– Independent Work Report Spring 2025 –

Robustness Verification for Ensemble Learning Methods

Samuel Sanft
Adviser: Aarti Gupta

Abstract

The goal of this paper is to propose a new method for approximating the minimum and maximum outputs of decision tree-based ensemble learning methods over a bounded input. To this end, a data structure for representing these models and algorithms for merging and pruning decision trees are proposed, which can be used to set lower and upper bounds on the output of a model. This method can be applied to common ensemble methods including bagging, random forests, and gradient boosted trees. This paper details how these algorithms can be used in conjunction with existing optimization techniques to solve verification problems such as local robustness verification. This paper demonstrates how the proposed method outperforms current SMT-based methods for local robustness verification by several orders of magnitude. It also attempts to use this method to compare the effect of model choice on local robustness in both regression and classification settings.

1. Introduction

Introduction

2. Goals

Goals

3. Background Work

3.1. Decision Trees

Formally define decision trees.

3.2. Ensemble Methods

Explain Bagging, Random Forests, Gradient-boosted Trees.

3.3. Robustness/Safety Verification

Explain robustness verification.

3.4. Satisfiability Modulo Theories

Explain SMT.

3.5. Optimization Techniques

Explain particle swarm optimization (PSO) and branch and bound.

4. Approach

In order to prove a robustness query, one must demonstrate that $y_{\min} \leq f_{\min} \leq f_{\max} \leq y_{\max}$, where $y_{\min}, y_{\max}$ are constants provided by the query and $f_{\min}, f_{\max}$ are respectively the minimum and maximum outputs of a model f given a set of inputs Ω. While calculating $f_{\min}, f_{\max}$ for an ensemble of decision trees is an NP-hard problem, calculating the minimum and maximum outputs of a single decision tree can be done efficiently by finding the minimum and maximum values among its set of leaves. We can set a lower bound on $f_{\min}$ by summing the minimum values of every decision tree within f's ensemble and an upper bound on $f_{\max}$ by summing the maximum values of every decision tree within f's ensemble. More formally, if $f(x) = \sum f_i(x)$, where each f_i is a decision tree, then:

$$f_{\text{min\_bound}} = \sum \min_x f_i(x), f_{\text{max\_bound}} = \sum \max_x f_i(x)$$

This paper will refer to $f_{\text{min\_bound}}, f_{\text{max\_bound}}$ as the outer bounds, since $f_{\text{min\_bound}} \leq f_{\min} \leq f(x) \leq f_{\max} \leq f_{\text{max\_bound}}$. A robustness query can therefore be proven by demonstrating that $y_{\min} \leq f_{\text{min\_bound}} \leq f_{\text{max\_bound}} \leq y_{\max}$. A robustness query can be disproven by providing an input $x \in \Omega$, such that $f(x) < y_{\min}$ or $f(x) > y_{\max}$.

For simplicity, all robustness queries will be treated as two seperate queries: a minimum query and a maximum query. A minimum query can be proven by showing that $y_{\min} \leq f_{\min}$, given a set of inputs Ω, while a maximum query can be proven by showing that $f_{\max} \leq y_{\max}$, given a set of inputs Ω. A robustness query can be proven by proving its corresponding minimum and maximum queries, while it can be disproven by disproving either one.

This paper proposes a pruning algorithm and a merging algorithm that can be used to refine the outer bounds in order to successfully prove a robustness query. It also describes how PSO can be used to find counterexamples in order to disprove a query.

4.1. Decision Tree Algorithms

4.1.1. Prune The proposed pruning algorithm prunes the branches of a decision tree that are unreachable, given a boxed set of inputs Ω. It does so by successively applying `pruneLeft` and `pruneRight` to a decision tree along each axis. Given a decision tree, an axis k, and a constant c, `pruneLeft` removes any branches that are unreachable when $x_k \leq c$ while `pruneRight` removes any branches that are unreachable when $x_k \geq c$. If $x_{\min}, x_{\max}$ are the bounds for Ω, then a decision tree can be pruned by successively calling `pruneLeft` on every axis k with the kth entry of $x_{\max}$ and `pruneRight` on every axis k with the kth entry of $x_{\min}$.

Pseudo-code for `pruneLeft` is provided below. It is a recursive algorithm that traverses along the structure of the tree, starting at the root. The code for `pruneRight` follows the same pattern, but if `axis == tree.axis` it drops the left subtree and prunes the right subtree when `c >= tree.threshold` and prunes only the left subtree while keeping the right subtree intact otherwise.

```
pruneLeft(tree, axis, c):
    if tree.isLeaf:
        # Don't prune leaves
        return tree

    if axis != tree.axis:
        # Prune both subtrees
        l = pruneLeft(tree.left, axis, c)
        r = pruneLight(tree.right, axis, c)
        return Split(l, r, tree.axis, tree.threshold)
```

```
    if c <= tree.threshold:
        # Drop right subtree, prune left subtree
        return pruneLeft(tree.left, axis, c)
    else:
        # Prune only right subtree
        r = pruneLeft(tree.right, axis, c)
        return Split(tree.left, r, tree.axis, tree.threshold)
```

Listing 1: pruneLeft algorithm

`pruneLeft` and `pruneRight` both run in linear or sublinear time, with respect to the size of a tree, since they don't visit any node more than once. When Ω only contains a small percentage of the samples within the training dataset, applying this procedure can significantly reduce the size of a decision tree.

