Robustness of Optimal Randomized Decision Trees With Dynamic Programming

Valentijn Götz[1]
Supervisor(s): Emir Demirović[1], Koos van der Linden[1]
[1]EEMCS, Delft University of Technology, The Netherlands

A Thesis Submitted to EEMCS Faculty Delft University of Technology,
In Partial Fulfilment of the Requirements
For the Bachelor of Computer Science and Engineering
January 29, 2023

Name of the student: Valentijn Götz
Final project course: CSE3000 Research Project
Thesis committee: Emir Demirović, Koos van der Linden, Frans Oliehoek

Abstract

Decision tree learning is widely done heuristically, but advances in the field of optimal decision trees have made them a more prominent subject of research. However, current methods for optimal decision trees tend to overlook the metric of robustness. Our research wants to find out whether the robustness of optimal decision trees can be improved by incorporating randomization. To achieve this, we added randomization to the existing MurTree algorithm, and performed experiments to compare the robustness. The results show that adding randomization improves the robustness of the decision tree but lowers the out of sample accuracy.

1 Introduction

Decision tree learning is a machine learning method that produces tree models, which are called decision trees. The goal of these decision trees is to label and classify data. In every internal node of this tree, a query gets made to determine whether a data-point goes to the left or right side of the tree. When it reaches a leaf node, the data-point gets assigned a label corresponding to that leaf node. The major benefit of this model is that it is easy to interpret for humans, and does not necessarily require technical knowledge to understand. This makes them especially attractive to use in contexts where interpretability and readability are important, such as in healthcare.
Creating decision trees that are optimal, however, has been shown to be NP-Complete [8], which leads them to be very computationally expensive to compute.
For that reason, decision tree learning has historically been implemented using heuristic algorithms, e.g. CART [5] or ID3 [9]. Heuristic approaches use a heuristic to approximate a good split at every node, these approaches have fast runtimes, which enables them to solve the problem in a reasonable amount of time. The drawback is that these approaches do not guarantee that the resulting decision tree has the least amount of misclassifications, and thus are not always able to create decision trees that are optimal from a global perspective.
The benefit of optimal trees compared to heuristic trees is that they generally offer greater accuracy outside the training data and are seen as a better representation of the data. Computing these optimal decision trees was first considered to be computationally intractable, but through advances in research, multiple viable approaches to constructing optimal decision trees have been found.
One approach is to pose the problem as a mixed integer optimization problem [2] and use MIO techniques to produce optimal trees. Other methods such as DL8.5 [1] and MurTree [6] make use of dynamic programming to construct optimal decision trees. Dynamic programming uses memoization, where you store solutions to subproblems in a cache to improve runtime.
Most common algorithms for calculating decision trees have a deterministic rule for splitting the dataset at predicate nodes, but Blanquero et al. [4] proposed a different approach, the Optimal Randomized Classification Tree (ORCT). In an ORCT, every instance in a dataset has its own probability to branch left or right when in an internal node depending on the value of the associated feature. As a result, each instance has its own individual probabilities to be assigned a certain label. So instead of ending up in a single leaf node, each leaf node now has a certain probability to be reached.
One potential benefit that the randomization could have, is that it can improve the robustness of the model. Robustness means that a model is more resistant to malicious attacks and noise in the data. These things can impact the accuracy of the classification significantly, such as in the one pixel attack [11], where changing one pixel changes the classification result entirely.
Despite the benefits that robustness provides, this metric does not get investigated in the paper by Blanquero et al. [4]. Furthermore, the effects of randomization on decision trees has only

scarcely been studied in the literature.
So the question we strive to answer in this work is how the incorporation of randomization affects the robustness of decision trees.
To answer this question, we made a modified version of the MurTree [6] algorithm and added randomized splits as in ORCT [4].
More specifically, our contributions are as follows:

- The addition of discrete feature values to dynamic programming methods. Current dynamic programming methods only use binary feature values for creating decision trees.
- A way to randomize the splits within a decision tree.
- An experimental study to determine the effects of randomization on the robustness of optimal decision trees.

