A Comparison of Traditional Decision Trees and Hoeffding Trees

Abstract

Decision trees are a widely used classification technique in machine learning. Traditional trees such as ID3, C4.5, or CART are trained in a batch setting, where the entire dataset is available upfront. Hoeffding Trees (also known as Very Fast Decision Trees) extend this concept to data streams, enabling incremental learning with statistical guarantees. This paper provides a short introduction to how both types of trees are learned, highlighting their algorithmic steps, key differences, and typical use cases.

1 Introduction

Decision trees are hierarchical models that recursively split the input space according to feature values, producing a tree where leaves represent class predictions. They are interpretable, relatively fast to train, and effective for many classification tasks. However, traditional algorithms assume all training data are available in memory, which is impractical for large or streaming datasets.

Hoeffding Trees address this limitation by applying the Hoeffding bound to decide when splits are statistically justified, enabling learning from potentially infinite streams without storing all data.

2 Learning a Traditional Decision Tree

A traditional decision tree is learned in a *batch* mode. The high-level process is:

1. Start with the full dataset at the root node.
2. If all examples belong to the same class, create a leaf node.
3. Otherwise, compute a splitting criterion (e.g., information gain, Gini index) for each feature.
4. Select the feature with the highest score and split the dataset accordingly.
5. Recurse on each child node until stopping conditions are met.

2.1 Algorithmic Sketch

Algorithm 1 Batch Decision Tree Training

```
function TrainDecisionTree(data, features)
    if StoppingCondition(data) then
        return Leaf(MajorityClass(data))
    end if
    best_feature ← SelectBestSplit(data, features)
    node ← DecisionNode(best_feature)
    for each value v of best_feature do
        subset ← {x ∈ data : x[best_feature] = v}
        child ← TrainDecisionTree(subset, features \ best_feature)
        AddBranch(node, v, child)
    end for
    return node
end function
```

3 Learning a Hoeffding Tree

In contrast, a Hoeffding Tree processes data in a *streaming* fashion:

1. Start with a single leaf at the root.
2. For each new example, traverse the tree to the corresponding leaf.
3. Update sufficient statistics at that leaf.
4. Periodically evaluate possible splits at the leaf.
5. Use the Hoeffding bound to decide if the best split is statistically better than the second-best.
6. If confident, split the leaf into child nodes.

3.1 Hoeffding Bound

The Hoeffding inequality ensures that with probability $1-\delta$, the observed mean of a random variable converges to its true mean within an error margin ϵ:

$$\epsilon = \sqrt{\frac{R^2 \ln(1/\delta)}{2n}}$$

where R is the range of the random variable, n is the number of observations, and δ is the desired confidence.

3.2 Algorithmic Sketch

Algorithm 2 Hoeffding Tree Training

```
Initialize root as a Leaf
for each incoming example (x, y) do
    leaf ← TraverseTree(root, x)
    UpdateStatistics(leaf, x, y)
    if SeenEnoughExamples(leaf) then
        (best, best_score) ← FindBestSplit(leaf)
        (second, second_score) ← FindSecondBestSplit(leaf)
        ε ← HoeffdingBound(R, δ, n)
        if best_score − second_score > ε then
            SplitLeaf(leaf, best)
        end if
    end if
end for
```

4 Comparison

Aspect	Traditional Tree	Hoeffding Tree
Data assumption	Full dataset in memory	Streaming / infinite
Learning mode	Batch (offline)	Incremental (online)
Splitting criterion	Chosen after full scan	Chosen via Hoeffding bound
Memory usage	Stores dataset or stats	Stores only leaf stats
Adaptability	Static after training	Grows with new data

Table 1: Comparison of traditional and Hoeffding decision trees

5 Conclusion

Traditional decision trees are simple, interpretable, and effective when the full dataset is available. However, they are unsuitable for streaming contexts. Hoeffding Trees extend the decision tree paradigm to data streams, using statistical bounds to make reliable splitting decisions without revisiting old data. This makes them highly valuable for large-scale, real-time machine learning systems.

A Helper Functions for Decision Tree Training

Algorithm 3 Stopping Condition

```
function StoppingCondition(data)
    if all examples have same class then
        return True
    else if features empty OR max depth reached then
        return True
    else if size(data) < min_samples then
        return True
    else
        return False
    end if
end function
```

Algorithm 4 Majority Class

```
function MajorityClass(data)
    counts ← frequency of each class
    return class with highest count
end function
```

Algorithm 5 Select Best Split

```
function SelectBestSplit(data, features)
    best_score ← −∞
    for feature in features do
        score ← ComputeSplitScore(data, feature)
        if score > best_score then
            best_score ← score
            best_feature ← feature
        end if
    end for
    return best_feature
end function
```

Algorithm 6 Compute Split Score (Information Gain)

```
function ComputeSplitScore(data, feature)
    H_p ← Entropy(labels(data))
    H_c ← 0
    for value in unique(feature) do
        subset ← {x ∈ data : x[feature] = value}
        weight ← size(subset)/size(data)
        H_c ← H_c + weight × Entropy(labels(subset))
    end for
    return H_p − H_c
end function
```

B Helper Functions for Hoeffding Tree Training

Algorithm 7 Traverse Tree

```
function TraverseTree(node, x)
    if node is Leaf then
        return node
    end if
    feature ← node.split_feature
    value ← x[feature]
    child ← node.branch(value)
    return TraverseTree(child, x)
end function
```

Algorithm 8 Update Statistics

```
function UpdateStatistics(leaf, x, y)
    leaf.class_counts[y] ← leaf.class_counts[y] + 1
    for feature in x do
        value ← x[feature]
        leaf.feature_value_counts[feature][value][y] ← leaf.feature_value_counts[feature][value][y]
    + 1
    end for
end function
```

Algorithm 9 Check if Leaf Has Enough Examples

```
function SeenEnoughExamples(leaf)
    if leaf.total_seen ≥ MIN_INSTANCES then
        return True
    else
        return False
    end if
end function
```

Algorithm 10 Find Best Split

```
function FindBestSplit(leaf)
    best_score ← −∞
    best_feature ← None
    for feature in leaf.feature_value_counts do
        score ← ComputeSplitScore(leaf.statistics, feature)
        if score > best_score then
            best_score ← score
            best_feature ← feature
        end if
    end for
    return (best_feature, best_score)
end function
```

Algorithm 11 Hoeffding Bound

```
function HoeffdingBound(R, δ, n)
    ϵ ← sqrt(R^2 ln(1/δ) / 2n)
    return ϵ
end function
```

Algorithm 12 Split Leaf

```
function SplitLeaf(leaf, best_feature)
    node ← DecisionNode(split_feature = best_feature)
    for value in possible_values(best_feature) do
        child ← LeafNode()
        AddBranch(node, value, child)
    end for
    Replace leaf with node in the tree
end function
```