Decision Trees

Farzad Rahim khanian
Università degli Studi di Milano
Email: farzad.rahimkhanian@studenti.unimi.it
Matriculation Number: 987094

1 Introduction

Decision trees are foundational machine learning models, widely recognized for their interpretability and ease of implementation. This report details the systematic evolution of our project, starting with a basic binary implementation of the ID3 algorithm and progressively enhancing it with ideas inspired by C4.5 and CART to address its inherent limitations.

The original ID3 algorithm lacks critical features, such as support for continuous attributes, robust handling of missing data, and mechanisms to control overfitting through pruning. To address these challenges, we incorporated an approach inspired by C4.5 to handle missing data effectively and implemented dynamic splitting thresholds for continuous features. To combat overfitting—an issue that arises as trees grow deeper and begin to capture noise—we applied cost-complexity pruning after tree construction.

To optimize model performance, we employed Bayesian optimization for hyperparameter tuning, allowing us to efficiently explore the parameter space while minimizing computational overhead. Additionally, we extended the algorithm by integrating advanced decision criteria, such as gain ratio and Gini impurity, resulting in a more flexible and powerful decision tree implementation.

All code in this project has been implemented from scratch, ensuring full control over the algorithms while maintaining compatibility with the scikit-learn library. This compatibility enables us to leverage scikit-learn's extensive tools and capabilities to enhance and analyze our custom models.

The project culminates with the implementation of Random Forest, which utilizes our enhanced decision tree as its base learner. This ensemble approach represents the final step in our iterative development process, combining the strengths of individual trees for improved prediction accuracy and robustness.

2 Dataset

The dataset used in this study consists of 61,069 samples of hypothetical mushrooms based on 173 species, with an average of 353 samples per species. Each sample is labeled as either

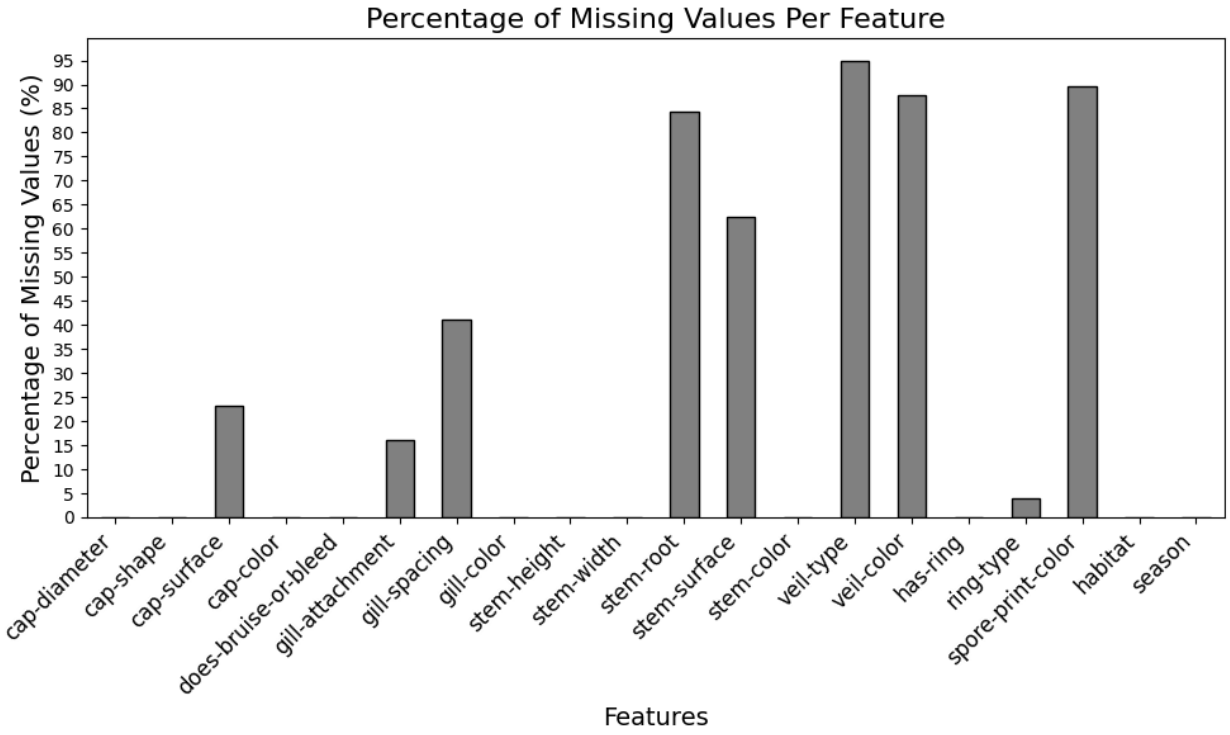

Figure 1: Missing percentage for categorical features in the secondary mushroom dataset, illustrating varying rates of missing data that may affect model preprocessing and performance.

edible (e) or poisonous (p), where the poisonous class also includes mushrooms of unknown edibility for practical classification purposes. This results in a binary classification task.

The dataset exhibits a moderate class imbalance, with 33,888 poisonous samples (55.5%) and 27,181 edible samples (44.5%). Given this imbalance, stratification will be crucial in future applications of this dataset, such as during training and cross-validation, to ensure that class proportions are preserved across splits and that the model is exposed to a representative distribution of the data.

The dataset contains 20 features, comprising:

- 3 continuous features: cap-diameter, stem-height, and stem-width.
- 17 categorical features.

Only categorical features in the dataset contain missing values, with missing rates varying widely across features. The presence of features with high missing rates poses challenges for data imputation and preprocessing. Imputing these features using statistical methods, such as the mode value, risks introducing significant bias, especially when the missing rate exceeds 50%. For algorithms that do not inherently handle missing data, such as ID3, one practical solution is to exclude features with exceptionally high missing rates (e.g., veil-type) and impute the remaining ones using the mode of the respective feature.

