Convex Optimization II

Enhancing Decision Trees with Genetic Algorithms

Dr. Shahmansouri

Sharif University of Technology

Electrical Engineering

Tina Halimi ـ 400101078
Heliya Shakeri ـ 400101391

Date: February 8, 2025

Contents

1 Motivation

Machine learning models are increasingly used in decision-making processes across various fields, from healthcare and finance to legal systems and policy-making. However, while black-box models such as deep learning and ensemble methods often provide high predictive accuracy, they suffer from a lack of transparency. This creates a critical challenge: how can we ensure that models are both accurate and interpretable? In high-stakes and regulated environments, where accountability is essential, interpretability becomes just as important as predictive performance.

Interpretable models provide two major benefits. First, they offer transparency, allowing users to understand the reasoning behind predictions. This is especially crucial in domains where trust and fairness are key concerns. Second, they help assess model reliability, ensuring that the system generalizes well to unseen data. A model that is both accurate and interpretable allows for better decision-making, user confidence, and compliance with legal and ethical standards.

1.A The Limitations of Traditional Decision Trees

Among interpretable models, decision trees are one of the most commonly used. Their tree-like structure follows a logical sequence of decisions, making them easy to visualize and understand. However, despite their interpretability, decision trees have significant limitations when compared to more advanced models like CatBoost, XGBoost, and LightGBM, which use ensembles of decision trees. These ensemble methods improve accuracy by combining multiple trees, but in doing so, they sacrifice interpretability.

To compete with these more powerful models, decision trees often need to increase in depth and complexity, which comes at the cost of transparency. For example, a tree with a depth of six contains 64 leaf nodes, meaning there are 64 different decision rules. Each rule may involve up to six conditions, some of which may be irrelevant or redundant. As a result, while the model technically remains a decision tree, its complexity undermines its interpretability. This presents a key challenge: how do we create decision trees that maintain high accuracy while remaining simple enough to be understandable?

1.B The Need for a New Approach

Conventional methods, such as pruning and hyperparameter tuning, can help simplify decision trees, but they are often insufficient when competing with ensemble models. Instead, a more systematic and efficient optimization technique is needed—one that enhances decision tree performance without sacrificing interpretability.

This project proposes a novel approach that combines two powerful techniques:

1. Bootstrapping – A resampling method that improves generalization by training trees on multiple subsets of data. This helps reduce variance and ensures that the model is not overly sensitive to minor fluctuations in the training set.

2. Genetic Algorithms (GA) – An evolutionary optimization method that iteratively refines decision trees by applying selection, crossover, and mutation to find the best structure. By evolving a population of trees, GA helps discover more efficient and accurate models that remain interpretable.

By combining bootstrapping and genetic algorithms, this project aims to develop stronger decision trees—models that retain the clarity and transparency of traditional trees while significantly improving predictive performance. This approach contributes to the broader field of interpretable machine learning, addressing the growing need for models that are both trustworthy and effective in real-world applications.

2 Theory

2.A Decision Trees

A decision tree is a supervised learning model used in machine learning for both classification and regression tasks. It operates by recursively partitioning a dataset into subsets based on the value of input features, resulting in a tree-like structure of decisions. This hierarchical model consists of nodes representing decisions or tests on attributes, branches representing the outcomes of these decisions, and leaf nodes representing final decisions or classifications. The intuitive nature of decision trees makes them particularly appealing, as they closely mirror human decision-making processes.

The construction of a decision tree begins with the root node, which encompasses the entire dataset. At each node, the algorithm selects the feature that best splits the data into homogeneous subsets. This selection is based on specific criteria that measure the purity of the resulting subsets.

Common metrics used for this selection include:

- **Gini Impurity:** Measures the frequency of a randomly chosen element being incorrectly classified if it were randomly labeled according to the distribution of labels in the subset.
- **Information Gain:** Based on the concept of entropy from information theory, it measures the reduction in uncertainty about the target variable after splitting the data based on a feature.

