CSE 472 - Decision Tree, Random Forest & Extra Trees

Complete Project Documentation

Course: CSE 472 - Machine Learning Lab
Assignment: 2 - Tree-Based Classification Algorithms
Institution: BUET (Bangladesh University of Engineering and Technology)

Table of Contents

---

1. Project Overview

This project implements three tree-based machine learning classification algorithms **from scratch** (without using sklearn's tree models):

Algorithm	Description
Decision Tree	Single tree that makes decisions by splitting data based on feature values
Random Forest	Ensemble of 100 decision trees with bootstrap sampling and random feature selection
Extra Trees	Similar to Random Forest but uses random thresholds instead of finding optimal ones

What This Project Does

1. Loads two classification datasets (Iris and Wine)
2. Splits data into 70% training and 30% testing
3. Trains 6 models (3 custom implementations + 3 sklearn versions)
4. Evaluates all models using Accuracy, F1-Score, and AUROC
5. Saves detailed results to text files

---

2. Folder Structure

```
OFFLINE-02(DecisionTree)/
│
├── readme.md                      # This documentation file
├── CSE_472_Assignment_2_DT.pdf    # Assignment specification
│
├── src/                           # All source code
│   ├── decision_tree.py           # Custom Decision Tree implementation
│   ├── random_forest.py           # Custom Random Forest implementation
│   ├── extra_trees.py             # Custom Extra Trees implementation
│   ├── run.py                     # Main entry point - runs experiments
│   └── report.py                  # Report generator with PDF and figures
│
├── results/                       # Output folder (created automatically)
│   ├── results.txt                # Combined results for all datasets
│   ├── 2005080_report.pdf         # Generated PDF report
│   └── figures/                   # Generated visualization graphs
│       ├── depth_analysis.png
│       ├── estimators_analysis.png
│       ├── final_comparison.png
│       └── generalization_gap.png
│
├── .venv/                         # Python virtual environment
│   └── ...                        # Virtual environment files
│
└── __pycache__/                   # Python cache (auto-generated)
    └── ...                        # Compiled Python files
```

File Descriptions

File	Purpose	Lines of Code
`src/run.py`	Main experiment runner - trains models, evaluates, saves results	~240
`src/report.py`	Generates PDF report with graphs and analysis	~665
`src/decision_tree.py`	Core Decision Tree with Gini impurity splitting	~150
`src/random_forest.py`	Ensemble of trees with bootstrap + feature randomness	~120
`src/extra_trees.py`	Ensemble with random thresholds (faster than RF)	~120

3. Prerequisites & Installation

Required Software

- **Python 3.8+** (tested with Python 3.11)
- **pip** (Python package manager)

Required Python Libraries

```
numpy>=1.20.0       # Numerical computations
scikit-learn>=1.0.0 # For datasets and comparison models
```

Installation Steps

```
# 1. Navigate to project folder
cd "path/to/OFFLINE-02(DecisionTree)"

# 2. Create virtual environment (recommended)
python -m venv .venv

# 3. Activate virtual environment
# Windows:
.venv\Scripts\activate
# Linux/Mac:
source .venv/bin/activate

# 4. Install dependencies
pip install numpy scikit-learn
```

---

4. Datasets Explained

We use two classic machine learning datasets from sklearn:

4.1 Iris Dataset

The Iris dataset is one of the most famous datasets in machine learning, created by Ronald Fisher in 1936.

Property	Value
Total Samples	150
Features	4
Classes	3
Train/Test Split	105 / 45

Features (measurements in cm):

1. `sepal_length` - Length of the sepal
2. `sepal_width` - Width of the sepal
3. `petal_length` - Length of the petal
4. `petal_width` - Width of the petal

Classes (types of iris flowers):

- 0 = Iris Setosa
- 1 = Iris Versicolor
- 2 = Iris Virginica

Sample Data:

```
sepal_length  sepal_width  petal_length  petal_width  class
5.1           3.5          1.4           0.2          0 (Setosa)
7.0           3.2          4.7           1.4          1 (Versicolor)
6.3           3.3          6.0           2.5          2 (Virginica)
```

4.2 Wine Dataset

The Wine dataset contains chemical analysis of wines from three different cultivars in Italy.

Property	Value
Total Samples	178
Features	13
Classes	3
Train/Test Split	124 / 54

Features (chemical properties):

1. Alcohol
2. Malic acid
3. Ash
4. Alcalinity of ash
5. Magnesium
6. Total phenols
7. Flavanoids
8. Nonflavanoid phenols
9. Proanthocyanins
10. Color intensity
11. Hue
12. OD280/OD315 of diluted wines
13. Proline

Classes (wine cultivars):

- 0 = Class 0 (59 samples)
- 1 = Class 1 (71 samples)
- 2 = Class 2 (48 samples)

---

5. What is Scikit-Learn?

Scikit-learn (sklearn) is the most popular machine learning library for Python. It provides:

What We Use from Sklearn

```
# 1. Loading datasets
from sklearn.datasets import load_iris, load_wine

# 2. Splitting data into train/test
from sklearn.model_selection import train_test_split

# 3. For comparison - sklearn's tree implementations
from sklearn.tree import DecisionTreeClassifier
from sklearn.ensemble import RandomForestClassifier, ExtraTreesClassifier

# 4. For AUROC metric calculation
from sklearn.metrics import roc_auc_score
from sklearn.preprocessing import label_binarize
```

Why Compare with Sklearn?

We implement our own Decision Tree, Random Forest, and Extra Trees from scratch. Then we compare our results with sklearn's optimized implementations to verify our code is correct.

Important: Our custom implementations do NOT use sklearn's tree models. We only use sklearn for:

- Loading datasets
- Splitting data
- Calculating AUROC metric
- Comparison (running sklearn models separately)

---

6. Algorithm Explanations

6.1 Decision Tree

A Decision Tree is like a flowchart that asks questions about your data to make predictions.

