Enver Eren 2019402189

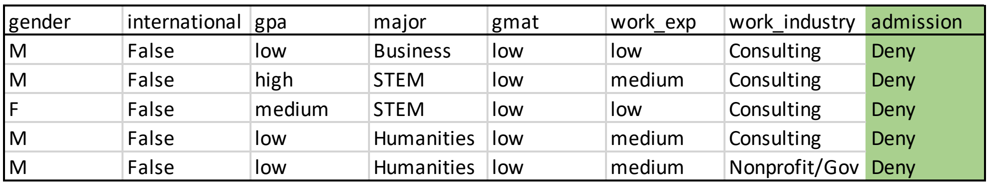
Homework 4

Dataset Information:

MBA Admissions Dataset, from Kaggle: Poornapradnya21/mba-csv
This dataset contains profiles of MBA program applicants, including their academic background, test scores, and whether or not they were admitted. I used about 500 clean rows as data.

After quantile-based discretization (so that all attributes are categorical), I used the following columns from the dataset:
1. **gender** - Gender of the applicant
2. **international** - Boolean indicating if the student is international
3. **gpa** - Applicant’s GPA category (discretizated)
4. **major** - Applicant’s undergraduate major
5. **gmat** - GMAT score category (discretizated)
6. **work_exp** - Work experience category (discretizated)
7. **work_industry** - Applicant’s industry
8. **admission** – Target variable. (Admit, Waitlist, Deny)

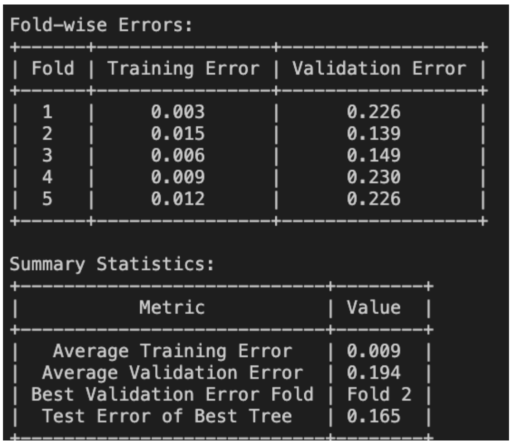
Head of the dataset:



gender	international	gpa	major	gmat	work_exp	work_industry	admission
M	False	low	Business	low	low	Consulting	Deny
M	False	high	STEM	low	medium	Consulting	Deny
F	False	medium	STEM	low	low	Consulting	Deny
M	False	low	Humanities	low	medium	Consulting	Deny
M	False	low	Humanities	low	medium	Nonprofit/Gov	Deny

5-Fold Cross-Validation:

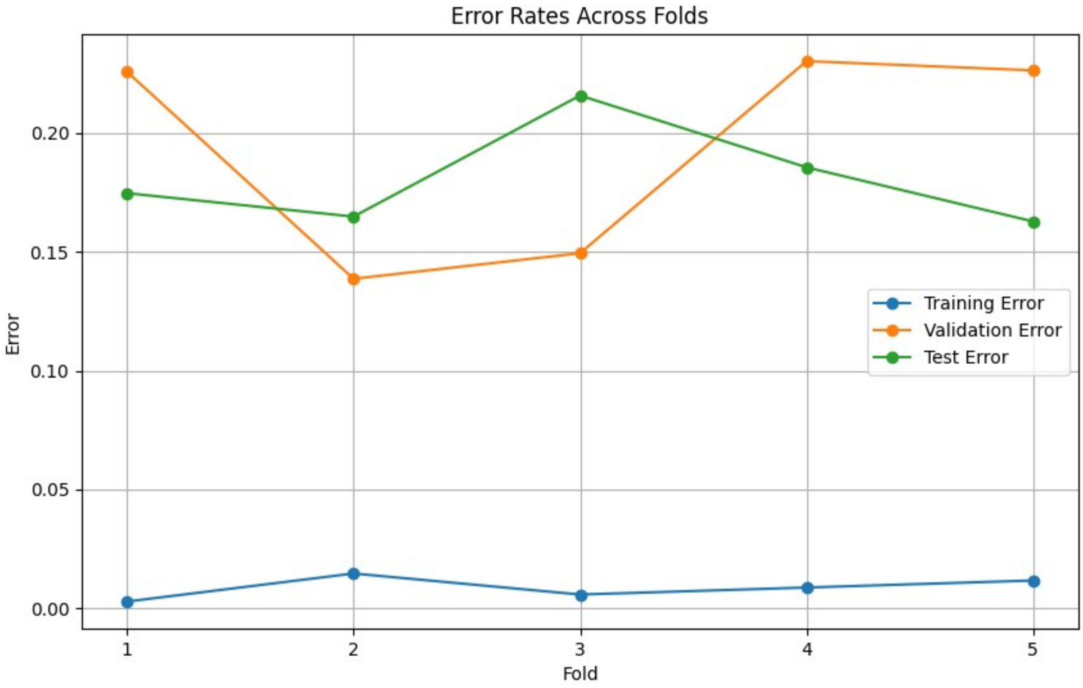
To evaluate the performance of the decision tree, I used 5-fold cross-validation on the training data. The procedure was as follows:
1. Split the dataset into training (80%) and testing (20%).
2. Divide the training portion into 5 folds (subsets) of equal size.
3. For each fold i:
    - Validation set: the ith fold
    - Training set: the remaining 4 folds combined
    - Train a decision tree model on the 4 folds
    - Evaluate on the validation fold, and record the validation error
4. Average the training errors and validation errors across all folds.
5. Choose the best model based on the lowest validation error.
6. Test the best tree (model) with the test data