The rest of this paper is structured as follows. The second section gives an overview of different decision tree learning approaches in the literature. The third section explains the definitions and notations that are used in this work. The fourth section describes the contributions this paper has made. The fifth section covers the experimental setup used and the results that were obtained. The sixth section discusses the reproducibility of our paper. In the last section, we conclude the paper and go into depth about limitations of our method and possible further research.

2 Related works

The decision tree learning method is diverse and has many different variations and approaches. In this section, we give an overview of the relevant decision tree learning approaches in the literature.

Heuristic decision trees
Among decision tree learning algorithms, heuristic approaches are popular for their scalability and fast runtimes. They start off with one node and keep expanding the tree based on some heuristic to evaluate the quality of the split. Examples of heuristic decision tree algorithms are CART [5], ID3 [9] and C4.5 [10].

Optimal decision trees
An optimal decision tree is the best possible decision tree given a certain metric, such as accuracy. Making optimal decision trees has been proven to be NP-complete [8], and their scalability and runtimes are generally inferior to their heuristic counterparts. One benefit that Bertsimas and Dunn [2] has shown is that optimal decision trees have higher out of sample accuracy than heuristic methods. There are multiple approaches to creating optimal decision trees, we will describe some of them in the following paragraphs.

Mixed integer programming
One way to create optimal decision trees is to apply Mixed Integer Optimization (MIO) methods. These methods create decision trees by formulating the problem in a specific manner. They specify a target to minimize, such as a misclassification score, and a set of constraints which the decision tree must adhere to. Then they use a generally available MIO solver such as Gurobi to solve the formulated problem. Some examples of decision trees methods with MIO are papers by Bertsimas and Shioda [3] and Bertsimas and Dunn [2].

Dynamic programming
Another approach for creating optimal decision trees is to make use of dynamic programming. These methods store optimal subtrees in a cache to avoid redundantly recalculating subtrees that were encountered before. Methods that apply dynamic programming include DL8 [7], DL8.5 [1] and MurTree [6].

MurTree
The MurTree [6] algorithm uses exhaustive search to find optimal decision trees. It applies multiple optimizations to improve the runtime. One of the optimizations is that they use dynamic programming to cache optimal subtrees. It also uses upper and lower bounds similar to DL8.5 [1] to prune the search space. Additionally, it introduced a specialized algorithm specifically for trees of depth two, which improved the runtime significantly.

Optimal randomized classification trees
Blanquero et al. [4] use a continuous-optimization method to create optimal decision trees. Integer values in mixed-integer optimization problems typically are slower than continuous values. To remove the integer decision variables in the MIO methods, it introduces randomization to their approach. Instead of deterministically going to the left or right of the tree, instances now have a probability that determines which side of the tree it goes to.

3 Preliminaries

In this section, the necessary theoretical background and the notations used throughout the paper are given.
A feature f is an encoding of information, this can take many forms such as binary, discrete or continuous. When a number of features relate to the same instance, they can be grouped together in a vector called a feature vector. An instance I is a pair composed of a feature vector and a label that represents the class of the instance. A dataset D is a collection of these instances. In a dataset, the order of the features is the same for all instances, i.e. the value of the i-th feature of every instance corresponds to the same property.
Every feature that occurs in a dataset get stored in a set F, and all the possible labels that can be achieved in a dataset, get stored in a set C.
A binary tree is a tree model with the constraint that every node can have at most two children. Every tree model consists of two types of nodes: internal and external nodes (also called non-leaf and leaf nodes). Internal nodes are nodes that have at least one child, and in the context of decision trees are called predicate nodes, while external nodes have no children and are called classification nodes.
A decision tree is a machine learning model that makes use of tree models to label and classify data. It does so by checking the value of a feature at every predicate node, and comparing that to a threshold, the outcome of which determines whether they go to the left or right side of the tree. When it eventually reaches a classification node, it assigns that instance a label. If the label assigned by the classification node does not correspond to the label associated with the instance, a misclassification occurs. Given a decision tree and a dataset, the misclassification score is the amount of instances the decision tree misclassifies in that dataset.
One goal of decision tree learning is to minimize this misclassification score, only if this misclassification score has the lowest amount possible can the decision tree be called optimal. Decision trees have a depth d, which is the maximum number of ancestors that occur, or in other words, the maximum number of predicate nodes one can encounter when going down the tree. They