How it works:

```
Is petal_length <= 2.45?
├── YES → Predict: Setosa (Class 0)
└── NO → Is petal_width <= 1.75?
         ├── YES → Predict: Versicolor (Class 1)
         └── NO → Predict: Virginica (Class 2)
```

Key Concepts:

1. **Splitting Criterion (Gini Impurity):**

- Measures how "pure" a node is
- Formula: `Gini = 1 - Σ(probability²)`
- Gini = 0 means all samples are same class (pure)
- Gini = 0.5 means samples are evenly split (impure)

2. **Information Gain:**

- Measures how much a split improves purity
- `Information Gain = Parent Gini - Weighted Children Gini`
- We pick the split with highest information gain

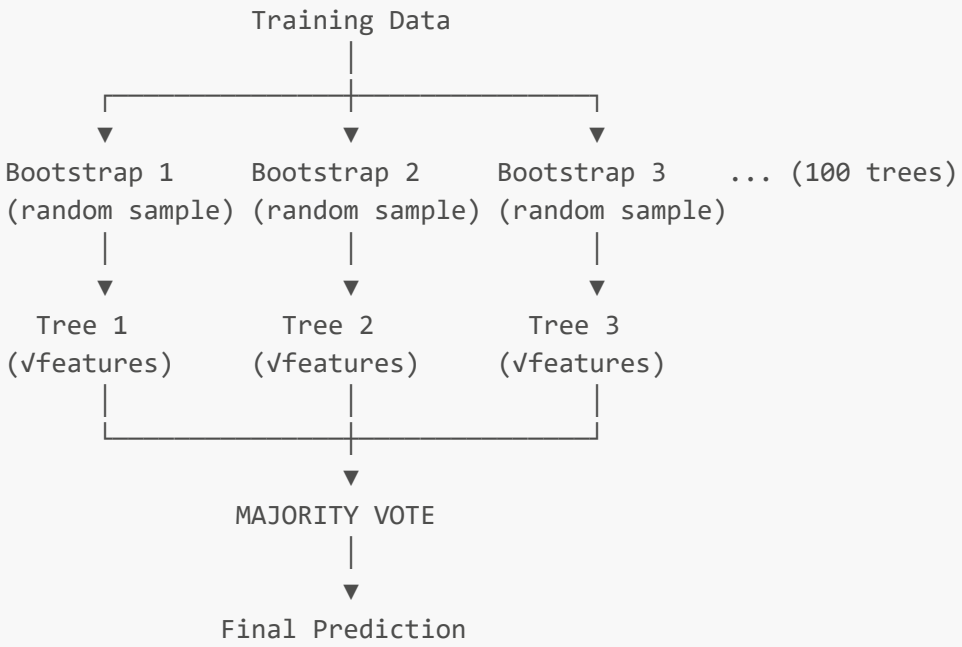
3. **Stopping Conditions:**

- Maximum depth reached (default: 10)
- Node is pure (all same class)
- Too few samples to split (default: 2)

6.2 Random Forest

Random Forest = Many Decision Trees working together

How it works:



Key Concepts:

1. **Bootstrap Sampling:**

- For each tree, randomly select N samples WITH replacement
- Some samples may appear multiple times, some not at all
- Creates diversity among trees

2. **Feature Randomness:**

- At each split, only consider √(total_features) random features
- Iris: √4 = 2 features per split
- Wine: √13 ≈ 3 features per split

3. **Majority Voting:**

- Each tree makes a prediction
- Final prediction = most common prediction among all trees

6.3 Extra Trees (Extremely Randomized Trees)

Extra Trees is like Random Forest but even more random!

Key Difference:

Aspect	Random Forest	Extra Trees
Threshold Selection	Tries all possible thresholds, picks best	Picks ONE random threshold
Speed	Slower (exhaustive search)	Faster (random selection)
Variance	Lower	Higher (more trees needed)

How Extra Trees picks a threshold:

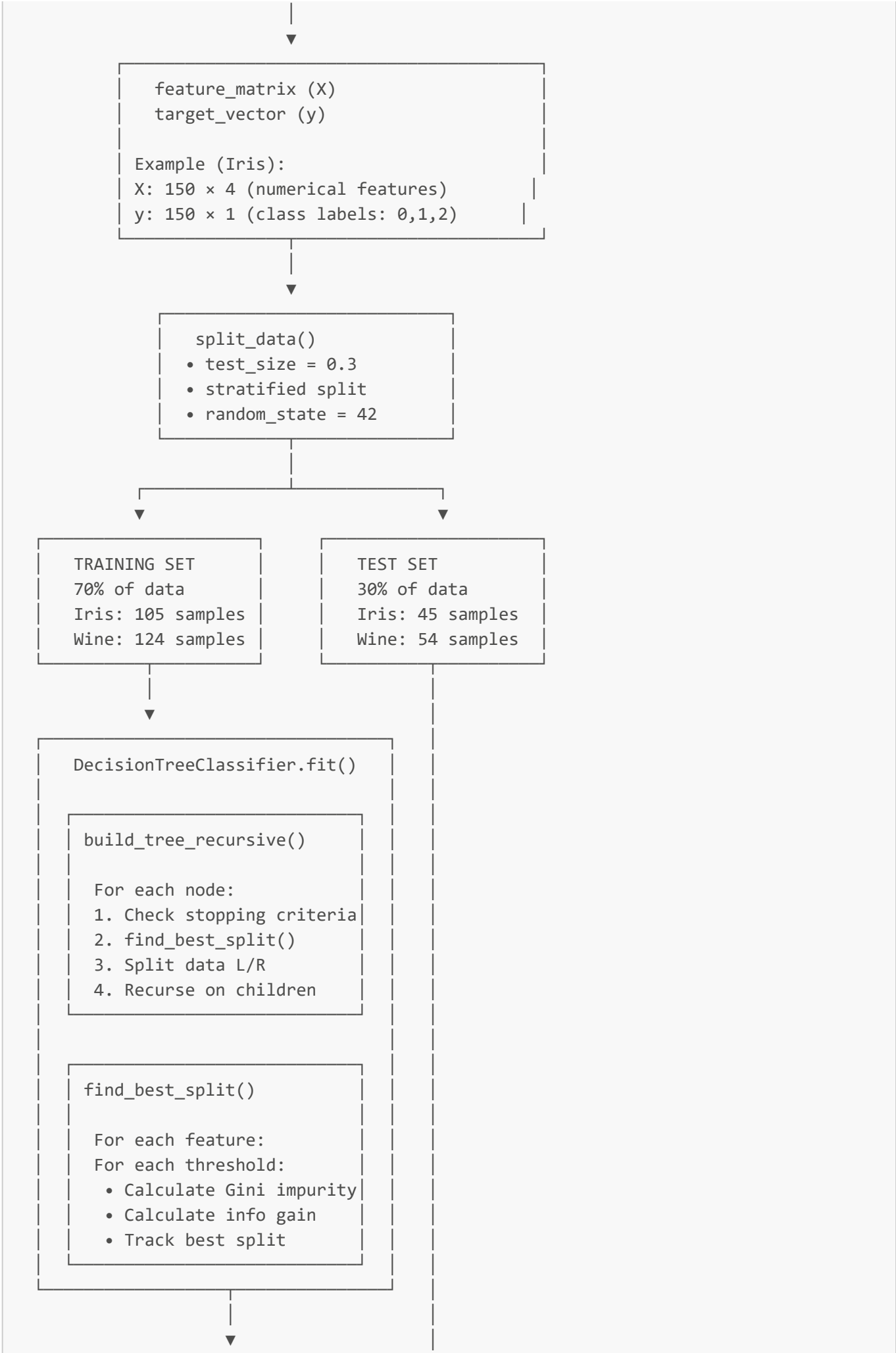
```
# Random Forest (tries all)
for threshold in all_unique_values:
    calculate_gain(threshold)
pick_best_threshold()

# Extra Trees (random)
random_threshold = random.uniform(min_value, max_value)
use(random_threshold)
```

