Statistical Methods for Machine Learning

Nicolo Cesa-Bianchi

2023-2024

Master in Data Science for Economics

Student

Name: Hilal Mente

Matr. Number: 47252A

Tree Predictor for Mushroom Classification

1. Introduction

The classification of objects into predefined categories is a critical problem in various fields, ranging from healthcare to finance and even agriculture. One of the key challenges in such problems is to build models that can accurately distinguish between these categories based on a set of input features. Decision trees have become one of the most popular methods to address these challenges due to their interpretability and ease of implementation. This project aims to apply a decision tree classifier to a dataset containing attributes of mushrooms, in order to predict whether they are poisonous or edible.

Background

Mushroom classification is an important problem, especially in areas where wild mushrooms are harvested for consumption. Certain mushroom species are edible, while others are highly toxic, and even a small mistake in identification can be fatal. A reliable classification system can be life-saving in these situations. Decision trees, known for their hierarchical and tree-like structure, can help solve this problem by classifying mushrooms based on their observable characteristics, such as cap shape, color, and habitat.

Project Overview

The purpose of this project is to build a binary classification model to predict whether a mushroom is poisonous or edible based on its physical attributes. The dataset used in this project contains several categorical features representing characteristics like cap shape, gill color, and habitat, among others. Our approach focuses on implementing a custom decision tree classifier, evaluating its performance, and tuning hyperparameters to improve accuracy.

Motivation

The mushroom classification problem serves as a strong example of the broader class of binary classification tasks, where the aim is to categorize inputs into one of two classes. This project is motivated by the desire to explore the effectiveness of decision trees for such tasks, especially in situations where interpretability and transparency are crucial. Moreover, by building the decision tree from scratch, I seek to gain a deeper understanding of the underlying mechanisms of this powerful algorithm.

Challenges

Some of the challenges I anticipate in this project include:

1. Handling Missing Data

   The dataset contains missing values in several columns, such as gill-spacing, cap-surface, gill-attachment, and ring-type. To address this, columns with a large proportion of missing data (more

than 80%) were removed to avoid skewing the results, while columns with smaller amounts of missing data were imputed with the most frequent value. This ensures that the dataset remains complete, preventing any issues with missing values during model training. Handling missing data in this way helps the model by:

Reducing Bias:

Removing columns with excessive missing values prevents the model from making decisions based on incomplete or inaccurate data.

Improving Consistency:

Imputing missing values ensures that the dataset is fully utilized and avoids reducing the sample size unnecessarily. It also maintains consistency across the features, enabling the model to learn better from the data.

However, it is important to note that even though imputation can provide a solution for missing data, it may introduce some bias, especially if the missing values are not random.

2. Categorical Feature Encoding

In this project, the dataset contains 21 columns, 18 of which are categorical. Depending on whether these categorical features exhibit ordinal (ordered) or nominal (unordered) properties, different encoding techniques have been applied to prepare the data for modeling.

Encoding Approach Implementation

The encoding steps were implemented as follows:

One-Hot Encoding:

The pandas.get_dummies function was used to convert the specified nominal categorical features into multiple binary columns. Each unique value in a feature becomes a new column, and the presence or absence of the value in each row is marked with 1 or 0, respectively.

Label Encoding:

For ordinal or binary features, LabelEncoder from sklearn.preprocessing was used. This method assigns an integer value to each unique category in a feature, ensuring that any ordinal relationship between categories is preserved.

By applying these encoding techniques, the categorical data is converted into a format suitable for use in machine learning models, without imposing false assumptions of order or proximity in nominal features. This careful consideration ensures that the model remains unbiased and better equipped to make accurate predictions.

3. Choosing Optimal Hyperparameters

   The performance of decision trees can be highly sensitive to hyperparameters like max_depth and the splitting criterion. Selecting optimal values for these parameters is crucial to avoid overfitting or underfitting.

4. Balancing Interpretability and Performance

   While decision trees are easy to interpret, they can become complex and prone to overfitting if not properly tuned. A balance between model complexity and interpretability is essential for producing useful results.

In the following sections, I will outline the data analysis and preparation process, describe the decision tree algorithm, and evaluate the model's performance based on various metrics.

2. Data Description

In this section, I provide a detailed description of the dataset used for building the mushroom classification model. The dataset consists of various features representing the physical characteristics of mushrooms, along with the target label indicating whether a mushroom is poisonous or edible.

Dataset Overview

The dataset contains a total of 21 columns (features), 18 of which are categorical. These features describe different observable attributes of mushrooms such as cap shape, color, gill size, and habitat. The target variable, class, is binary and indicates whether a mushroom is poisonous (1) or edible (0). The dataset has a total of **61,069 data points**.

Features

Each feature in the dataset plays an important role in helping the model differentiate between poisonous and edible mushrooms. Below is an overview of the key features, their descriptions, and the percentage of missing values in each feature:

