + **Structure discovery methods**
* Due to its categorical nature, it is often used in experiments involving **decision trees, rule-based learning, and expert systems**.

### **Source & Availability**

The **Car Evaluation Database** was originally developed as part of a demonstration for the **DEX methodology** and is publicly available for research and educational purposes. It can be accessed from the [**UCI Machine Learning Repository**](https://archive.ics.uci.edu/ml/datasets/Car+Evaluation).

## **4. Workflow:**

* **Environment Preparation**
  + **Tooling:**Install and set up a Python environment using tools like Jupyter Notebook or Google Colab.
  + **Library Installation:**Ensure you have all necessary libraries installed, such as:
    - **pandas** and **NumPy** for data manipulation
    - **Matplotlib** and **Seaborn** for visualization
    - **Ucimlrepo:** Fetch dataset from UCI repository
    - **scikit-learn** for additional machine learning tasks

**2. Data Understanding & Exploration**

- Load dataset and examine structure

- Review metadata and variable information

- Perform initial data inspection:

* - Check shape and columns
* - Examine data types (df.info())
* - Generate descriptive statistics (df.describe())
* - Analyze class distribution

**3. Exploratory Data Analysis (EDA)**

- Visualize target variable distribution

- Analyze categorical features:

* - Buying price
* - Maintenance cost
* - Number of doors
* - Passenger capacity
* - Luggage boot size
* - Safety rating

- Create correlation heatmap of encoded features

- Investigate key feature interactions (safety vs. passenger capacity)

**4. Data Preprocessing**

- Check for missing values

- Verify for duplicate entries

- Implement preprocessing pipeline:

* - Encode categorical features (LabelEncoder)
* - Separate features and target variable
* - Split data into train/test sets (70/30 split with stratification)

**5. Model Development - Decision Tree Classifier**

* - Initialize baseline Decision Tree Classifier
* - Train model on training data
* - Evaluate performance:
* Accuracy score
* Classification report
* Confusion matrix
* - Visualize decision tree structure
* - Extract and examine decision rules

**6. Model Optimization**

- Implement pre-pruning via GridSearchCV:

* - Tune hyperparameters (max\_depth, min\_samples\_split, etc.)
* - Select best model based on cross-validation

- Apply post-pruning with cost complexity pruning:

* - Calculate ccp\_alphas
* - Evaluate accuracy across alpha values
* - Select optimal pruning strength

- Compare pre-pruning vs. post-pruning results

**7. Feature Importance Analysis**

- Extract and rank feature importances

- Visualize relative importance of features

- Interpret business implications of key features

**8. Alternative Approach - Decision Tree Regressor**

- Implement regression-based approach

- Train Decision Tree Regressor

- Evaluate performance:

* - Mean squared error
* - Rounded prediction accuracy
* - Classification metrics

**9. Documentation & Reporting**

- Generate visualizations of key findings

- Document interpretation of results

- Prepare model evaluation metrics

- Save final models and visualizations

## **5. Code Explanation:**

5.1 Import the Important Libraries:

| # Import pandas library for data manipulation and analysis import pandas as pd  # Import numpy library for numerical computations import numpy as np  # Import seaborn library for data visualization import seaborn as sns  # Import matplotlib library for creating static, animated, and interactive visualizations import matplotlib.pyplot as plt   # Import train\_test\_split function from scikit-learn for splitting data into training and testing sets from sklearn.model\_selection import train\_test\_split  # Import LabelEncoder from scikit-learn for encoding categorical variables from sklearn.preprocessing import LabelEncoder  # Import DecisionTreeClassifier, DecisionTreeRegressor, plot\_tree, and export\_text from scikit-learn for building decision tree models from sklearn.tree import DecisionTreeClassifier, DecisionTreeRegressor, plot\_tree, export\_text  # Import accuracy\_score, classification\_report, confusion\_matrix, ConfusionMatrixDisplay, and mean\_squared\_error from scikit-learn for evaluating model performance from sklearn.metrics import (accuracy\_score, classification\_report, confusion\_matrix, ConfusionMatrixDisplay, mean\_squared\_error)  # Import GridSearchCV from scikit-learn for performing grid search hyperparameter tuning from sklearn.model\_selection import GridSearchCV  # Import randint from scipy.stats for generating random integers for hyperparameter tuning from scipy.stats import randint  # Import export\_graphviz from scikit-learn for exporting decision tree models to Graphviz format from sklearn.tree import export\_graphviz  # Import graphviz library for visualizing decision tree models import graphviz  # Import fetch\_ucirepo from ucimlrepo for fetching datasets from the UCI Machine Learning Repository from ucimlrepo import fetch\_ucirepo |
| --- |

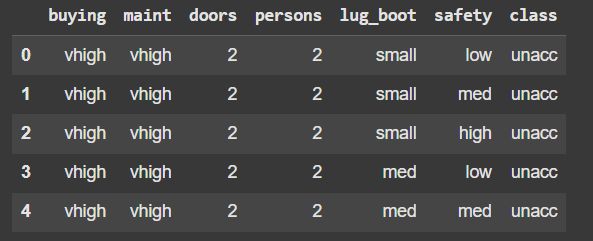
5.2

| # Import the fetch\_ucirepo function from the ucimlrepo library to fetch datasets from the UCI Machine Learning Repository from ucimlrepo import fetch\_ucirepo  # Fetch the Car Evaluation dataset (id=19) from the UCI Machine Learning Repository car\_evaluation = fetch\_ucirepo(id=19)  # Extract the feature data (X) and target data (y) from the fetched dataset # X contains the input features X = car\_evaluation.data.features  # y contains the target variable y = car\_evaluation.data.targets  # Print the metadata associated with the dataset, such as the dataset name, description, and citation print(car\_evaluation.metadata)  # Print information about the variables in the dataset, such as their names, data types, and descriptions print(car\_evaluation.variables) |
| --- |

This code snippet demonstrates how to fetch the Car Evaluation dataset from the UCI Machine Learning Repository using the ucimlrepo library. It extracts the feature data (X) and target data (y) from the dataset and prints the metadata and variable information associated with the dataset. This dataset can be used for classification tasks, such as predicting the car's acceptance based on its attributes.

| # Import the pandas library and assign it the alias 'pd' for convenience import pandas as pd  # Fetch the Car Evaluation dataset (id=19) from the UCI Machine Learning Repository car\_evaluation = fetch\_ucirepo(id=19)  # Combine the feature data and target data into a single Pandas DataFrame # The axis=1 argument specifies that the concatenation should occur along the columns (i.e., horizontally) df = pd.concat([car\_evaluation.data.features, car\_evaluation.data.targets], axis=1)  # Display the first few rows of the DataFrame using the head() method df.head() |
| --- |

This code snippet fetches the Car Evaluation dataset and combines its feature and target data into a single Pandas DataFrame. It then displays the first few rows of the DataFrame using the head() method, allowing for a quick preview of the data.



| # Print a concise summary of the DataFrame, including data types and summary statistics df.info() |
| --- |

# Output:

# <class 'pandas.core.frame.DataFrame'>

# RangeIndex: 1728 entries, 0 to 1727

# Data columns (total 7 columns):

# # Column Non-Null Count Dtype

# --- ------ -------------- -----

# 0 buying 1728 non-null object

# 1 maint 1728 non-null object

# 2 doors 1728 non-null object

# 3 persons 1728 non-null object

# 4 lug\_boot 1728 non-null object

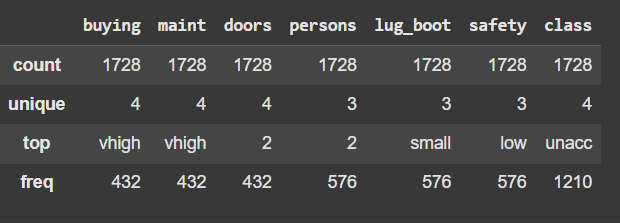
# 5 safety 1728 non-null object

# 6 class 1728 non-null object

# dtypes: object(7)

| # Print summary statistics for the DataFrame, excluding NaN values df.describe() |
| --- |

# Output:



This code snippet provides an overview of the Car Evaluation dataset. It displays the shape of the DataFrame, column names, data types, and summary statistics. The summary statistics include count, unique values, top values, and frequency of each variable.

Let’s See class Distribution:

print(df['class'].value\_counts())

class

unacc 1210

acc 384

good 69

vgood 65

Name: count, dtype: int64

In conclusion, this analysis provides an overview of the Car Evaluation dataset. The dataset consists of 1728 samples, each described by 6 attributes (buying, maint, doors, persons, lug\_boot, and safety) and classified into one of four classes (unacc, acc, good, and vgood).

The class distribution is imbalanced, with:

- 1210 (70%) samples classified as "unacc" (unacceptable)

- 384 (22%) samples classified as "acc" (acceptable)

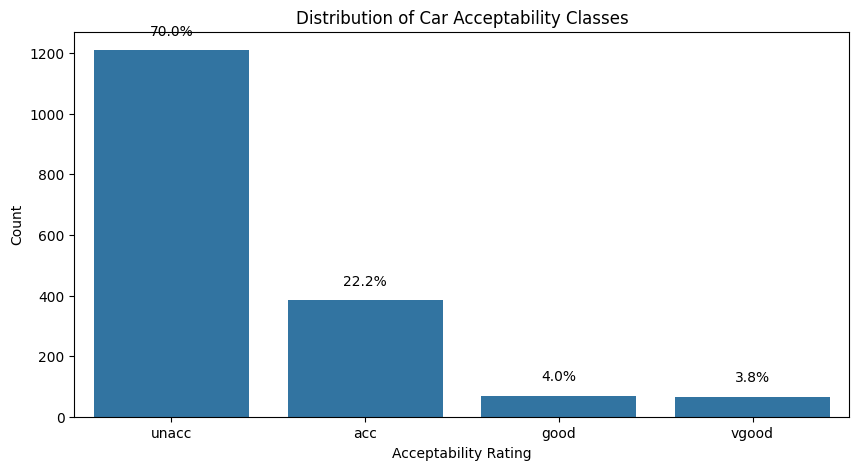
- 69 (4%) samples classified as "good"

- 65 (4%) samples classified as "vgood" (very good)

This imbalance may impact the performance of machine learning models trained on this dataset. Future analysis can focus on handling the class imbalance and developing predictive models to classify cars into their respective categories.