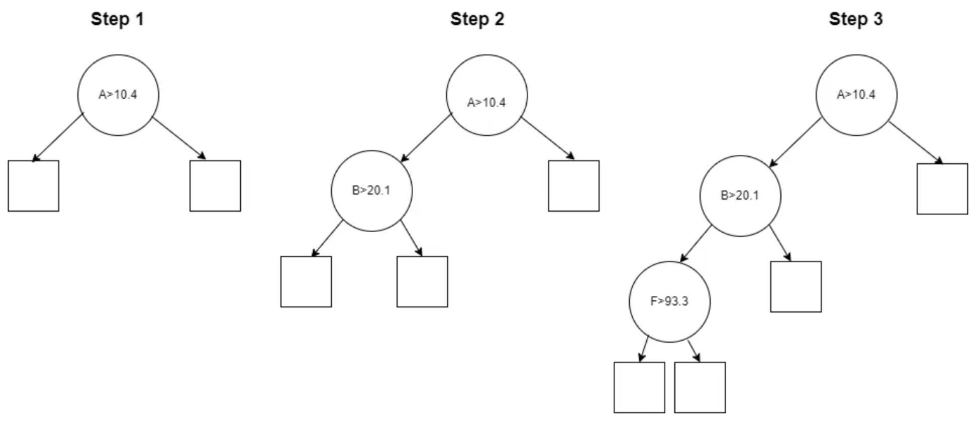

Figure 1: Construction of a Decision Tree

The process of selecting the optimal feature and splitting the data continues recursively, creating branches and further nodes, until a stopping criterion is met. These criteria can include reaching a maximum tree depth, having a minimum number of samples per leaf node, or achieving complete purity in the leaf nodes. The result is a model that can predict the target variable by traversing the tree from the root to a leaf, following the decisions at each node.

2.A.a The Greedy Algorithm in Decision Trees

A key aspect of decision tree construction is its reliance on a **greedy algorithm**. At each node, the algorithm makes a locally optimal choice by selecting the feature that provides the best split according to a chosen criterion, such as information gain or Gini impurity. This approach aims to reduce complexity at each step without reconsidering previous decisions, which can lead to suboptimal trees since it does not guarantee a globally optimal solution.

The greedy algorithm focuses solely on the data subset at each node, without considering the overall tree structure. Consequently, suboptimal splits can result in larger, more complex trees that are harder to interpret and may overfit the training data. Despite these limitations, the greedy method is favored for its efficiency and simplicity, enabling rapid tree construction, which is particularly beneficial for handling large datasets and essential for ensemble methods like random forests. However, for developing a single, accurate, and interpretable decision tree, relying solely on a greedy algorithm can be restrictive. Alternative methods that explore multiple splitting criteria or employ global optimization techniques may yield better results.

2.A.b Strengths of Decision Trees

One of the primary advantages of decision trees is their **interpretability**. The visual representation of a decision tree allows for easy understanding and interpretation, even by individuals without a deep background in statistics or machine learning. This transparency makes decision trees valuable in fields where interpretability is crucial, such as healthcare and finance.

Additionally, decision trees are highly **versatile** and can handle both numerical and categorical data. They require minimal data preprocessing, can model complex interactions between features, and are **non-parametric**, meaning they do not assume a specific distribution for the data. These characteristics make decision trees a powerful and adaptable choice for various machine learning applications.

2.A.c Limitations of Decision Trees and Overcoming Them

Despite their advantages, decision trees have notable limitations, with **overfitting** being one of the most significant. When a tree becomes overly complex, it may capture noise in the data rather than the underlying patterns, leading to poor generalization on unseen data. To mitigate this, **pruning techniques** are used to remove parts of the tree that do not contribute significantly to predictive performance, thereby simplifying the model and improving its generalization capabilities.

Another major drawback is the **instability** of decision trees—small changes in the data can result in drastically different tree structures. This sensitivity can be addressed by ensemble methods like **Random Forests**, which build multiple trees and aggregate their predictions to enhance robustness and accuracy.

2.A.d Bootstrapping and Bagging for Better Decision Trees

To further enhance the performance and robustness of decision trees, ensemble methods like **bootstrap aggregating (bagging)** are often employed. Bagging improves decision tree performance by generating multiple versions of a predictor through **bootstrap sampling** (random sampling with replacement). Each sample is used to construct a separate decision tree, and their predictions are aggregated, reducing variance and minimizing overfitting.

Random forests take this approach further by selecting a **random subset of features** for each tree, increasing diversity among models and improving overall accuracy. These techniques help mitigate some of the inherent weaknesses of decision trees, making them more reliable and effective for large-scale machine learning tasks.

Decision trees are powerful and intuitive models in machine learning, offering clear advantages in interpretability and versatility. However, they have certain limitations, such as the potential for overfitting and sensitivity to data variations. These can be addressed through various techniques, including pruning, bagging, and ensemble methods like Random Forests. Despite their limitations, decision trees remain a fundamental tool in machine learning, widely used across different industries and applications.