3 Preprocessing

Effective preprocessing is essential for ensuring the success of machine learning models, particularly decision trees, which require tailored handling of categorical and continuous features. Our preprocessing approach is designed to preserve interpretability while ensuring compatibility with decision tree algorithms.

3.1 Handling Categorical Features Without One-Hot Encoding

Many machine learning algorithms, such as logistic regression, support vector machines, and neural networks, require one-hot encoding for categorical variables. However, decision trees, including ID3, C4.5, CART, and ensemble methods like Random Forest and Gradient Boosting Machines, do not require one-hot encoding and can handle categorical features natively.

One-hot encoding introduces several drawbacks when used in tree-based models:

- **Increased dimensionality** – When categorical features have many unique values, one-hot encoding significantly expands the number of features, leading to increased memory consumption and computational inefficiency.
- **Loss of categorical relationships** – One-hot encoding treats all categories as independent binary features, disregarding any ordinal or hierarchical relationships that may exist within the data.
- **Suboptimal feature selection** – Decision trees split nodes based on entropy or Gini impurity reduction. With one-hot encoding, a tree can only split on one category at a time. This means a category will only be selected if, by itself, it reduces entropy more than any other feature. This limitation prevents the algorithm from evaluating categorical variables holistically.

Some advanced tree-based algorithms, such as LightGBM, allow categorical variables to be specified directly. These methods evaluate combinations of categorical values at each split, rather than isolating individual categories, resulting in more efficient and accurate decision-making. For this reason, our decision tree implementation does not use one-hot encoding, preserving both interpretability and computational efficiency.

3.2 Handling Missing Data

Handling missing data is a critical step in preprocessing, as improper treatment can introduce bias or lead to unreliable predictions. Our approach varies depending on the proportion of missing values in each feature.

3.2.1 Dropping Features with High Missing Rates

Features with excessive missing values (above 50%) are removed, as imputing them may introduce significant bias. The following features are removed from the dataset:

- stem-surface (62.4% missing)
- stem-root (84.4% missing)
- veil-type (94.8% missing)
- veil-color (87.9% missing)
- spore-print-color (89.6% missing)

By eliminating these features, we reduce potential biases while preserving the integrity of the dataset.

3.2.2 Imputing Features with Low to Moderate Missing Rates

For features with a moderate level of missing values, we apply mode imputation, replacing missing values with the most frequent category. This method is computationally efficient and ensures minimal information loss. The following features are imputed:

- cap-surface (23.1% missing)
- gill-attachment (16.2% missing)
- ring-type (4.0% missing)
- gill-spacing (41.0% missing)

In the initial implementation of ID3, this manual imputation is necessary to handle missing data. However, in the final implementation, missing values are handled dynamically during training using an approach inspired by C4.5, eliminating the need for manual imputation.

3.3 Encoding Target Labels

The dataset contains a binary classification target, which is encoded as follows:

- Edible (e) $\rightarrow 0$
- Poisonous (p) $\rightarrow 1$

This encoding ensures compatibility with tree-based algorithms while maintaining interpretability.

3.4 Train-Test Split

To prepare the dataset for training and evaluation, we divide it into training and testing sets with an 80:20 ratio. Given the class imbalance in the dataset (55.5% poisonous, 44.5% edible), stratified sampling is applied during the split. Stratification ensures that the class proportions remain consistent across both the training and test sets, preventing bias in model evaluation.

4 Methodology

4.1 Training Setup and Parallel Processing

All models in this study are trained using two primary stopping criteria:

- Maximum tree depth, ranging from 2 to 30
- Minimum number of samples required for a split, ranging from 2 to 3000, to regulate tree expansion.

Given that the evaluation process involves training multiple models iteratively across different hyperparameter configurations, parallel processing is utilized to enhance computational efficiency. The *joblib* library in Python enables concurrent execution of multiple training instances, significantly reducing processing time and allowing for efficient experimentation.

4.2 Phase 1: Implementing and Modifying ID3

4.2.1 Overview of the ID3 Algorithm

The Iterative Dichotomiser 3 (ID3) algorithm constructs a decision tree by recursively partitioning the dataset using the feature that maximizes information gain at each step. The tree expands until all samples in a node belong to the same class or a predefined stopping criterion is met.

The original version of ID3 only supports categorical features and does not include a mechanism for handling missing values. As a result, in the initial implementation, missing values are addressed through preprocessing, where features with high missing rates are removed, and moderate missing values are imputed using the mode of the respective feature.

4.2.2 Steps of the ID3 Algorithm

The ID3 algorithm follows these steps:

1. Compute the entropy of the dataset:

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

where c is the number of classes, and p_i is the proportion of samples belonging to class i.

2. For each feature, compute the information gain:

$$IG(S, X) = H(S) - \sum_{v \in X} \frac{|S_v|}{|S|} H(S_v) \tag{2}$$

where S_v represents the subset of data where feature X takes value v.

3. Select the feature with the highest information gain and split the dataset accordingly.

4. Recursively apply the process to child nodes until stopping criteria are met:

   - All samples in a node belong to the same class.
   - The maximum allowed depth is reached.
   - The number of samples in a node falls below the minimum split threshold.

5. Assign a class label to leaf nodes based on majority voting if further splits are not possible.

4.2.3 Handling Continuous Features in ID3

The original ID3 algorithm does not support continuous features. To extend its functionality, the implementation was modified to:

- Sort continuous feature values in ascending order.
- Identify potential split points between consecutive values.
- Compute the information gain for each possible threshold and select the best split.
- Perform binary splitting, creating two child nodes: one for values $\leq$ the threshold and another for values $>$ the threshold.