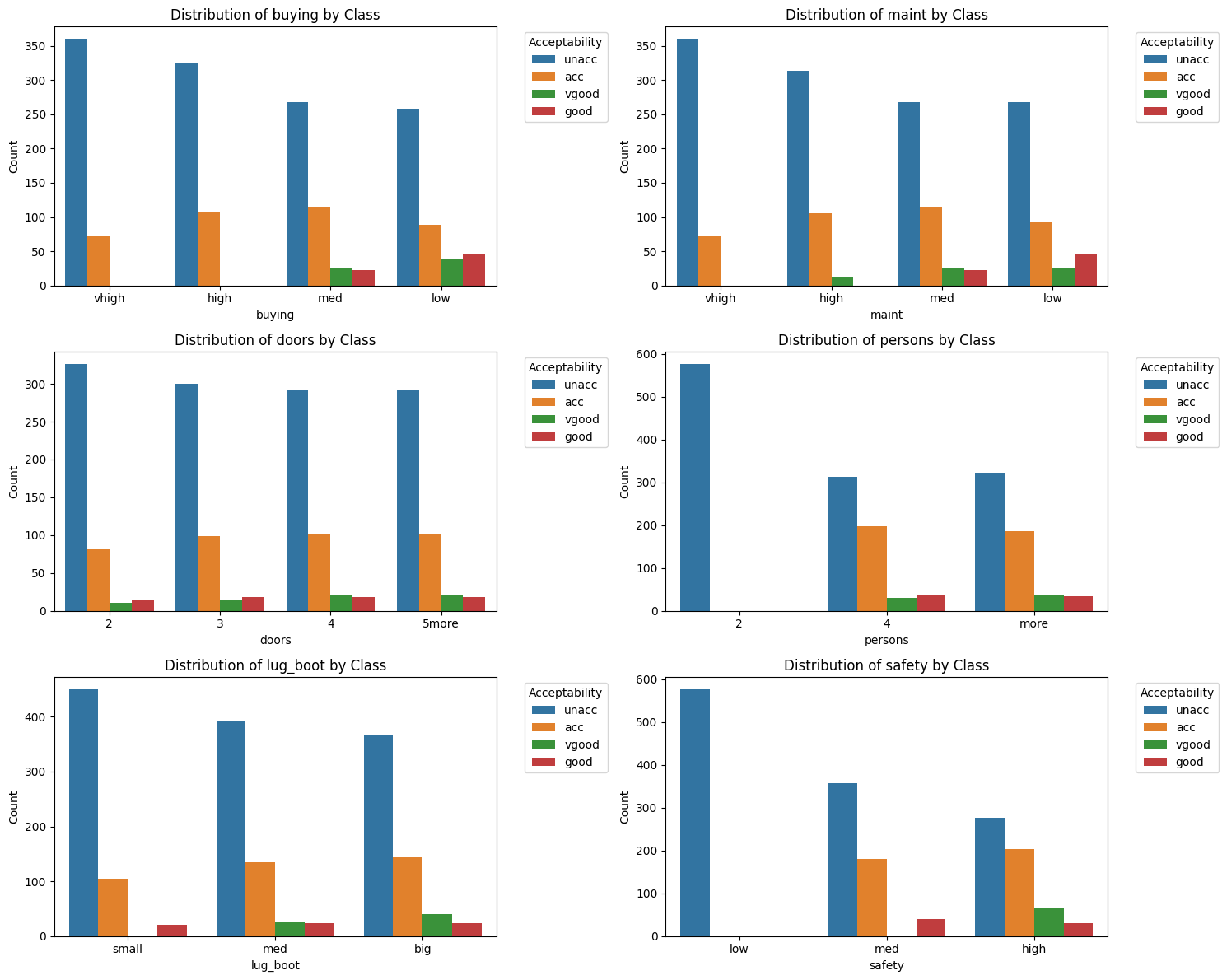
## EDA:

| # 1. Target Distribution # ======================  # Create a new figure with a specified size plt.figure(figsize=(10, 5))  # Use seaborn's countplot function to create a bar chart of the target variable ('class') # The 'order' parameter is used to sort the bars in descending order of frequency ax = sns.countplot(data=df, x='class', order=df['class'].value\_counts().index)  # Set the title of the plot plt.title('Distribution of Car Acceptability Classes')  # Set the labels for the x and y axes plt.xlabel('Acceptability Rating') plt.ylabel('Count')  # Calculate the total number of samples total = len(df)  # Iterate over each bar in the plot for p in ax.patches:  # Calculate the percentage of samples in each class  percentage = f'{100 \* p.get\_height()/total:.1f}%'    # Get the x and y coordinates of the top of the bar  x = p.get\_x() + p.get\_width()/2  y = p.get\_height() + 50    # Annotate the bar with its percentage value  ax.annotate(percentage, (x, y), ha='center')  # Display the plot plt.show() |
| --- |



From the chart, it is clear that the vast majority of cars are classified as “unacceptable” (around 70%), while about one-fifth are deemed “acceptable” (roughly 22%). Only a small fraction—less than 10% combined—are rated “good” or “very good.” This indicates that most cars in the dataset fall into the lowest acceptability category, with relatively few vehicles meeting higher standards.

| # ================================= # 2. Categorical Features Analysis # =================================  # Define the list of categorical features categorical\_features = ['buying', 'maint', 'doors', 'persons', 'lug\_boot', 'safety']  # Create a new figure with a specified size plt.figure(figsize=(15, 12))  # Iterate over each categorical feature for I, feature in enumerate(categorical\_features, 1):  # Create a subplot for each feature  plt.subplot(3, 2, I)    # Use seaborn's countplot function to create a bar chart of the feature  # The 'hue' parameter is used to color the bars by class  # The 'dodge' parameter is used to create a dodged bar chart  sns.countplot(data=df, x=feature, hue='class', dodge=True)    # Set the title of the subplot  plt.title(f'Distribution of {feature} by Class')    # Set the labels for the x and y axes  plt.xlabel(feature)  plt.ylabel('Count')    # Add a legend to the subplot  plt.legend(title='Acceptability', bbox\_to\_anchor=(1.05, 1), loc='upper left')    # Adjust the layout of the subplots  plt.tight\_layout()    # Display the plot  plt.show() |
| --- |



Key Patterns from Car Evaluation Data

Buying & Maintenance Costs

* "vhigh" and "high" buying or maintenance costs are predominantly labeled as "unacc."
* Lower costs ("med" or "low") have a better chance of being rated "acc," "good," or "vgood," though "unacc" remains common.

Number of Doors & Number of Persons

* The "doors" attribute shows "unacc" as the largest category across all options, but there is a slight increase in "acc," "good," and "vgood" ratings when more doors are available.
* The "persons" attribute has a strong effect:  
  + Cars with only 2 seats are mostly labeled as "unacc."
  + Cars with 4 or more seats yield higher "acc," "good," and "vgood" ratings.

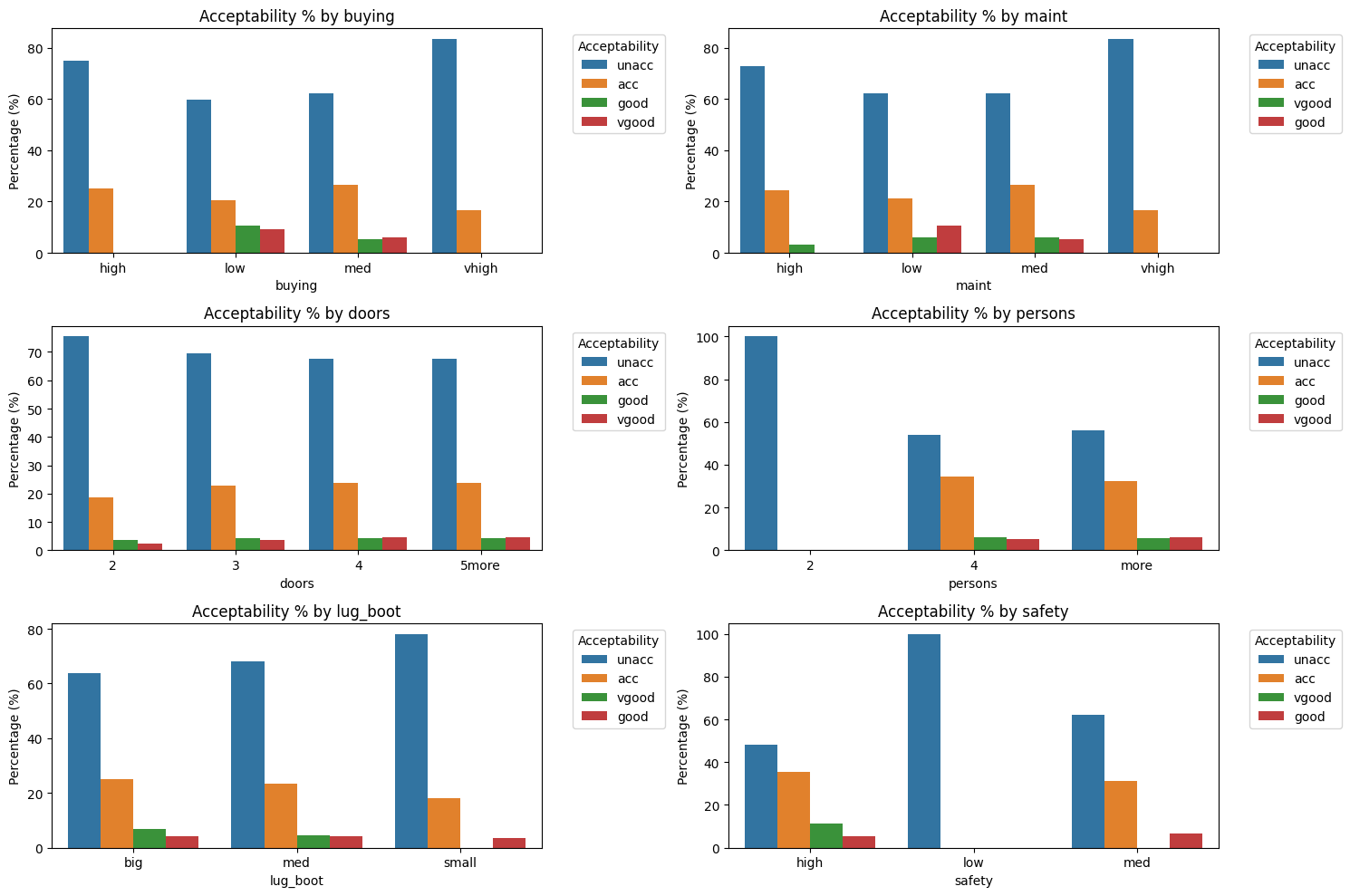
Luggage Boot Size & Safety

* For "lug\_boot," a "small" capacity skews heavily toward "unacc," whereas "big" luggage capacity sees somewhat more favorable ratings.
* "Safety" is one of the most decisive attributes:  
  + "Low" safety is almost entirely labeled "unacc."
  + "Med" and especially "high" safety correlate with significantly higher "acc," "good," and "vgood" ratings.

