Decision Tree

Report: Decision Tree Implementation from Scratch

Submitted by: Arya Patil

Project: Decision Tree Classifier (Implemented from Scratch in Python)

---

1. Introduction

The main objective of this project was to implement a **Decision Tree Classifier** completely from scratch, without using scikit-learn's built-in decision tree functions.

This was an assignment given by my interviewer to:

- Test my **fundamental understanding** of machine learning algorithms.
- Evaluate whether I can **design, structure, and implement code in an industry-standard way**.
- Check if I can **explain my approach** clearly through both code and documentation.

The dataset chosen for testing was the **Breast Cancer Wisconsin dataset**, which is a well-known dataset for binary classification problems.

---

2. Problem Statement

The task was to build a **binary classification model** using the Decision Tree algorithm from scratch.

A Decision Tree should be able to:

1. Select the best **feature** to split on.
2. Choose the best **threshold value** for splitting.
3. Build the tree recursively until stopping conditions are met.
4. Assign **leaf nodes** with class labels.
5. Predict new data points by traversing the tree.

Constraints:

- Do not use `DecisionTreeClassifier` or similar pre-built libraries.
- Follow **industry-style templates** (separate implementation and training files).
- Evaluate the model performance on real data.

---

3. Project Structure

```
DecisionTree.py   → Implementation of the Decision Tree Classifier
train.py          → Script for training, testing, and evaluating the model
```

- **DecisionTree.py**: Contains the `Node` and `DecisionTree` classes, along with helper methods such as entropy, information gain, and splitting logic.

- **train.py**: Loads dataset, splits data, trains the decision tree, makes predictions, and calculates accuracy.

4. Methodology

4.1 How a Decision Tree Works

A decision tree is a **flowchart-like model** where:

- Each **internal node** tests a feature against a threshold.
- Each **branch** represents the outcome of that test.
- Each **leaf node** represents a class label (prediction).

To decide the best split, the algorithm uses **Information Gain** (based on Entropy).

$$InformationGain = Entropy(parent) - WeightedAverage[Entropy(children)]$$

Where:

$$Entropy(S) = -\sum p(x)log(p(x))$$

This ensures that each split reduces the randomness (uncertainty) in the dataset.

4.2 Implementation Steps

a) Node Class

- The `Node` class represents a **single node** in the decision tree.
- It stores:
  - `feature` → which feature index was used for splitting.
  - `threshold` → value used to divide data.
  - `left` and `right` → references to child nodes.
  - `l_value` → class label if it is a leaf node.
- A helper method `check_leaf_node()` checks if the current node is terminal.

b) DecisionTree Class

The main `DecisionTree` class includes all logic for building and using the tree.

Key parameters:

- `min_sample_split` → minimum samples required to split further.
- `max_depth` → maximum depth allowed.
- `n_features` → number of features to consider (adds randomness, useful for Random Forests).

Key methods:

1. **fit(x, y)**

- Entry point for training the model.
- Calls `_grow_tree()` to build recursively.

2. **predict(x)**
    - Loops through test samples and calls `_traverse_tree()` for each.
3. **_traverse_tree(i, node)**
    - Traverses tree from root to leaf for a given input.
    - At each step, compares the feature value with threshold and moves left or right.
4. **_grow_tree(x, y, depth)**
    - Recursive method that:
        - Checks stopping conditions (max depth, pure labels, not enough samples).
        - If stopping, creates a leaf node with `_most_common_label()`.
        - Otherwise, finds the **best split** and grows left and right child nodes.
5. **_best_split(x, y, selected_features)**
    - For each feature and threshold, calculates information gain.
    - Selects the best feature-threshold pair.
6. **_information_gain(y, feature_col, threshold)**
    - Calculates parent entropy.
    - Splits data into left and right groups.
    - Computes weighted child entropy.
    - Returns information gain.
7. **_entropy(y)**
    - Computes entropy based on class label distribution.
8. **_split(feature_col, threshold)**
    - Splits data indices into left and right based on threshold.
9. **_most_common_label(y)**
    - Returns the most frequent class label (used for leaf nodes).

---

c) Training & Evaluation (train.py)

Steps followed in `train.py`:

1. Loaded Breast Cancer dataset using scikit-learn.
2. Split data into training and test sets using `train_test_split`.
3. Initialized the decision tree classifier with `max_depth=10`.
4. Trained the model with `clf.fit(X_train, y_train)`.
5. Made predictions with `clf.predict(X_test)`.
6. Calculated accuracy manually using:

$$accuracy = \frac{Correct\_Predictions}{Total\_Predictions}$$

5. Results

- Dataset: **Breast Cancer Wisconsin**
- Total samples: 569
- Classes: 2 (Malignant = 0, Benign = 1)
- Training/Test split: 80% / 20%
- Decision Tree Depth: 10

Model Performance:

- Accuracy on Test Data: **~94%**

This indicates the model is working as expected and is able to classify correctly most of the time.

6. Challenges Faced

- Understanding recursion for `_grow_tree()` (tree keeps calling itself).
- Avoiding infinite recursion by setting proper **stopping conditions**.
- Debugging entropy and information gain formulas.
- Making sure array indexing for left and right splits was correct.

7. Learning Outcomes

Through this project I learned:

- How decision trees are implemented **step by step**.
- How information gain drives the splitting decisions.
- How to **structure code** in an industry-like format (separate modules).
- Practical experience with **debugging ML algorithms**.
- That even without scikit-learn, we can build working ML models.

8. Future Improvements

- Add a **tree visualization function** (to see splits clearly).
- Implement **pruning** to avoid overfitting.
- Extend the model into **Random Forests** by building multiple trees.
- Add more performance metrics (precision, recall, F1-score).

9. Conclusion

This project successfully demonstrates a **from-scratch Decision Tree implementation** for binary classification.

The model achieves ~94% accuracy on test data and confirms that the algorithm works correctly.

The assignment helped me gain:

- **Conceptual clarity** about decision trees.
- **Hands-on coding skills** in Python for ML.
- **Confidence** in implementing algorithms without depending only on libraries.

This was a valuable learning experience that combined both theory and practice.