For a given continuous feature X, the optimal split threshold T^* is determined as:

$$T^* = \arg\max_T IG(S, X, T) \tag{3}$$

where $IG(S, X, T)$ represents the information gain for a split at threshold T.

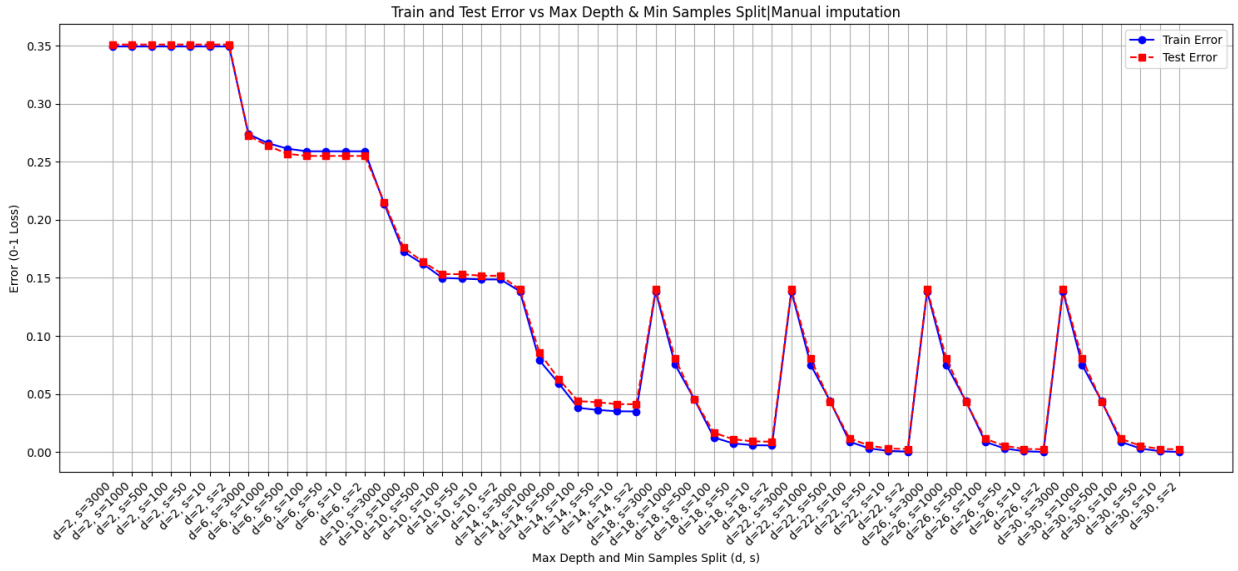

Figure 2: Training and test error curves with manual imputation of missing data for`max_depth` from 2 to 30, with the `min_samples_split` varying from 3000 to 2 for each depth value.

4.3 Phase 2: Native Missing-Data Handling Approach

In this phase, we enhance the decision tree to handle missing data natively, eliminating the need for preprocessing-based imputation. Our approach is inspired by C4.5 but is simplified for computational efficiency. Additionally, we introduce a penalty parameter to regulate the influence of features with high missing rates on the tree structure.

4.3.1 Skipping Missing Values During Split Selection

For each feature, information gain is computed only on the subset of samples that contain non-missing values for that feature. Missing values do not contribute to the split threshold or category selection. This follows the core idea of C4.5, where missing values are ignored when evaluating a feature's utility. However, in contrast to C4.5, which distributes missing values probabilistically, we adopt a simpler deterministic approach by skipping missing values entirely during gain calculation.

4.3.2 Hard Assignment of Missing Values at Training Time

Once the best feature is selected for a node, samples are divided into three groups:

- Those with non-missing values that go to the left child.
- Those with non-missing values that go to the right child.
- Those with missing values for the selected feature.

To determine where the missing samples should be assigned, we compute the ratio of non-missing samples that ended up in the left and right child nodes:

$$\text{missing\_left\_fraction} = \frac{L}{L + R} \tag{4}$$

where L and R represent the number of non-missing samples assigned to the left and right child nodes, respectively. If the fraction is greater than or equal to 0.5, all missing samples are assigned to the left; otherwise, they are assigned to the right. This fraction is stored in `node.missing_left_fraction` for consistent handling at prediction time.

4.3.3 Prediction with Missing Values

When making predictions for new samples that have missing values for the feature at an internal decision node, we use the stored `missing_left_fraction`:

- If missing_left_fraction ≥ 0.5, the sample is directed to the left subtree.
- Otherwise, it is directed to the right subtree.

This method ensures that missing values are assigned in accordance with the majority tendency observed in the training data, minimizing bias in predictions.

4.3.4 Effectiveness of the Approach

- **Skipping missing values** during split selection assumes that missingness is approximately random per feature, avoiding systematic bias in information gain calculations.
- **Hard assignment of missing values** ensures that missing samples are allocated proportionally to the observed distribution of non-missing data. Unlike naive imputation methods, which may introduce bias, this method preserves the natural structure of the dataset.
- **Prevents missing values from distorting the tree**: Without proper handling, a feature with a high missing rate could still be selected if its observed data has a strong predictive signal. The introduced penalty mechanism addresses this issue.