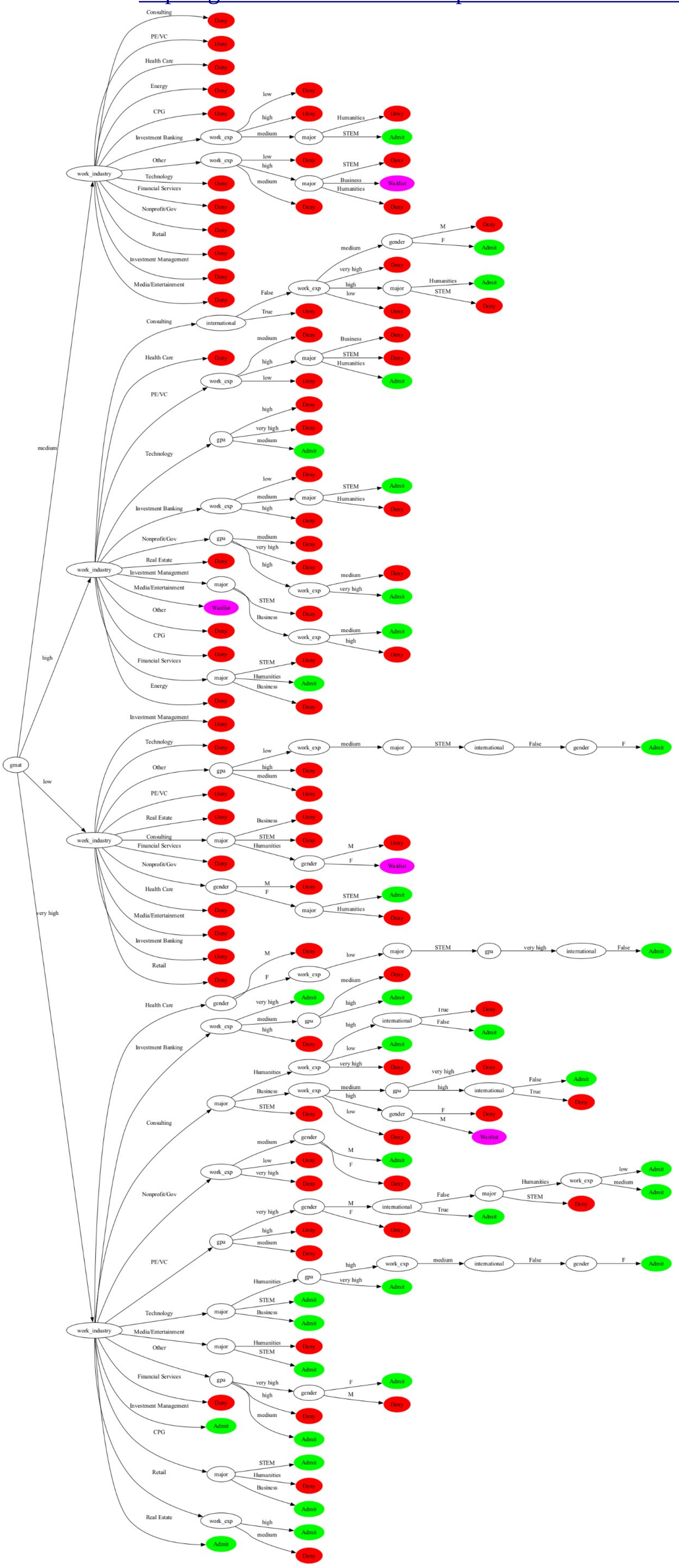
```
Fold-wise Errors:
+------+----------------+------------------+
| Fold | Training Error | Validation Error |
+------+----------------+------------------+
|  1   |     0.003      |      0.226       |
|  2   |     0.015      |      0.139       |
|  3   |     0.006      |      0.149       |
|  4   |     0.009      |      0.230       |
|  5   |     0.012      |      0.226       |
+------+----------------+------------------+

Summary Statistics:
+---------------------------+--------+
|           Metric          | Value  |
+---------------------------+--------+
|   Average Training Error  | 0.009  |
|  Average Validation Error | 0.194  |
| Best Validation Error Fold| Fold 2 |
|  Test Error of Best Tree  | 0.165  |
+---------------------------+--------+
```