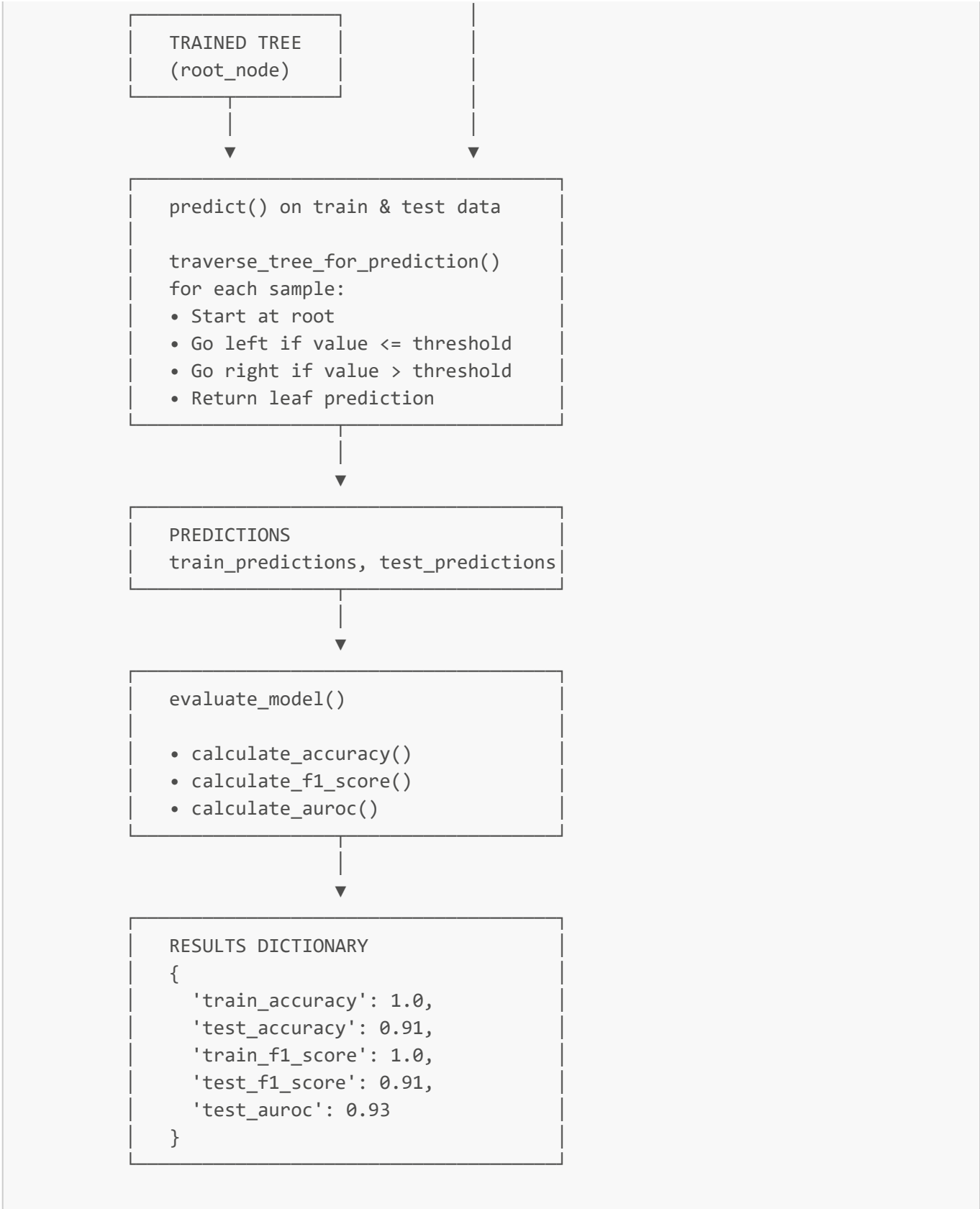
---

7. Data Flow Diagrams

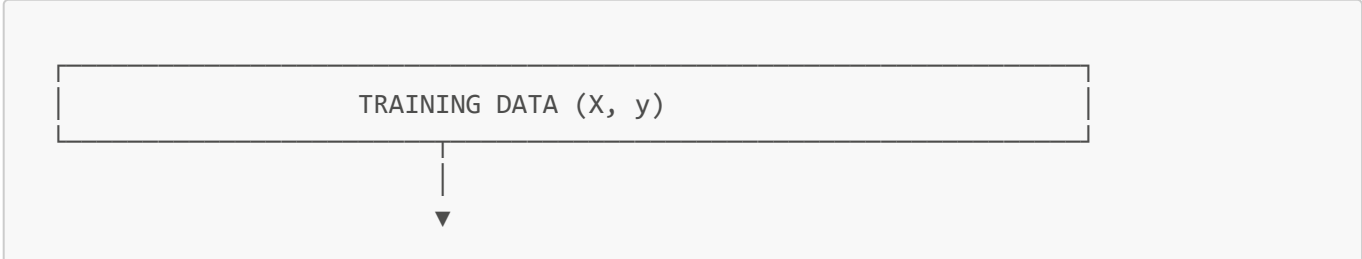
7.1 Decision Tree Data Flow

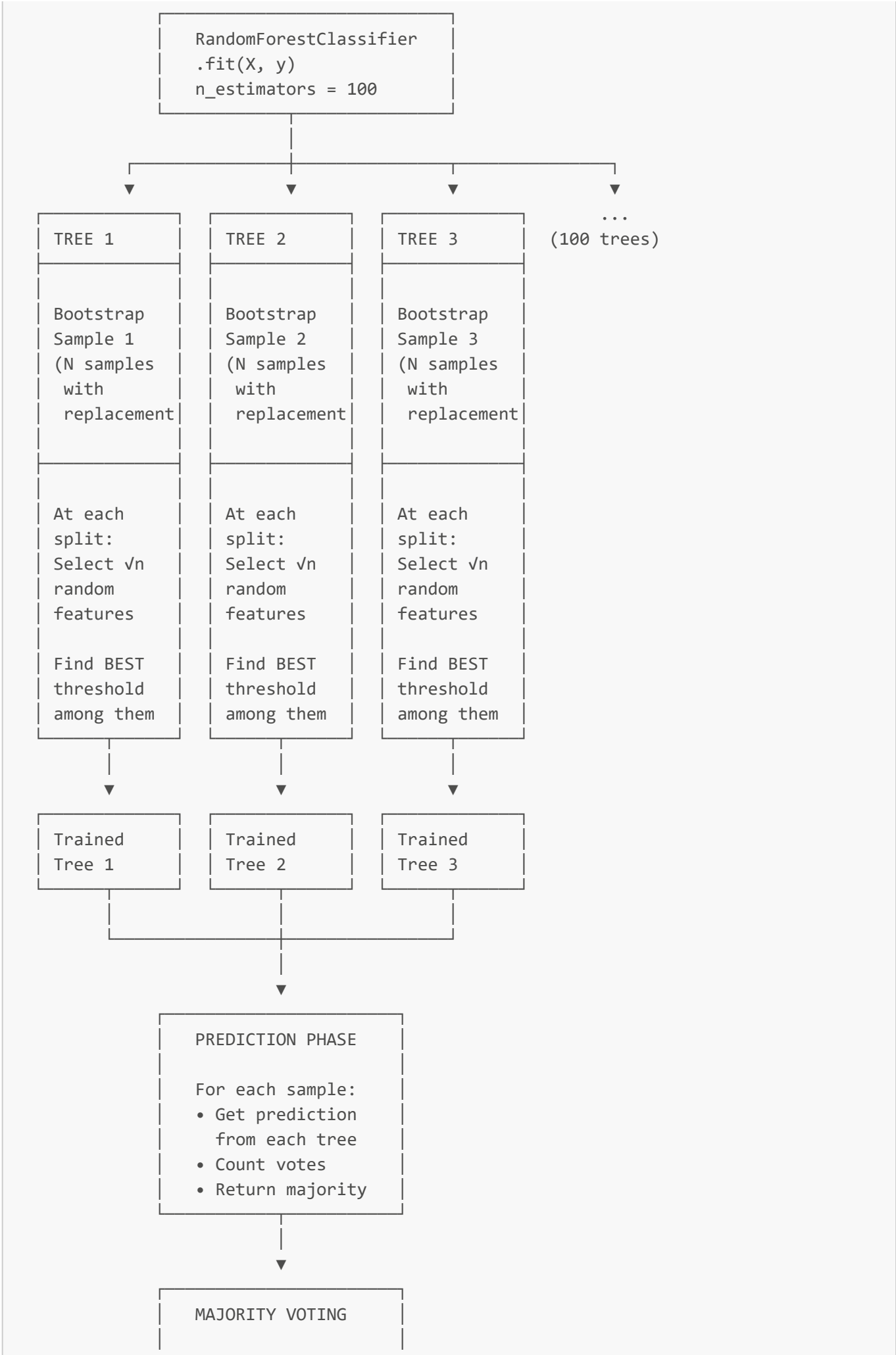
```
┌─────────────────────────────────────────────────────────────┐
│                    INPUT: RAW DATASET                       │
│              (Iris: 150×4 | Wine: 178×13)                   │
└─────────────────────────┬───────────────────────────────────┘
                          │
                          ▼
            ┌─────────────────────────────┐
            │   load_dataset()            │
            │  • sklearn.load_iris()      │
            │  • sklearn.load_wine()      │
            └──────────────┬──────────────┘
                           │
```


feature_matrix (X)
target_vector (y)
Example (Iris):
X: 150 × 4 (numerical features)
y: 150 × 1 (class labels: 0,1,2)
split_data()
• test_size = 0.3
• stratified split
• random_state = 42
TRAINING SET
70% of data
Iris: 105 samples
Wine: 124 samples
TEST SET
30% of data
Iris: 45 samples
Wine: 54 samples
DecisionTreeClassifier.fit()
build_tree_recursive()
For each node:
1. Check stopping criteria
2. find_best_split()
3. Split data L/R
4. Recurse on children
find_best_split()
For each feature:
For each threshold:
• Calculate Gini impurity
• Calculate info gain
• Track best split