Conclusion

While "unacc" is the most frequent category across almost all attribute values, higher safety, greater passenger capacity, and lower buying/maintenance costs significantly increase the likelihood of better car acceptability ratings.

| # ================================== # 3. Stacked Percentage Bar Charts # ==================================  # Create a new figure with a specified size plt.figure(figsize=(15, 10))  # Iterate over each categorical feature for i, feature in enumerate(categorical\_features, 1):  # Create a subplot for each feature  plt.subplot(3, 2, i)    # Calculate the percentage of each class for each category in the feature  prop\_df = (df.groupby(feature)['class']  .value\_counts(normalize=True) # Normalize counts to proportions  .mul(100) # Convert proportions to percentages  .rename('percentage') # Rename the column  .reset\_index()) # Reset the index    # Use seaborn's barplot function to create a stacked percentage bar chart  sns.barplot(data=prop\_df, x=feature, y='percentage', hue='class')    # Set the title of the subplot  plt.title(f'Acceptability % by {feature}')    # Set the label for the y-axis  plt.ylabel('Percentage (%)')    # Add a legend to the subplot  plt.legend(title='Acceptability', bbox\_to\_anchor=(1.05, 1), loc='upper left')    # Adjust the layout of the subplots  plt.tight\_layout()    # Display the plot  plt.show() |
| --- |



Key Trends from Percentage-Based Plots

Buying Price & Maintenance Cost

* "vhigh" or "high" costs strongly correlate with "unacc" ratings.
* As costs drop to "med" or "low," the share of "acc," "good," and "vgood" increases, though "unacc" remains significant.

Doors & Persons

* Fewer doors (2 or 3) yield a higher proportion of "unacc." More doors (4 or 5+) slightly improve acceptability.
* The "persons" attribute has a pronounced effect:  
  + Cars that seat only 2 people are overwhelmingly labeled as "unacc."
  + Cars with 4 or more seats lead to notably higher "acc," "good," and "vgood" percentages.

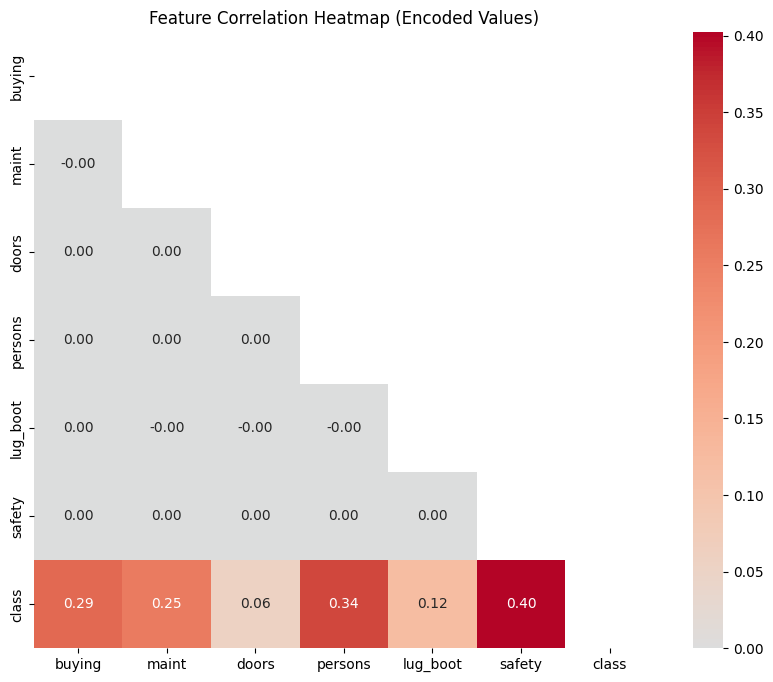
Luggage Boot Size & Safety

* A "big" luggage boot is more likely to earn higher acceptability ratings than a "small" one.
* Safety is especially influential:  
  + "Low" safety ratings are almost entirely labeled as "unacc."
  + "High" safety significantly increases the share of "acc," "good," and "vgood."

Conclusion

Lower buying and maintenance costs, the ability to seat more passengers, a larger luggage capacity, and higher safety ratings all tend to improve a car’s acceptability.

| # ================================== # 4. Pairwise Relationships (Heatmap) # ==================================  # Create a temporary copy of the DataFrame to encode categorical variables temp\_df = df.copy()  # Encode categorical variables temporarily for correlation analysis for col in categorical\_features + ['class']:  # Use pd.factorize to encode categorical variables as numerical values  temp\_df[col] = pd.factorize(temp\_df[col])[0]  # Create a new figure with a specified size plt.figure(figsize=(10, 8))  # Calculate the correlation matrix of the temporary DataFrame corr = temp\_df.corr()  # Create a mask to hide the upper triangle of the correlation matrix mask = np.triu(np.ones\_like(corr, dtype=bool))  # Use seaborn's heatmap function to create a heatmap of the correlation matrix sns.heatmap(corr, mask=mask, annot=True, cmap='coolwarm', center=0, fmt='.2f')  # Set the title of the heatmap plt.title('Feature Correlation Heatmap (Encoded Values)')  # Display the plot plt.show() |
| --- |

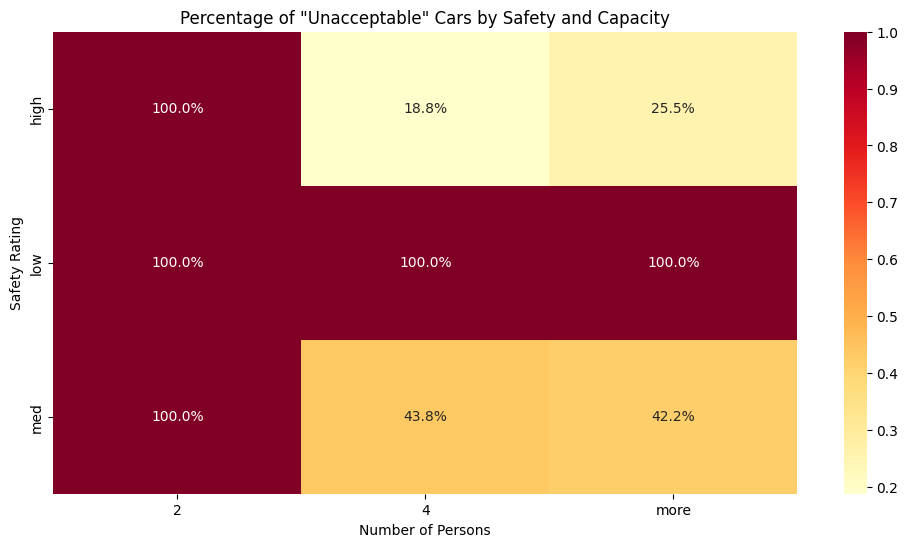


**Conclusion**

Safety has the strongest correlation with the car’s class (0.40), followed by persons (0.34), buying (0.29), and maint (0.25). Doors (0.06) and lug\_boot (0.12) show relatively weaker correlations with the class.

Correlations among the features themselves are mostly near zero, indicating they’re largely independent of each other.

| # ================================== # 5. Safety vs Persons (Key Features) # ==================================  # Create a new figure with a specified size plt.figure(figsize=(12, 6))  # Perform a cross-tabulation to create a heatmap of the relationship between safety, persons, and class # The cross-tabulation calculates the percentage of cars that are unacceptable for each combination of safety and persons ct = pd.crosstab(df['safety'], df['persons'], values=df['class'], aggfunc=lambda x: (x == 'unacc').mean())  # Use seaborn's heatmap function to create a heatmap of the cross-tabulation sns.heatmap(ct, annot=True, fmt='.1%', cmap='YlOrRd')  # Set the title of the heatmap plt.title('Percentage of "Unacceptable" Cars by Safety and Capacity')  # Set the labels for the x and y axes plt.xlabel('Number of Persons') plt.ylabel('Safety Rating')  # Display the plot plt.show() |
| --- |



**Conclusion :**

* When safety is low, all cars (100%) are “unacceptable,” regardless of capacity.
* When safety is medium, having only 2 persons still results in 100% “unacceptable,” but the rate drops to around 43% for 4 or more persons.
* When safety is high and 2 persons, 100% of cars are “unacceptable.”
* However, for 4 or more persons, the “unacceptable” percentage falls to about 19–25%.

## **Data Preprocessing;**

# Check for null value

df.isnull().sum() # 0

So, There is no null value in this dataset.

# Check for duplicate rows

print(df.duplicated().sum()) #0

There is no duplicate value in this Dataset.

## **As, It is imbalanced Dataset:**

**unacc: 1210 instances**

**acc: 384 instances**

**good: 69 instances**

**vgood: 65 instances**

## Rationale for Merging

**Simplifying the Classification Task**

By merging "good" and "vgood" into "acc", you convert the problem from a four-class classification into a binary classification (i.e., acceptable vs. unacceptable). This approach can reduce model complexity and might help the classifier concentrate on the primary decision boundary—whether a car is acceptable or not.