By removing any branches of a decision tree that are unreachable when $x \in \Omega$, it is possible to improve the outer bounds $f_{\text{min\_bound}}, f_{\text{max\_bound}}$. Applying this procedure to every tree in an ensemble can also make further analysis significantly more efficient, thanks to a reduction in size of the model. The `pruneLeft` and `pruneRight` algorithms are also a key component of the `merge` algorithm.

4.1.2. Merge The proposed merge algorithm takes two trees f_1, f_2 as input and produces a new tree as output f', such that $f'(x) = f_1(x) + f_2(x)$ for any x. It is a recursive function that traverses the structure of both trees. Since $\min_x f_1(x) + \min_x f_2(x) \leq \min_x(f_1(x) + f_2(x))$ and $\max_x f_1(x) + \max_x f_2(x) \geq \max_x(f_1(x) + f_2(x))$, we can improve the outer bounds by successively applying `merge` to pairs of trees within an ensemble. This reduces the total number of trees within the ensemble, although it may significantly (in certain cases, exponentially) increase the size of each tree.

Partial pseudo-code for the `merge` algorithm is provided below. `addConst` returns a copy of a tree, where each leaf value has been modified by adding a given constant.

```
merge(tree1, tree2):
    # If one or both trees are a leaf, add value to all leaves in other tree
    if tree1.isLeaf:
        return addConst(tree2, tree1.val)
    if tree2.isLeaf:
        return addConst(tree1, tree2.val)
```

```
    # Trees split on seperate axes at top node
    if tree1.axis != tree2.axis:
        l = mergeTrees(tree1.left, tree2)
        r = mergeTrees(tree1.right, tree2)
        return Split(l, r, tree1.axis, tree1.threshold)

    # Trees split on same axis
    axis = tree1.axis
    t1 = tree1.threshold
    t2 = tree2.threshold
    if t1 < t2:
        l = merge(tree1.left, pruneLeft(tree2.left, axis, t1))
        c = merge(pruneLeft(tree1.right, axis, t2), pruneRight(tree2.left, axis, t1))
        r = merge(pruneRight(tree1.right, axis, t2), tree2.right)
        return Split(l, Split(c, r, axis, t2), axis, t1)
    else if t2 < t1:
        l = merge(pruneLeft(tree1.left, axis, t2), tree2.left)
        c = merge(pruneRight(tree1.left, axis, t2), pruneLeft(tree2.right, axis, t1))
        r = merge(tree1.right, pruneRight(tree2.right, axis, t1))
        return Split(l, Split(c, r, axis, t1), axis, t2)
    else:
        l = merge(tree1.left, tree2.left)
        r = merge(tree1.right, tree1.right)
        return Split(l, r, axis, t1)
```

Listing 2: merge algorithm

In cases where the two trees have no correlating structure (`tree1` and `tree2` have no nodes in common that split along the same axis), the number of leaves in the resulting merged tree is equal to the product of the numbers of leaves in `tree1` and the number of leaves in `tree2`. However, when the trees have some correlation in their structure (which is generally expected to be true for any two trees in an ensemble, since they trained on similar data), the size may be smaller. In other words, successively merging trees within an ensemble may take exponential time and space in the worst case, but polynomial time and space in practical cases. This also means that merging trees in an ensemble may be more inefficient when the dimension increases (since trees have fewer splitting nodes with matching axes) or in settings like random forests, where trees are intentionally trained on different subsets of features, in order to reduce correlation.

One possible optimization for merging trees is to use a branch and bound technique. A modified `merge` function with branch and bound optimization that focuses on lowering $f_{\text{max\_bound}}$ would take two trees, f_1, f_2, and a constant c. If $\max_x f_1 + \max_x f_2 \leq c$, then instead of merging the two

trees, the function returns a single leaf whose value is $\max_x f_1 + \max_x f_2$. In other words, subtrees of f' whose maximum value is guaranteed to be less than c are replaced by a single leaf, helping to manage an explosion of size and complexity. This means that the guarantee $f'(x) = f_1(x) + f_2(x)$ is replaced by the guarantee $f'(x) \geq f_1(x) + f_2(x)$. This optimization can also be applied to create a modified `merge` function that focuses on increasing $f_{\text{min\_bound}}$. The choice of c greatly impacts the effectiveness of this optimization. Choosing a larger c will increase the number of branches that get replaced (decreasing the overall size of the result), but picking a c that is too large may limit how much $f_{\text{max\_bound}}$ is decreased when merging trees.