also have a size n, which is the amount of predicate nodes inside the tree. The size of a binary decision tree has an upper bound of $2^d - 1$, because each node can only have 2 children.
Each decision tree learning approach has different constraints and limitations. In our approach, only discrete features are supported, and any continuous features have to be discretized beforehand. This is done to make it possible to exhaustively construct every possible decision tree, given a constraint on the maximum depth and size.
Dynamic programming uses memoization to improve the runtime. Memoization is the process of storing the results of running a function with certain parameters in a cache. When those parameters occur again, the results of the cache gets retrieved instead of running the whole function again.
In this paper, we have a cache that stores optimal decision trees and one that stores lower bounds. The results from the cache gets retrieved based on branch information. This branch B is a set of internal nodes it passed through and can be divided into two sets: B_L and B_R, which contains the nodes that went to left and to the right respectively.
Randomized trees have probabilistic splits instead of deterministic. The probability to go down the left side of a tree is given by the formula: $p_{It} = F(\mu, \sigma, f_I)$. Here I is the instance in the dataset, t is the internal node, F is the randomization function, μ is the center of the randomization function, σ is the standard deviation of the function and f_I is the value of the feature inside instance I that gets evaluated.
The probability of an instance reaching a certain leaf node can be calculated by taking the product of all the parent probabilities. This probability is denoted as: $P_I = \Pi_{t \in B_L}(p_{It})\Pi_{t \in B_R}(1 - p_{It})$
In this formula, I is the instance, P_I is the probability that this instance reaches that leaf node, B_L is a set of parents which branched left, and B_R is a set of parents that branched right.

To prune the search space, upper and lower bounds are used. When the misclassification score or the lower bound of a tree exceeds the upper bound, we determine the decision tree to be infeasible and prune that part of the search space.

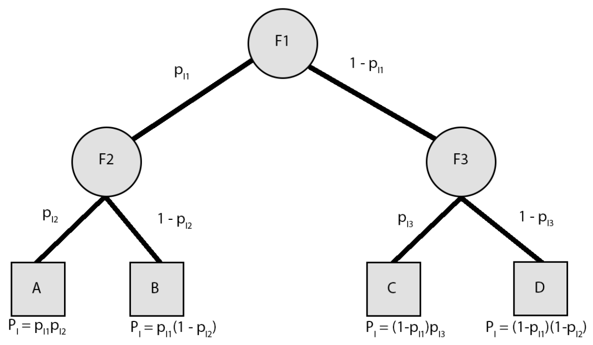

Figure 1: An example of a randomized decision tree.

4 Method

Our algorithm for building optimal decision trees uses the MurTree algorithm as a base, but some modifications were made to it. We added support for discrete feature values instead of just binary. This change is important because the randomization is not possible on binary features. We also introduced randomized splits, which uses a step function to determine the probability of going down a certain branch.

In the following sections, we will be giving a high-level overview of the algorithm, and also go more in depth about the workings of the new functionalities.

4.1 High level overview

In Eq. 1 we give a dynamic programming formulation of the problem. Where $T(\mathcal{D}, B, d, n)$ is the misclassification score when using those parameters.

$$T(\mathcal{D}, B, d, n) = \begin{cases} T(\mathcal{D}, B, d, 2^d - 1) & n > 2^d - 1 \\ T(\mathcal{D}, B, n, n) & d > n \\ \sum_{I \in D} P_I - max\{\sum_{I \in D} P_I * (class(I) = c) : c \in C\} & n = 0 \vee d = 0 \\ min\{T(D, B_L + f, d-1, n-1-i) + T(D, B_R + f, d-1, i) & \\ : f \in F, k \in [0, f_{max}], i \in [0, n-1]\} & n > 0 \wedge d > 0 \end{cases}$$

(1)