**Addressing Class Imbalance**

Even after merging, you'll have an imbalance (unacc: 1210 vs. acc: 518), but the grouping helps by pooling the minority acceptable classes. It can also make the cost of misclassification easier to interpret from a business perspective if the end goal is simply to decide if a car meets acceptable criteria.

**Reducing Variance in Minor Categories**

Models trained on very low-frequency classes (like 69 or 65 samples) may have high variance and unstable predictive performance. Merging can lead to more robust estimates for the acceptable class.

**Interpretability**

A binary decision (acceptable versus unacceptable) is often easier for stakeholders to understand and act upon than four finely divided categories.

print("Before merging:")

print(df['class'].value\_counts())

# Merge 'good' and 'vgood' into 'acc'

df['class'] = df['class'].replace({'good': 'acc', 'vgood': 'acc'})

print("\nAfter merging:")

print(df['class'].value\_counts())

Before merging:

class

unacc 1210

acc 384

good 69

vgood 65

Name: count, dtype: int64

After merging:

class

unacc 1210

acc 518

Name: count, dtype: int64

So, Approx it is balanced (70:30) . so, we can go ahead.

| # Define a function to preprocess the data def preprocess\_data(df):  """  Preprocess the data by encoding categorical features and the target variable,  separating features and target, and splitting the data into training and testing sets.    Parameters:  df (DataFrame): The input DataFrame.    Returns:  X\_train (DataFrame): The training features.  X\_test (DataFrame): The testing features.  y\_train (Series): The training target.  y\_test (Series): The testing target.  class\_mapping (dict): A dictionary mapping the encoded target values to their original labels.  """    # Create a copy of the DataFrame to avoid modifying the original data  df\_processed = df.copy()    # Initialize a LabelEncoder to encode categorical features and the target variable  le = LabelEncoder()    # Define the categorical feature columns  categorical\_cols = ['buying', 'maint', 'doors', 'persons', 'lug\_boot', 'safety']    # Encode the categorical features using the LabelEncoder  for col in categorical\_cols:  df\_processed[col] = le.fit\_transform(df[col])    # Encode the target variable using the LabelEncoder  df\_processed['class'] = le.fit\_transform(df['class'])    # Create a dictionary mapping the encoded target values to their original labels  class\_mapping = {i: label for i, label in enumerate(le.classes\_)}    # Print the class mapping for interpretation  print("\nClass mapping:", class\_mapping)    # Separate the features and target  X = df\_processed.drop('class', axis=1)  y = df\_processed['class']    # Split the data into training and testing sets using stratified sampling  X\_train, X\_test, y\_train, y\_test = train\_test\_split(X, y, test\_size=0.3, random\_state=42, stratify=y)    # Return the preprocessed data  return X\_train, X\_test, y\_train, y\_test, class\_mapping  # Step 2: Train/Test Split of Data X\_train, X\_test, y\_train, y\_test, class\_mapping = preprocess\_data(df) |
| --- |

This code preprocesses the Car Evaluation dataset by: - Encoding categorical features and the target variable using LabelEncoder - Separating features and target - Splitting the data into training (70%) and testing sets (30%) using stratified sampling The preprocessed data is then returned for further analysis or modeling.

## **Step 3: Model Building - Decision Tree Classifier**

def build\_decision\_tree\_classifier(X\_train, y\_train, X\_test, y\_test, class\_mapping):

Step 1: Initialize and Fit the Classifier

| clf = DecisionTreeClassifier(random\_state=42) clf.fit(X\_train, y\_train) |
| --- |

- Initialize a Decision Tree Classifier with a random state of 42 for reproducibility.

- Fit the classifier to the training data (X\_train and y\_train).

**Step 2: Make Predictions and Evaluate the Model**

| y\_pred = clf.predict(X\_test) accuracy = accuracy\_score(y\_test, y\_pred) print(f"Initial Accuracy: {accuracy:.4f}") |
| --- |

- Make predictions on the test data (X\_test) using the trained classifier.

- Calculate the accuracy of the model by comparing the predicted labels (y\_pred) with the actual labels (y\_test).

- Print the initial accuracy of the model.

**Step 3: Classification Report and Confusion Matrix**

| print("\nClassification Report:") print(classification\_report(y\_test, y\_pred, target\_names=class\_mapping.values()))  cm = confusion\_matrix(y\_test, y\_pred) disp = ConfusionMatrixDisplay(confusion\_matrix=cm, display\_labels=class\_mapping.values()) disp.plot(cmap=plt.cm.Blues) plt.title("Confusion Matrix - Initial Decision Tree") plt.show() |
| --- |

- Print a classification report that includes precision, recall, F1 score, and support for each class.

- Create a confusion matrix to evaluate the model's performance.

- Display the confusion matrix as a heatmap.

**Step 4: Visualize the Decision Tree**

| plt.figure(figsize=(30, 15)) plot\_tree(clf, filled=True, feature\_names=X\_train.columns, class\_names=list(class\_mapping.values()), rounded=True) plt.title("Initial Decision Tree (Unpruned)") plt.savefig('decision\_tree.png', dpi=300) plt.show() |
| --- |

- Create a plot of the decision tree using the plot\_tree function.

- Customize the plot by setting the figure size, filling the nodes with colors, and displaying feature names and class names.

- Save the plot as an image file named "decision\_tree.png".

Step 5: Print Decision Tree Rules

| tree\_rules = export\_text(clf, feature\_names=list(X\_train.columns)) print("\nDecision Tree Rules (first 100 lines):") print("\n".join(tree\_rules.split("\n")[:100])) |
| --- |

- Export the decision tree rules as a text string using the export\_text function.

- Print the first 100 lines of the decision tree rules.

**The function returns the trained Decision Tree Classifier.**

The Output:

=== Decision Tree Classifier ===

Initial Accuracy: 0.9884

Classification Report:

precision recall f1-score support

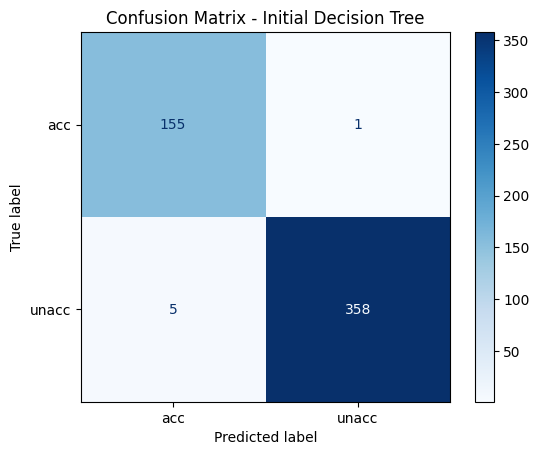
acc 0.97 0.99 0.98 156

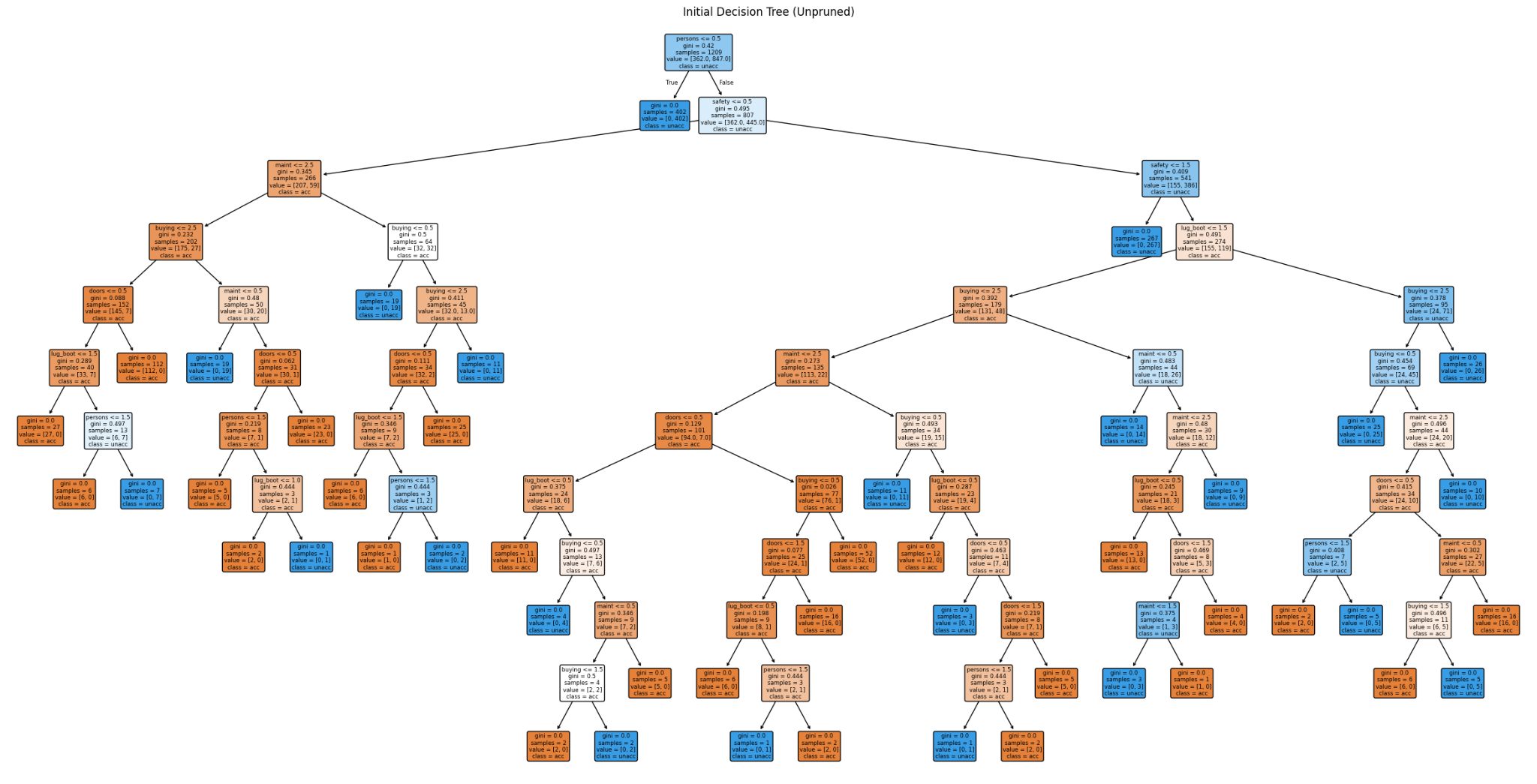
unacc 1.00 0.99 0.99 363

accuracy 0.99 519

macro avg 0.98 0.99 0.99 519

weighted avg 0.99 0.99 0.99 519

****

****

**Link:** [**https://res.cloudinary.com/dqcvlj2il/image/upload/v1744278172/1\_zibakb.png**](https://res.cloudinary.com/dqcvlj2il/image/upload/v1744278172/1_zibakb.png)

**Final Decision Tree Model**

**Overall Performance**

* Accuracy: 0.9884, indicating an exceptionally strong classification capability.
* The model demonstrates excellent performance across both classes, with precision, recall, and F1-scores above 0.97.
* The model is particularly strong at identifying “unacc” cars, which is the majority class.

