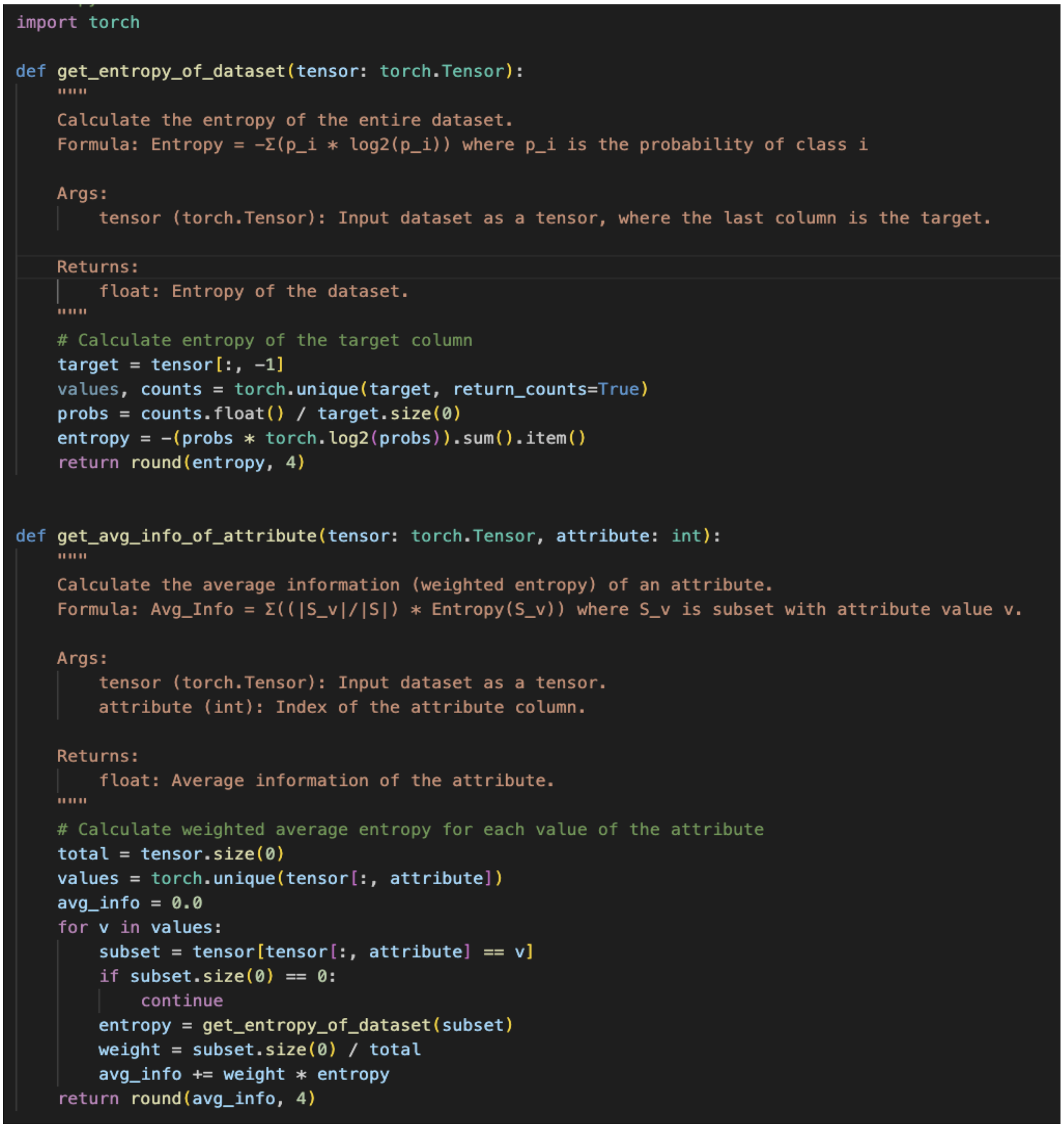
UE23CS352A: Machine Learning