There are 4 possible cases that can happen.
In the first case we limit the amount of nodes by taking advantage of the binary tree structure. When the amount of nodes is higher than what is possible given the depth, we set n to the maximum possible nodes in a full binary tree which is $2^d - 1$.
In the second case, we limit the depth by using another property of binary trees, namely that it is impossible for the depth to exceed the number of nodes. Thus, in this case we limit the depth to the amount of nodes.
In the third case, either the depth or the amount of nodes equals zero, which means it is no longer possible to expand the tree. So we create a classification node and have to choose which label to give to that node. This is done by choosing the label which gives the lowest expected value of the misclassification score. We do this by iterating over every instance in the dataset and every possible label to give to the classification node. The term (class(I) = c)) equals 1 if the label of the instance equals label c, otherwise it equals 0. So it sums the probability of reaching the classification node for every instance, but the probability only gets added to the sum if the label of the instance and the node match. When the label with the highest probability gets determined, it subtracts that probability from the total probability to obtain the misclassification score for that leaf node.
In the fourth case, both the number of nodes and the depth is larger than 0. This means we can expand the tree and add another split. It checks every possible split it can make, and selects the one that results in the lowest possible misclassification score. So for every possible feature, for every possible value that feature can take, and for every way to distribute the budget of nodes, it checks the misclassification score by recursively calculating the left and right side of the tree and adding them together.

4.2 Algorithm

We use the MurTree [6] algorithm and implemented changes to support discrete feature values and randomized splits. These changes, however, were not compatible with some of the optimizations that the MurTree algorithm uses and thus had to be removed. One example of this is the removal of the specialized depth two solver. This solver requires that the dataset can be divided, but in our algorithm the dataset can not be altered for the randomization to work.

4.2.1 Discrete feature values

One of the changes we have made to the MurTree algorithm is shown in algorithm 1. The algorithm is identical to the MurTree version except for an additional loop that takes place to iterate over every possible value that a feature can take on.

Algorithm 1 *MurTree.GeneralCase(D,B,d,n,UB)*, this algorithm corresponds to the fourth case in Eq. 1. For the most part identical to the MurTree version.

```
1:  $T_{best} \leftarrow ClassificationNode(D)$
2:  if $LeafMisclassification(D) > UB$ then
3:      $T_{best} \leftarrow \emptyset$
4:  $LB \leftarrow RetrieveLowerBoundFromCache(D, B, d, n)$
5:  $RLB \leftarrow \infty$
6:  $n_{max} \leftarrow min\{2^{(d-1)}, n-1\}$
7:  $n_{min} \leftarrow (n - 1 - n_{max})$
8:  for $splitting\ feature\ f \in F$ do
9:      for $k \in [0, f_{max}]$ do
10:         if $Misclassifications(T_{best}) = LB$ then
11:             break
12:         if $Misclassifications(T_{best}) = LB$ then
13:             break
14:         for $n_L \in [n_{min}, n_{max}]$ do
15:             $n_R \leftarrow n - 1 - n_L$
16:             $UB' \leftarrow min\{UB, Misclassifications(T_{best} - 1\}$
17:             $(T, LB_{local}) \leftarrow MurTree.SolveSubTreeGivenRootFeature(D, B, f(k), d, n_L, n_R, UB')$
18:             if $T \neq \emptyset$ then
19:                 $T_{best} \leftarrow T$
20:             else
21:                 $RLB \leftarrow min\{RLB, LB_{local}\}$
22: if $Misclassifications(T_{best}) \leq UB$ then
23:     $StoreOptimalSubtreeInCache(T_{best}, D, B, d, n)$
24: else
25:     $LB \leftarrow max\{LB, UB + 1\}$
26:     if $RLB < \infty$ then
27:         $LB \leftarrow max\{LB, RLB\}$
28:     $StoreLowerBoundInCache(D, B, d, n, LB)$
29: $ReplaceDatasetForSimilarity(D, B, d)$
30: return $T_{best}$
```

An in depth explanation of the algorithm can be found in the MurTree [6] paper. The changes we have made is the addition of the for loop on line 9, and on line 17 the splitting feature now gets passed along with a feature value k.