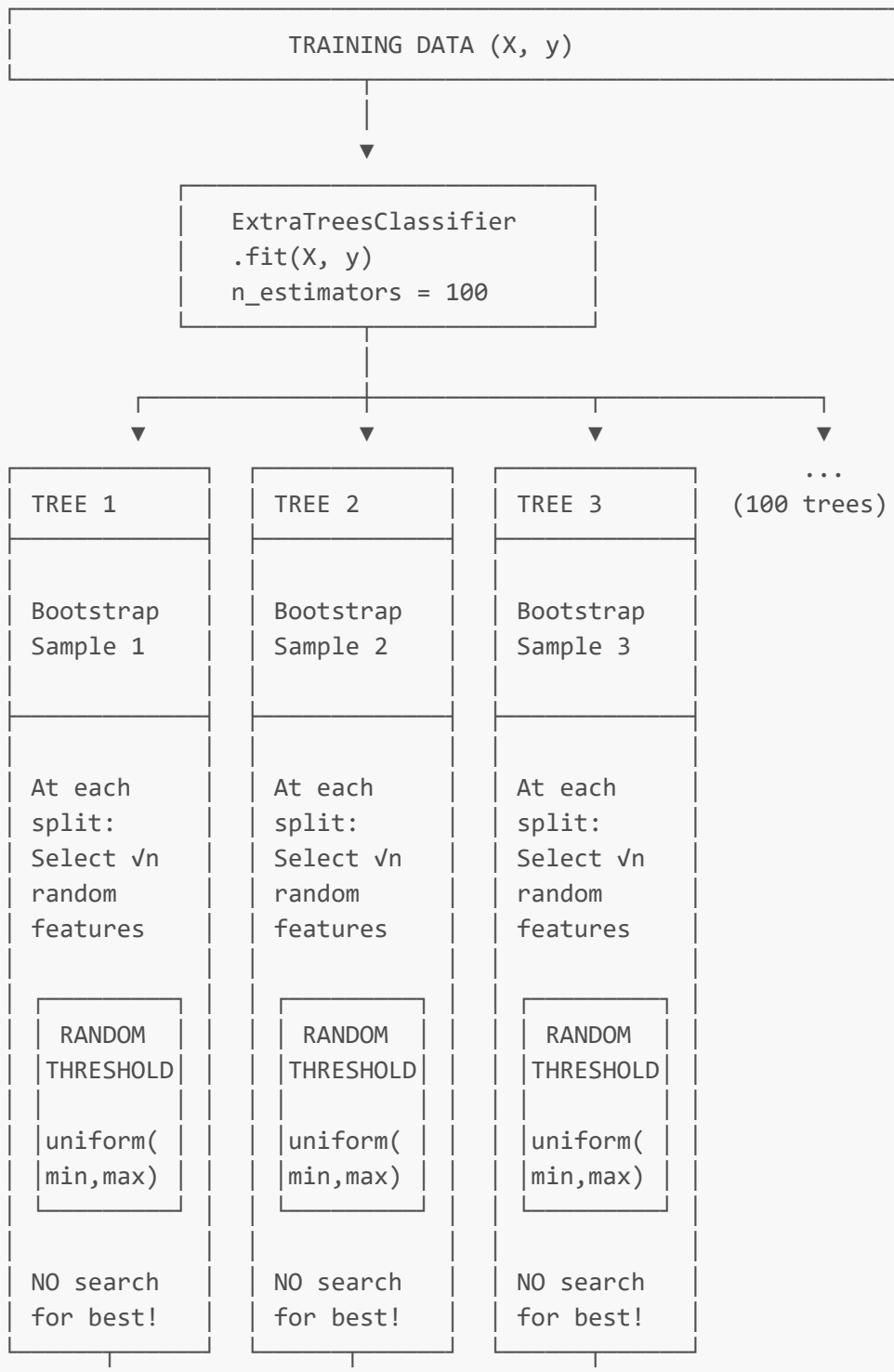
7.2 Random Forest Data Flow




RandomForestClassifier
.fit(X, y)
n_estimators = 100
TREE 1
TREE 2
TREE 3
...
(100 trees)
Bootstrap Sample 1 (N samples with replacement
Bootstrap Sample 2 (N samples with replacement
Bootstrap Sample 3 (N samples with replacement
At each split: Select √n random features
Find BEST threshold among them
Trained Tree 1
Trained Tree 2
Trained Tree 3
PREDICTION PHASE
For each sample:
• Get prediction from each tree
• Count votes
• Return majority
MAJORITY VOTING

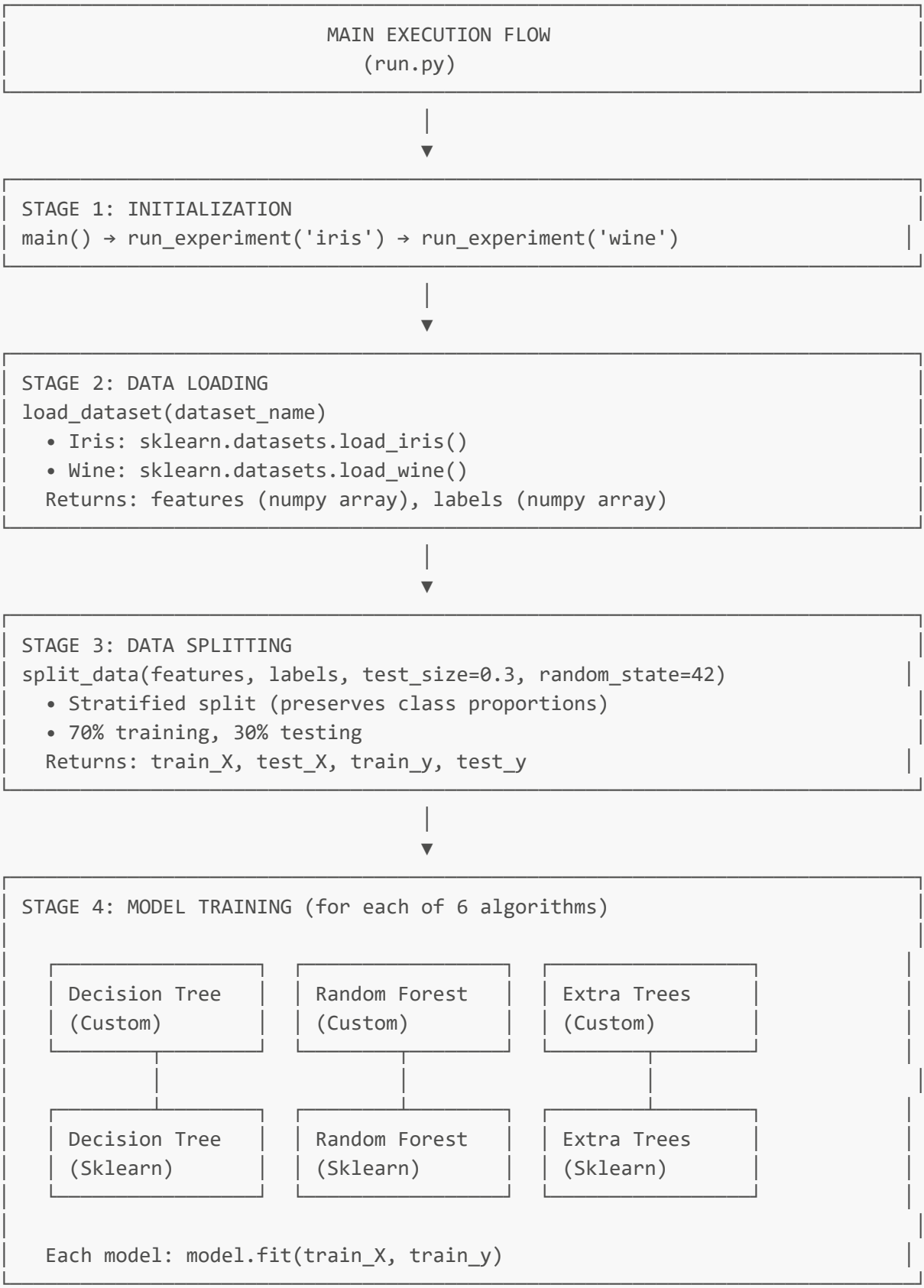
```
                │   Tree 1: Class 0     │
                │   Tree 2: Class 1     │
                │   Tree 3: Class 0     │
                │   ...                 │
                │   Tree 100: Class 0   │
                │                       │
                │   Votes: {0:67, 1:33} │
                │   Winner: Class 0     │
                └───────────────────────┘
```

