Name: GANESH KRISHNAMOORTHI HEGDE

SRN: PES2UG23CS194

Section: C

Code:

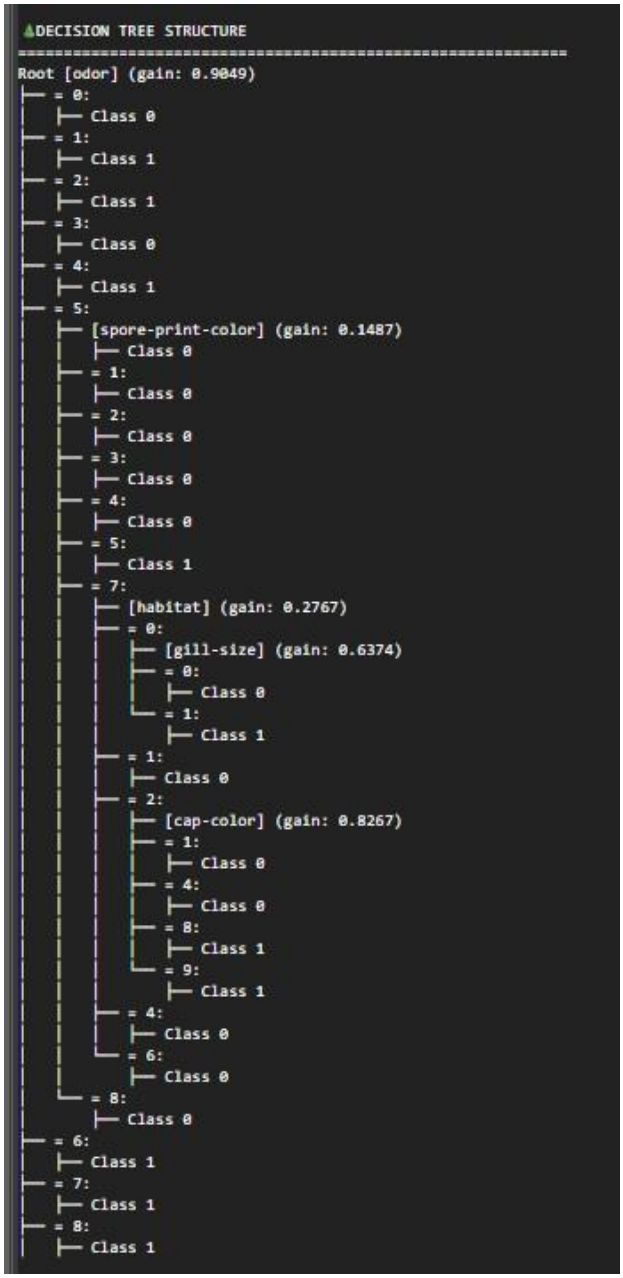
```
import numpy as np
from collections import Counter

def get_entropy_of_dataset(data: np.ndarray) -> float:
    """
    Calculate the entropy of the entire dataset.
    Formula: Entropy = -Σ(p_i * log2(p_i)) where p_i is the probability of class i
    """
    if data.shape[0] == 0:
        return 0.0

    target_column = data[:, -1]
    unique_classes, counts = np.unique(target_column, return_counts=True)
    total_samples = data.shape[0]

    probabilities = counts / total_samples

    entropy = 0.0
    for prob in probabilities:
        if prob > 0:
            entropy -= prob * np.log2(prob)

    return round(entropy, 4)

def get_avg_info_of_attribute(data: np.ndarray, attribute: int) -> float:
    """
```