4.3.5 Introducing a Missing Data Penalty

To further refine missing data handling, we introduce a penalty term that prevents features with high missing rates from dominating the tree structure. Instead of using the raw fraction of non-missing samples, we introduce a tunable penalty exponent, denoted as `missing_penalty_power`. This modifies the gain calculation as follows:

$$\text{penalized\_gain} = \text{raw\_gain} \times (\text{fraction\_nonmissing})^{\text{missing\_penalty\_power}} \tag{5}$$

where:

$$\text{fraction\_nonmissing} = \frac{\#\text{samples with non-missing feature}}{\#\text{total samples}}. \tag{6}$$

By adjusting `missing_penalty_power`, we control the extent to which missing data influences split selection:

- A higher penalty reduces the likelihood that a feature with a high missing rate will be selected.
- A lower penalty allows features with missing values to still be used if they provide significant information gain.

Through experimental trials, we found that setting `missing_penalty_power` $= 2$ effectively **prevents features with high missing rates from dominating splits**, while still allowing moderately missing features to contribute meaningfully.

4.3.6 Limitations and Future Enhancements

While this method effectively handles missing values, some refinements could further improve the approach:

- **Fractional Assignment (C4.5-style)**: Instead of hard assignment, missing values could be probabilistically distributed across child nodes, requiring weighted sample counts.
- **Surrogate Splitting (CART-style)**: When a primary split feature is missing, alternative features could be used to decide the sample's direction.

These extensions, while computationally more complex, could improve model flexibility and robustness.

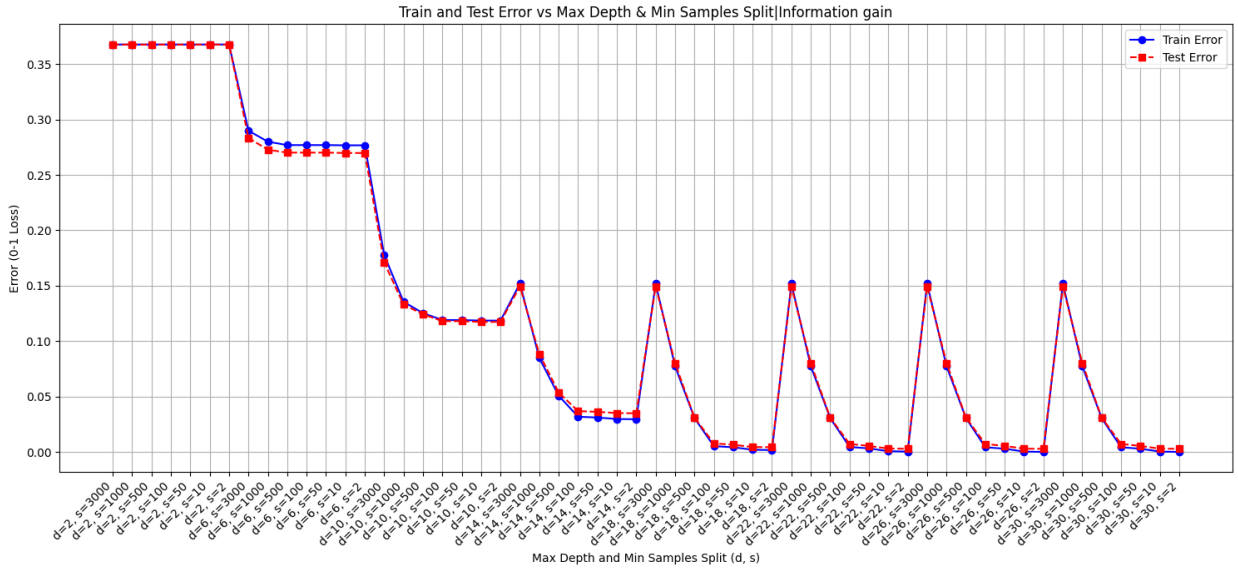

Figure 3: Training and test error curves using default splitting criterion(Information Gain) and native handling of missing data for `max_depth` from 2 to 30, with the `min_samples_split` varying from 3000 to 2 for each depth value. The missing data penalty is `missing_penalty_power=2`.

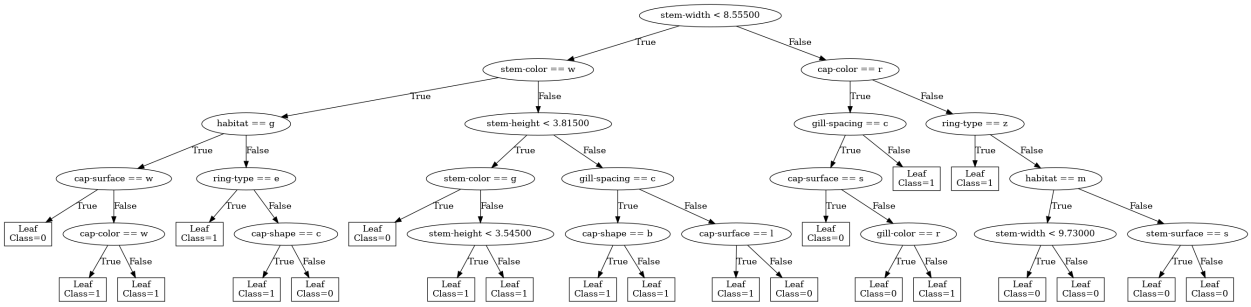

Figure 4: Tree constructed with default splitting criterion(Information Gain) at `max_depth=5` and `min_samples_split=2`.

4.4 Alternative Splitting Criteria

In the initial implementation of ID3, information gain was used as the primary criterion for selecting the best split at each node. However, information gain has limitations, particularly in handling categorical features with many unique values. To address this, gain ratio, used in C4.5, and Gini index, used in CART, are introduced as alternative splitting criteria to refine the decision tree's performance.

4.4.1 Gain Ratio (C4.5)

Gain ratio is an extension of information gain that penalizes features with many unique values, preventing the model from favoring high-cardinality categorical features, which could lead to overfitting.