Five-fold cross-validation is a robust method for assessing how well a model generalizes. It splits the training data into five subsets, using one subset as a validation set while training on the remaining four. This process repeats so that each subset serves as the validation set exactly once, and its average error becomes a more stable estimate of true performance than a single training-validation split. By cycling through all subsets, every example is used for both training and validation, thereby reducing overfitting and variance in error estimates. This approach is especially beneficial for smaller datasets and model selection, since it provides multiple reliable evaluations without sacrificing much data for validation. After identifying the best-performing fold, we then test that model's performance on the held-out test set, adding an extra layer of confirmation that our chosen model generalizes well.

Error Plots:

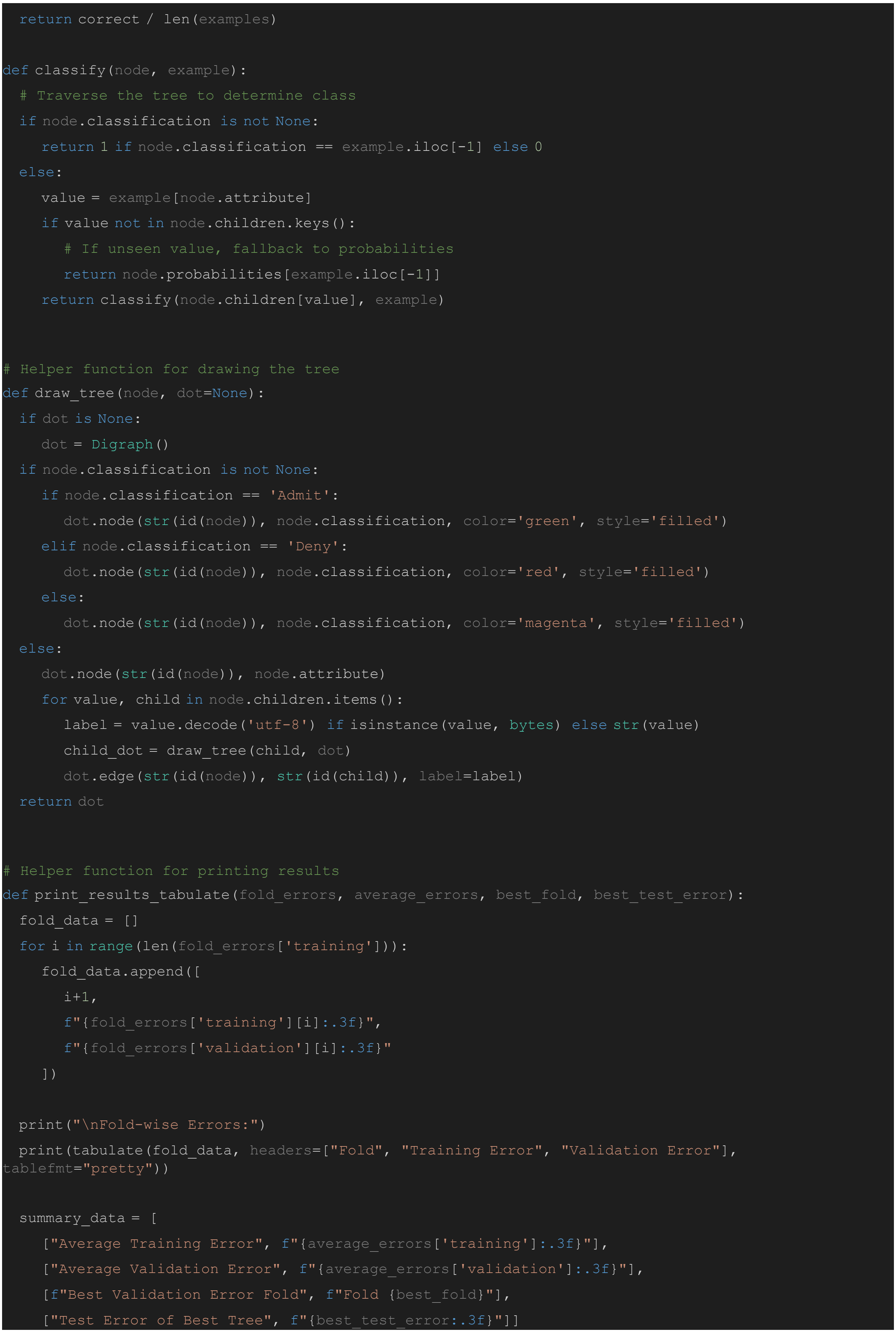


Final/Best Decision Tree: https://github.com/enverern/Cmpe480/blob/main/HW4/final_tree.png