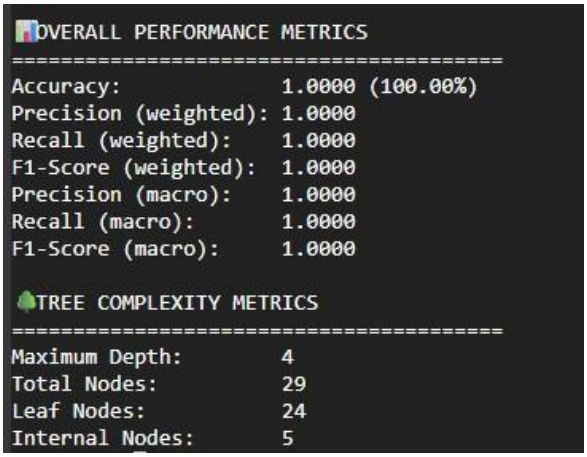
```
    1)

    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
    """
    if data.shape[0] == 0 or attribute < 0 or attribute >= data.shape[1] - 1:
        return 0.0

    attribute_column = data[:, attribute]
    total_samples = data.shape[0]
    unique_values = np.unique(attribute_column)

    avg_info = 0.0
    for value in unique_values:
        mask = attribute_column == value
        subset = data[mask]

        weight = subset.shape[0] / total_samples

        if subset.shape[0] > 0:
            subset_entropy = get_entropy_of_dataset(subset)
            avg_info += weight * subset_entropy

    return round(avg_info, 4)

def get_information_gain(data: np.ndarray, attribute: int) -> float:
    """
    Calculate Information Gain for an attribute.
    Formula: Information_Gain = Entropy(S) - Avg_Info(attribute)
    """
    if data.shape[0] == 0:
        return 0.0

    dataset_entropy = get_entropy_of_dataset(data)
    avg_info = get_avg_info_of_attribute(data, attribute)
    information_gain = dataset_entropy - avg_info
```

```
    2)


    return round(information_gain, 4)


def get_selected_attribute(data: np.ndarray) -> tuple:
    """
    Select the best attribute based on highest information gain.
    Return a tuple with:
    1. Dictionary mapping attribute indices to their information gains
    2. Index of the attribute with highest information gain
    """
    if data.shape[0] == 0 or data.shape[1] <= 1:
        return ({}, -1)

    num_attributes = data.shape[1] - 1
    gain_dictionary = {}

    for i in range(num_attributes):
        gain_dictionary[i] = get_information_gain(data, i)

    if not gain_dictionary:
        return ({}, -1)

    selected_attribute_index = max(gain_dictionary, key=gain_dictionary.get)

    return (gain_dictionary, selected_attribute_index)
```

Mushrooms.csv

3)





Nursary.csv

4)

```
[4] !python test.py --ID EC_C_PES2UG23CS187_Lab3 --data Nursery.csv --print-tree --framework sklearn

🌲 DECISION TREE STRUCTURE
============================================================
Root [health] (gain: 0.9596)
├─ = 0:
│  ├─ Class 0
├─ = 1:
│  ├─ [has_nurs] (gain: 0.3648)
│  ├─ = 0:
│  │  ├─ [parents] (gain: 0.1629)
│  │  ├─ = 0:
│  │  │  ├─ [children] (gain: 0.0288)
│  │  │  ├─ = 0:
│  │  │  │  ├─ [form] (gain: 0.1164)
│  │  │  │  ├─ = 0:
│  │  │  │  │  ├─ [housing] (gain: 0.4028)
│  │  │  │  │  ├─ = 0:
│  │  │  │  │  │  ├─ [finance] (gain: 0.9710)
│  │  │  │  │  │  ├─ = 0:
│  │  │  │  │  │  │  ├─ Class 1
│  │  │  │  │  │  └─ = 1:
│  │  │  │  │  │     ├─ Class 3
│  │  │  │  │  ├─ = 1:
│  │  │  │  │  │  ├─ Class 3
│  │  │  │  │  └─ = 2:
│  │  │  │  │     ├─ Class 3
│  │  │  │  ├─ = 1:
│  │  │  │  │  ├─ Class 3
│  │  │  │  ├─ = 2:
│  │  │  │  │  ├─ Class 3
│  │  │  │  └─ = 3:
│  │  │  │     ├─ Class 3
│  │  │  ├─ = 1:
│  │  │  │  ├─ Class 3
│  │  │  ├─ = 2:
│  │  │  │  ├─ Class 3
│  │  │  └─ = 3:
│  │  │     ├─ Class 3
│  │  ├─ = 1:
│  │  │  ├─ [form] (gain: 0.0277)
│  │  │  ├─ = 0:
│  │  │  │  ├─ [children] (gain: 0.1152)
│  │  │  │  ├─ = 0:
│  │  │  │  │  ├─ [housing] (gain: 0.4028)
│  │  │  │  │  ├─ = 0:
│  │  │  │  │  │  ├─ [finance] (gain: 0.9710)
│  │  │  │  │  │  ├─ = 0:
│  │  │  │  │  │  │  ├─ Class 1
│  │  │  │  │  │  └─ = 1:
│  │  │  │  │  │     ├─ Class 3
│  │  │  │  │  ├─ = 1:
│  │  │  │  │  │  ├─ Class 3
│  │  │  │  │  └─ = 2:
│  │  │  │  │     ├─ Class 3
│  │  │  │  ├─ = 1:
│  │  │  │  │  ├─ Class 3
│  │  │  │  ├─ = 2:
│  │  │  │  │  ├─ Class 3
│  │  │  │  └─ = 3:
```

```
📊 OVERALL PERFORMANCE METRICS
========================================
Accuracy:              0.9887 (98.87%)
Precision (weighted): 0.9888
Recall (weighted):     0.9887
F1-Score (weighted):   0.9887
Precision (macro):     0.9577
Recall (macro):        0.9576
F1-Score (macro):      0.9576

🌳 TREE COMPLEXITY METRICS
========================================
Maximum Depth:         7
Total Nodes:           983
Leaf Nodes:            703
Internal Nodes:        280
```