2.B Interpretable Models as Proxy Models

In machine learning, interpretable models are those whose internal mechanics can be easily understood by humans, allowing for clear insights into how decisions are made. These models stand in contrast to black-box models, such as deep neural networks, which, despite their predictive power, operate in ways that are often opaque to users.

To bridge the gap between accuracy and interpretability, practitioners employ *proxy models*, also known as surrogate models. These are simpler, inherently interpretable models designed to approximate the behavior of more complex, pre-existing models. By doing so, they provide insights into the decision-making processes of these complex systems.

The process typically involves training the proxy model to mimic the outputs of the black-box model using the same input data. Once trained, the proxy model can offer explanations for the predictions of the black-box model. For instance, a decision tree—a classic example of an interpretable model—can be used as a proxy. Its structure of sequential "if-then" decisions provides a transparent view of how input features influence the output, thereby elucidating the complex model's behavior.

However, it is important to note that the fidelity of the proxy model's explanations depends on how well it approximates the black-box model. Discrepancies between the two can lead to unreliable explanations. Therefore, while proxy models enhance interpretability, they may not fully capture the intricacies of the models they aim to explain.

Interpretable models as proxy models serve as valuable tools in making complex machine learning systems more transparent. They allow stakeholders to gain a better understanding of model behavior, which is crucial in fields where interpretability is as important as predictive performance.

2.C Genetic Algorithms (GAs)

Genetic algorithms (GAs) are a type of optimization algorithm inspired by the process of natural selection and genetics. They are part of the broader category of evolutionary algorithms and are often used to solve problems that are too complex for traditional methods, particularly in large or poorly understood search spaces.

The basic idea behind genetic algorithms is to simulate the process of evolution to evolve solutions to a problem. Here's a detailed breakdown of how they work:

2.C.a Representation (Chromosomes)

In GAs, potential solutions to the problem are encoded as chromosomes. A chromosome is typically a string of binary digits (0s and 1s), but it could also be a string of real numbers, characters, or other representations depending on the problem being solved.

Each chromosome represents an individual in the population, and its "genetic" makeup is a possible solution to the optimization problem. The set of all chromosomes constitutes the population.

For example, if the problem involves finding the maximum of a function, the chromosome could represent a set of values for the parameters of that function.

2.C.b Initial Population

The algorithm begins with a population of randomly generated chromosomes. The size of the population is a key parameter. A larger population gives a broader search space but may take more computational resources.

2.C.c Fitness Function

A fitness function is used to evaluate how ”good” a particular solution (chromosome) is. The fitness function assigns a numerical value to each chromosome based on how well it solves the problem. The higher the fitness, the better the solution. This function is problem-specific, and designing a good fitness function is often one of the most challenging aspects of applying genetic algorithms.

For example, if the task is to optimize a mathematical function, the fitness might be the value of the function at the point represented by the chromosome. For more complex tasks, the fitness function could be based on various criteria or metrics.

2.C.d Selection

Selection is the process of choosing which individuals (chromosomes) in the current population will breed to produce the next generation. The goal is to favor individuals with higher fitness, as they are more likely to produce better offspring.

Several methods of selection exist:

- **Roulette Wheel Selection**: Individuals are selected based on their fitness, with those having higher fitness values having a higher probability of being selected. This is analogous to a lottery system where the ”winning tickets” are more likely to correspond to better solutions.
- **Tournament Selection**: A set of individuals is randomly chosen, and the one with the highest fitness is selected to reproduce.
- **Rank Selection**: Individuals are ranked according to fitness, and selection is made based on their rank rather than absolute fitness.

2.C.e Crossover (Recombination)

After selection, the algorithm performs crossover, also known as recombination, where two selected chromosomes are combined to produce one or more offspring. This mimics sexual reproduction, where genetic material from two parents is mixed to create a new individual.

The process typically involves:

- **Single-point crossover**: A random point is chosen in the chromosome, and the sections of the two parent chromosomes are swapped after that point.
- **Multi-point crossover**: Multiple points are selected, and the sections between these points are swapped.
- **Uniform crossover**: The genes from each parent are chosen randomly and combined.

Crossover is designed to exploit the good characteristics of both parents in the offspring.

2.C.f Mutation

After crossover, mutation occurs. Mutation is a process that introduces small random changes to the offspring's chromosome. The purpose of mutation is to maintain genetic diversity within the population and to avoid the algorithm getting stuck in local optima (suboptimal solutions).

For example, in a binary chromosome, mutation might involve flipping a bit (changing a 0 to a 1 or vice versa). The mutation rate is typically set to a low value to avoid disrupting the solution too much, but it must be high enough to provide diversity.