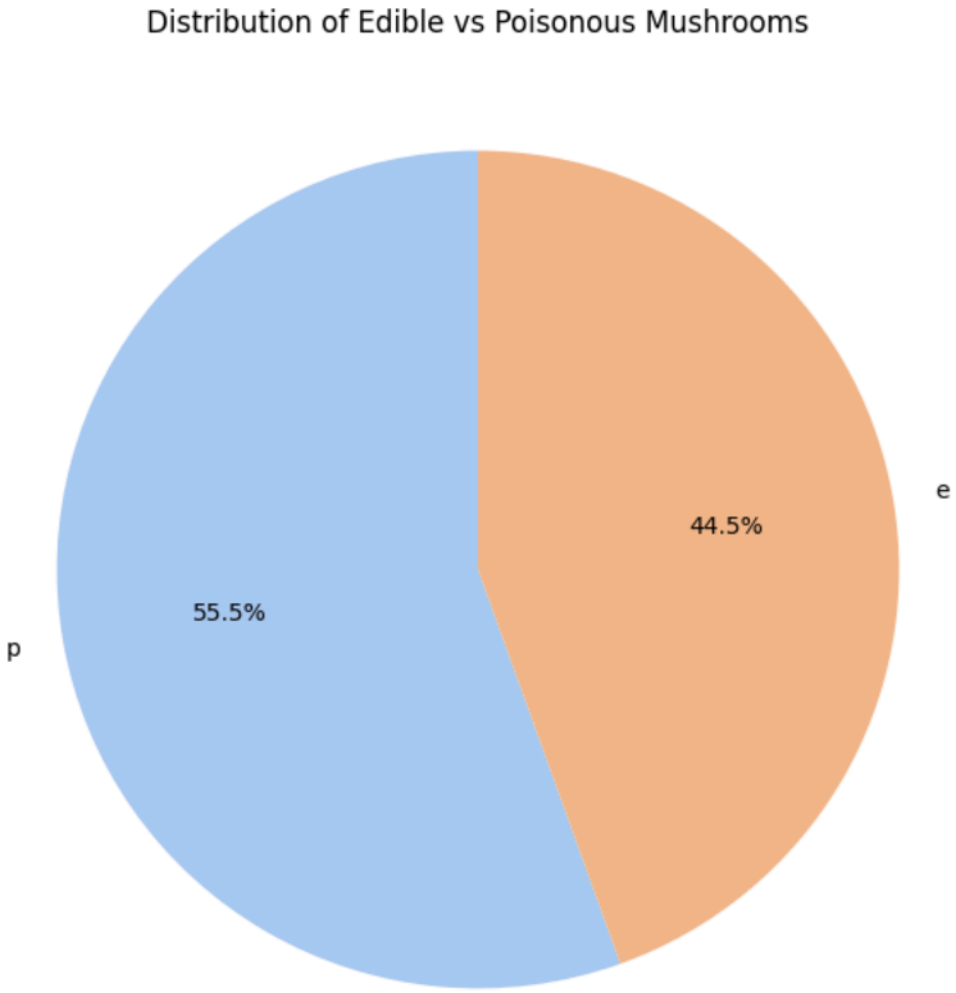
Feature	Description	% Missing Values
class	Edible or poisonous mushroom (target)	0.00%
cap-diameter	Diameter of the mushroom's cap	0.00%
cap-shape	Shape of the mushroom's cap	0.00%
cap-surface	Surface texture of the mushroom's cap	23.12%
cap-color	Color of the mushroom's cap	0.00%
does-bruise-or-bleed	Whether the mushroom bruises or bleeds	0.00%
gill-attachment	How the gills are attached to the stem	16.18%
gill-spacing	The spacing between the gills	41.04%
gill-color	Color of the mushroom's gills	0.00%
stem-height	Height of the mushroom's stem	0.00%
stem-width	Width of the mushroom's stem	0.00%
stem-root	Root structure of the mushroom's stem	84.39%
stem-surface	Surface texture of the mushroom's stem	62.43%
stem-color	Color of the mushroom's stem	0.00%
veil-type	Type of veil covering the mushroom	94.80%
veil-color	Color of the veil covering the mushroom	87.86%
has-ring	Whether the mushroom has a ring on its stem	0.00%
ring-type	Type of ring on the stem	4.05%
spore-print-color	Color of the mushroom's spore print	89.60%
habitat	Natural habitat where the mushroom grows	0.00%
season	Season during which the mushroom grows	0.00%

Exploratory Data Analysis (EDA)

To better understand the dataset and prepare it for the decision tree model, I conducted exploratory data analysis (EDA). EDA helps identify patterns, distributions, and potential issues in the data, such as missing values or outliers, which can affect model performance.

1. Distribution of Edible vs. Poisonous Mushrooms

The pie chart below shows the distribution of edible and poisonous mushrooms in the dataset. I observe that 55.5% of the mushrooms are poisonous, while 44.5% are edible.
This relatively balanced distribution of the target classes ensures that the model will not be biased towards any class during training.

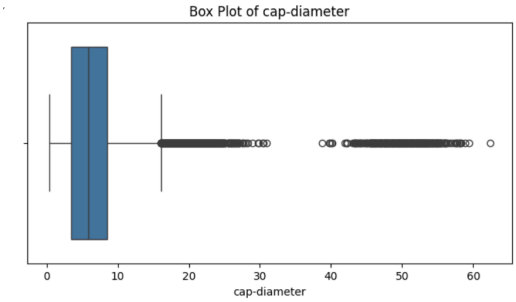


2. Box Plots for Numerical Columns

The following box plots represent the distributions of numerical features (cap-diameter, stem-height, stem-width). From these plots, I observe several outliers, particularly in the stem-width and cap-diameter features.

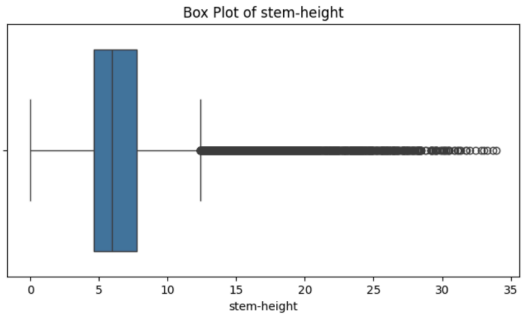
Box Plot of Cap Diameter:

Displays the spread of cap diameters, showing potential outliers in larger values.



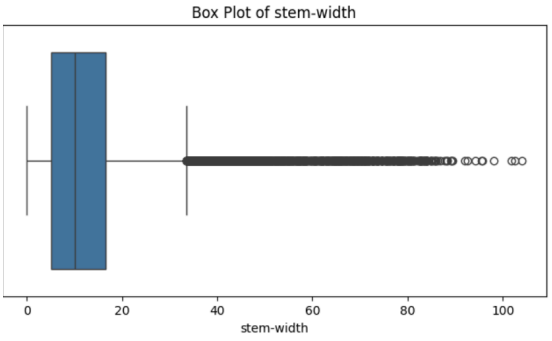
Box Plot of Stem Height:

Highlights that most mushrooms have stem heights below 10, but several extreme outliers exist.