4.2.2 Randomization function

The function that we use to calculate the probabilities is a cumulative logistics distribution. Figure 1 and 2 shows examples of this function. This function is centered around a specific value and the standard deviation determines how spread out the function is. When the standard deviation equals zero the function is equivalent to a split in a deterministic decision tree.

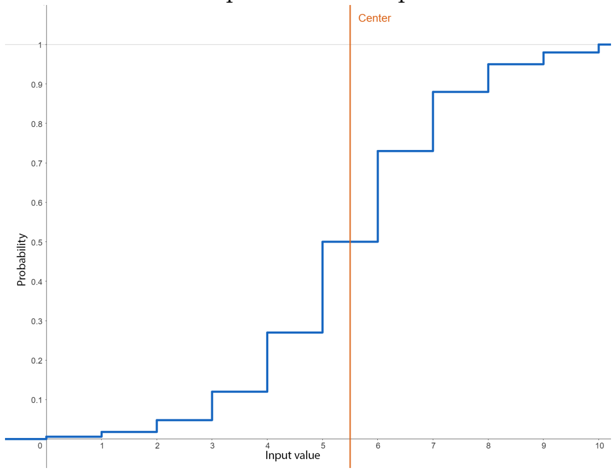

Figure 2: An example of a randomized function with a center of 5 and standard deviation of 1. The y axis gives the probability of going to the left side of the tree, and the x axis gives the input value for the randomization function.

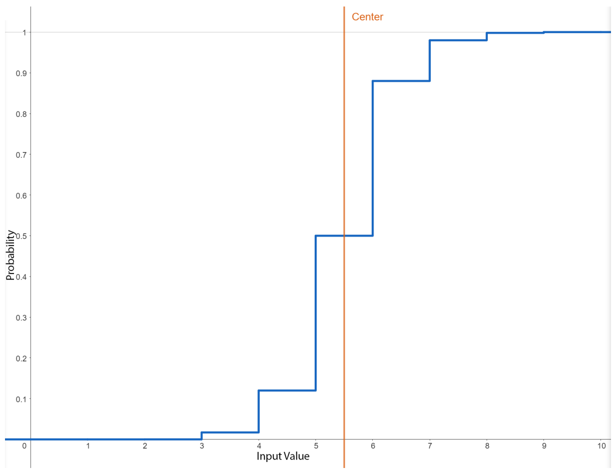

Figure 3: An example of a randomized function with a center of 5 and standard deviation of 0.5. The function is less spread out than the previous figure, which means the randomization effect is weaker.

With the addition of randomization, the misclassification cost has to be calculated in a completely different way. Algorithm 2 shows how the new misclassification cost gets calculated.

Algorithm 2 *MurTree.leafMisclassification(D,B,σ)*, this algorithm corresponds to the third case in Eq. 1. Calculates the lowest misclassifications possible for a single leaf node.

```
// Keep track of the probability of each label
for label $c \in C$ do
    $labelProbability[c] \leftarrow 0$
// Calculate for every instance the probabilty of reaching the node
for instance $I \in D$ do
    // leafProbability becomes the probability to reach the leaf node, to calculate it we multiply each probability of the parent nodes in the branch.
    $leafProbability \leftarrow 1$
    $B_L \leftarrow LeftBranch(B)$
    $B_R \leftarrow RightBranch(B)$
    // All the splitting features that went left in the tree multiply the probability with p
    for splitting feature $f \in B_L$ do
        // The randomization function centers at splitting feature f, has a standard deviation of $\sigma$ and evaluates with the corresponding feature in instance I
        $predicateProbability \leftarrow RandomizationFunction(f, \sigma, I.getFeature(f))$
        $leafProbability \leftarrow leafProbability * predicateProbability$
    // All the splitting features that went right in the tree multiply the probability with (1-p)
    for splitting feature $f \in B_R$ do
        $predicateProbability \leftarrow (1 - RandomizationFunction(f, I.getFeature(f))$
        $leafProbability \leftarrow leafProbability * predicateProbability$
    $labelProbability[I.getLabel()] \leftarrow labelProbability[I.getLabel()] + leafProbability$
$highestProbability \leftarrow 0$
$totalProbability \leftarrow 0$
for label $c \in C$ do
    $totalProbability \leftarrow totalProbability + labelProbability[c]$
    $highestProbability \leftarrow max\{highestProbability, labelProbability[c]\}$
return $totalProbability - highestProbability$
```