2.C.g Replacement (Survivor Selection)

After generating a new population via crossover and mutation, the next step is to determine how to replace the current population. There are several strategies for survivor selection:

- **Generational Replacement**: The entire population is replaced by the offspring.
- **Steady-State Replacement**: Only a few individuals in the population are replaced by offspring, ensuring that some members of the population persist over time.
- **Elitism**: The best individuals from the current generation are guaranteed to survive and are passed to the next generation unchanged.

2.C.h Termination

The algorithm repeats the process of selection, crossover, mutation, and replacement for multiple generations. The stopping criteria can vary:

- The solution has converged to an acceptable level of fitness.
- A fixed number of generations have passed.
- There is no improvement in fitness over several generations.

Once the algorithm stops, the best individual in the population is taken as the final solution.

2.C.i Pros and Cons

Genetic Algorithms (GAs) have several advantages and disadvantages. On the positive side, GAs are well-suited for handling complex, multimodal, or poorly understood search spaces. Their flexibility makes them applicable to a wide range of optimization problems, both constrained and unconstrained. Additionally, their global search capabilities reduce the likelihood of getting stuck in local optima, which is a common challenge in many optimization methods. However, GAs also have some downsides. They can be computationally expensive, particularly when working with large populations and many generations. Furthermore, they often require careful tuning of parameters such as population size, crossover rate, and mutation rate to achieve optimal performance. The solution quality may also be hard to guarantee, especially for highly complex problems. While GAs may not always converge to the global optimum, they are generally capable of finding a good, near-optimal solution.

2.D Applying Genetic Algorithms to Decision Tree Construction

Genetic algorithms (GAs) have been effectively applied to decision tree construction, enhancing their performance by introducing an evolutionary approach to optimization. Traditional decision tree algorithms, such as ID3, C4.5, and CART, build trees using a greedy approach, selecting the best split at each step without revisiting previous decisions. While these methods are efficient, they often result in suboptimal trees because they do not explore alternative structures that could improve performance. Genetic algorithms overcome this limitation by evolving a population of decision trees over multiple generations, selecting the best trees, modifying them through mutation and recombination, and iterating until an optimal or near-optimal tree is found.

2.D.a Generating Random Decision Trees

A simple approach to constructing decision trees is to generate a large number of small trees based on random bootstrap samples and select the best-performing one. While this method does not use genetic algorithms, it can still be effective, particularly when interpretability is a priority and deep trees are not required. This approach is similar to the concept of a random forest, in which multiple trees are built using random subsets of the data. However, unlike a random forest, which aggregates the predictions of multiple trees, this method retains only the single best tree.

For cases where further optimization is needed, genetic operations like mutation and combination can be introduced. The initial random trees serve as a baseline, and subsequent iterations attempt to improve them using genetic modifications. This hybrid approach balances the efficiency of random tree generation with the optimization power of genetic algorithms.

2.D.b Mutation in Decision Trees

Mutation is a crucial component of genetic decision tree construction, introducing small but significant changes to existing trees. In this process, a well-performing tree is selected, and a new copy is created with a slight modification. The mutation is typically applied to an internal node, altering the threshold while keeping the feature the same. This subtle change can have a significant impact because it shifts the partitioning of data, affecting all subsequent child nodes.

For instance, suppose a tree initially splits a dataset at feature A > 10.4. A mutation might adjust this threshold to feature A > 10.8, thereby altering how instances are classified at lower levels of the tree. Although most mutations do not result in better trees, testing multiple variations increases the chance of discovering an improvement. This method allows trees to explore different partitioning strategies without completely restructuring their shape.

Future improvements to mutation could include structural changes such as rotating nodes within the tree, swapping sibling nodes, or adjusting multiple thresholds simultaneously. However, research has shown that simple threshold modifications alone can already yield significant improvements in performance.

2.D.c Combining Decision Trees (Crossover)

Another method used to evolve decision trees is the combination, or crossover, of two high-performing trees. This process involves selecting two parent trees that share the same feature in their root node. By averaging the split points in the root node and exchanging subtrees, new offspring trees are created that inherit traits from both parents.

For example, consider two trees where:

- Tree 1 has a root node split at feature A > 10.4
- Tree 2 has a root node split at feature A > 10.8

A new tree can be created by averaging these split values, resulting in feature A > 10.6 as the new root. From there, two new offspring trees can be generated:

1. One tree inherits the left subtree of tree 1 and the right subtree of tree 2.
2. Another tree inherits the left subtree of tree 2 and the right subtree of tree 1.