**Confusion Matrix Highlights**

**Unacceptable Class (“unacc”)**

* The model demonstrates exceptional confidence and accuracy in rejecting unacceptable cars, with:
* - Precision: 1.00: The model is virtually flawless in its ability to correctly identify unacceptable cars, with almost no false positives.
* - Recall: 0.99: The model is highly effective in capturing almost all instances of unacceptable cars, with only a tiny fraction of false negatives.
* - F1-score: 0.99: The model's overall performance on the unacceptable class is outstanding, reflecting its ability to balance precision and recall.

**Acceptable Class (“acc”, including original acc, good, and vgood)**

* While still strong, the model's performance on the acceptable class shows slightly more variability:
* - Precision: 0.97: The model is highly accurate in identifying acceptable cars, but with a slightly higher rate of false positives compared to the unacceptable class.
* - Recall: 0.99: The model is still highly effective in capturing almost all instances of acceptable cars, with only a small fraction of false negatives.
* - F1-score: 0.98: The model's overall performance on the acceptable class is very strong, reflecting its ability to balance precision and recall.

Overall, the model's performance suggests that it is highly effective in distinguishing between acceptable and unacceptable cars, with only minor variations in its performance between the two classes.

**Key Decision Rules**

* The model primarily splits on "persons" (seating capacity) as the first decision-making attribute, followed by "safety", indicating their dominant influence on acceptability.
* Once the model differentiates based on capacity and safety, it refines the classification using cost factors such as "maint" (maintenance cost) and "buying" price.
* Attributes like "doors" and "lug\_boot" (luggage capacity) come into play in later splits, providing finer distinctions in borderline cases.
* The decision tree structure strongly aligns with earlier insights: higher safety and greater seating capacity significantly boost a car’s rating, while cost-related factors help determine its final classification.

## **Applying Pruning Techniques**

This code applies pruning techniques to the decision tree classifier to prevent overfitting and improve its performance on unseen data.

| # Pruning Techniques def apply\_pruning\_techniques(X\_train, y\_train, X\_test, y\_test, class\_mapping):  print("\n=== Applying Pruning Techniques ===")   # --------------------------------------  # Pre-Pruning using Grid Search  # --------------------------------------  print("\n--- Pre-Pruning with Grid Search ---")   # Define the parameter grid for hyperparameter tuning  param\_grid = {  'max\_depth': [3, 5, 7, 10, None], # Control the depth of the tree to prevent overfitting  'min\_samples\_split': [2, 5, 10], # Minimum samples required to split a node  'min\_samples\_leaf': [1, 2, 4], # Minimum samples required at a leaf node  'criterion': ['gini', 'entropy'], # Splitting criteria  'max\_features': [ 'sqrt', 'log2'] # Number of features to consider when splitting  }   # Perform Grid Search with 5-fold cross-validation to find the best parameters  grid\_search = GridSearchCV(DecisionTreeClassifier(random\_state=42),  param\_grid, cv=5, scoring='accuracy')  grid\_search.fit(X\_train, y\_train)   # Retrieve the best decision tree model  best\_clf = grid\_search.best\_estimator\_  print(f"Best parameters: {grid\_search.best\_params\_}")  print(f"Best cross-validation accuracy: {grid\_search.best\_score\_:.4f}")   # Evaluate the best model on the test set  y\_pred = best\_clf.predict(X\_test)  accuracy = accuracy\_score(y\_test, y\_pred)  print(f"Test accuracy with pre-pruning: {accuracy:.4f}")   # Visualize the pre-pruned decision tree  plt.figure(figsize=(30, 15))  plot\_tree(best\_clf, filled=True, feature\_names=X\_train.columns,  class\_names=list(class\_mapping.values()), rounded=True)  plt.title("Decision Tree with Pre-Pruning")  plt.savefig('decision\_tree\_Pre\_Pruning.png', dpi=300)  plt.show()   # --------------------------------------  # Post-Pruning using Cost Complexity Pruning  # --------------------------------------  print("\n--- Post-Pruning with Cost Complexity Pruning ---")   # Get the cost complexity pruning path to determine optimal alpha values  path = best\_clf.cost\_complexity\_pruning\_path(X\_train, y\_train)  ccp\_alphas, impurities = path.ccp\_alphas, path.impurities # Extract alphas and impurities   # Train multiple decision trees with different ccp\_alpha values  clfs = []  for ccp\_alpha in ccp\_alphas:  clf = DecisionTreeClassifier(random\_state=42, ccp\_alpha=ccp\_alpha)  clf.fit(X\_train, y\_train)  clfs.append(clf)   # Compute accuracy scores for training and testing sets  train\_scores = [clf.score(X\_train, y\_train) for clf in clfs]  test\_scores = [clf.score(X\_test, y\_test) for clf in clfs]   # Plot accuracy vs. alpha values to visualize impact of pruning  fig, ax = plt.subplots()  ax.set\_xlabel("alpha")  ax.set\_ylabel("accuracy")  ax.set\_title("Accuracy vs alpha for training and testing sets")  ax.plot(ccp\_alphas, train\_scores, marker='o', label="train", drawstyle="steps-post")  ax.plot(ccp\_alphas, test\_scores, marker='o', label="test", drawstyle="steps-post")  ax.legend()  plt.show()   # Select the best alpha value that maximizes test set accuracy  best\_alpha = ccp\_alphas[np.argmax(test\_scores)]  print(f"Best alpha for post-pruning: {best\_alpha:.4f}")   # Train the final decision tree model with the optimal alpha value  final\_clf = DecisionTreeClassifier(random\_state=42, ccp\_alpha=best\_alpha)  final\_clf.fit(X\_train, y\_train)   # Evaluate the final pruned model on the test set  y\_pred = final\_clf.predict(X\_test)  accuracy = accuracy\_score(y\_test, y\_pred)  print(f"Test accuracy with post-pruning: {accuracy:.4f}")   # Visualize the final pruned decision tree  plt.figure(figsize=(30, 15))  plot\_tree(final\_clf, filled=True, feature\_names=X\_train.columns,  class\_names=list(class\_mapping.values()), rounded=True)  plt.title("Final Pruned Decision Tree")  plt.savefig('Final\_Pruned\_decision\_tree.png', dpi=300)  plt.show()   return final\_clf   # Step 4: Apply pruning techniques to improve model generalization final\_clf = apply\_pruning\_techniques(X\_train, y\_train, X\_test, y\_test, class\_mapping) |
| --- |

Here's a detailed explanation of the code:

**1. Applying Pruning Techniques to Improve Model Generalization**

This section applies pruning techniques to a decision tree classifier to improve its generalization performance on unseen data.

**2.Pre-Pruning using Grid Search**

Pre-pruning involves tuning hyperparameters to prevent overfitting. Grid search is used to find the optimal hyperparameters.

1. **- Hyperparameter Tuning:** The hyperparameter grid includes parameters such as max\_depth, min\_samples\_split, min\_samples\_leaf, criterion, and max\_features.
2. **- Grid Search with Cross-Validation:** Grid search is performed with 5-fold cross-validation to find the best parameters.
3. **- Best Model Selection:** The best decision tree model is retrieved based on the grid search results.
4. **- Evaluation on Test Set:** The best model is evaluated on the test set to estimate its generalization performance.
5. **- Visualization:** The pre-pruned decision tree is visualized using the plot\_tree function.