The information gain for a feature X is given by:

$$IG(S, X) = H(S) - \sum_{v \in X} \frac{|S_v|}{|S|} H(S_v) \tag{7}$$

where $H(S)$ represents the entropy of the dataset, and S_v represents subsets of data split by feature X. While information gain selects the feature that maximizes entropy reduction, it tends to favor categorical features with many distinct values, as splitting on such features can create many small, pure subsets.

C4.5 addresses this issue by introducing the gain ratio, which normalizes information gain by the intrinsic value of the feature:

$$IV(X) = -\sum_{v \in X} \frac{|S_v|}{|S|} \log_2 \frac{|S_v|}{|S|} \tag{8}$$

The gain ratio is then computed as:

$$GR(S, X) = \frac{IG(S, X)}{IV(X)} \tag{9}$$

This normalization prevents the selection of features that create a large number of small partitions unless the gain remains significantly high after the adjustment. Continuous features with many possible threshold values may also be penalized under this criterion, as their intrinsic value can be large, reducing their gain ratio. This can lead to cases where a continuous feature is not selected for expansion despite having a high raw information gain.

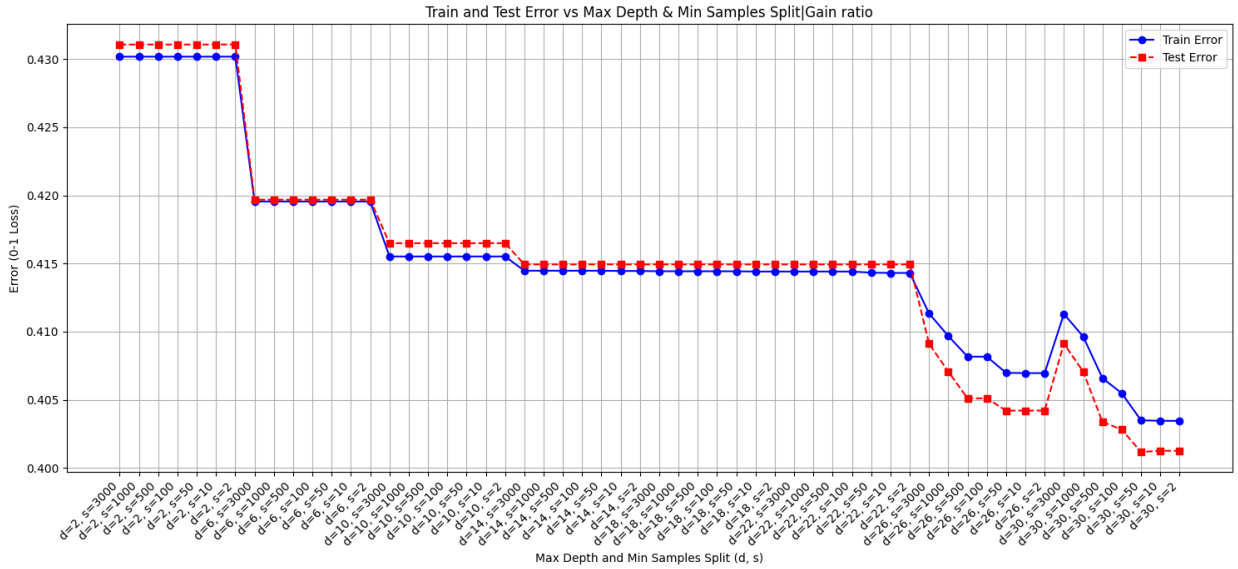

Figure 5: Training and test error curves using Gain Ratio as splitting criterion and native handling of missing data for `max_depth` from 2 to 30, with the `min_samples_split` varying from 3000 to 2 for each depth value. The missing data penalty is `missing_penalty_power=2`.

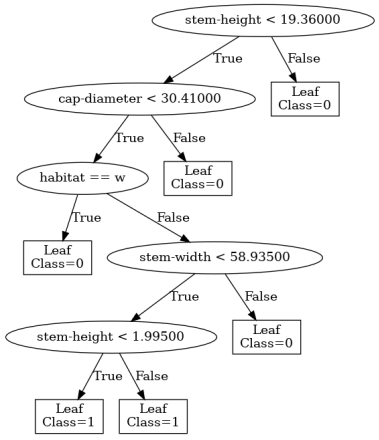

Figure 6: Tree constructed with Gain Ratio as splitting criterion at `max_depth=5` and `min_samples_split=2`.

4.4.2 Gini Index (CART)

The Gini index, used in the CART algorithm, is another alternative splitting criterion. Unlike entropy-based measures, which focus on reducing uncertainty, the Gini index measures the probability of a randomly chosen sample being incorrectly classified if it were assigned a label based on the current distribution.

The Gini impurity for a dataset S is defined as:

$$Gini(S) = 1 - \sum_{i=1}^{c} p_i^2 \tag{10}$$

where p_i is the proportion of samples belonging to class i. Like entropy, the Gini index measures impurity, but it tends to be computationally faster as it avoids logarithmic calculations.

The Gini index for a feature X is computed as:

$$Gini(S, X) = \sum_{v \in X} \frac{|S_v|}{|S|} Gini(S_v) \tag{11}$$

where $Gini(S_v)$ is the impurity of each subset after the split.

While the Gini index and information gain often result in similar tree structures, Gini tends to create more balanced splits, as it does not overemphasize rare categories. Unlike gain ratio, the Gini index does not explicitly penalize high-cardinality categorical features, but it remains computationally efficient, making it well-suited for large datasets.

Both gain ratio and Gini index provide alternative approaches to tree construction, addressing the limitations of information gain in different ways. The choice of splitting criterion ultimately depends on dataset characteristics and computational constraints.