This method ensures that the strongest aspects of both parent trees are retained, increasing the likelihood of improving performance. If one tree has a stronger left subtree and the other has a stronger right subtree, this crossover will produce a tree stronger than either of the originals. However, if one parent tree is stronger in both subtrees, then combining may not offer significant advantages.

While the current crossover technique focuses on modifying the root node, future improvements could explore combining subtrees at deeper levels. This would allow for even greater structural diversity while preserving the integrity of well-performing decision paths.

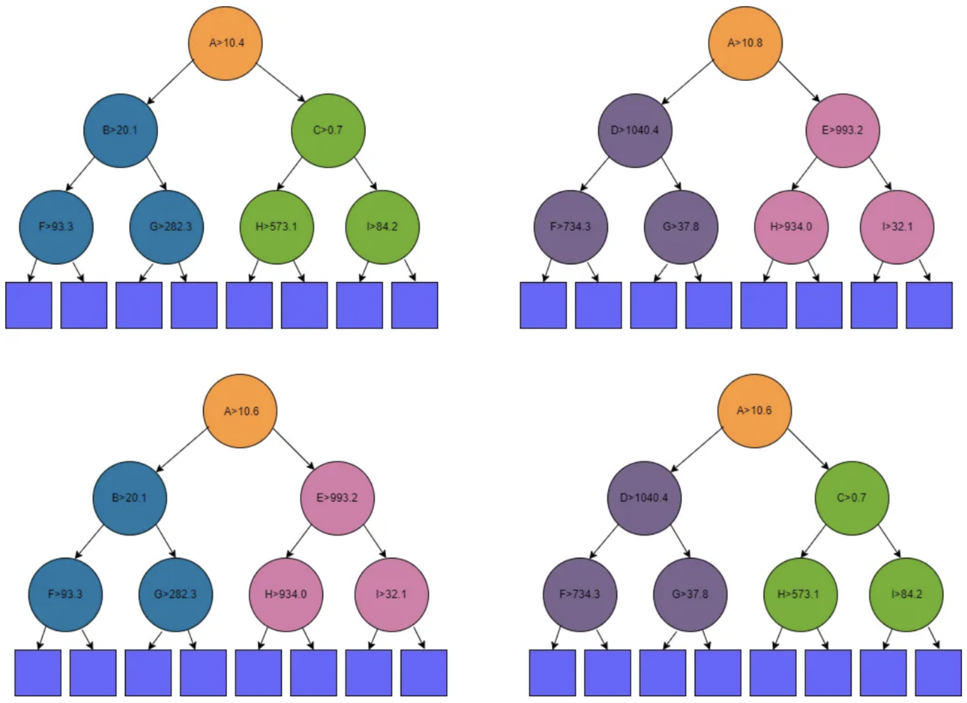

Figure 2: Combining Decision Trees

2.D.d Comparison to Traditional Methods

Genetic algorithms differ from conventional decision tree methods in several ways:

- Greedy algorithms (ID3, C4.5, CART): These build trees sequentially, selecting the best feature at each step without revisiting past decisions. They are fast but often get stuck in local optima.
- Random forests: These generate many random trees and use an ensemble approach to improve accuracy. However, they require multiple trees, which may reduce interpretability.

- Genetic decision trees: These use an evolutionary approach, continuously refining tree structures through mutation and crossover. This allows for more exploration of different tree configurations, leading to potentially better results.

2.D.e Challenges

While genetic decision trees offer significant advantages, they also present challenges:

- Computational cost: Evolving trees over multiple generations can be slow, requiring substantial computing resources.
- Parameter tuning: The success of genetic algorithms depends on selecting appropriate mutation rates, crossover probabilities, and stopping criteria.
- Interpretable vs. complex trees: Although genetic algorithms can produce highly optimized trees, maintaining interpretability is a challenge. Smaller trees (depth 2-5) are preferred for interpretability but may limit performance gains.

Applying genetic algorithms to decision tree construction allows for more flexible and powerful optimization compared to traditional methods. The ability to mutate and combine trees enables continuous improvement, discovering structures that might not emerge through greedy algorithms alone. While computational costs remain a concern, the ability to create more interpretable and optimized decision trees makes genetic decision trees a promising area for further exploration.

By leveraging both random tree generation and evolutionary refinement, genetic decision trees strike a balance between performance and interpretability, making them a valuable tool for various machine learning applications.