4.2. Verification Approach

4.2.1. Inner Bounds & Counter-examples Counter-examples for a minimum or maximum query can be found by using PSO. PSO is an good choice for this task thanks to its ease of implementation and efficiency, its robustness to early convergence to local minima/maxima, and its ability to optimize non-differentiable and non-continuous functions. PSO is not guaranteed to find f_{min} or f_{max}, but if it finds any x, such that $f(x) \leq y_{\text{min}}$ or $f(x) \geq y_{\text{max}}$, then the query can be immediately disproven. PSO's ability to find samples close to f_{min} or f_{max} depends on the overall smoothness of f, which in turn depends on the smoothness of the target function it is trained to replicate. If the target function has many local minima and maxima, the likelihood of PSO converging to a value far away from the true f_{min} or f_{max} increases.

Even if PSO is unable to disprove a query, it can provide a best known sample x' that comes closest to y_{min} or y_{max}, depending on if PSO was used to minimize or maximize the function. This best known sample can be helpful for improving the outer bounds, as it can be used to set bounds for the branch and bound optimization of the `merge` algorithm.

4.2.2. Outer Bounds Outer bounds for an ensemble can be improved by using the proposed algorithms described in subsection 4.1. Given a minimum or maximum query, the first step to improve the outer bounds is to prune the trees within the ensemble in order to remove any branches that are unreachable given the input bounds. Then pairs of trees within the ensemble can be

successively merged until $y_{\text{min}} \leq f_{\text{min\_bound}}$ in the case of a minimum query or $f_{\text{max\_bound}} \leq y_{\text{max}}$ in the case of a maximum query. If a best known sample x' exists, this x' can be used to generate bounds for the branch and bound optimization. For example, when merging two trees f_i, f_j, the bound $f_i(x') + f_j(x')$ could be used. This method of merging trees is useful because it merges the trees one at a time, meaning no extra work needs to be done once the query is proven.

4.3. Verification Steps

The proposed process for attempting to prove a minimum or maximum query is as follows:

1. Prune the ensemble to remove any unreachable branches given the input bounds.
2. Use PSO to search for counterexamples/best known sample.
3. Improve outer bound by successively calling merge, using best known sample for branch and bound optimization.

After each step, $f_{\text{min\_bound}}$ or $f_{\text{max\_bound}}$ can be calculated in order to prove the query and end the process early. Since merging the entire ensemble into a single tree is often impossible given practical time and space constraints, a limit can be placed on the number of merges that are performed in step 3. This process is not guaranteed to prove or disprove a query, but strengthening the hyperparameters for PSO (e.g. increasing number of samples per iteration or the total number of iterations) or increasing the number of merges in step 3 can help.

5. Implementation

5.1. Data Structures

For the purposes of proving minimum and maximum queries, the following C struct for representing decision trees is proposed.

```
struct tree {
    bool isLeaf;
    uint32_t dim;
    double val;
    struct splitInfo * split;
};

```

```c
struct splitInfo {
    uint32_t axis;
    double threshold;
    struct tree * left;
    struct tree * right;
    double min;
    double max;
    uint32_t depth;
    uint64_t size;
};
```

Listing 3: decision tree data structure

Each node in a tree has it's own struct and can be treated as it's own tree. A linked list representation of the tree (as opposed to the list representation used by libraries like scikit-learn) [ADD CITATION] is necessary for performing the prune and merge algorithms described in subsection 4.1. For leaf nodes, only a `struct tree` is necessary, and the `split` field is set to `NULL`. For splitting nodes, a `struct tree` and `struct splitNode` is allocated. The `axis` and `threshold` fields determine the location of the split, while `left` and `right` point to the left and right subtrees. Minimum and maximum values are calculated at construction and stored so that they can be accessed in constant time (this is necessary in order to calculate $f_{\text{min\_bound}}, f_{\text{max\_bound}}$ for an ensemble more efficiently).

Using this data structure, the output of a decision tree can be evaluated recursively as follows:

```c
double treeEval(const struct tree * t, const double x[]) {
    if (t->isLeaf) {
        return t->val;
    }

    // Tree has splitting node
    struct split = t->splitInfo;
    if (x[split->axis] <= split->threshold) {
        return treeEval(split->left, x);
    } else {
        return treeEval(split->right, x);
    }
}
```

Listing 4: decision tree evaluations

5.2. Software

A library for constructing ensembles of decision trees and proving robustness queries was implemented in Python, for the purposes of this paper. Python is an ideal language for this task thanks to its ease of use and extensive support for ensemble and other machine learning methods. This library allows conversion from scikit-learn's models, including support for gradient-boosted trees, random forests, and bagging estimators. This means that a model can be created and trained using scikit-learn, then converted to the data structure presented in Listing 3 and used for proving robustness and minimum/maximum queries using the library presented along with this paper.

In order to improve runtime and memory efficiency, the code for creating, evaluating, pruning, merging and freeing decision trees was implemented directly in C. Wrappers for these functions were written in Python using the C Foreign Function Interface [ADD CITATION] so that these functions could be called from within the Python library.

6. Results and Evaluation

7. Future Work

8. Conclusion

9. Honor Statement

This paper represents my own work in accordance with University policy.

- Samuel Sanft

References