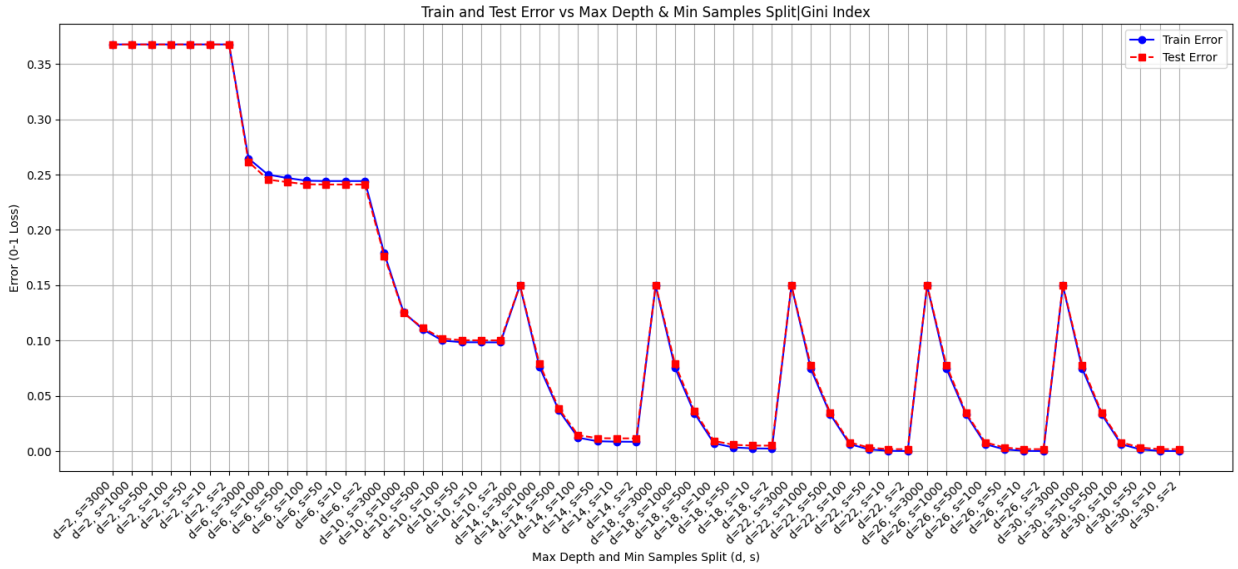

Figure 7: Training and test error curves using Gini Index as splitting criterion and native handling of missing data for `max_depth` from 2 to 30, with the `min_samples_split` varying from 3000 to 2 for each depth value. The missing data penalty is `missing_penalty_power=2`.

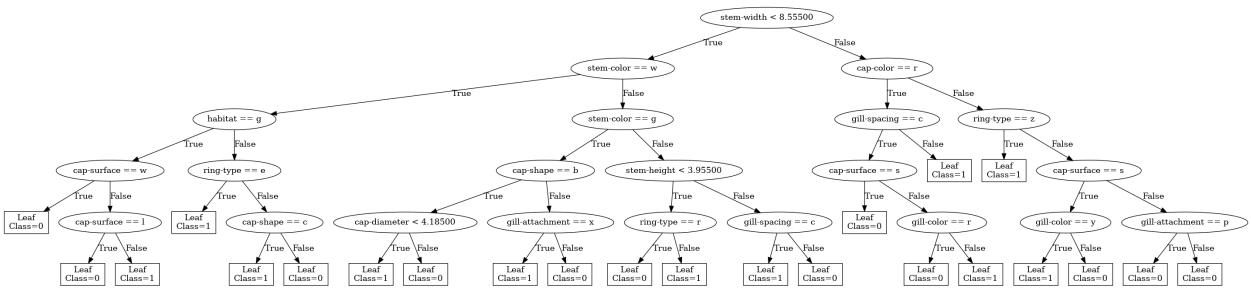

Figure 8: Tree constructed with Gini Index as splitting criterion at `max_depth=5` and `min_samples_split=2`.

4.5 Pruning

Cost-complexity pruning, used in the CART algorithm, provides a systematic method for balancing tree complexity and predictive accuracy. It sequentially prunes subtrees to reduce overfitting while maintaining classification performance. The process introduces a complexity parameter, α, which controls the trade-off between the number of leaves and the misclassification error. Given a fully grown decision tree T, the cost of a subtree T_t rooted at node t is defined as:

$$C_\alpha(T_t) = C(T_t) + \alpha \cdot |T_t| \tag{12}$$

where $C(T_t)$ is the classification error (or another impurity measure such as Gini) and $|T_t|$ is the number of leaf nodes. The parameter α penalizes tree size to prevent excessive branching.

Pruning is performed iteratively using **weakest-link pruning**:

1. Compute $\alpha_t = \frac{C(T_t) - C(T_t^*)}{|T_t| - |T_t^*|}$ for each internal node, where T_t^* is the subtree after collapsing node t into a leaf.
2. Identify the node with the smallest α_t and prune its subtree.
3. Repeat this process, generating a sequence of increasingly pruned trees.
4. Use cross-validation or a validation set to select the optimal α.

4.5.1 Effect of Different α Values

- $\alpha = 0$ results in minimal pruning, retaining a large tree that may overfit the training data.
- Small α values prune only weakly relevant branches, leading to slight complexity reduction.
- High α values aggressively prune the tree, reducing it to a shallow structure that may underfit the data.