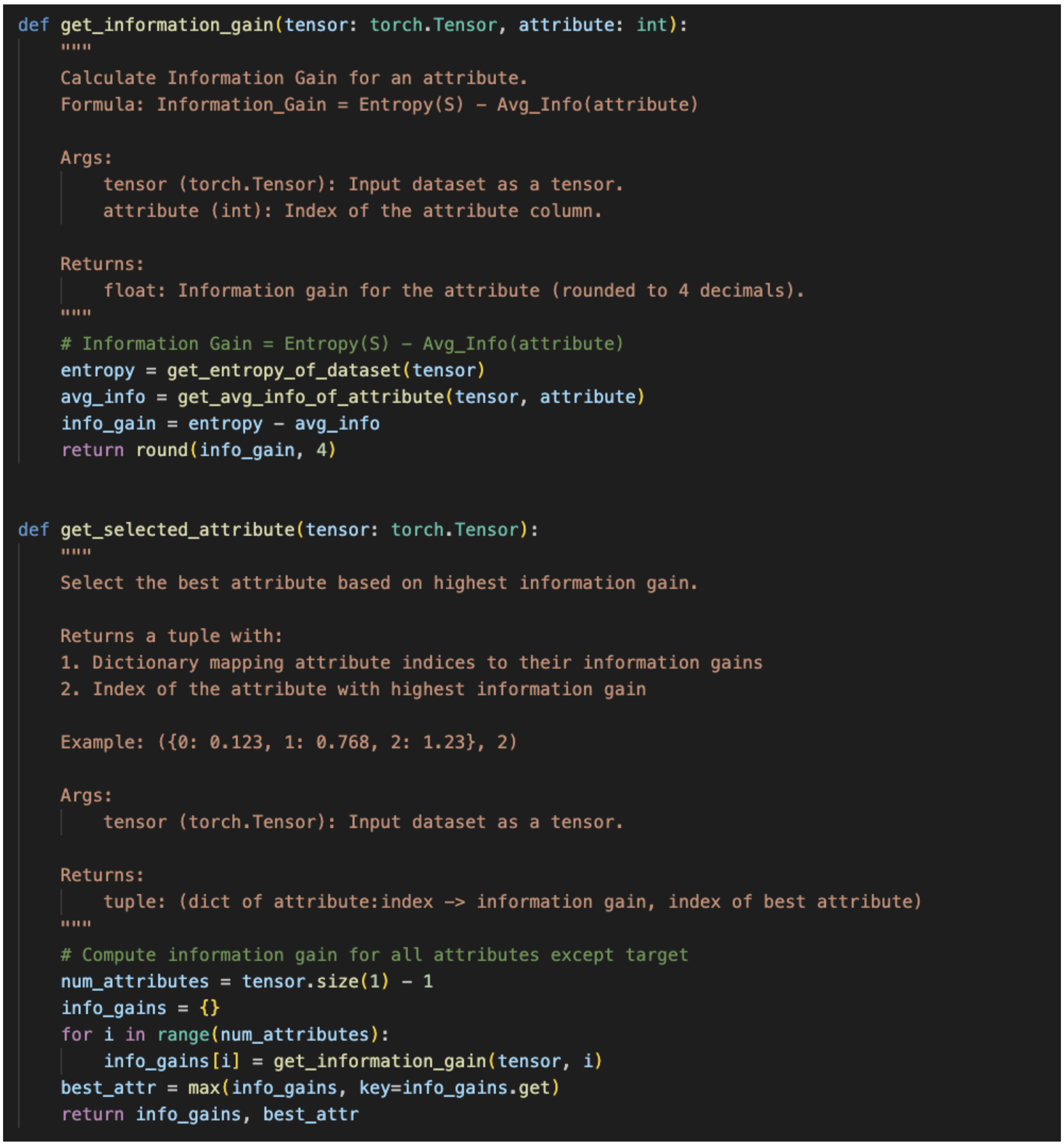
LAB 3: Decision Tree Classifier - Multi-Dataset Analysis