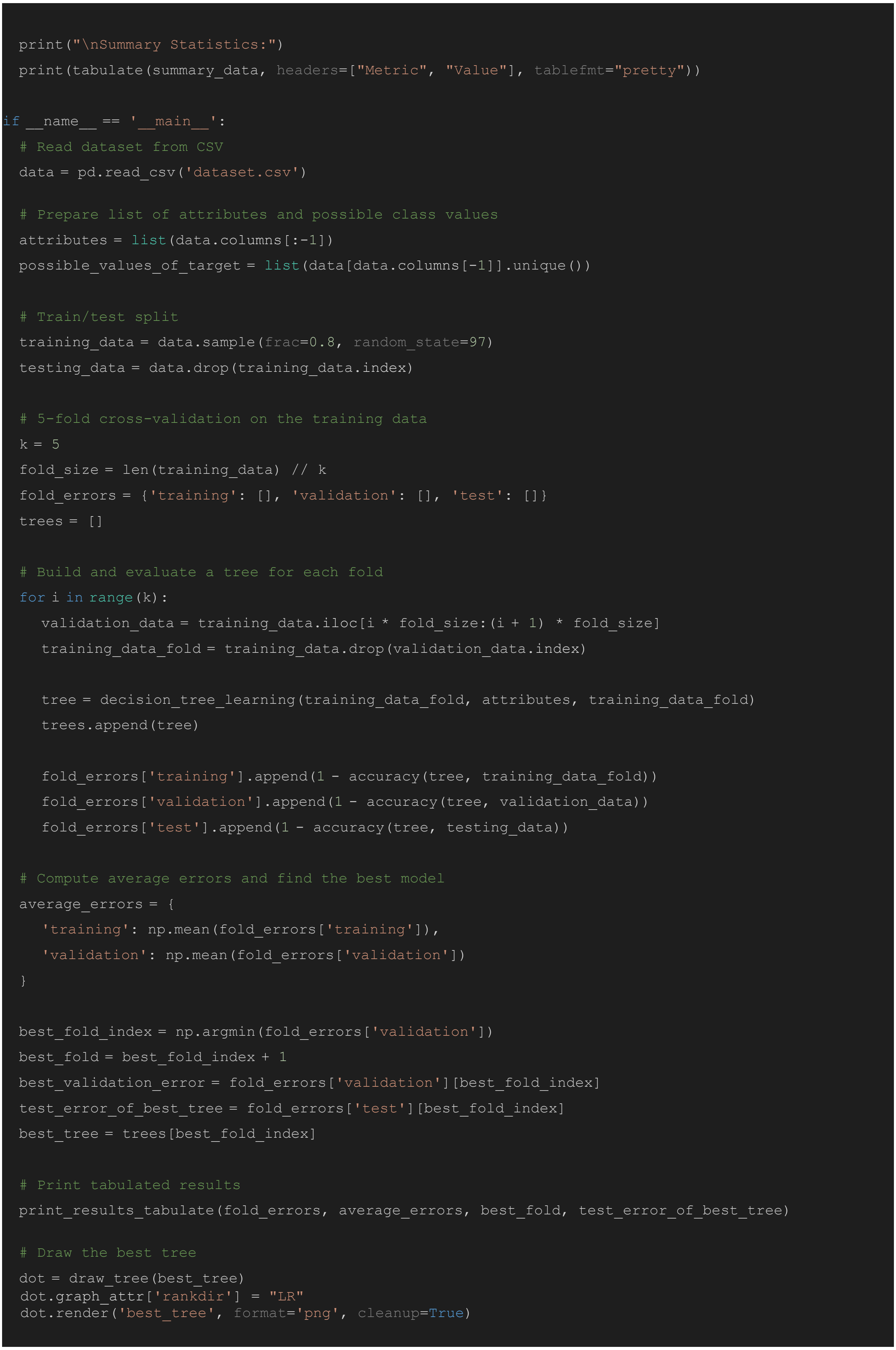


Source Code with Comments:

```python
import pandas as pd
import numpy as np
import math
import matplotlib.pyplot as plt
from graphviz import Digraph
from tabulate import tabulate

# Node in the decision tree
class Node:
  def __init__(self, attribute=None, value=None, classification=None, probabilities=None):
    self.attribute = attribute   # Which attribute to split on
    self.value = value        # Value of the parent's attribute
    self.children = {}             # Subtrees for each attribute value
    self.classification = classification # Leaf node class (None if non-leaf)
    self.probabilities = probabilities   # Class distribution at this node

  def add_child(self, value, node):
    # Attach a child node for a specific attribute value
    self.children[value] = node

def decision_tree_learning(examples, attributes, parent_examples):
  # If no data, use parent's majority class
  if len(examples) == 0:
    return plurality_value(parent_examples)
  # If all examples share the same class, create a leaf
  elif all_same_classification(examples):
    return Node(classification=examples[examples.columns[-1]].iloc[0],
            probabilities=get_probabilities(examples))
  # If no attributes left, use majority class of current subset
  elif len(attributes) == 0:
    return plurality_value(examples)
  else:
    # Choose best attribute to split on (highest info gain)
    A = argmax_importance(attributes, examples)
    index_of_A = attributes.index(A)
    new_attributes_list = attributes[:index_of_A] + attributes[index_of_A+1:]

    # Create root node for this branch
    tree = Node(attribute=A, probabilities=get_probabilities(examples))

    # Split examples on each value of the chosen attribute
    for value in examples[A].unique():
      exs = examples[examples[A] == value]
      subtree = decision_tree_learning(exs, new_attributes_list, examples)
      tree.add_child(value, subtree)
    return tree

def plurality_value(examples):
  # Return a node with the most common class
  return Node(classification=examples[examples.columns[-1]].mode()[0],
         probabilities=get_probabilities(examples))

def get_probabilities(examples):
  # Compute class probabilities at the node
  probabilities = examples[examples.columns[-1]].value_counts(normalize=True).to_dict()
  for key in possible_values_of_target:
    if key not in probabilities.keys():
      probabilities[key] = 0
  return probabilities

def all_same_classification(examples):
  # Check if all examples belong to one class
  return len(examples[examples.columns[-1]].unique()) == 1

def argmax_importance(attributes, examples):
  if len(attributes) == 1:
    return attributes[0]
  max_importance = 0