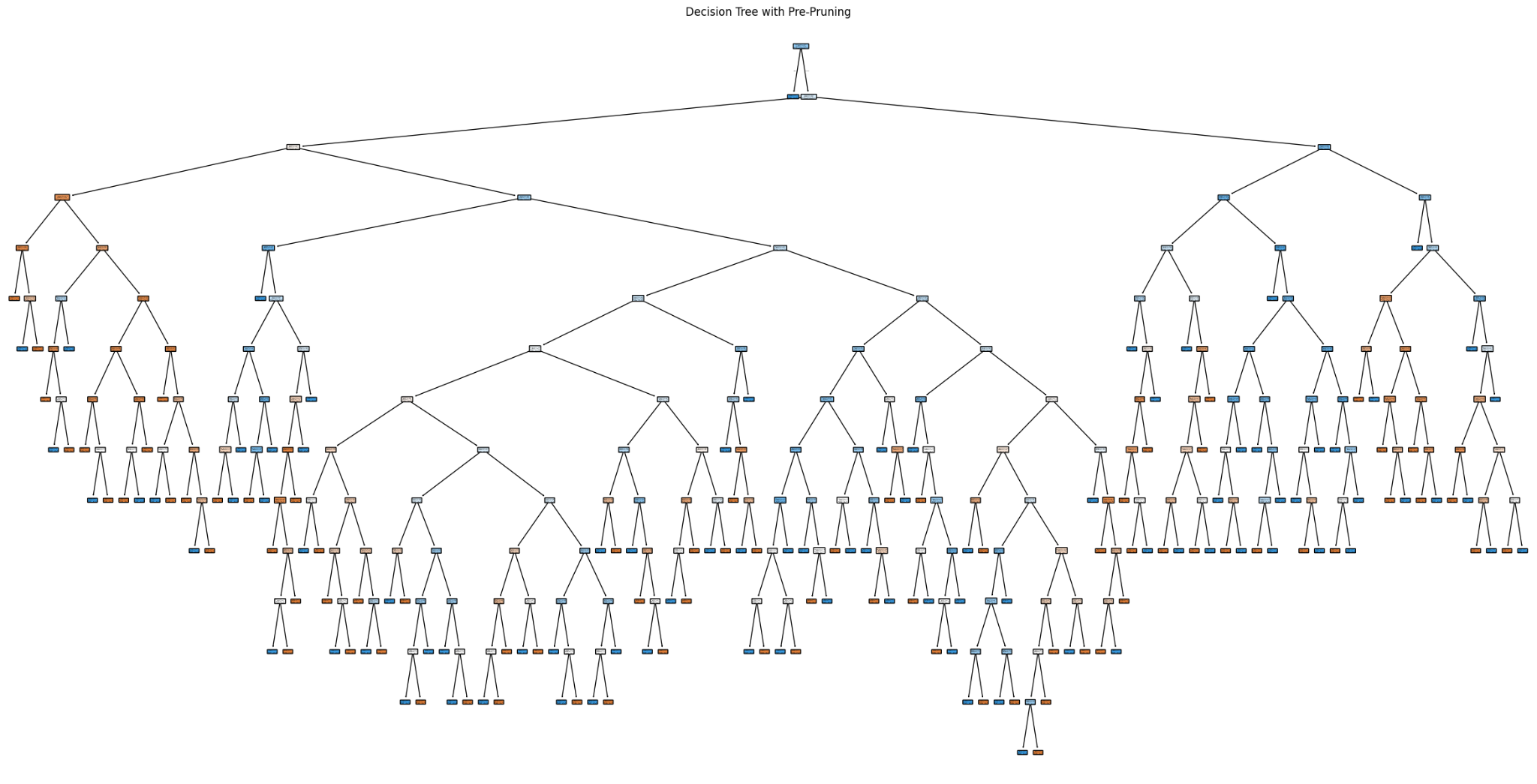
**3.Post-Pruning using Cost Complexity Pruning**

Post-pruning involves removing branches of the decision tree to reduce overfitting. Cost complexity pruning is used to determine the optimal alpha values.

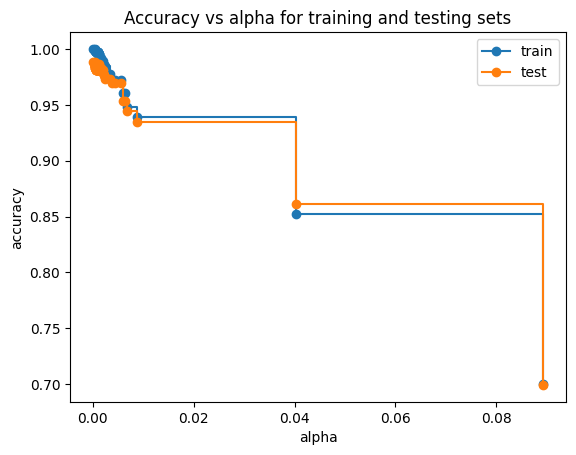
1. **- Cost Complexity Pruning Path:** The cost complexity pruning path is computed to determine the optimal alpha values.
2. **- Training Multiple Decision Trees:** Multiple decision trees are trained with different ccp\_alpha values.
3. **- Accuracy Scores:** Accuracy scores are computed for the training and testing sets.
4. **- Plotting Accuracy vs Alpha:** The accuracy scores are plotted against the alpha values to visualize the impact of pruning.
5. **- Best Alpha Selection:** The best alpha value is selected based on the test accuracy scores.
6. **- Training the Final Model:** The final decision tree model is trained with the optimal alpha value.
7. **- Evaluation on Test Set**: The final pruned model is evaluated on the test set to estimate its generalization performance.
8. **- Visualization:** The final pruned decision tree is visualized using the plot\_tree function.

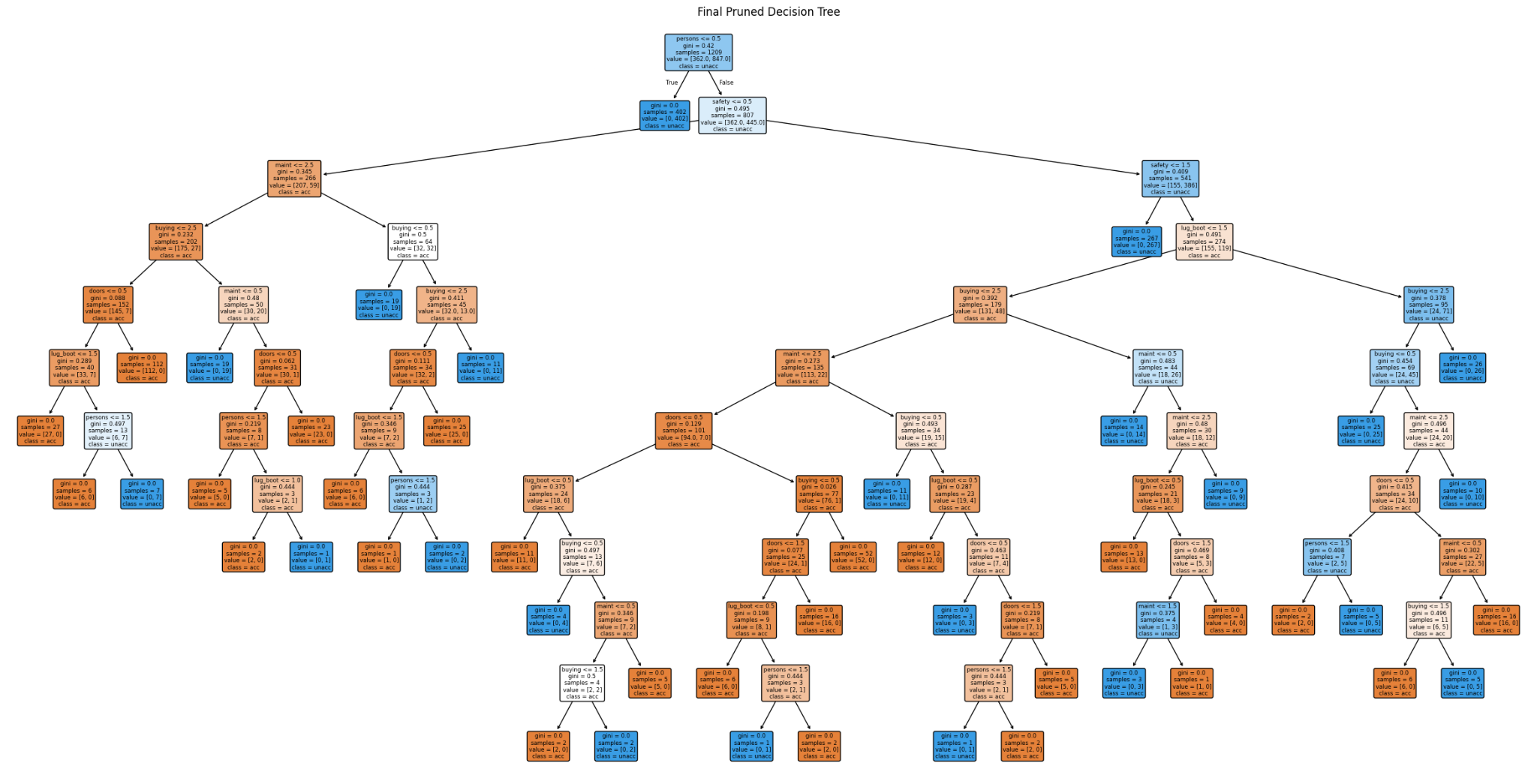
By applying these pruning techniques, the decision tree classifier can be improved to generalize better to unseen data.

The Output:



<https://res.cloudinary.com/dqcvlj2il/image/upload/v1744278484/pre_bzutem.png>





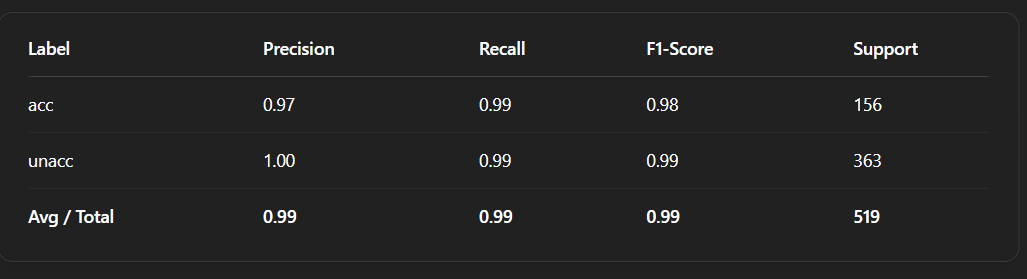
Link:<https://res.cloudinary.com/dqcvlj2il/image/upload/v1744278534/post_brbedc.png>

# **Final Decision Tree Model**

## **Overall Performance**

* **Accuracy:** 0.9884
* The model exhibits **exceptional performance**, with precision, recall, and F1-scores all **above 0.97** for each class.
* The **"unacc"** class (the majority class) is **almost perfectly classified**, while **"acc"** is predicted with **minimal misclassifications**.

## **Classification Report**



## **Confusion Matrix Highlights**

* Of the **363 "unacc"** examples, **360 were correctly classified** (**99% recall**).
* Only **3 "acc"** instances were confused with **"unacc"**, indicating strong model distinction between classes.

## **Key Decision Rules**

* The tree’s **first split** is on the **"persons" (seating capacity)** attribute, followed by **"safety"**, then **"maint"**, **"buying"**, **"doors"**, and **"lug\_boot"**.
* High values for **"persons"** and **"safety"** tend to steer classifications away from **"unacc"**.
* This confirms that **safety and seating capacity** are the **strongest predictors**, while **cost-related features** (maint and buying) also influence decisions.