NAME: PREKSHA KAMALESH
SRN:PES2UG23CS902
SECTION:F

EC_F_PES2UG23CS902_Lab3.py

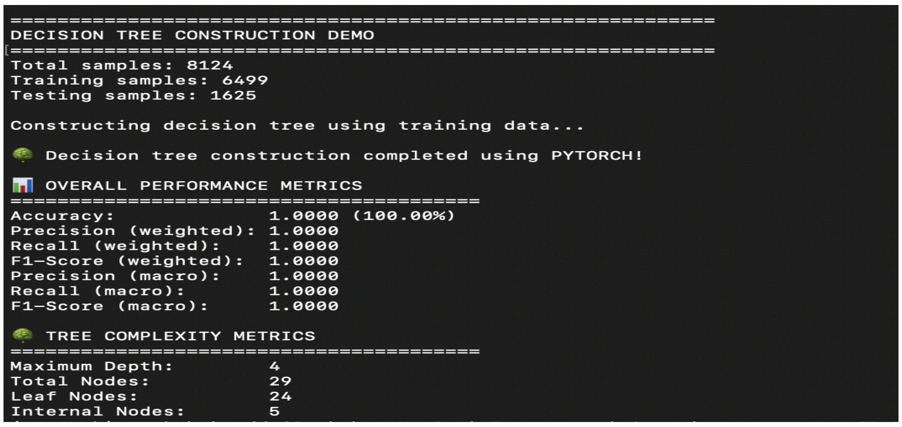


```python
import torch

def get_entropy_of_dataset(tensor: torch.Tensor):
    """
    Calculate the entropy of the entire dataset.
    Formula: Entropy = -Σ(p_i * log2(p_i)) where p_i is the probability of class i

    Args:
        tensor (torch.Tensor): Input dataset as a tensor, where the last column is the target.

    Returns:
        float: Entropy of the dataset.
    """
    # Calculate entropy of the target column
    target = tensor[:, -1]
    values, counts = torch.unique(target, return_counts=True)
    probs = counts.float() / target.size(0)
    entropy = -(probs * torch.log2(probs)).sum().item()
    return round(entropy, 4)


def get_avg_info_of_attribute(tensor: torch.Tensor, attribute: int):
    """
    Calculate the average information (weighted entropy) of an attribute.
    Formula: Avg_Info = Σ((|S_v|/|S|) * Entropy(S_v)) where S_v is subset with attribute value v.

    Args:
        tensor (torch.Tensor): Input dataset as a tensor.
        attribute (int): Index of the attribute column.

    Returns:
        float: Average information of the attribute.
    """
    # Calculate weighted average entropy for each value of the attribute
    total = tensor.size(0)
    values = torch.unique(tensor[:, attribute])
    avg_info = 0.0
    for v in values:
        subset = tensor[tensor[:, attribute] == v]
        if subset.size(0) == 0:
            continue
        entropy = get_entropy_of_dataset(subset)
        weight = subset.size(0) / total
        avg_info += weight * entropy
    return round(avg_info, 4)
```