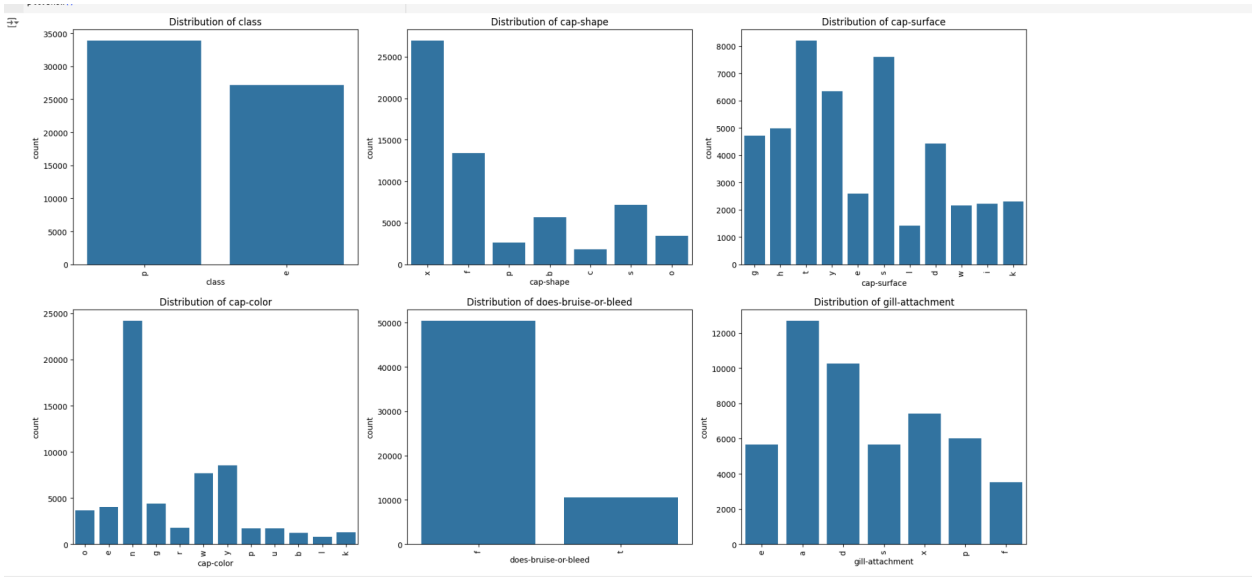
Box Plot of Stem Width:

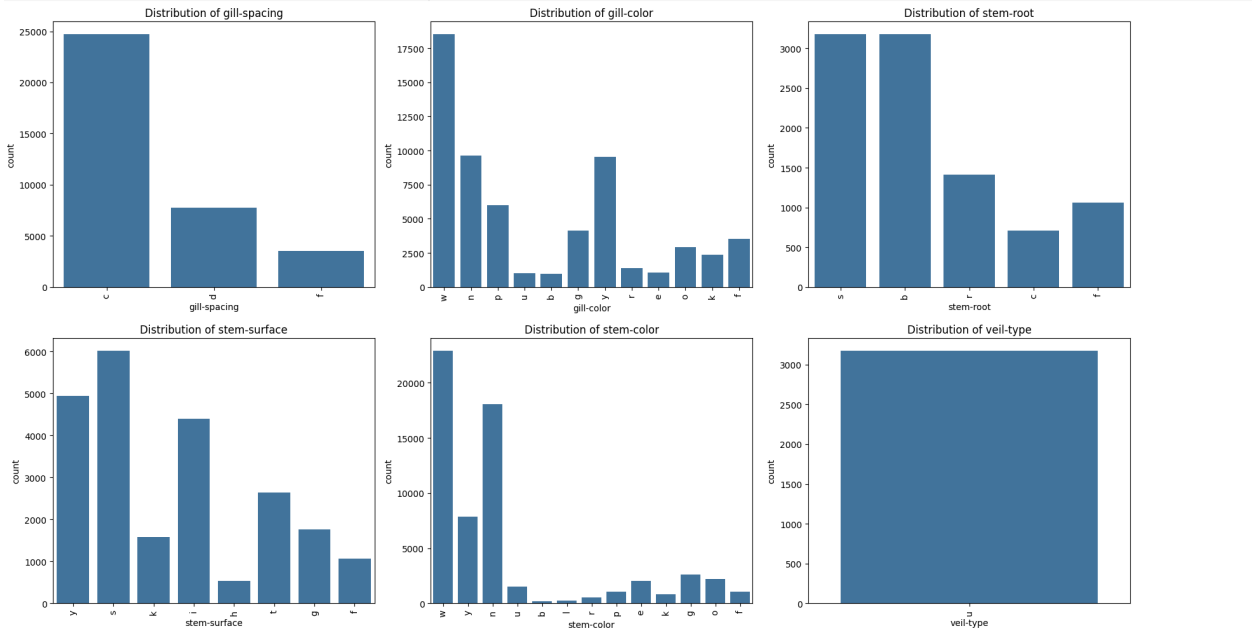
Indicates a large number of outliers, suggesting that certain mushrooms have much wider stems compared to the majority. Outliers could affect the model's performance, and I may need to consider normalization or further investigation.



3. Categorical Feature Distributions

Below, I present the distribution of various categorical features such as cap-shape, gill-attachment, gill-color, and others. These bar charts provide an overview of the most common values in each feature.
The bar plots clearly show the skewness in some categorical features. For example, certain values in cap-shape or cap-color are much more frequent, indicating that the model might heavily rely on these features when making predictions.