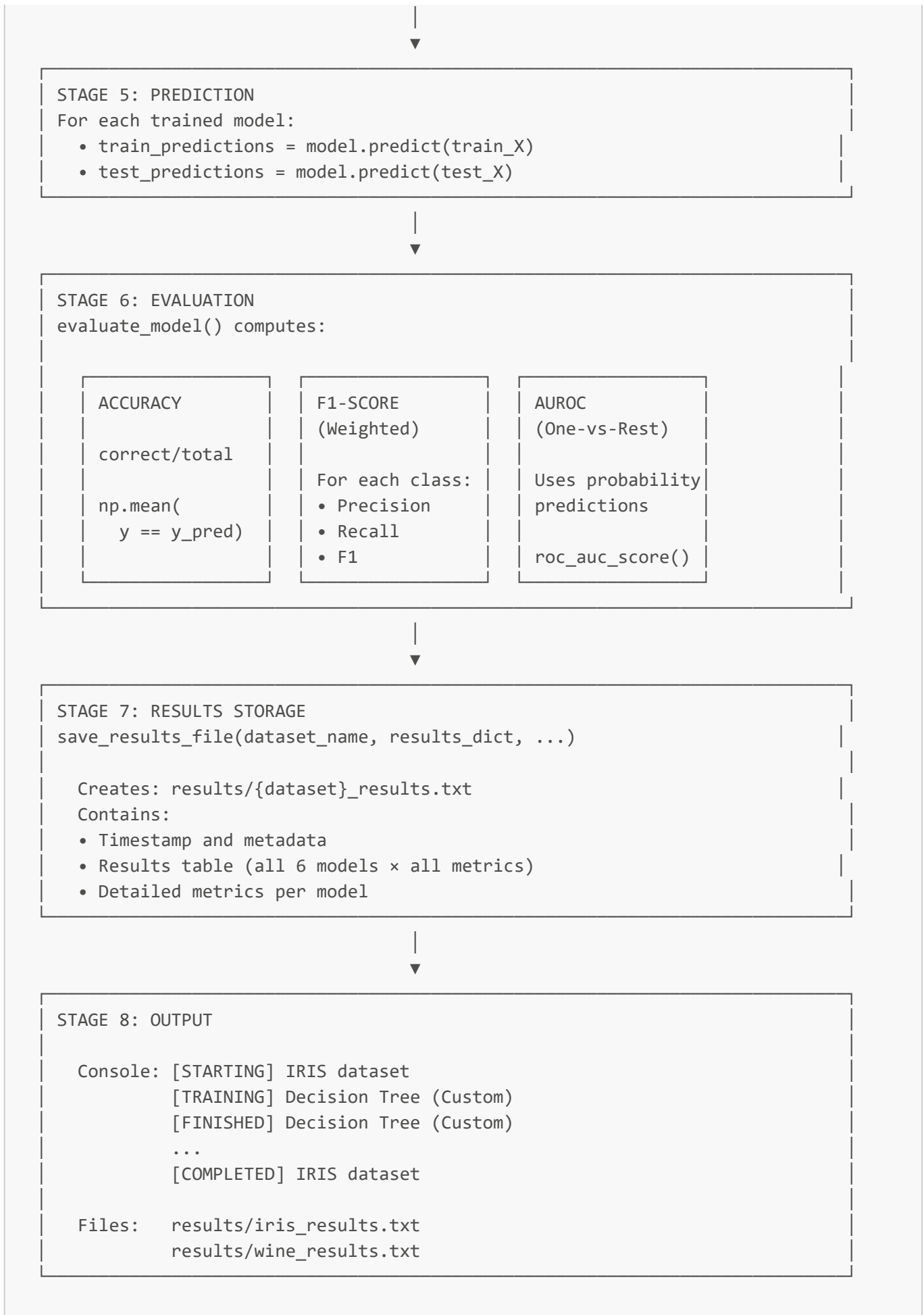
7.3 Extra Trees Data Flow



```
        │               │               │
        ▼               ▼               ▼
        (Same majority voting as Random Forest)
```

7.4 Complete Processing Pipeline



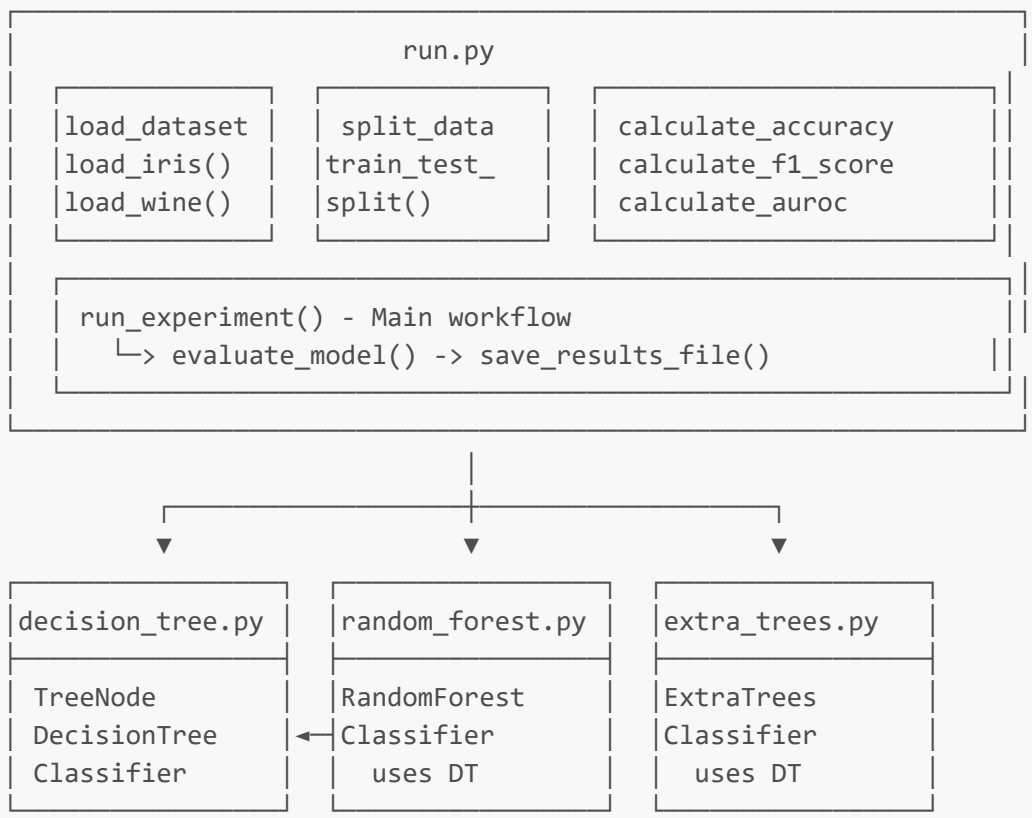


7.5 Algorithm Comparison Table

Aspect	Decision Tree	Random Forest	Extra Trees
Number of Trees	1	100	100
Bootstrap Sampling	No	Yes	Yes
Feature Selection	All features	√n random features	√n random features
Threshold Selection	Best (exhaustive)	Best (exhaustive)	Random (one sample)
Training Speed	Fast	Medium	Fast
Prediction Speed	Fast	Medium	Medium
Overfitting Risk	High	Low	Low
Variance	High	Medium	High (but averaged)
Bias	Low	Low	Slightly Higher

8. Code Architecture

7.1 Class Diagram



7.2 decision_tree.py Structure

```
class TreeNode:
    """Represents a single node in the decision tree"""
```

```
    - split_feature_index   # Which feature to split on
    - split_threshold       # Value to compare against
    - left_subtree          # Samples <= threshold
    - right_subtree         # Samples > threshold
    - class_prediction      # Prediction (only for leaf nodes)

class DecisionTreeClassifier:
    """Main Decision Tree implementation"""

    # Training
    - fit(X, y)                    # Train the tree
    - build_tree_recursive()       # Recursively build tree
    - find_best_split()            # Find optimal split
    - calculate_information_gain() # Compute gain
    - calculate_gini_impurity()    # Compute Gini

    # Prediction
    - predict(X)                   # Get predictions
    - traverse_tree_for_prediction() # Walk down tree
    - predict_probability()        # Get class probabilities
```