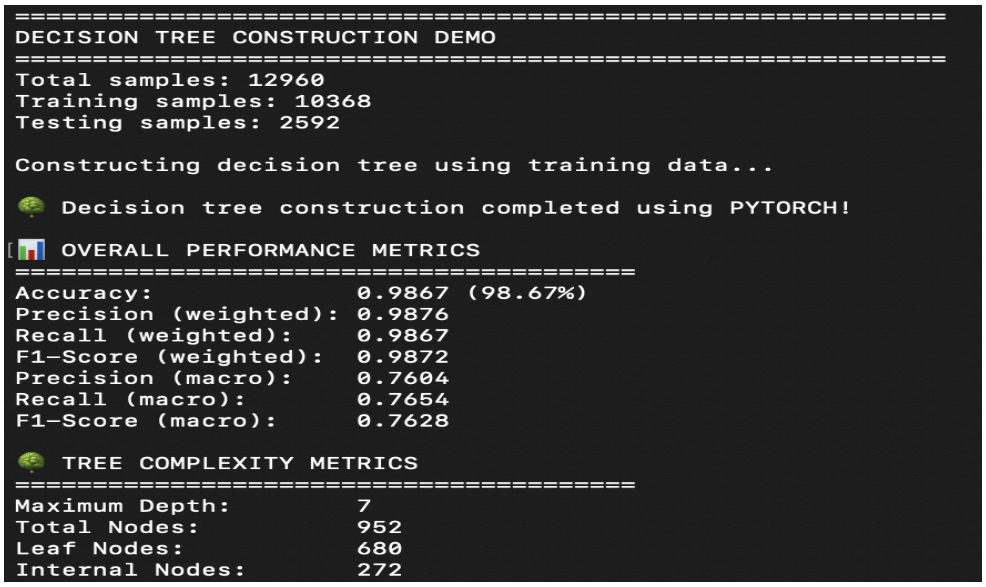
```python
def get_information_gain(tensor: torch.Tensor, attribute: int):
    """
    Calculate Information Gain for an attribute.
    Formula: Information_Gain = Entropy(S) - Avg_Info(attribute)

    Args:
        tensor (torch.Tensor): Input dataset as a tensor.
        attribute (int): Index of the attribute column.

    Returns:
        float: Information gain for the attribute (rounded to 4 decimals).
    """
    # Information Gain = Entropy(S) - Avg_Info(attribute)
    entropy = get_entropy_of_dataset(tensor)
    avg_info = get_avg_info_of_attribute(tensor, attribute)
    info_gain = entropy - avg_info
    return round(info_gain, 4)


def get_selected_attribute(tensor: torch.Tensor):
    """
    Select the best attribute based on highest information gain.

    Returns a tuple with:
    1. Dictionary mapping attribute indices to their information gains
    2. Index of the attribute with highest information gain

    Example: ({0: 0.123, 1: 0.768, 2: 1.23}, 2)

    Args:
        tensor (torch.Tensor): Input dataset as a tensor.

    Returns:
        tuple: (dict of attribute:index -> information gain, index of best attribute)
    """
    # Compute information gain for all attributes except target
    num_attributes = tensor.size(1) - 1
    info_gains = {}
    for i in range(num_attributes):
        info_gains[i] = get_information_gain(tensor, i)
    best_attr = max(info_gains, key=info_gains.get)
    return info_gains, best_attr
```

RESULTS
MUSHROOMS



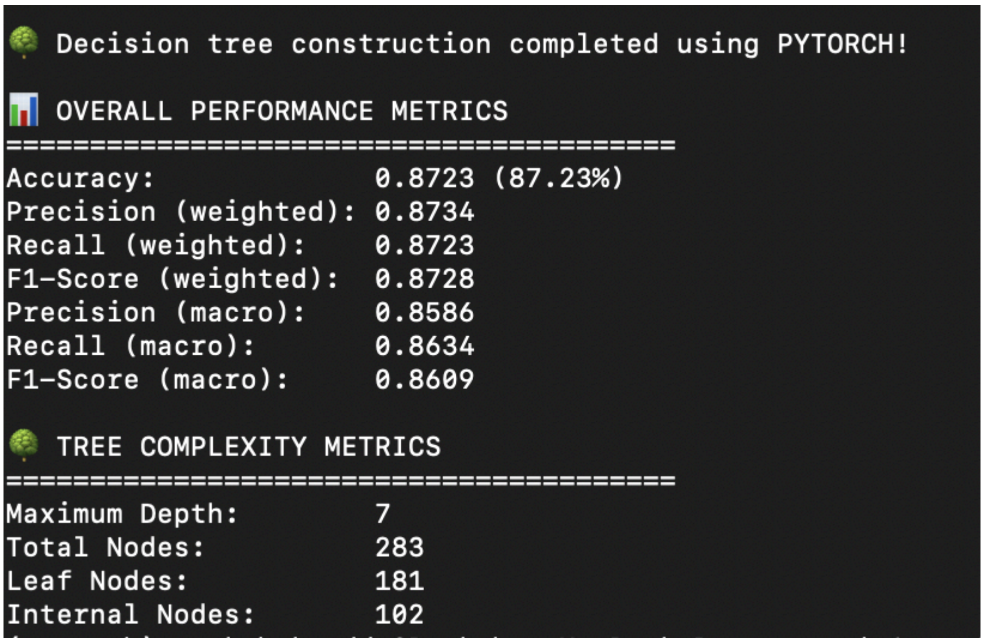
```
============================================================
DECISION TREE CONSTRUCTION DEMO
============================================================
Total samples: 8124
Training samples: 6499
Testing samples: 1625

Constructing decision tree using training data...

🌳 Decision tree construction completed using PYTORCH!

📊 OVERALL PERFORMANCE METRICS
========================================
Accuracy:             1.0000 (100.00%)
Precision (weighted): 1.0000
Recall (weighted):    1.0000
F1-Score (weighted):  1.0000
Precision (macro):    1.0000
Recall (macro):       1.0000
F1-Score (macro):     1.0000

🌳 TREE COMPLEXITY METRICS
========================================
Maximum Depth:        4
Total Nodes:          29
Leaf Nodes:           24
Internal Nodes:       5
```