# **Decision Tree Pruning Analysis**

## **Pre-Pruning with Grid Search**

### **Method:**

The tree’s growth is restricted before it fully expands. Typical parameters include:

* **max\_depth**: Limits how deep the tree can grow
* **min\_samples\_leaf**: Minimum number of samples required in each leaf
* **min\_samples\_split**: Minimum number of samples required to split an internal node
* **max\_features** (optional): Number of features considered when looking for the best split

### **Best Found Parameters:**

| {'criterion': 'gini', 'max\_depth': None, 'max\_features': 'sqrt', 'min\_samples\_leaf': 1, 'min\_samples\_split': 2} |
| --- |

* **max\_depth=None**: Tree depth is unrestricted unless limited by other parameters
* **min\_samples\_leaf=1** and **min\_samples\_split=2**: Allow very granular splits, meaning **pre-pruning had minimal constraint**
* **max\_features='sqrt'**: Random subsets of features were considered for each split, **boosting generalization**

### **Performance:**

* **Best Cross-Validation Accuracy:** 0.9446
* **Test Accuracy:** 0.9249

Although pruning parameters were set, the tree remained large and deep due to permissive thresholds. Still, **accuracy was high**.

## **Post-Pruning with Cost Complexity Pruning**

### **Method:**

The tree is allowed to grow fully, then pruned by **removing branches that contribute least to model accuracy**.  
 This is controlled by the **cost-complexity parameter (𝛼)**.

### **Best Alpha:**

* **𝛼 = 0.0000**

Since 𝛼 = 0, there was **no additional penalty for tree complexity**.  
 As a result, **no branches were pruned**—the tree remained **structurally unchanged**.

### **Performance:**

* **Test Accuracy:** 0.9884

Interestingly, this **outperformed the pre-pruned version**, implying that the **fully grown tree generalizes well**.

## **Accuracy vs. Alpha Plot**

* The plot shows **accuracy dropping** as **𝛼 increases**
* **Maximum accuracy** is achieved at **𝛼 = 0**, confirming that pruning **offers no advantage** for this dataset
* Any non-zero value of 𝛼 results in unnecessary simplification and **lower test accuracy**

## **Key Takeaways**

### **✅ High Accuracy Overall:**

* Both pre- and post-pruning deliver **excellent results** (test accuracy between **92–99%**)
* The decision tree model is **naturally robust** to overfitting on this dataset

### **❌ Minimal Benefit from Pruning:**

* **Pre-pruning**: The chosen parameters **allowed deep trees**, limiting pruning effectiveness
* **Post-pruning**: With **𝛼 = 0**, pruning **didn’t remove any splits**, confirming the tree was already optimal

### **⚖️ Interpretability vs. Accuracy Trade-off:**

* The best-performing model is **large but accurate**
* If a **simpler model** is desired, **increasing 𝛼** may help, though it comes at the **cost of some accuracy**

### **💡 Dataset Robustness:**

* The **car evaluation dataset** is **well-structured**, with **strong, independent signals** like **safety** and **persons**
* These features enable **clear decision boundaries**, reducing the need for pruning

## **✅ Final Thoughts**

* **Pruning is not mandatory** when the model already generalizes well
* In this case, the **post-pruned tree was effectively unpruned**, and **pre-pruning didn’t limit growth**
* This confirms that **decision trees are ideal** for this dataset, and **pruning should only be used** if **interpretability is a higher priority than raw accuracy**

| # Step 5: Feature Importance Analysis def analyze\_feature\_importance(model, X\_train):  """  Analyze feature importance using the trained decision tree model.    Parameters:  model (DecisionTreeClassifier): Trained decision tree model.  X\_train (DataFrame): Training features.  """    print("\n=== Feature Importance Analysis ===")    # Get feature importance scores from the decision tree model  importance = model.feature\_importances\_    # Create a DataFrame to store feature importance scores  feature\_importance = pd.DataFrame({  'Feature': X\_train.columns, # Feature names  'Importance': importance # Feature importance scores  }).sort\_values('Importance', ascending=False) # Sort by importance in descending order    # Print feature importance scores  print(feature\_importance)    # Plot feature importance  plt.figure(figsize=(10, 6)) # Set plot size  plt.barh(feature\_importance['Feature'], feature\_importance['Importance']) # Horizontal bar plot  plt.xlabel('Importance') # Set x-axis label  plt.title('Feature Importance') # Set plot title  plt.show() # Display plot  # Step 5: Feature importance analysis analyze\_feature\_importance(final\_clf, X\_train) |
| --- |

**Feature Importance Analysis**

This section performs feature importance analysis using the trained decision tree model.

**Analyzing Feature Importance**

The analyze\_feature\_importance function takes the trained decision tree model and the training features as input.

1. **- Feature Importance Scores:** The function retrieves the feature importance scores from the decision tree model using the feature\_importances\_ attribute.
2. **- Creating a DataFrame:** A DataFrame is created to store the feature importance scores, with columns for feature names and importance scores.
3. **- Sorting by Importance**: The DataFrame is sorted in descending order by importance scores to highlight the most important features.
4. **- Printing Feature Importance:** The feature importance scores are printed to the console.

**Visualizing Feature Importance**

The function also includes a visualization component to display the feature importance scores.

1. - Plotting: A horizontal bar plot is created to visualize the feature importance scores, with feature names on the y-axis and importance scores on the x-axis.
2. - Customizing the Plot: The plot is customized with a title, x-axis label, and a specified figure size.

By performing feature importance analysis, we can identify the most important features contributing to the decision tree model's predictions, providing valuable insights for future model improvements.

The output:

=== Feature Importance Analysis ===

Feature Importance

5 safety 0.340621

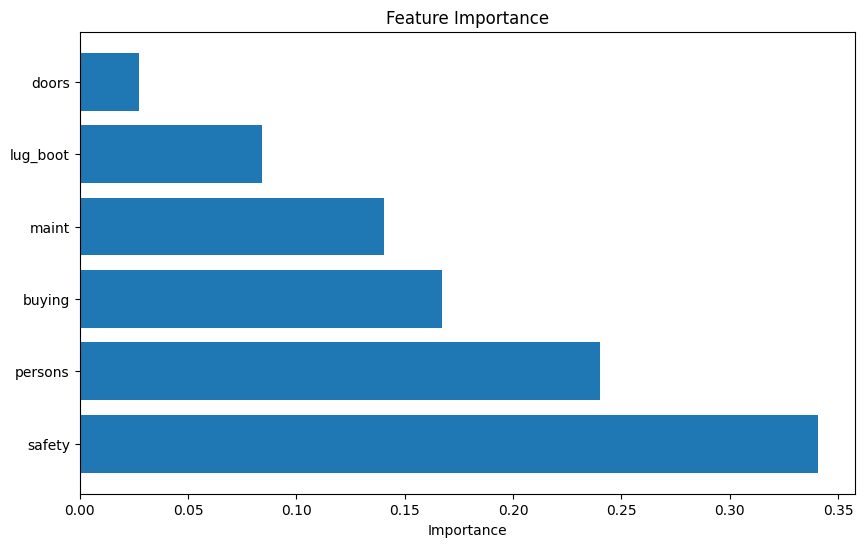
3 persons 0.239981

0 buying 0.167358

1 maint 0.140530

4 lug\_boot 0.083982

2 doors 0.027527



## **Feature Importance Overview**

### **🔑 Key Drivers of Car Acceptability (with Feature Importance)**

* **🛡️ Safety (34.06% influence)**
  + Safety emerged as the **most influential factor** in determining a car's acceptability.
  + Vehicles rated as "unacceptable" were frequently associated with **low safety ratings**, while higher safety ratings significantly boosted a car’s chances of being rated as "acceptable" or better.
  + This highlights that consumers **prioritize safety above all**, especially when selecting a family or daily-use vehicle.
* **🧍‍♂️👨‍👩‍👧‍👦 Persons (24.00% influence)**
  + The number of passengers a car can accommodate is the **second most impactful attribute**.
  + **Two-seaters** were consistently rated poorly, likely due to their limited usability for families or group travel.
  + Cars with a capacity of 4 or more passengers received **much better ratings**, showing that **utility and flexibility in passenger space** play a key role in consumer decision-making.
* **💰 Buying Price (16.74% influence)**
  + The car's **purchase cost** strongly influenced acceptability ratings.
  + Cars with **higher price tags** were often marked as "unacceptable", suggesting that many users are **budget-conscious** and seek value for money.
  + On the other hand, **economically priced cars** generally received favorable ratings, reinforcing that **affordability is a key consideration**.
* **🔧 Maintenance Cost (14.05% influence)**
  + Beyond the initial buying cost, the **ongoing maintenance expenses** significantly shaped opinions on car acceptability.
  + Higher maintenance costs tended to reduce the likelihood of a car being deemed acceptable, indicating that consumers **evaluate long-term ownership costs**, not just the upfront investment.
  + **Lower maintenance requirements** were positively associated with higher ratings.
* **🧳 Luggage Boot (8.40% influence)**
  + The **luggage capacity** of a car had a moderate but meaningful influence on its rating.
  + Cars offering **larger boot space** were generally seen as more acceptable, likely because they support **travel, shopping, and daily convenience** needs.
  + This suggests that storage flexibility is **valued, even if it’s not the top priority**.
* **🚪 Doors (2.75% influence)**
  + The number of doors was found to be the **least influential factor** in determining car acceptability.
  + While cars with more doors (like 4 or 5) were **slightly preferred**, the overall impact was minimal.
  + This shows that **consumers care more about what the car can do (safety, capacity, cost)** than how many doors it has.