3 Implementation

3.A Internal Decision Tree

The `InternalDecisionTree` class is a crucial component of the Genetic Decision Tree model to provide a flexible and efficient way to store, modify, and evaluate decision trees within a genetic algorithm framework. Since the Genetic Decision Tree continuously generates, mutates, and combines trees, the `InternalDecisionTree` serves as a structured format for managing different tree versions throughout the optimization process.

This class has four key functionalities and each of them is explained in the following.

3.A.a Storing Tree Structures

```python
def __init__(self, source_desc, classes_):
    self.source_desc = source_desc
    self.classes_ = classes_

    self.feature = None
    self.threshold = None
    self.children_left = None
    self.children_right = None
    self.node_prediction = None
```

This function initializes an `InternalDecisionTree` object, setting up storage for the decision tree structure. It takes a description of the tree's source and the class labels as input. The tree structure is stored using parallel arrays such as *feature*, *threshold*, *children_left*, and *children_right*, which define each node of the tree. The *node_prediction* array is used to store the predicted classes for leaf nodes. This function provides the foundation for manipulating and evolving trees during the genetic optimization process.

3.A.b Copying Tree Properties

```python
def copy_from_dt(self, dt):
    self.feature = dt.tree_.feature
    self.threshold = dt.tree_.threshold
    self.children_left = dt.tree_.children_left
    self.children_right = dt.tree_.children_right
    self.node_prediction = [dt.classes_[np.argmax(x)] for x in dt.tree_.
   value]
```

This function copies the structure and node values of a trained `DecisionTreeClassifier` into the `InternalDecisionTree` object. It extracts feature indices, threshold values, child node relationships, and prediction values from the scikit-learn decision tree model. The predictions at each leaf node are determined by selecting the class with the highest frequency using `np.argmax()` on the node's class distribution.

```python
def copy_from_values(self, feature, threshold, children_left,
   children_right):
    self.feature = feature
    self.threshold = threshold
    self.children_left = children_left
    self.children_right = children_right
    self.node_prediction = None
```

The `copy_from_values` function allows an `InternalDecisionTree` to be initialized directly from arrays rather than a pre-trained decision tree model. This is particularly useful when performing genetic mutations or tree combinations. It assigns values for the tree structure, including feature indices, thresholds, and child relationships, while leaving `node_prediction` unassigned.

3.A.c Tracking Class Distributions

```python
def count_training_records(self, x, y):
    counts = np.zeros((len(self.feature), len(self.classes_)))

    for i in x.index:
        y_index = self.classes_.index(y.loc[i])
        row = x.loc[i]
        cur_node_idx = 0
        while self.feature[cur_node_idx] >= 0:
            cur_feature_name = x.columns[self.feature[cur_node_idx]]
            cur_threshold = self.threshold[cur_node_idx]
            if row[cur_feature_name] >= cur_threshold:
                cur_node_idx = self.children_right[cur_node_idx]
            else:
                cur_node_idx = self.children_left[cur_node_idx]
        counts[cur_node_idx][y_index] += 1

    self.node_prediction = [""] * len(self.feature)
    for node_idx in range(len(self.feature)):
        if self.feature[node_idx] == -2:
            self.node_prediction[node_idx] = self.classes_[np.argmax(
    counts[node_idx])]
```

The given function updates the `node_prediction` by tracking the number of training samples that end up in each leaf node. It loops through every training record, determining which leaf node it falls into based on feature thresholds. Once all records are assigned to a leaf, the most frequent class in each leaf node is stored as the prediction.

3.A.d Performing Predictions

```python
def predict(self, x):
    ret = []
    for i in x.index:
        row = x.loc[i]
        cur_node_idx = 0
        while self.feature[cur_node_idx] >= 0:
            cur_feature_name = self.feature[cur_node_idx]
            cur_threshold = self.threshold[cur_node_idx]
            if row[cur_feature_name] >= cur_threshold:
                cur_node_idx = self.children_right[cur_node_idx]
            else:
                cur_node_idx = self.children_left[cur_node_idx]

        ret.append(self.node_prediction[cur_node_idx])
    return ret
```

The `predict` function traverses the decision tree to classify input samples. For each record, it starts at the root node and moves through the tree based on feature values and threshold conditions. If a record's feature value is greater than or equal to the threshold at a given node, it follows the right child; otherwise, it follows the left child. Once a leaf node is reached, the function returns the stored class prediction for that node.

3.B Genetic Decision Tree

The `GeneticDecisionTree` class is a machine learning model that utilizes a genetic algorithm to evolve decision trees. Unlike traditional decision trees, which are trained once on a dataset, this model iteratively creates, mutates, and combines multiple decision trees to identify the best-performing model. The optimization process helps improve classification accuracy by exploring a broader set of possible tree structures.