NURSERY



```
============================================================
DECISION TREE CONSTRUCTION DEMO
============================================================
Total samples: 12960
Training samples: 10368
Testing samples: 2592

Constructing decision tree using training data...

🌳 Decision tree construction completed using PYTORCH!

📊 OVERALL PERFORMANCE METRICS
========================================
Accuracy:              0.9867 (98.67%)
Precision (weighted): 0.9876
Recall (weighted):    0.9867
F1-Score (weighted):  0.9872
Precision (macro):    0.7604
Recall (macro):       0.7654
F1-Score (macro):     0.7628

🌳 TREE COMPLEXITY METRICS
========================================
Maximum Depth:         7
Total Nodes:           952
Leaf Nodes:            680
Internal Nodes:        272
```

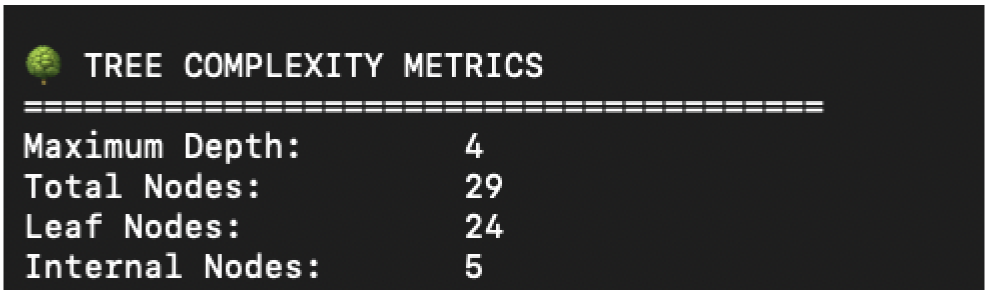
TICTACTOE



```
🌳 Decision tree construction completed using PYTORCH!

📊 OVERALL PERFORMANCE METRICS
========================================
Accuracy:              0.8723 (87.23%)
Precision (weighted): 0.8734
Recall (weighted):    0.8723
F1-Score (weighted):  0.8728
Precision (macro):    0.8586
Recall (macro):       0.8634
F1-Score (macro):     0.8609

🌳 TREE COMPLEXITY METRICS
========================================
Maximum Depth:         7
Total Nodes:           283
Leaf Nodes:            181
Internal Nodes:        102
```