7.3 random_forest.py Structure

```
class RandomForestClassifier:
    """Ensemble of Decision Trees with bootstrap + feature randomness"""

    - n_estimators = 100           # Number of trees
    - ensemble_trees = []          # List of trained trees

    # Training
    - fit(X, y)                    # Train all 100 trees
    - get_max_features_count()     # Calculate √features
    - modify_tree_for_random_forest() # Add feature randomness

    # Prediction
    - predict(X)                   # Get ensemble prediction
    - get_majority_vote()          # Count votes from all trees
    - predict_probability()        # Get probabilities
```

7.4 extra_trees.py Structure

```
class ExtraTreesClassifier:
    """Like Random Forest but with random thresholds"""

    # Same structure as RandomForest, but:
    - modify_tree_for_extra_trees() # Uses random threshold selection
```

---

9. How to Run

Quick Start

```
# 1. Open terminal/command prompt
# 2. Navigate to project folder
cd "path/to/OFFLINE-02(DecisionTree)"

# 3. Activate virtual environment
.venv\Scripts\activate   # Windows
source .venv/bin/activate # Linux/Mac

# 4. Run the experiment
python src/run.py

# 5. Generate PDF report
python src/report.py
```

Expected Console Output

```
==================================================
CSE 472 - Tree Ensemble Experiments
==================================================

[STARTING] IRIS dataset
  [TRAINING] Decision Tree (Custom)
  [FINISHED] Decision Tree (Custom)
  [TRAINING] Random Forest (Custom)
  [FINISHED] Random Forest (Custom)
  [TRAINING] Extra Trees (Custom)
  [FINISHED] Extra Trees (Custom)
  [TRAINING] Decision Tree (Sklearn)
  [FINISHED] Decision Tree (Sklearn)
  [TRAINING] Random Forest (Sklearn)
  [FINISHED] Random Forest (Sklearn)
  [TRAINING] Extra Trees (Sklearn)
  [FINISHED] Extra Trees (Sklearn)
[COMPLETED] IRIS dataset

[STARTING] WINE dataset
  ... (same pattern)
[COMPLETED] WINE dataset

==================================================
All experiments completed!
Results saved to: results/
==================================================
```

10. Complete Code Walkthrough

Step 1: Program Starts (src/run.py)

```
if __name__ == "__main__":
    main()
```

When you run `python src/run.py`, Python executes `main()`:

```
def main():
    all_results = {}
    for dataset in ['iris', 'wine']:
        all_results[dataset] = run_experiment(dataset, random_state=42)
```

Step 2: run_experiment() - The Main Workflow

```
def run_experiment(dataset_name, random_state=42):

    # STEP 2a: Load the dataset
    features, labels = load_dataset(dataset_name)
    # features = numpy array of shape (150, 4) for Iris
    # labels = numpy array of shape (150,) with values 0, 1, or 2

    # STEP 2b: Split into train/test (70% / 30%)
    train_features, test_features, train_labels, test_labels = split_data(
        features, labels, test_size=0.3, random_state=random_state
    )
    # train_features: (105, 4), test_features: (45, 4) for Iris

    # STEP 2c: Create all 6 models
    algorithms = [
        ('Decision Tree (Custom)', DecisionTreeClassifier(...)),
        ('Random Forest (Custom)', RandomForestClassifier(...)),
        ('Extra Trees (Custom)', ExtraTreesClassifier(...)),
        ('Decision Tree (Sklearn)', SklearnDecisionTree(...)),
        ('Random Forest (Sklearn)', SklearnRandomForest(...)),
        ('Extra Trees (Sklearn)', SklearnExtraTrees(...)),
    ]

    # STEP 2d: Train and evaluate each model
    for algo_name, model in algorithms:
        model.fit(train_features, train_labels)  # TRAINING
        eval_results = evaluate_model(model, ...)  # EVALUATION
        results[algo_name] = eval_results

    # STEP 2e: Save results to file
    save_results_file(dataset_name, results, ...)
```

Step 3: Training a Decision Tree

When `model.fit(X, y)` is called:

```
def fit(self, X, y):
    feature_data = np.array(X)      # Convert to numpy array
    target_labels = np.array(y)     # Convert to numpy array
    self.root_node = self.build_tree_recursive(feature_data, target_labels,
depth=0)
```

The recursive tree building:

```
def build_tree_recursive(self, feature_data, target_labels, depth):
    # Check stopping conditions
    if (depth >= self.max_depth or
        all_same_class or
        too_few_samples):
        return TreeNode(class_prediction=most_common_class)

    # Find the best way to split
    best_feature, best_threshold = self.find_best_split(...)

    # Split data into left and right
    left_mask = feature_data[:, best_feature] <= best_threshold
    right_mask = feature_data[:, best_feature] > best_threshold

    # Recursively build subtrees
    left_child = self.build_tree_recursive(feature_data[left_mask], ...)
    right_child = self.build_tree_recursive(feature_data[right_mask], ...)

    return TreeNode(split_feature_index=best_feature,
                    split_threshold=best_threshold,
                    left_subtree=left_child,
                    right_subtree=right_child)
```

Step 4: Finding Best Split

```
def find_best_split(self, feature_data, target_labels, total_features):
    best_gain = -1

    # Try every feature
    for feature_index in range(total_features):
        feature_values = feature_data[:, feature_index]

        # Try every unique value as threshold
        for threshold in np.unique(feature_values):
            gain = self.calculate_information_gain(target_labels, feature_values,
```

```
threshold)

            if gain > best_gain:
                best_gain = gain
                best_feature = feature_index
                best_threshold = threshold

    return best_feature, best_threshold
```

Step 5: Making Predictions

```
def predict(self, feature_data):
    predictions = []
    for sample in feature_data:
        prediction = self.traverse_tree_for_prediction(sample, self.root_node)
        predictions.append(prediction)
    return np.array(predictions)

def traverse_tree_for_prediction(self, sample, node):
    # If leaf node, return the class prediction
    if node.is_leaf_node():
        return node.class_prediction

    # Otherwise, go left or right based on threshold
    if sample[node.split_feature_index] <= node.split_threshold:
        return self.traverse_tree_for_prediction(sample, node.left_subtree)
    else:
        return self.traverse_tree_for_prediction(sample, node.right_subtree)
```

Step 6: How Random Forest Differs

```
def fit(self, X, y):
    for tree_index in range(100):  # 100 trees
        # Bootstrap sampling (random sample with replacement)
        bootstrap_indices = np.random.choice(total_samples, total_samples,
replace=True)
        bootstrap_features = feature_data[bootstrap_indices]
        bootstrap_labels = target_labels[bootstrap_indices]

        # Create tree with modified split function
        tree = DecisionTreeClassifier(...)
        tree = self.modify_tree_for_random_forest(tree, max_features_count)

        # Train on bootstrap sample
        tree.fit(bootstrap_features, bootstrap_labels)
        self.ensemble_trees.append(tree)
```