Handling Categorical Features

The dataset contains 18 categorical features. Categorical data is not directly usable by machine learning models and must be transformed into numerical representations. Depending on whether these categorical features exhibit ordinal (ordered) or nominal (unordered) properties, different encoding techniques were applied.

One-Hot Encoding:

One-Hot Encoding was applied to features with nominal data (i.e., categories that do not have a natural order). This technique was used for the following features:

- cap-shape
- gill-attachment
- cap-surface
- cap-color
- gill-color
- stem-color
- ring-type
- habitat

This technique converts each unique category in a feature into a new binary column (0 or 1). For example, a feature like cap-shape with multiple unique values is split into several columns, each representing the presence (1) or absence (0) of a specific shape. One-Hot Encoding ensures that no unintended ordinal relationships are introduced between categories.

Label Encoding:

Label Encoding was applied to features that either had a natural order or were binary. The following features were encoded using this method:

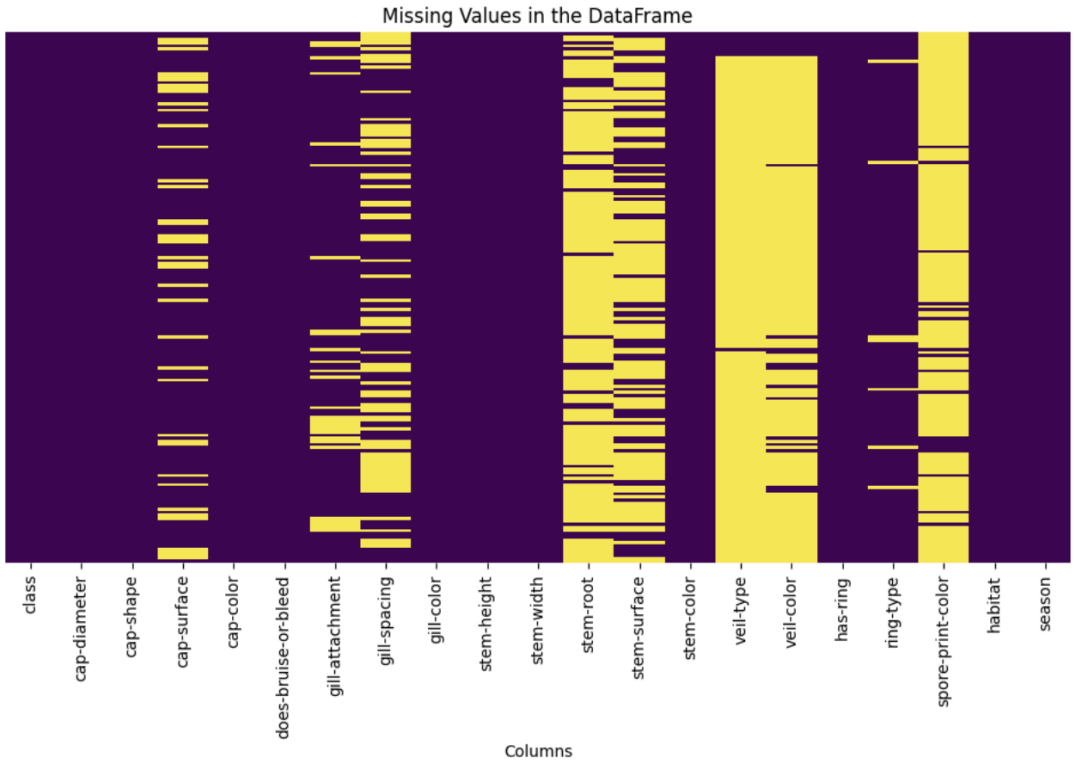
- does-bruise-or-bleed
- gill-spacing
- class
- has-ring
- season

Label Encoding assigns an integer to each category, preserving the ordinal relationship where applicable. For instance, seasons can be encoded with values like Spring = 0, Summer = 1, Fall = 2, and Winter = 3, reflecting the natural progression of seasons.

By using a combination of One-Hot Encoding and Label Encoding, I transformed the categorical data into a format suitable for machine learning models, without imposing false assumptions about the relationships between categories.

4. Missing Data Visualization

The heatmap below visualizes the missing values across the dataset. Features like veil-type, spore-print-color, and veil-color contain significant amounts of missing data, which was handled during preprocessing by either removing these columns or imputing the missing values.



As discussed earlier, columns with over 50% missing values were dropped, while others were imputed using the most frequent value.

Handling Missing Data

One of the key challenges in working with this dataset was handling missing values. The percentage of missing data varied widely across different features, and the following strategies were applied:

1. **Dropping Columns with Over 50% Missing Data**:

   Columns with a high percentage of missing values (more than 50%) were dropped from the dataset, as imputing such large amounts of missing data could introduce bias. These columns included:

   - veil-type (94.80% missing)
   - spore-print-color (89.60% missing)
   - veil-color (87.86% missing)
   - stem-root (84.39% missing)
   - stem-surface (62.43% missing)

2. **Imputing Missing Values in Columns with 5% to 50% Missing Data**:

   For columns with fewer missing values, the most frequent value (mode) was used to fill in the missing data. These columns included:

   - gill-spacing (41.04% missing)
   - cap-surface (23.12% missing)
   - gill-attachment (16.18% missing)
   - ring-type (4.05% missing)

By handling missing data in this way, I reduced potential biases and maintained a cleaner, more consistent dataset for modeling.

Data Summary

- **Number of Data Points**: 61,069
- **Total Number of Features**: 21 (including the target variable class)
- **Categorical Features**: 18
- **Dropped Columns**: veil-type, spore-print-color, veil-color, stem-root, stem-surface (due to over 50% missing data)
- **Imputation Strategy**: Columns with 5% to 50% missing data were filled using the most frequent value (majority)
- **Encoding**: One-Hot Encoding and Label Encoding were applied based on the nature of the categorical features.