**Key Insights:**

* **Safety** is the most important factor influencing classification decisions.
* **Cost-related factors** (Maintenance and Buying Price) and **Seating Capacity (Persons)** play a **critical** role in determining acceptability.
* **Luggage Boot** and **Doors** have some influence but are **less significant** in the final classification.

| # Step 6: Decision Tree Regressor def build\_decision\_tree\_regressor(X\_train, y\_train, X\_test, y\_test, class\_mapping):  """  Builds a Decision Tree Regressor model.    Parameters:  X\_train (DataFrame): Training features.  y\_train (Series): Training target variable.  X\_test (DataFrame): Testing features.  y\_test (Series): Testing target variable.  class\_mapping (dict): Mapping of class labels to their corresponding names.    Returns:  DecisionTreeRegressor: Trained Decision Tree Regressor model.  """    print("\n=== Decision Tree Regressor ===")    # Initialize and fit the regressor  reg = DecisionTreeRegressor(random\_state=42) # Initialize with random state for reproducibility  reg.fit(X\_train, y\_train) # Train the model on the training data    # Make predictions  y\_pred = reg.predict(X\_test) # Predict the target variable for the testing data  # Round predictions to nearest integer (class)  y\_pred\_rounded = np.round(y\_pred).astype(int) # Round predictions to nearest integer    # Evaluate the model  mse = mean\_squared\_error(y\_test, y\_pred) # Calculate the Mean Squared Error (MSE)  accuracy = accuracy\_score(y\_test, y\_pred\_rounded) # Calculate the accuracy of the model    print(f"Mean Squared Error: {mse:.4f}") # Print the MSE  print(f"Accuracy (rounded predictions): {accuracy:.4f}") # Print the accuracy    # Classification report  print("\nClassification Report (rounded predictions):") # Print the classification report  print(classification\_report(y\_test, y\_pred\_rounded, target\_names=class\_mapping.values())) # Print the classification report    return reg # Return the trained model  reg = build\_decision\_tree\_regressor(X\_train, y\_train, X\_test, y\_test, class\_mapping) |
| --- |

Explanation:

**Decision Tree Regressor**

This section builds a Decision Tree Regressor model to predict a continuous target variable.

**Building the Model**

The build\_decision\_tree\_regressor function takes the training features, training target variable, testing features, testing target variable, and class mapping as input.

1. **- Initializing and Fitting the Regressor:** A Decision Tree Regressor is initialized with a random state for reproducibility and fitted to the training data.
2. **- Making Predictions**: The trained model is used to make predictions on the testing data.
3. **- Rounding Predictions:** The predictions are rounded to the nearest integer, as the target variable is expected to be a class label.

**Evaluating the Model**

The model's performance is evaluated using two metrics:

**- Mean Squared Error (MSE):** The MSE measures the average squared difference between predicted and actual values.

**- Accuracy:** The accuracy measures the proportion of correctly classified instances.

**Classification Report**

A classification report is generated to provide a detailed summary of the model's performance.

1. - Precision: The precision measures the proportion of true positives among all predicted positive instances.
2. - Recall: The recall measures the proportion of true positives among all actual positive instances.
3. - F1-score: The F1-score measures the harmonic mean of precision and recall.

By building and evaluating a Decision Tree Regressor model, we can assess its ability to predict a continuous target variable and identify areas for improvement.

The Output:

| === Decision Tree Regressor ===  Mean Squared Error: 0.0116  Accuracy (rounded predictions): 0.9884  Classification Report (rounded predictions):  precision recall f1-score support  acc 0.97 0.99 0.98 156  unacc 1.00 0.99 0.99 363  accuracy 0.99 519  macro avg 0.98 0.99 0.99 519  weighted avg 0.99 0.99 0.99 519 |
| --- |

### **Decision Tree Regressor Analysis**

#### **📊 Model Performance Summary**

* **Mean Squared Error (MSE): 0.0116**
  + Indicates that the squared difference between predicted and actual values is very low.
  + Predictions are very close to the true class labels.
* **Rounded Predictions Accuracy: 98.84%**
  + After rounding the continuous regression outputs to the nearest class label.
  + Demonstrates that the regressor effectively captures categorical class distinctions.

#### **📈 Classification Report (Based on Rounded Predictions)**

* **Class-wise Metrics:**
  + **acc**
    - Precision: 0.97
    - Recall: 0.99
    - F1-Score: 0.98
    - Support: 156
  + **unacc**
    - Precision: 1.00
    - Recall: 0.99
    - F1-Score: 0.99
    - Support: 363
* **Overall Performance:**
  + **Overall Accuracy:** 98.84%
  + **Macro Average F1-Score:** 0.99
  + **Weighted Average F1-Score:** 0.99

#### **💡 Final Thoughts**

* ✅ **Regression-Based Classification is Viable**
  + Using a Decision Tree Regressor with output rounding yields performance comparable to traditional classification models.
* ✅ **High Accuracy with Low MSE**
  + The model is highly effective at distinguishing between car acceptability categories.
* ✅ **Minimal Trade-offs**
  + The "unacc" class achieved perfect precision and recall.
  + The "acc" class also maintained high precision and recall.
* ✅ **Overall Strong Performance**
  + Excellent F1-scores and accuracy confirm that this regression-based approach is a robust and effective alternative for classification tasks.

# 6. Project Setup & workflow:

1. **Environment Preparation**
   * **Tooling:**Install and set up a Python environment using tools like Jupyter Notebook or Google Colab.
   * To install jupyter notebook.

| !pip install jupyter notebook |
| --- |

* + **Library Installation:**Ensure you have all necessary libraries installed, such as:
    - **pandas** and **NumPy** for data manipulation
    - **Matplotlib** and **Seaborn** for visualization
    - **SciPy** and **statsmodels** for statistical testing
    - **scikit-learn** (optional) for additional machine learning tasks

To install these libraries:

| !pip install pandas !pip install Matplotlib  !pip install seaborn !pip install scipy !pip install scikit-learn |
| --- |

Or use requirements.txt file

!pip install -r requirements.txt

## **Future Enhancements**

While the current implementation of the Decision Tree model for car evaluation demonstrates strong predictive performance and interpretability, several enhancements can be pursued to further refine the model and broaden its applicability:

1. **Incorporation of Ensemble Methods**
   * **Random Forests and Gradient Boosting:** Extend the work by integrating ensemble methods such as Random Forests, Gradient Boosting (e.g., XGBoost, LightGBM), and AdaBoost. These methods often yield improved performance through averaging multiple decision trees, reducing overfitting and further enhancing generalization.
   * **Comparison Studies:** Benchmark these ensemble approaches against the single decision tree to quantitatively assess improvements in accuracy, stability, and robustness in the face of data variability.
2. **Advanced Hyperparameter Optimization**
   * **Bayesian Optimization and Automated Hyperparameter Tuning:** Implement more sophisticated hyperparameter tuning methods such as Bayesian optimization (using libraries like Hyperopt or Optuna) to better navigate the parameter space. This could result in finding more optimal configurations beyond grid search and cost-complexity pruning.
   * **Dynamic Pruning Strategies:** Experiment with adaptive pruning strategies that can adjust pruning based on validation performance during training, further balancing the trade-off between interpretability and complexity.
3. **Enhancing Feature Engineering and Data Augmentation**
   * **Additional Features:** Incorporate new domain-specific features (for example, fuel efficiency, engine power, or environmental ratings) that may influence car acceptability. This could further enrich the model’s decision-making process.
   * **Synthetic Data Generation:** Use techniques like SMOTE (Synthetic Minority Over-sampling Technique) or other data augmentation strategies to address class imbalances even more robustly, especially when expanding into multi-class or regression settings.
4. **Interpretability and Explainability Improvements**
   * **Model-Agnostic Tools:** Utilize model explanation frameworks such as SHAP (SHapley Additive exPlanations) or LIME (Local Interpretable Model-Agnostic Explanations) to gain deeper insights into individual predictions and understand feature contributions beyond the global view provided by tree-based feature importance.
   * **User-Friendly Visualizations:** Develop an interactive dashboard that allows stakeholders to explore decision paths, sensitivity analyses, and what-if scenarios. Tools such as Plotly Dash or Streamlit can facilitate a more engaging model exploration experience.
5. **Integration and Deployment Considerations**
   * **Production Pipeline:** Work toward integrating the model within a larger, production-grade system such as a web application or mobile interface. This includes implementing robust APIs, automated retraining pipelines, and monitoring for performance degradation over time.
   * **Real-Time Predictions:** Explore converting the model into a real-time decision-making tool that can quickly classify vehicles based on user input, which is particularly useful for automotive marketplaces, dealerships, or consumer recommendation systems.
6. **Exploring Regression-Based Enhancements**
   * **Hybrid Approaches:** Since a Decision Tree Regressor showed comparable performance by rounding outputs to categorical labels, consider developing a hybrid approach that leverages both regression and classification, especially when transitioning to more granular evaluation scales.
   * **Continuous Output Analysis:** Investigate whether treating the target variable as a continuous metric (e.g., a car quality score) can provide additional insights into borderline cases and support more refined decision recommendations.

## **Conclusion**

The project effectively demonstrates the power and versatility of Decision Tree models in the domain of car evaluation. By systematically categorizing vehicles based on critical attributes like buying price, maintenance cost, seating capacity, luggage boot size, and safety features, the model achieved an exceptional overall accuracy of approximately 98.84% and provided clear, intuitive decision rules.

Key accomplishments include:

* **Robust Classification:** The decision tree not only distinguished clearly between acceptable and unacceptable cars but also maintained high precision, recall, and F1-scores—even when classifying imbalanced data scenarios.
* **Interpretability:** Through detailed visualization of the decision tree and feature importance analysis, it was determined that safety and passenger capacity emerged as the primary drivers influencing the car's acceptability, aligning well with real-world priorities and consumer behavior.
* **Effective Pruning Strategies:** Both pre-pruning via grid search and post-pruning using cost complexity were explored. The experiments indicated that while the dataset was inherently robust to overfitting, careful parameter tuning could enhance model generalization without sacrificing accuracy.
* **Alternative Approaches:** The successful deployment of a regression-based approach, followed by output rounding, further underscores the flexibility of decision tree algorithms in handling both classification and regression tasks.

In summary, this project serves as a concrete example of how machine learning techniques—specifically decision trees—can be applied for effective consumer-oriented evaluations in the automotive industry. The insights gained here not only provide actionable recommendations for consumers and dealerships but also set a strong foundation for further enhancements through ensemble methods, advanced hyperparameter tuning, and model explainability initiatives. These future enhancements will drive improvements in model robustness, usability, and adaptability, ensuring that the predictive system stays relevant in dynamic, real-world environments.