  max_attribute = None
  for attribute in attributes:
    importance = importance_of_attribute(attribute, examples)
    if importance >= max_importance:
      max_importance = importance
      max_attribute = attribute
  return max_attribute

def importance_of_attribute(attribute, examples):
  # Info gain = Entropy - Weighted child entropies
  return entropy(examples) - remainder(attribute, examples)

def entropy(examples):
  # Compute entropy of given examples
  e = 0
  for value in examples[examples.columns[-1]].unique():
    p = len(examples[examples[examples.columns[-1]] == value]) / len(examples)
    e -= p * math.log2(p)
  return e

def remainder(attribute, examples):
  # Weighted sum of entropies for each value
  r = 0
  for value in examples[attribute].unique():
    exs = examples[examples[attribute] == value]
    r += (len(exs) / len(examples)) * entropy(exs)
  return r

def accuracy(tree, examples):
  # Proportion of examples correctly classified by the tree
  correct = 0
  for i in range(len(examples)):
    result = classify(tree, examples.iloc[i])
    correct += result
```



```
  return correct / len(examples)

def classify(node, example):
  # Traverse the tree to determine class
  if node.classification is not None:
    return 1 if node.classification == example.iloc[-1] else 0
  else:
    value = example[node.attribute]
    if value not in node.children.keys():
      # If unseen value, fallback to probabilities
      return node.probabilities[example.iloc[-1]]
    return classify(node.children[value], example)


# Helper function for drawing the tree
def draw_tree(node, dot=None):
  if dot is None:
    dot = Digraph()
  if node.classification is not None:
    if node.classification == 'Admit':
      dot.node(str(id(node)), node.classification, color='green', style='filled')
    elif node.classification == 'Deny':
      dot.node(str(id(node)), node.classification, color='red', style='filled')
    else:
      dot.node(str(id(node)), node.classification, color='magenta', style='filled')
  else:
    dot.node(str(id(node)), node.attribute)
    for value, child in node.children.items():
      label = value.decode('utf-8') if isinstance(value, bytes) else str(value)
      child_dot = draw_tree(child, dot)
      dot.edge(str(id(node)), str(id(child)), label=label)
  return dot


# Helper function for printing results
def print_results_tabulate(fold_errors, average_errors, best_fold, best_test_error):
  fold_data = []
  for i in range(len(fold_errors['training'])):
    fold_data.append([
      i+1,
      f"{fold_errors['training'][i]:.3f}",
      f"{fold_errors['validation'][i]:.3f}"
    ])

  print("\nFold-wise Errors:")
  print(tabulate(fold_data, headers=["Fold", "Training Error", "Validation Error"],