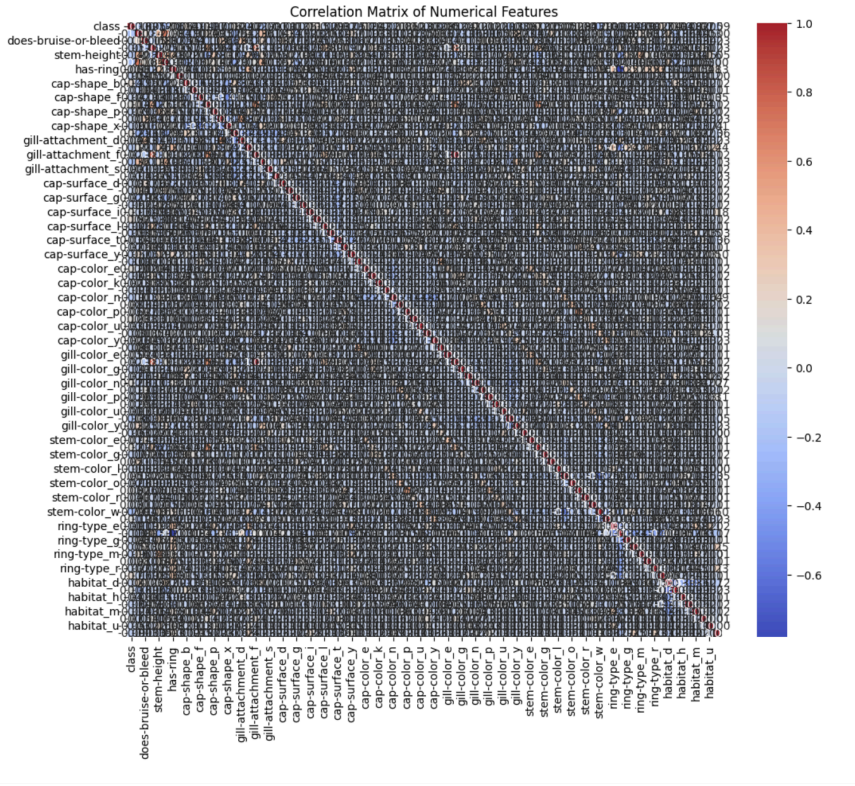
This thorough data preprocessing step ensures that the dataset is ready for modeling, with missing data handled effectively and categorical data encoded appropriately.

Correlation Analysis

Correlation analysis is a crucial step in understanding the relationships between features in a dataset, especially in machine learning where multicollinearity between features can affect the model's performance. In this section, I analyze the correlation between numerical and encoded categorical features in the dataset.

Correlation Matrix for All Features

I computed the correlation matrix for all features in the dataset, which includes both numerical and encoded categorical variables. The heatmap below visualizes the correlation matrix, where the color intensity represents the strength of the correlation between pairs of features.



From the correlation matrix, I observe the following key insights:

Strong Positive Correlations

Strong positive correlations indicate that as one feature increases, the other feature also tends to increase. Here are the key strong positive correlations (above 0.5):

1. **Cap Diameter and Stem Width (0.695)**: This suggests that mushrooms with larger caps tend to have wider stems.
2. **Gill Spacing and Gill Attachment (0.795)**: This strong positive correlation indicates that the gill spacing and gill attachment are closely related, meaning that specific gill attachment types are more likely to exhibit specific spacing patterns.
3. **Gill Attachment and Gill Color (1.0)**: The perfect correlation indicates that these features are likely identical in terms of the way they are encoded, suggesting redundancy between these features.
4. **Cap Color and Stem Color (0.524)**: Certain cap and stem colors are frequently observed together in mushrooms, which could be useful for classification.

Strong Negative Correlations

Strong negative correlations indicate that as one feature increases, the other feature tends to decrease. Here are the key strong negative correlations (below -0.5):

1. **Has Ring and Ring Type (-0.780)**: This negative correlation indicates that the presence of a ring is strongly associated with specific ring types.
2. **Stem Color (n) and Stem Color (w) (-0.502)**: This negative correlation shows that certain stem colors are mutually exclusive, meaning that specific stem color categories do not co-occur.
3. **Habitat (d) and Habitat (g) (-0.626)**: Mushrooms found in habitat type ‘d’ are unlikely to be found in habitat type ‘g’, indicating distinct environmental preferences.

Weak Correlations

Weak correlations (between -0.3 and 0.3) suggest little to no linear relationship between features. Most correlations with the target feature class (edible or poisonous) fall within this range, indicating that no single feature has a strong linear influence on whether a mushroom is edible or poisonous. Here are some examples:

1. **Cap Diameter and Class (-0.181)**: The size of the mushroom's cap has a weak negative correlation with whether the mushroom is poisonous or not.
2. **Does Bruise or Bleed and Class (-0.019)**: The bruising property has almost no correlation with the class of the mushroom.

This indicates that there is no dominant linear relationship between individual features and the target variable, which suggests that a decision tree algorithm might be well-suited for this problem, as it can capture non-linear relationships and interactions between multiple features.

Conclusion of Correlation Analysis

The correlation analysis revealed that while some features are strongly correlated with one another, no single feature has a significant linear relationship with the target variable. This highlights the need for algorithms like decision trees, which can capture non-linear patterns and combinations of features to make accurate predictions.

Correlation with Target Variable Class

Interestingly, there were no strong positive or negative correlations between any feature and the target variable class (poisonous vs edible). This implies that no single feature can determine whether a mushroom is poisonous or edible. Instead, the classification task likely requires the combination of multiple features to accurately predict the target variable.
Here is a summary of the correlations between class and other features:

Weak Negative Correlation:

- Cap Diameter (-0.181)
- Stem Width (-0.197)
- Habitat (-0.078 to 0.085)

Weak Positive Correlation:

Some other features like habitat and stem color showed weak correlations with the target. Since no feature is strongly correlated with the target, this reinforces the importance of using more complex methods (such as decision trees) that can handle interactions between features to predict whether a mushroom is poisonous or edible.