In line 3-11, for every possible label we sum the probabilities of the instances with that label to reach this node. Then in line 12-16 we determine which label has the highest chance of reaching that node, and calculate the probability of reaching that node regardless of label. In line 17 it returns the misclassification score by subtracting the probability of reaching that node by the probability of correctly classifying the node.

5 Results/Experiment

In this section, we will compare the robustness of our algorithm when using different parameters for the randomization. With the aim of finding out whether increasing the randomization also results in a more robust decision tree.

5.1 Datasets and processing

The datasets used for training and experimentation are publicly available datasets gathered from the UCI machine learning repository. There are multiple processing steps applied to make the

datasets compatible with our algorithm.
First of all, missing values were handled by removing any feature that has a missing value for any of the instances in the dataset (the column gets removed). Secondly, for any nominal feature, binary dummy features are created that correspond to each possible value it can take on. Lastly, both ordinal and continuous values are discretized using equal width discretization, in this method values get divided into bins. The size of these bins is $(max - min)/n$, where n is the desired number of bins and each bin is the same size. The runtime of the algorithm is greatly affected by the number of bins that features can take on, so a trade-off has to be made between accuracy and speed. For this paper, we have decided to restrict the number of bins to ten.
For the experiments, the datasets get split in two. The first part constitutes 75 percent of the dataset and is used for the training of the decision tree. The second part constitutes the remaining 25 percent and is used for doing the experiments.

5.2 Robustness metric

To measure robustness, we assume that there is an attacker who has knowledge about the underlying decision tree model and tries to alter the classification result by modifying feature values. The attacker can perturb the value either up or down, and will try to change a correct classification into a wrong one. We measure the most misclassifications that an attacker can cause in the out of sample part of the dataset.

5.3 Results

Table 1 contains information about the datasets that were used in the experiments.

Table 1: Datasets that were used, along with dataset size, amount of features and amount of labels

Dataset	|D|	|F|	|C|
anneal	199	11	6
abalone	1044	10	28
crx	172	13	2
haberman	76	3	2

The following graphs show the results of the experiments. The y-axis measures the accuracy, which is the proportion between the correct classifications and the size of the dataset, and the x-axis shows the randomization in the form of the standard deviation that was used to train the decision tree. These graphs contain two lines, the full line is the overall out of sample accuracy and the dashed line is the accuracy after an adversary performed an attack. All the experiments are performed on trees of depth two.