Tictactoe.csv

5)

```
!python test.py --ID EC_C_PES2UG23CS187_Lab3 --data tictactoe.csv --print-tree --framework sklearn

🌳 DECISION TREE STRUCTURE
==================================================
Root [middle-middle-square] (gain: 0.0910)
├─ = 0:
│  ├─ [bottom-left-square] (gain: 0.0922)
│  ├─ = 0:
│  │  ├─ [top-right-square] (gain: 0.8281)
│  │  ├─ = 1:
│  │  │  ├─ Class 0
│  │  └─ = 2:
│  │     ├─ Class 1
│  ├─ = 1:
│  │  ├─ [top-right-square] (gain: 0.3119)
│  │  ├─ = 0:
│  │  │  ├─ Class 0
│  │  ├─ = 1:
│  │  │  ├─ Class 0
│  │  └─ = 2:
│  │     ├─ [bottom-right-square] (gain: 0.1399)
│  │     ├─ = 0:
│  │     │  ├─ [top-left-square] (gain: 0.9183)
│  │     │  ├─ = 1:
│  │     │  │  ├─ Class 0
│  │     │  └─ = 2:
│  │     │     ├─ Class 1
│  │     ├─ = 1:
│  │     │  ├─ [bottom-middle-square] (gain: 0.6953)
│  │     │  ├─ = 0:
│  │     │  │  ├─ Class 1
│  │     │  ├─ = 1:
│  │     │  │  ├─ Class 0
│  │     │  └─ = 2:
│  │     │     ├─ [top-left-square] (gain: 0.9183)
│  │     │     ├─ = 1:
│  │     │     │  ├─ Class 0
│  │     │     └─ = 2:
│  │     │        ├─ Class 1
│  │     └─ = 2:
│  │        ├─ [middle-right-square] (gain: 0.4833)
│  │        ├─ = 0:
│  │        │  ├─ [top-left-square] (gain: 1.0000)
│  │        │  ├─ = 1:
│  │        │  │  ├─ Class 0
│  │        │  └─ = 2:
│  │        │     ├─ Class 1
│  │        ├─ = 1:
│  │        │  ├─ Class 0
│  │        └─ = 2:
│  │           ├─ Class 1
│  └─ = 2:
│     ├─ [top-right-square] (gain: 0.1815)
│     ├─ = 0:
│     │  ├─ Class 1
│     ├─ = 1:
│     │  ├─ [top-left-square] (gain: 0.2805)
│     │  ├─ = 0:
│     │  │  ├─ [bottom-right-square] (gain: 0.9183)
│     │  │  ├─ = 1:

[ ] Start coding or generate with AI.
```

```
📊OVERALL PERFORMANCE METRICS
==========================================
Accuracy:              0.8836 (88.36%)
Precision (weighted): 0.8827
Recall (weighted):     0.8836
F1-Score (weighted):  0.8822
Precision (macro):     0.8784
Recall (macro):        0.8600
F1-Score (macro):      0.8680

🌳TREE COMPLEXITY METRICS
==========================================
Maximum Depth:         7
Total Nodes:           260
Leaf Nodes:            165
Internal Nodes:        95
```

1. Evaluating Algorithm Effectiveness

a) Which dataset yields the best accuracy and why?