5. Model Implementation

In this project, I implemented a custom decision tree from scratch to classify mushrooms as poisonous or edible. The model implementation also incorporates an **interactive flow**, where the user is prompted to make decisions regarding the model's parameters during execution. This interactive approach enhances flexibility and user involvement, allowing for either automatic selection of the best parameters via grid search or manual input of custom parameters.

1. Interactive Flow for Parameter Selection

When running the code, the user is presented with a choice: they can either use the best hyperparameters found via grid search or enter custom parameters manually. This input-driven flow makes the model more adaptable to the user's preferences and allows for experimentation with different model configurations. The overall flow is as follows:

Pseudo Code for Interactive Flow

1. Present the user with a choice:
    a. Use the best parameters found by grid search.
    b. Enter custom parameters manually.
2. If the user chooses 'yes':
    a. Perform grid search to find the optimal parameters (max_depth, split_function, entropy_threshold).
    b. Train the model using the best parameters.
    c. Evaluate the model on the test set and display the results.
3. If the user chooses 'no':
    a. Prompt the user to input custom parameters for max_depth, split_function and entropy_threshold.
    b. Train the model using the custom parameters.
    c. Evaluate the model on the test set and display the results.

4. If the input is invalid, display an error message and prompt the user to try again.
5. Handle any exceptions that might occur during the process and display an error message.

Explanation of the Interactive Flow

1. Grid Search Option: If the user chooses to use the best parameters found by grid search (yes), the program will execute the grid search across the parameter grid (including max_depth,entropy_threshold and split_function) and return the best-performing parameters. The decision tree is then trained using these parameters, and the test set is evaluated.
2. Custom Parameters Option: If the user prefers to manually specify the parameters (no), they are prompted to enter the maximum tree depth and splitting criterion. If the splitting criterion is scaled entropy, the user needs to enter the entropy threshold as another parameter. The decision tree is then trained based on these custom parameters, and the model's performance is evaluated on the test set.
3. Error Handling: The program also includes error handling to ensure that invalid inputs or runtime errors are managed gracefully, providing informative error messages to the user.

This interactive flow makes the model implementation more dynamic, allowing people to engage directly with the model-building process. The flexibility to either rely on grid search for hyperparameter tuning or manually adjust parameters enables people to experiment with different configurations and observe their impact on the model's performance.

2. Decision Tree

Below is the pseudo code that describes the overall flow of the decision tree algorithm, from training the model to making predictions.

Pseudo Code for Decision Tree Algorithm

1. Start with the entire dataset (X, y) where X is the feature set and y is the target label (class).
2. Create a root node for the tree.
3. Define a stopping condition (e.g., maximum tree depth or minimum samples per split).
4. At each node, check if the stopping condition is met:
    a. If true, label the node as a leaf node and assign a class label (the majority class in the subset).
    b. If false, find the best split by:
        i. Iterating through all features.
        ii. For each feature, evaluate all possible thresholds (split points).
        iii. Calculate the split criterion (Gini, Entropy, or Squared Impurity) for each possible split.
        iv. Select the split that minimizes (or maximizes, depending on the criterion) the impurity.
        v. If the entropy change after the split is smaller than the predefined threshold, stop splitting.
        vi. Select the split that minimizes (or maximizes, depending on the criterion) the impurity.
5. Partition the dataset into two subsets based on the selected split (left and right subsets).
6. Recursively apply steps 4-5 to the left and right subsets to create child nodes.
7. Repeat until the entire tree is built or the stopping condition is met.
8. For predictions, traverse the tree starting from the root:
    a. For each test sample, move down the tree based on feature values.
    b. When a leaf node is reached, return the class label of that node as the prediction.

Detailed Explanation of Each Step:

1. **Start with the Entire Dataset**: The algorithm begins by considering the entire dataset. The goal is to partition the data into smaller subsets until each subset is as "pure" as possible (i.e., contains mostly one class).
2. **Create the Root Node**: The root node represents the entire dataset. At this stage, no decisions have been made, and the algorithm is tasked with finding the best feature to split the data.
3. **Stopping Condition**: The algorithm will stop splitting the data if certain conditions are met. Common stopping conditions include:
    - Reaching a maximum tree depth (max_depth).
    - Having fewer than the minimum number of samples required to make a split (min_samples_split).

- All samples in the node belong to the same class (pure node).
- Entropy Threshold: If the reduction in entropy after a split is smaller than a predefined entropy threshold, further splitting is halted. This ensures that the tree does not keep splitting on trivial differences that don't improve classification much.

4. **Find the Best Split**: The core of the decision tree algorithm is finding the best feature and threshold to split the data. To do this:
    - **Iterate through all features**: The algorithm evaluates each feature one by one.
    - **For each feature, test all possible thresholds**: The algorithm tests different thresholds (split points) to decide where to split the data. For continuous features, this could be any value between the minimum and maximum values of the feature.
    - **Calculate the impurity**: For each split, the algorithm computes the impurity of the resulting subsets. The impurity measures how "mixed" the classes are in each subset. Lower impurity means the classes are more homogeneous. The most common impurity measures are:
        - **Gini Impurity**: $G = 1 - \sum p_i^2$ where p_i is the proportion of class i.
        - **Entropy**: $H = -\sum p_i \, log(p_i)$, where p_i is the proportion of class i.
        - **Squared Impurity**: Similar to Gini impurity, but squares the probabilities which is $1 - \sum p_i\text{^}4$.
    - **Select the best split**: The algorithm selects the feature and threshold that result in the lowest impurity (or highest, depending on the measure used).
5. **Partition the Data**: Once the best split is found, the dataset is divided into two subsets: one for samples that satisfy the split condition and another for samples that do not. This process creates two child nodes for the decision tree.
6. **Recursively Apply the Algorithm**: The algorithm repeats the process (steps 4-5) for each child node, continuing to split the data until the stopping condition is met. As the tree grows deeper, each node represents a smaller and more homogeneous subset of the data.
7. **Building the Tree**: This recursive process continues until the tree is fully built, meaning no further splits can be made or the stopping condition has been satisfied. The leaf nodes at the bottom of the tree represent the final class labels.
8. **Prediction**: After the tree is built, it can be used for predictions. For a given test sample:
    - The algorithm starts at the root and moves down the tree, making decisions based on the feature values of the sample.
    - When the algorithm reaches a leaf node, it assigns the class label associated with that node as the prediction.