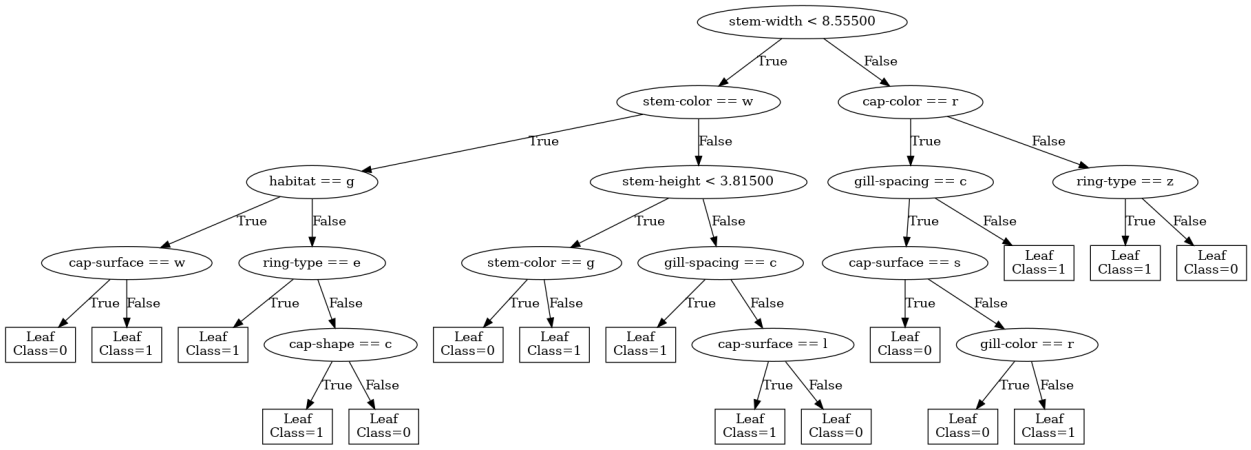

Figure 9: Cost-complexity pruning has been performed on a tree with the same parameters seen in Figure 4. All the pure nodes have been pruned without loss of accuracy. `alpha=0`

4.5.2 Alternative Forms of Pruning

Manually setting `max_depth` and `min_samples_split` can act as implicit pruning methods. A strict depth constraint prevents the tree from growing excessively deep, while increasing the minimum split size reduces the number of small, unreliable branches. Cost-complexity pruning provides a systematic way to control model complexity, ensuring a balance between overfitting and generalization.

4.6 Random Forest Implementation

A random forest is an ensemble learning method that constructs multiple decision trees and aggregates their predictions to improve accuracy and robustness. This implementation creates a basic random forest classifier using a custom decision tree model as the base estimator. The model performs bootstrap sampling, selects random feature subsets, and combines the predictions of individual trees using majority voting.

4.6.1 Model Construction and Training

The random forest is implemented as a class that inherits from `BaseEstimator` and `ClassifierMixin`, ensuring compatibility with the `scikit-learn` API. It consists of multiple instances of a specified base decision tree, trained independently on different bootstrap samples of the dataset.

Each tree in the ensemble is trained as follows:

1. A bootstrap sample of the dataset is generated by randomly selecting training instances with replacement.

2. A subset of features is randomly chosen for splitting at each node, determined by the `max_features` parameter.

3. The base decision tree is trained on the sampled data using only the selected feature subset.

4. The trained tree is added to the ensemble.

4.6.2 Prediction Using Majority Voting

For inference, the trained trees independently predict the class of each sample. The final prediction is determined through majority voting across all trees:

1. Each tree makes a prediction using its selected feature subset.

2. The predictions from all trees are collected into an array.

3. The most frequently predicted class is assigned as the final output.

This majority voting strategy reduces variance and improves robustness compared to a single decision tree.

4.6.3 Model Evaluation

The `score` function computes the accuracy of the model by comparing predicted labels to ground truth labels. Since random forests are designed to reduce overfitting, they generally perform better than a single decision tree, particularly on complex datasets.

This implementation provides a flexible and customizable framework for random forests while allowing experimentation with different decision tree algorithms as base estimators.

4.7 Hyperparameter Tuning Using Bayesian Optimization

Hyperparameter tuning plays a crucial role in optimizing the performance of machine learning models. In this study, we utilize Bayesian optimization, a sequential model-based approach, to efficiently explore the hyperparameter space and identify the best-performing configuration. The tuning process is conducted using BayesSearchCV from the *scikit-optimize* library, ensuring compatibility with our scikit-learn-based models.

4.7.1 Bayesian Optimization: Theory and Advantages

Bayesian optimization is a probabilistic method that models the objective function using a surrogate model, typically a Gaussian process. Unlike exhaustive search methods such as grid search, Bayesian optimization balances exploration and exploitation by sequentially selecting promising hyperparameter values based on previous evaluations.

The tuning process follows these steps:

1. **Surrogate Model Construction**: A probabilistic model estimates the relationship between hyperparameters and model performance.

2. **Acquisition Function Evaluation**: The acquisition function determines the next set of hyperparameters to test, prioritizing regions of high uncertainty or expected improvement.

3. **Model Evaluation and Update**: The model is trained with the suggested hyperparameters, and the surrogate model is updated with the new results.

4. **Iterative Refinement**: The process repeats, refining the search space with each iteration to efficiently converge toward an optimal solution.

Compared to traditional hyperparameter tuning methods:

- **Grid Search** requires evaluating all possible hyperparameter combinations, making it computationally infeasible for large search spaces.
- **Random Search** samples hyperparameters randomly, improving efficiency over grid search but lacking systematic refinement.
- **Bayesian Optimization** learns from previous evaluations to intelligently guide the search, reducing the number of evaluations required to find an optimal configuration.