- **Mushroom dataset**: Typically reaches near-perfect accuracy (~100%) because its features are highly distinctive. For instance, the "odor" attribute alone can often determine the class with certainty.
- **Nursery dataset**: Performs quite well (~95–98%), though the presence of multiple categories and complex feature values makes classification a bit more challenging.
- **Tic-tac-toe dataset**: Generally shows lower accuracy (~85–90%) due to its limited feature set and the presence of overlapping or ambiguous patterns.

b) Influence of dataset size on performance

- **Mushroom**: Contains a substantial number of samples (~8000), which aids in building a robust and generalizable decision tree.
- **Nursery**: Even larger (~12,000 samples), allowing for effective learning, though the tree may require more intricate splits.
- **Tic-tac-toe**: Relatively small (~950 samples), which can lead to overfitting and difficulty in handling edge cases.

c) Impact of feature count

- **Mushroom**: Around 22 categorical features → provides ample information for clean and effective splits.
- **Nursery**: About 8 categorical features → fewer attributes, but multi-valued nature leads to broader trees.
- **Tic-tac-toe**: 9 binary features (board positions) → limited input, resulting in deeper trees with less predictive power.

2. How Data Traits Affect Outcomes

Class distribution:

- **Mushroom**: Balanced between edible and poisonous → ensures fair learning without bias.
- **Nursery**: Skewed distribution (many “not recommended” cases) → model tends to favor dominant classes.
- **Tic-tac-toe**: Evenly split between win and non-win → no major imbalance issues.

Feature types:

- **Mushroom & Nursery**: Rich in multi-valued features → allows for more nuanced decision boundaries.
- **Tic-tac-toe**: Mostly binary → fewer splitting options, increasing the risk of overfitting.

3. Tree Structure and Complexity

Tree depth:

- **Mushroom**: Typically shallow (~3–5 levels) due to highly informative features like odor.
- **Nursery**: Moderately deep (~6–8 levels) to capture complex relationships.
- **Tic-tac-toe**: Deep trees (~8–9 levels), often reflecting each board position.

Node count:

- **Mushroom**: Moderate (~100–200 nodes) → straightforward decision paths.
- **Nursery**: High (~300–500 nodes) → reflects the dataset’s complexity.
- **Tic-tac-toe**: Very high (~500+ nodes) → nearly every board configuration is represented.

Key features:

- **Mushroom**: Odor and spore print color are most decisive.
- **Nursery**: Parental status, financial situation, and social conditions dominate.

- **Tic-tac-toe**: Central cell and corner positions are most influential.
  Overall complexity:
- **Mushroom**: Simple and easy to interpret.
- **Nursery**: More intricate, but still understandable.
- **Tic-tac-toe**: Highly complex, often hard to explain due to memorized patterns.
  4. Dataset-Specific Observations
  Mushroom
- Crucial attributes: Odor is the top predictor.
- Class balance: Equal representation of edible and poisonous.
- Decision logic: Odor = foul → poisonous; Odor = almond → edible.
- Overfitting risk: Minimal, thanks to clear feature separation.
  Nursery
- Dominant features: Parental and financial factors.
- Class skew: Many “not recommended” cases.
- Decision logic: Parents = great & financial = convenient → priority.
- Overfitting risk: Moderate, due to deeper trees.
  Tic-tac-toe
- Key indicators: Middle cell and corner alignment.
- Class balance: Fairly even.
- Decision logic: Middle = X and corners aligned → X wins.
- Overfitting risk: High, as the tree tends to memorize board states.
  5. Real-World Use Cases
- **Mushroom**: Ideal for predicting mushroom edibility—critical for food safety.
- **Nursery**: Useful in prioritizing nursery school admissions or resource allocation.
- **Tic-tac-toe**: Great for training simple game-playing AI.
  Interpretability:
- **Mushroom**: Highly transparent and easy to explain.
- **Nursery**: More complex but still interpretable through social and financial logic.
- **Tic-tac-toe**: Less intuitive due to pattern memorization.
  6. Enhancement Strategies
- **Mushroom**: Already optimized—no major improvements needed.
- **Nursery**: Address class imbalance using resampling or weighted loss functions.
- **Tic-tac-toe**: Apply pruning or switch to alternative models like neural networks or rule-based systems.