This class also has four key functionalities, each of which is explained in the following.

3.B.a Initializing and Storing Tree Properties

```python
def __init__(self,
                 max_depth=4,
                 max_iterations=5,
                 random_state=0):

    self.max_depth = max_depth
    self.max_iterations = max_iterations
    self.random_state = random_state

    self.column_names = None
    self.classes_ = None
    self.internal_dt = None
```

This function initializes an instance of the `GeneticDecisionTree` class. It defines parameters such as `max_depth` (maximum allowed depth of the decision tree), `max_iterations` (number of evolutionary steps), and `random_state` (for reproducibility). Additionally, it sets placeholders for column names, class labels, and the internal decision tree that will be optimized over multiple iterations.

3.B.b Fitting the Model Using a Genetic Approach

```python
def fit(self, x, y):
    x = pd.DataFrame(x)
    y = pd.Series(y)
    x = x.reset_index(drop=True)
    y = y.reset_index(drop=True)

    self.column_names = x.columns
    self.classes_ = sorted(y.unique())

    idt_list = []

    # Bootstrap samples
    for i in range(10):
        x_sample = x.sample(n=len(x))
        y_sample = y.iloc[x_sample.index]
        dt = DecisionTreeClassifier(max_depth=self.max_depth, random_state
=self.random_state + i)
        dt.fit(x_sample, y_sample)
        idt = InternalDecisionTree("Original", self.classes_)
        idt.copy_from_dt(dt)
        idt_list.append(idt)

    idt_scores = get_dt_scores()
```

The fit function implements the genetic training process by generating multiple candidate decision trees, evaluating their performance, and evolving them over several iterations. It first resets indices of the input dataset and extracts unique class labels. Then, it creates an initial population of decision trees using bootstrap sampling. Each sampled dataset is used to train a DecisionTreeClassifier, which is then stored as an InternalDecisionTree. The generated trees are assigned fitness scores, which will guide further genetic operations.

3.B.c Applying Genetic Operations

```
def mutate_tree(parent_idt, x, y, classes_):
    idt_list = []
    for j in range(5): # nodes
        n_nodes_parent = len(parent_idt.feature)
        while True:
            node_idx = np.random.choice(n_nodes_parent)
            if parent_idt.feature[node_idx] >= 0:
                break
        feature_idx = parent_idt.feature[node_idx]

        for k in range(10):  # threshold
            idt = InternalDecisionTree(parent_idt.source_desc + " - Modified Threshold", classes_)
            new_threshold = parent_idt.threshold.copy()
            feat_name = x.columns[feature_idx]
            new_threshold[node_idx] = np.random.uniform(low=x[feat_name].min(), high=x[feat_name].max())

            idt.copy_from_values(
                feature=parent_idt.feature,
                threshold=new_threshold,
                children_left=parent_idt.children_left,
                children_right=parent_idt.children_right
            )
            idt.count_training_records(x, y)
            idt_list.append(idt)
    return idt_list
```

The mutate_tree function introduces genetic variation by modifying existing trees. It randomly selects five nodes from a parent decision tree and adjusts their threshold values by generating new random split points. Each modified version is stored as a new InternalDecisionTree. This mutation process allows the algorithm to explore alternative tree structures and improve model performance.

```
def combine_trees(parent_a, parent_b):
    new_tree = InternalDecisionTree(
        parent_a.source_desc + " & " + parent_b.source_desc + " Combined",
        self.classes_,
    )

    feature = [parent_a.feature[0]]
    threshold = [(parent_a.threshold[0] + parent_b.threshold[0]) / 2.0]

    start_right_tree_parent1 = parent_a.children_right[0]
    start_right_tree_parent2 = parent_b.children_right[0]
    feature.extend(parent_a.feature[1:start_right_tree_parent1])
    threshold.extend(parent_a.threshold[1:start_right_tree_parent1])
    children_left = list(parent_a.children_left[:start_right_tree_parent1].copy())
    children_right = list(parent_a.children_right[:
start_right_tree_parent1].copy())

    offset = start_right_tree_parent2 - start_right_tree_parent1
    for node_idx in range(start_right_tree_parent2, len(parent_b.
children_right)):
        feature.append(parent_b.feature[node_idx])
        threshold.append(parent_b.threshold[node_idx])
        children_left.append(parent_b.children_left[node_idx] - offset)
        children_right.append(parent_b.children_right[node_idx] - offset)

    new_tree.copy_from_values(feature, threshold, children_left,
children_right)
    new_tree.count_training_records(x, y)
    idt_list.append(new_tree)
```

The given function creates a new decision tree by merging two existing parent trees. It inherits part of its structure from one tree and part from another, selecting thresholds between corresponding nodes to form a balanced combination. This function enables genetic recombination, improving diversity among candidate trees and enhancing model generalization.