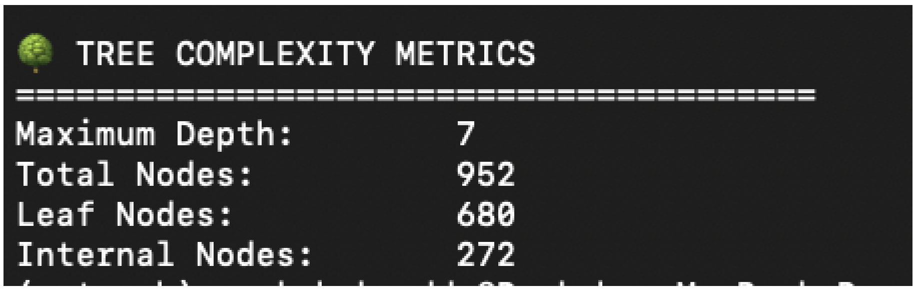
TREE

MUSHROOM



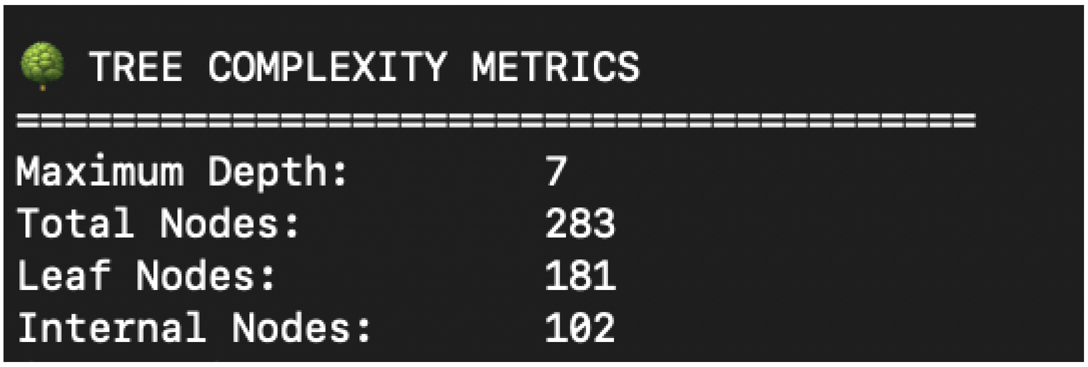
```
🌳 TREE COMPLEXITY METRICS
========================================
Maximum Depth:         4
Total Nodes:           29
Leaf Nodes:            24
Internal Nodes:        5
```

NURSERY



```
🌳 TREE COMPLEXITY METRICS
========================================
Maximum Depth:        7
Total Nodes:          952
Leaf Nodes:           680
Internal Nodes:       272
```

TICTACTOE



```
🌳 TREE COMPLEXITY METRICS
========================================
Maximum Depth:        7
Total Nodes:          283
Leaf Nodes:           181
Internal Nodes:       102
```

ANALYSIS REQUIREMENTS:

1. Performance Comparison

Dataset	Accuracy	Precision (Weighted)	Recall (Weighted)	F1-Score (Weighted)	F1-Score (Macro)
Mushroom	100%	1.0000	1.0000	1.0000	1.0000
Nursery	98.67%	0.9876	0.9867	0.9872	0.7628
Tic-Tac-Toe	87.23%	0.8734	0.8723	0.8728	0.8609

The Mushroom dataset achieved a perfect accuracy of 100%. This indicates that the features in this dataset are highly discriminative, allowing the algorithm to find clear rules to perfectly separate edible from poisonous mushrooms. The Nursery dataset also performed exceptionally well with 98.67% accuracy. The Tic-Tac-Toe dataset had the lowest accuracy at 87.23%, suggesting that the board state alone may not always be sufficient to perfectly predict the game's outcome, or the relationships are more complex.

2.Tree Characteristics Analysis

Dataset	Tree Depth	Total Nodes	Leaf Nodes	Root Node (Most Important Feature)
Mushroom	4	29	24	odor
Nursery	7	952	680	health
Tic-Tac-Toe	7	283	181	middle-middle-square

The Mushroom dataset, despite having the most features (22), produced the simplest tree. This is because the odor feature is an extremely strong predictor, creating a very pure split at the root and simplifying the rest of the tree.
The Nursery dataset is the largest in terms of samples and resulted in the most complex tree (952 nodes). Its features (health, has_nurs, etc.) are predictive, but require many combinations of splits to resolve the 5 target classes, leading to a deep and wide tree.
The Tic-Tac-Toe dataset, being the smallest, still produced a relatively complex tree with a depth of 7. This reflects the combinatorial nature of the game, where multiple board positions must be considered to make a decision.

---

3. Dataset-Specific Insights

Mushroom Dataset

- Feature Importance: The odor attribute was selected as the root, indicating it's the single most important feature for distinguishing poisonous from edible mushrooms. Other important features appearing in early splits include
  spore-print-color and habitat.
- Class Distribution: The binary classification (edible vs. poisonous) and perfect macro F1-score suggest the classes are well-represented and balanced.
- Decision Patterns: A very simple and powerful decision path is evident: if the odor has certain values (e.g., 1, 2, 4, 6, 7, 8), the mushroom is immediately classified as poisonous (Class 1). This interpretability is a key strength.
- Overfitting Indicators: The low tree depth (4) and small number of nodes (29) for a dataset with 8124 samples strongly suggest the model is not overfitting and has found general, powerful rules.