tablefmt="pretty"))

  summary_data = [
    ["Average Training Error", f"{average_errors['training']:.3f}"],
    ["Average Validation Error", f"{average_errors['validation']:.3f}"],
    [f"Best Validation Error Fold", f"Fold {best_fold}"],
    ["Test Error of Best Tree", f"{best_test_error:.3f}"]]
```



```
  print("\nSummary Statistics:")
  print(tabulate(summary_data, headers=["Metric", "Value"], tablefmt="pretty"))

if __name__ == '__main__':
  # Read dataset from CSV
  data = pd.read_csv('dataset.csv')

  # Prepare list of attributes and possible class values
  attributes = list(data.columns[:-1])
  possible_values_of_target = list(data[data.columns[-1]].unique())

  # Train/test split
  training_data = data.sample(frac=0.8, random_state=97)
  testing_data = data.drop(training_data.index)

  # 5-fold cross-validation on the training data
  k = 5
  fold_size = len(training_data) // k
  fold_errors = {'training': [], 'validation': [], 'test': []}
  trees = []

  # Build and evaluate a tree for each fold
  for i in range(k):
    validation_data = training_data.iloc[i * fold_size:(i + 1) * fold_size]
    training_data_fold = training_data.drop(validation_data.index)

    tree = decision_tree_learning(training_data_fold, attributes, training_data_fold)
    trees.append(tree)

    fold_errors['training'].append(1 - accuracy(tree, training_data_fold))
    fold_errors['validation'].append(1 - accuracy(tree, validation_data))
    fold_errors['test'].append(1 - accuracy(tree, testing_data))

  # Compute average errors and find the best model
  average_errors = {
    'training': np.mean(fold_errors['training']),
    'validation': np.mean(fold_errors['validation'])
  }

  best_fold_index = np.argmin(fold_errors['validation'])
  best_fold = best_fold_index + 1
  best_validation_error = fold_errors['validation'][best_fold_index]
  test_error_of_best_tree = fold_errors['test'][best_fold_index]
  best_tree = trees[best_fold_index]

  # Print tabulated results
  print_results_tabulate(fold_errors, average_errors, best_fold, test_error_of_best_tree)

  # Draw the best tree
  dot = draw_tree(best_tree)
  dot.graph_attr['rankdir'] = "LR"
  dot.render('best_tree', format='png', cleanup=True)
```

```
# Plot errors across folds
plt.figure(figsize=(10, 6))
folds = range(1, k + 1)
plt.plot(folds, fold_errors['training'], marker='o', label='Training Error')
plt.plot(folds, fold_errors['validation'], marker='o', label='Validation Error')
plt.plot(folds, fold_errors['test'], marker='o', label='Test Error')
plt.xlabel('Fold')
plt.ylabel('Error')
plt.title('Error Rates Across Folds')
plt.xticks(folds)
plt.legend()
plt.grid(True)
plt.savefig('error_plot.png')
plt.close()
```