Explanation of Splitting Criteria

The decision tree's effectiveness depends largely on how it splits the data at each node. The three main splitting criteria are:

- Gini Impurity: Measures the "impurity" of a node. If all instances in the node belong to the same class, the Gini impurity is 0. The goal is to minimize Gini impurity when making splits.
- Entropy: Entropy measures the randomness or uncertainty in the data. The higher the entropy, the more mixed the classes are at a node. The goal is to minimize entropy by creating pure subsets.
    - Entropy threshold prevents the tree from splitting further when the information gain (reduction in entropy) from a potential split is below a certain threshold. This helps avoid creating nodes that do not significantly improve the model, thus controlling overfitting.

- Squared Impurity: Similar to Gini impurity but gives more weight to smaller class proportions, which can be useful when the classes are imbalanced.

Each of these criteria aims to create splits that result in the most homogeneous subsets, helping the tree make better decisions.

Hyperparameter Tuning and Optimization

To prevent overfitting or underfitting, I introduced hyperparameters to control the growth of the tree:

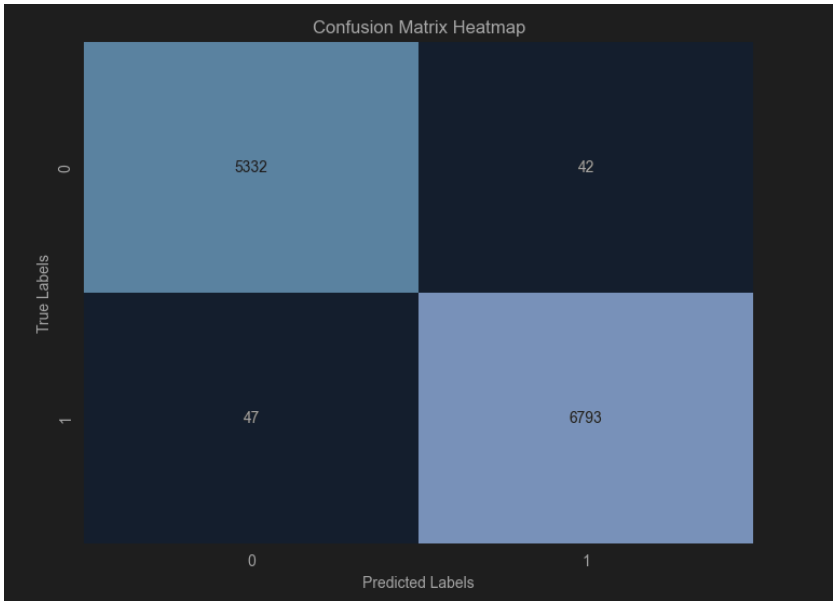
- Max Depth: Limits the depth of the tree to prevent it from growing too complex and capturing noise in the data.
- Min Samples Split: Ensures that a node is only split if it contains more than a minimum number of samples.
- Entropy Threshold: Prevents the tree from splitting further when the information gain (reduction in entropy) from a potential split is below a certain threshold. This helps avoid creating nodes that do not significantly improve the model, thus controlling overfitting.

I performed hyperparameter tuning using a grid search to find the best combination of these parameters based on cross-validation performance.

6. Model Performance

The model's performance was evaluated using the zero-one loss and accuracy metrics. Zero-one loss is the proportion of incorrect predictions made by the model. The decision tree achieved a zero-one loss of X% and accuracy of Y% on the test set.

The performance metrics show that the decision tree was able to capture the underlying patterns in the data, providing a strong baseline model for the classification task.



After implementing the decision tree and tuning its hyperparameters using grid search, I evaluated the model's performance. The grid search explored different combinations of max_depth and split_function to identify the optimal parameters that minimize the zero-one loss.