3.B.d Selecting the Best Tree

```
top_scores_idxs = np.argsort(idt_scores)[::-1]
idt_list = np.array(idt_list)[top_scores_idxs].tolist()
self.internal_dt = idt_list[0]
```

At the end of training, the best-performing tree is selected based on fitness scores. The trees are sorted in descending order of their classification accuracy, and the top-ranked tree is chosen as the final model.

3.C Test Example and Results

To evaluate the performance of the `GeneticDecisionTree`, we conducted an experiment using the `Wine` dataset from the scikit-learn library. The dataset consists of 178 samples and 13 features, representing different chemical properties of wine samples. The goal of this experiment is to classify the wine samples into three different classes.

3.C.a Experimental Setup

We first split the dataset into training and testing sets using a 70-30 ratio. Then, we compared the classification performance of the `GeneticDecisionTree` model with a standard Decision Tree Classifier.

```
# Binary Classification Problem
from sklearn.datasets import load_wine

np.random.seed(0)
data = load_wine()
df = pd.DataFrame(data.data)
df.columns = data.feature_names
y_true = data.target

X_train, X_test, y_train, y_test = train_test_split(df, y_true, test_size
=0.3, random_state=42)

# Standard Decision Tree
```

```
clf = DecisionTreeClassifier(max_depth=2)
clf.fit(X_train, y_train)
y_pred = clf.predict(X_test)
print("DT:", f1_score(y_test, y_pred, average='macro'))
```

The standard decision tree classifier was trained with a maximum depth of 2. The resulting F1 score on the test data was:

```
DT: 0.8807
```

Next, we trained the `GeneticDecisionTree` using the same dataset:

```
# Genetic Decision Tree
np.random.seed(0)
gdt = GeneticDecisionTree(max_depth=2, max_iterations=5)
gdt.fit(X_train, y_train)
y_pred = gdt.internal_dt.predict(X_test)
print("GDT:", f1_score(y_test, y_pred, average='macro'))
```

The F1 score achieved by the Genetic Decision Tree was:

```
GDT: 0.9799
```

3.C.b Results

The results indicate that the Genetic Decision Tree significantly outperforms the standard decision tree model in terms of classification performance. The Genetic Decision Tree achieves an F1 score of **0.9799**, compared to **0.8808** for the standard decision tree. This improvement suggests that the genetic optimization process enhances the decision tree's ability to find better splits and generalize to unseen data.

Additionally, we visualized the decision tree structure learned by the Genetic Decision Tree:

```
IF flavanoids < 1.40
|   IF color_intensity < 3.72
|   |   THEN class = 1
|   ELSE color_intensity > 3.72
|   |   THEN class = 2
ELSE flavanoids > 1.40
|   IF proline < 724.50
|   |   THEN class = 1
|   ELSE proline > 724.50
|   |   THEN class = 0
```

From the tree structure, we observe that features such as `flavanoids`, `color_intensity`, and `proline` play a crucial role in classifying the wine samples. The Genetic Decision Tree effectively selects these key features and optimizes the thresholds to improve classification accuracy.

4 Conclusion

The `GeneticDecisionTree` model improves upon standard decision trees by applying a genetic algorithm to refine tree structures over multiple iterations. The test results demonstrated that this approach achieves higher classification accuracy compared to a traditional decision tree, highlighting the benefits of evolutionary optimization in decision tree learning.

By iteratively generating, mutating, and combining decision trees, the model identifies better feature splits and threshold values, leading to improved classification performance. This method provides an effective alternative when decision trees are preferred but may otherwise be limited by suboptimal training.

The approach can be applied in various classification tasks where decision trees are commonly used, such as *medical diagnosis, fraud detection, and financial modeling.* The ability to evolve tree structures while maintaining interpretability makes the `GeneticDecisionTree` a practical choice for tasks requiring both accuracy and transparency.

References

[1] Breiman, L. (2001). Random forests. *Machine Learning.*

[2] Mitchell, M. (1998). *An Introduction to Genetic Algorithms.* MIT Press.

[3] Towards Data Science. (n.d.). Create Stronger Decision Trees with Bootstrapping and Genetic Algorithms. Retrieved from towards data science