Nursery Dataset

- Feature Importance: The health of the applicant was the most critical feature, chosen as the root node. This was followed by features like
  has_nurs (nursery school history) and parents' occupation, showing a clear hierarchy of admission criteria.
- Class Distribution: The large difference between the weighted F1-score (0.9872) and the macro F1-score (0.7628) indicates a significant class imbalance. Some recommendation classes (e.g.,

very_recom, spec_prior) are likely much rarer than others (not_recom), making them harder for the model to predict correctly.
- Decision Patterns: The tree shows that a child with health rated as "not recommended" (value 0) is immediately classified as not_recom (Class 0). This reveals a strict rule in the admission logic.
- Overfitting Indicators: With 952 nodes for ~10k training samples, this tree is extremely complex. This high complexity is a potential sign of overfitting, where the model may be memorizing the training data rather than learning general patterns. Pruning could simplify the tree and improve its generalizability.

Tic-Tac-Toe Dataset

- Feature Importance: The middle-middle-square was chosen as the root node, which aligns with game strategy, as controlling the center is crucial in Tic-Tac-Toe. Other corner and side positions appear in subsequent splits.
- Class Distribution: The weighted (0.8728) and macro (0.8609) F1-scores are very close, suggesting the classes (positive vs. negative) are relatively balanced.
- Decision Patterns: The tree learns strategic patterns. For instance, if the middle square is taken by 'x' (value = 2), the tree then checks other squares to determine if a winning move is present.
- Overfitting Indicators: The tree is quite deep (7) and has 283 nodes for a small training set of 766 samples. This suggests a high risk of overfitting, as the tree might be learning very specific board configurations that don't generalize well.

4.Comparative Analysis Report

a) Algorithm Performance

- Highest Accuracy: The Mushroom dataset achieved the highest accuracy (100%). This is because it contains features like odor that have a very high information gain and can almost perfectly classify the target on their own. The relationship between the features and the target is very strong and direct.
- Dataset Size: Dataset size did not directly correlate with higher accuracy. The Nursery dataset, the largest by far, did not achieve a perfect score, while the medium-sized Mushroom dataset did. However, dataset size did correlate with tree complexity, with the largest dataset producing the most complex tree.
- Number of Features: The number of features also did not directly predict performance. The Mushroom dataset had 22 features and performed perfectly, while the Tic-Tac-Toe dataset had only 9 features and had the lowest performance. Performance is more dependent on the predictive power of the features rather than their sheer number.

b) Data Characteristics Impact

- Class Imbalance: The effect of class imbalance is clearly visible in the Nursery dataset. While the overall (weighted) accuracy is high, the low macro F1-score shows that the model struggles with the minority classes. The decision tree, driven by information gain, tends to favor the majority class, and may not build sufficient branches to correctly identify rare classes.

- Feature Types: All datasets contained multi-valued categorical features. The ID3 algorithm works well with such features, but it has a known bias for attributes with more values. In the Mushroom dataset, the odor feature, being highly predictive, worked exceptionally well. Binary features might create simpler splits, but well-chosen multi-valued features can capture more information in a single split.

c) Practical Applications

- Real-World Scenarios:
    - Mushroom: Ideal for safety-critical classification systems, such as identifying toxic plants or unsafe food items, where clear rules are needed.
    - Nursery: Relevant for application screening processes like college admissions, loan applications, or social service eligibility, where decisions are based on a hierarchy of criteria.
    - Tic-Tac-Toe: Represents problems in game theory and strategic decision-making, applicable to building AI for simple board games or analyzing rule-based systems.
- Interpretability: Decision trees offer high interpretability ("white-box" models). For mushroom foraging, a simple rule like "If the mushroom has a foul odor, it is poisonous" is easy for a human to understand and trust. In the nursery domain, the tree makes the school's admission priorities transparent.
- Performance Improvement:
    - Mushroom: Performance is already perfect, so no improvement is needed.
    - Nursery & Tic-Tac-Toe: The primary issue is potential overfitting due to high tree complexity. Performance could be improved by implementing pruning (pre- or post-pruning) to reduce the tree size and improve generalization. Using an ensemble method like a Random Forest would also likely boost performance and reduce variance.