Grid Search Results

The grid search evaluated a total of 9 parameter combinations across 5-fold cross-validation, resulting in 45 iterations. The top 5 best results and the top 5 worst results are listed below:

```
Running grid search for best parameters...

Total number of combinations: 15  x  5 cv = 75 iterations

Zero-one loss: 0.44636 with params: {'max_depth': 15, 'split_function': 'scaled_entropy', 'entropy_threshold': 0.0001}
Zero-one loss: 0.44636 with params: {'max_depth': 10, 'split_function': 'scaled_entropy', 'entropy_threshold': 0.01}
Zero-one loss: 0.44636 with params: {'max_depth': 20, 'split_function': 'scaled_entropy', 'entropy_threshold': 0.001}
Zero-one loss: 0.44636 with params: {'max_depth': 20, 'split_function': 'scaled_entropy', 'entropy_threshold': 0.01}
Zero-one loss: 0.44636 with params: {'max_depth': 15, 'split_function': 'scaled_entropy', 'entropy_threshold': 0.01}
Zero-one loss: 0.44636 with params: {'max_depth': 15, 'split_function': 'scaled_entropy', 'entropy_threshold': 0.001}
Zero-one loss: 0.44636 with params: {'max_depth': 10, 'split_function': 'scaled_entropy', 'entropy_threshold': 0.001}
Zero-one loss: 0.44636 with params: {'max_depth': 10, 'split_function': 'scaled_entropy', 'entropy_threshold': 0.0001}
Zero-one loss: 0.44636 with params: {'max_depth': 20, 'split_function': 'scaled_entropy', 'entropy_threshold': 0.0001}
Zero-one loss: 0.12916 with params: {'max_depth': 10, 'split_function': 'squared_impurity'}
Zero-one loss: 0.12854 with params: {'max_depth': 10, 'split_function': 'gini'}
Zero-one loss: 0.02256 with params: {'max_depth': 15, 'split_function': 'gini'}
Zero-one loss: 0.02284 with params: {'max_depth': 15, 'split_function': 'squared_impurity'}
Zero-one loss: 0.00659 with params: {'max_depth': 20, 'split_function': 'squared_impurity'}
Zero-one loss: 0.00690 with params: {'max_depth': 20, 'split_function': 'gini'}

Top 5 Best Results:
Rank 1: Zero-one loss: 0.006591 with params: {'max_depth': 20, 'split_function': 'squared_impurity'}
Rank 2: Zero-one loss: 0.006898 with params: {'max_depth': 20, 'split_function': 'gini'}
Rank 3: Zero-one loss: 0.022557 with params: {'max_depth': 15, 'split_function': 'gini'}
Rank 4: Zero-one loss: 0.022843 with params: {'max_depth': 15, 'split_function': 'squared_impurity'}
Rank 5: Zero-one loss: 0.128544 with params: {'max_depth': 10, 'split_function': 'gini'}

Top 5 Worst Results:
Rank 1: Zero-one loss: 0.446362 with params: {'max_depth': 15, 'split_function': 'scaled_entropy', 'entropy_threshold': 0.001}
Rank 2: Zero-one loss: 0.446362 with params: {'max_depth': 15, 'split_function': 'scaled_entropy', 'entropy_threshold': 0.0001}
Rank 3: Zero-one loss: 0.446362 with params: {'max_depth': 20, 'split_function': 'scaled_entropy', 'entropy_threshold': 0.01}
Rank 4: Zero-one loss: 0.446362 with params: {'max_depth': 20, 'split_function': 'scaled_entropy', 'entropy_threshold': 0.001}
Rank 5: Zero-one loss: 0.446362 with params: {'max_depth': 20, 'split_function': 'scaled_entropy', 'entropy_threshold': 0.0001}

Best parameters: {'max_depth': 20, 'split_function': 'squared_impurity'}
Best score (zero-one loss): 0.006590932350834101
Zero-one loss on test set with best params: 0.007287
Test accuracy: 0.992713
Great result! The model performs very well with a good classification.
```

Top 5 Best Results:

1. **Zero-one loss: 0.006591** with params: {'max_depth': 20, 'split_function': 'squared_impurity'}
2. **Zero-one loss: 0.006898** with params: {'max_depth': 20, 'split_function': 'gini'}
3. **Zero-one loss: 0.022557** with params: {'max_depth': 15, 'split_function': 'gini'}
4. **Zero-one loss: 0.022843** with params: {'max_depth': 15, 'split_function': 'squared_impurity'}

5. **Zero-one loss: 0.128544** with params: {'max_depth': 10, 'split_function': 'gini'}

Top 5 Worst Results:

1. **Zero-one loss: 0.446362** with params: {'max_depth': 15, 'split_function': 'scaled_entropy', 'entropy_threshold': 0.001}
2. **Zero-one loss: 0.446362** with params: {'max_depth': 15, 'split_function': 'scaled_entropy', 'entropy_threshold': 0.0001}
3. **Zero-one loss: 0.446362** with params: {'max_depth': 20, 'split_function': 'scaled_entropy', 'entropy_threshold': 0.01}
4. **Zero-one loss: 0.446362** with params: {'max_depth': 20, 'split_function': 'scaled_entropy', 'entropy_threshold': 0.001}
5. **Zero-one loss: 0.446362** with params: {'max_depth': 20, 'split_function': 'scaled_entropy', 'entropy_threshold': 0.0001}

Best Parameters

The best-performing model used the following parameters:

- **Max Depth**: 20
- **Split Function**: Squared Impurity

With these parameters, the model achieved a **zero-one loss of 0.00659** on the validation set during grid search. After training the model using these parameters on the full training set and evaluating it on the test set, the model achieved a **zero-one loss of 0.00729** and the test **accuracy of 0.992713** on the test set, indicating that the model was able to accurately classify the majority of mushrooms in the test data.

7. Conclusion

In this project, I successfully implemented a custom decision tree classifier to predict whether mushrooms are poisonous or edible based on their physical attributes. Through various stages, including data preprocessing, feature encoding, correlation analysis, and model training, I built a model that demonstrated strong predictive performance.

Key Highlights:

- **Interactive Flow**: The interactive nature of the implementation allowed users to either choose the best parameters via grid search or manually enter custom parameters for model tuning. This flexibility enables experimentation with different configurations, fostering better understanding and control over the model-building process.

- **Grid Search Results**: The grid search helped us optimize the decision tree's performance by identifying the best combination of max_depth and split_function. The best parameters were found to be max_depth: 20 and split_function: squared_impurity, which resulted in a very low zero-one loss.

- **Model Performance**: After fine-tuning the decision tree model, the final evaluation on the test set yielded a **zero-one loss of 0.00729** and the test **accuracy of 0.992713**, indicating that the model was able to make highly accurate predictions.

Potential Improvements:

- **Feature Engineering**: Additional domain-specific knowledge could be used to create new features that better capture the relationships between the attributes and the target variable.
- **Hyperparameter Tuning**: While grid search provided an optimal solution, more advanced techniques such as randomized search or Bayesian optimization could be explored to further improve model performance and reduce the computational cost of hyperparameter tuning.

Conclusion:

The decision tree model built in this project demonstrates that physical characteristics of mushrooms can be effectively used to predict whether they are poisonous or edible. The low error rate achieved on the test set highlights the model's potential for practical applications, such as aiding in the safe identification of mushrooms in the wild. Further improvements, such as feature engineering and more advanced modeling techniques, could make the model even more powerful.