The modification for random feature selection:

```
def modify_tree_for_random_forest(self, tree, max_features_count):
    def modified_find_best_split(feature_data, target_labels, total_features):
        # Only consider √features random features (not all)
        selected_features = np.random.choice(total_features, max_features_count,
replace=False)

        # Find best split only among selected features
        for feature_index in selected_features:
            # ... same as regular decision tree
```

Step 7: How Extra Trees Differs

```
def modify_tree_for_extra_trees(self, tree, max_features_count):
    def modified_find_best_split(feature_data, target_labels, total_features):
        selected_features = np.random.choice(total_features, max_features_count,
replace=False)

        for feature_index in selected_features:
            feature_values = feature_data[:, feature_index]

            # KEY DIFFERENCE: Random threshold instead of trying all
            random_threshold = np.random.uniform(
                np.min(feature_values),
                np.max(feature_values)
            )

            information_gain = calculate_gain(random_threshold)
```

Step 8: Evaluation

```
def evaluate_model(model, train_features, test_features, train_labels,
test_labels):
    train_predictions = model.predict(train_features)
    test_predictions = model.predict(test_features)

    return {
        'train_accuracy': calculate_accuracy(train_labels, train_predictions),
        'test_accuracy': calculate_accuracy(test_labels, test_predictions),
        'train_f1_score': calculate_f1_score(train_labels, train_predictions),
        'test_f1_score': calculate_f1_score(test_labels, test_predictions),
        'train_auroc': calculate_auroc(train_labels, train_probs),
        'test_auroc': calculate_auroc(test_labels, test_probs),
    }
```

---

11. Evaluation Metrics

10.1 Accuracy

What it measures: Percentage of correct predictions

Formula:

```
Accuracy = (Correct Predictions) / (Total Predictions)
```

Code:

```
def calculate_accuracy(true_labels, predicted_labels):
    return np.mean(true_labels == predicted_labels)
```

Example:

```
True:      [0, 1, 2, 1, 0]
Predicted: [0, 1, 1, 1, 0]
Correct:   [✓, ✓, ✗, ✓, ✓] = 4 correct
Accuracy = 4/5 = 0.80 = 80%
```

10.2 F1-Score (Weighted)

What it measures: Balance between Precision and Recall

Key Terms:

- **Precision:** Of all predicted positives, how many are actually positive?
- **Recall:** Of all actual positives, how many did we find?
- **F1:** Harmonic mean of Precision and Recall

Formula:

```
Precision = TP / (TP + FP)
Recall = TP / (TP + FN)
F1 = 2 × (Precision × Recall) / (Precision + Recall)
```

Where:

- TP = True Positive (correctly predicted positive)
- FP = False Positive (incorrectly predicted positive)
- FN = False Negative (missed positive)

Weighted F1: Average F1 across all classes, weighted by number of samples in each class.

10.3 AUROC (Area Under ROC Curve)

What it measures: Model's ability to distinguish between classes

Range: 0 to 1

- 0.5 = Random guessing
- 1.0 = Perfect classification

For Multi-class: We use One-vs-Rest (OvR) approach and compute weighted average.

---

12. Results Output

Output Location

Results are saved to `results/` folder:

- `results/iris_results.txt`
- `results/wine_results.txt`

Sample Output Format

```
================================================================================
============================
TREE ENSEMBLE EXPERIMENTS - IRIS DATASET
Timestamp: 2026-01-23 10:30:00
Data: 150 samples (105 train, 45 test) | Features: 4
Total Execution Time: 0.50s
================================================================================
============================

RESULTS TABLE - ALL MODELS AND METRICS
--------------------------------------------------------------------------------
---------------------------
Algorithm                      Train Acc    Test Acc     Train F1     Test F1
Test AUROC
--------------------------------------------------------------------------------
---------------------------
Decision Tree (Custom)             100.00%       91.11%      1.0000      0.9107
0.9333
Random Forest (Custom)             100.00%       91.11%      1.0000      0.9107
0.9333
Extra Trees (Custom)               100.00%       93.33%      1.0000      0.9333
0.9500
Decision Tree (Sklearn)            100.00%       93.33%      1.0000      0.9327
N/A
Random Forest (Sklearn)            100.00%       88.89%      1.0000      0.8878
N/A
Extra Trees (Sklearn)              100.00%       91.11%      1.0000      0.9107
N/A
================================================================================
============================
```

Understanding Results

Metric	Meaning	Good Value
Train Acc	Accuracy on training data	Higher is better (but watch for overfitting)
Test Acc	Accuracy on unseen test data	Higher is better (main metric)
Train F1	F1-score on training data	1.0 is perfect
Test F1	F1-score on test data	Higher is better
Test AUROC	Area under ROC curve	Closer to 1.0 is better

Comparing Models

Custom vs Sklearn: Our implementations should produce similar (though not identical) results to sklearn. Small differences are normal due to:

- Different random number generation
- Implementation details
- Tie-breaking rules

---

13. Hyperparameters

What are Hyperparameters?

Hyperparameters are settings you choose BEFORE training. They control how the model learns.

Hyperparameters Used

Parameter	Value	Description
`max_depth`	10	Maximum depth of trees
`min_samples_split`	2	Minimum samples required to split a node
`n_estimators`	100	Number of trees (for RF and ET)
`max_features`	'sqrt'	Number of features to consider at each split
`bootstrap`	True	Whether to use bootstrap sampling
`random_state`	42	Seed for reproducibility

Hyperparameter Effects

max_depth:

- Lower = Simpler trees, may underfit
- Higher = Complex trees, may overfit

n_estimators:

- More trees = Better accuracy but slower training
- 100 is a common default

max_features:

- 'sqrt' = √(total features) - good for classification
- 'log2' = $\log_2$(total features) - another option
- More features = less randomness, trees more similar

random_state:

- Fixed value (42) ensures reproducible results
- Same code + same random_state = same results every time

Troubleshooting

Common Issues

1. ModuleNotFoundError: No module named 'numpy'

```
pip install numpy scikit-learn
```

2. ImportError: cannot import name 'DecisionTreeClassifier'

- Make sure you're in the correct directory
- Check that `decision_tree.py` exists in the same folder as `run.py`

3. Results folder not created

- The `results/` folder is created automatically when you run the experiment
- Make sure you have write permissions in the project directory

---

Summary

This project demonstrates:

1. ☑ **Decision Tree from scratch** - Binary recursive splitting with Gini impurity
2. ☑ **Random Forest from scratch** - 100 trees with bootstrap + feature randomness
3. ☑ **Extra Trees from scratch** - Random threshold selection for faster training
4. ☑ **Comparison with sklearn** - Validate our implementations
5. ☑ **3 Evaluation Metrics** - Accuracy, F1-Score, AUROC
6. ☑ **2 Datasets** - Iris (150 samples) and Wine (178 samples)

Key Takeaway: Tree ensembles (Random Forest, Extra Trees) generally outperform single Decision Trees because they reduce overfitting through averaging multiple diverse trees.

Last Updated: January 2026