4.7.2 Search Space and Tuning Procedure

To tune the decision tree hyperparameters, we define the following search space:

- **Maximum Depth** (max_*depth*): {2, 6, 10, 14, 18, 22, 26, 30}
- **Minimum Samples per Split** (min_*samples_split*): {3000, 1000, 500, 100, 50, 10, 2}
- **Missing Penalty Power** (missing_penalty_power): {0.5, 1.0, 1.5, 2.0}
- **Splitting Criterion**: {"information gain", "gain ratio", "gini"}

The search space includes hyperparameters that control tree complexity, regulate missing value influence, and determine the splitting criterion. Although an experimentally optimal value for missing_penalty_power was identified, it was still included in the tuning process to validate its effect across different conditions.

To ensure robust performance estimation, we applied stratified k-fold cross-validation with $k = 5$, maintaining class distribution across folds. The optimization process was performed over **30 iterations**, balancing computational efficiency with performance exploration. Due to high computational cost, RandomForest was excluded from tuning, as training multiple ensembles would have significantly increased execution time.

4.7.3 Results of Hyperparameter Tuning

After completing the tuning process, the best hyperparameter combination was found to be:

- **Splitting Criterion**: Information Gain
- **Maximum Depth**: 26
- **Minimum Samples per Split**: 2
- **Missing Penalty Power**: 1.5

The best accuracy achieved during cross-validation was **0.9981**, demonstrating that the optimized decision tree model effectively balances complexity and performance.

Bayesian optimization provided an efficient and systematic approach to hyperparameter tuning, allowing us to identify an optimal configuration with significantly fewer iterations than traditional search methods.

5 Conclusion

This study explored the behavior of decision trees under different splitting criteria—information gain, gain ratio, and the Gini index—examining their effect on model complexity, generalization, and potential overfitting. Through visualizing tree structures and analyzing training and test error trends, we assessed how these criteria influence decision-making and feature selection. The results obtained from Bayesian hyperparameter tuning provided further insight into optimal tree configurations and the robustness of the learned models.

The decision tree visualizations (Figures 4, 6, and 8) revealed that different splitting criteria significantly impact tree structure. The **information gain-based tree (Figure 4)** exhibited balanced expansion, leveraging both numerical and categorical splits to maximize entropy reduction. The **gain ratio-based tree (Figure 6)** showed more constrained growth, as gain ratio penalizes features with high cardinality, limiting excessive branching. The **Gini index-based tree (Figure 8)** resembled the information gain tree but with slight differences in feature selection, reflecting its emphasis on class homogeneity. These structural variations illustrate how each criterion prioritizes different feature properties, influencing model complexity.

The training and test error curves (Figures 3, 5, and 7) further demonstrate the impact of these criteria on generalization. **Figures 3 and 7 (information gain and Gini index, respectively)** show an overall decrease in error with increasing depth, but periodic fluctuations arise due to variations in **min_samples_split**, which ranged from **3000 to 2** at each depth level. These fluctuations are expected and do not indicate instability. **Figure 5 (gain ratio)**, in contrast, shows a smoother trend, suggesting that gain ratio regulates tree growth more effectively, preventing unnecessary splits and over-expansion.

A key focus of this study was determining whether the model suffered from **overfitting**, given the exceptionally high accuracy (**0.9981**) found through Bayesian hyperparameter

tuning. The optimal configuration—**max_depth = 26, min_samples_split = 2, missing_penalty_power = 1.5, and information gain** as the splitting criterion—suggests that deeper decision boundaries were necessary for optimal classification. However, the risk of overfitting remains a valid concern. Several factors contribute to the uncertainty in determining whether overfitting has occurred:

- **Cross-validation minimized memorization effects:** The five-fold stratified cross-validation method ensured that the model was tested on different subsets of the data, reducing the risk of overfitting. The consistency in accuracy across folds suggests that the model was not simply memorizing training examples.
- **The dataset consists of simulated mushrooms generated randomly:** Since the dataset is not derived from real-world biological characteristics but is instead generated procedurally, the decision tree may have learned patterns in the **random data distribution** rather than meaningful, interpretable relationships. If the generative process behind the dataset created **artificially strong feature separability**, this could explain the unusually high accuracy.
- **Class separability may naturally support high accuracy:** If certain categorical features provide strong distinguishing power, deeper decision trees can legitimately achieve near-perfect classification without overfitting.
- **Regularization effects from missing data handling:** The inclusion of **missing_penalty_power = 1.5** discouraged reliance on sparse features, reducing the potential for overfitting.
- **Absence of a sharp divergence between training and test error:** A significant gap between training and test performance is a hallmark of overfitting, but this was not observed. The error trends remain relatively stable, suggesting that the model is not merely memorizing data.

Despite these indicators, the **final test for overfitting would require evaluation on an independent dataset**. If the model maintains similar accuracy on completely unseen data, it would confirm that the deeper tree structure genuinely captures relevant patterns. However, if accuracy drops significantly, it would indicate that the model over-specialized to the dataset used in cross-validation.

In conclusion, while Bayesian hyperparameter tuning favored a deep tree, there is **no definitive evidence of overfitting** based on current observations. The dataset's artificial nature introduces additional uncertainty, as the decision tree may have learned patterns from the **random generative process** rather than meaningful biological relationships. The differences in tree structures and error trends across splitting criteria highlight how model complexity is influenced by the choice of split metric. Gain ratio enforces natural constraints on tree growth, while information gain and Gini index allow for more aggressive expansion. These findings emphasize the importance of **careful hyperparameter tuning, external validation, and dataset interpretation in decision tree learning** to ensure models generalize beyond their training data.

Declaration

I declare that this material, which I now submit for assessment, is entirely my own work and has not been taken from the work of others, save and to the extent that such work has been cited and acknowledged within the text of my work. I understand that plagiarism, collusion, and copying are grave and serious offences in the university and accept the penalties that would be imposed should I engage in plagiarism, collusion or copying. This assignment, or any part of it, has not been previously submitted by me or any other person for assessment on this or any other course of study.