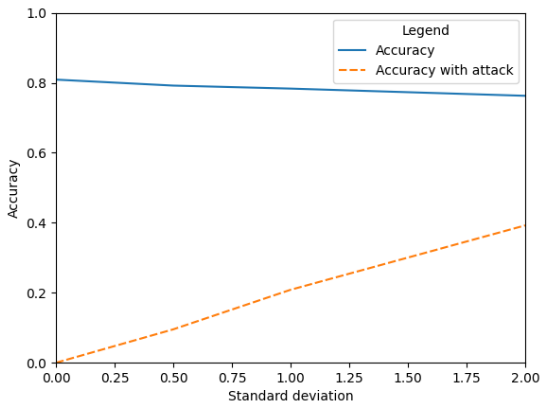

Figure 4: Experiment run on the anneal dataset

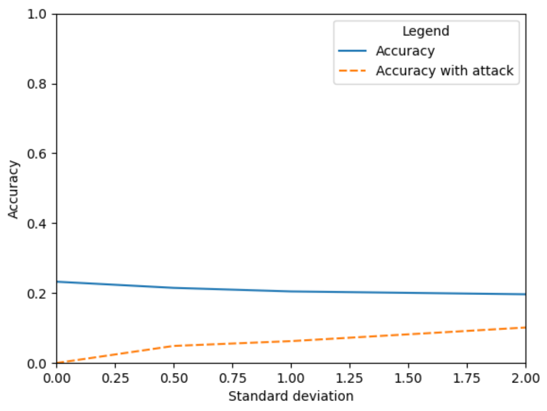

Figure 5: Experiment run on the abalone dataset

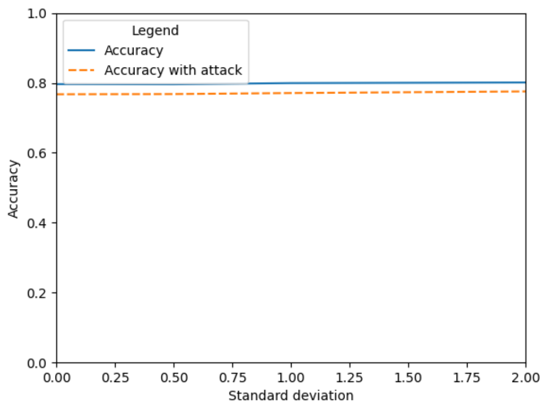

Figure 6: Experiment run on the crx dataset

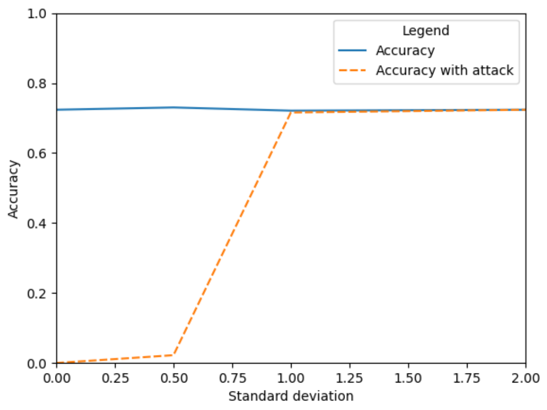

Figure 7: Experiment run on the haberman dataset

6 Responsible research

In this section, we describe which measures we took to ensure that our work is reproducible and done responsibly.

Datasets

The original datasets that were used in this paper are all publicly available. We also described the processing steps applied to the datasets, such that people can obtain the same dataset.

Algorithm

We describe our contributions in the method section, furthermore the source code for MurTree algorithm is available online. Which makes it possible to recreate this algorithm.

Hardware

All the experiments are run on a regular desktop pc, so the algorithm should be able to run on most devices. The results of the experiments are also not affected by variations in hardware, which is the case for measuring runtime, for example.

These facts combined means that people should be able to reproduce the experiments if they so desire.

7 Conclusion

In our research we noticed that adding randomization generally improved the robustness, however the out of sample accuracy drops as a result. So a trade-off has to be made between accuracy and robustness.
Despite these results, some major drawbacks have to be noted. First of all, the randomization is a detriment to the interpretability. The clear-cut decisions of a regular decision tree are replaced with a plethora of probabilities. People can no longer follow how a specific instance reaches a certain leaf node.

Secondly, the runtime and scalability have worsened significantly in comparison to the MurTree algorithm. A lot of optimizations had to be removed to accommodate for the addition of randomization. How to reimplement these optimizations or how to add new ones can be investigated in future work.
Furthermore, the depth of the trees used in the experiments are rather small due to time constraints. More research can be done on how the size of a tree affects robustness.

References

[1] Gaël Aglin, Siegfried Nijssen, and Pierre Schaus. "Learning Optimal Decision Trees Using Caching Branch-and-Bound Search". In: *Proceedings of the AAAI Conference on Artificial Intelligence* 34.04 (Apr. 2020), pp. 3146–3153. DOI: 10.1609/aaai.v34i04.5711. URL: https://ojs.aaai.org/index.php/AAAI/article/view/5711.

[2] Dimitris Bertsimas and Jack Dunn. "Optimal classification trees". In: *Machine Learning* 106.7 (July 2017), pp. 1039–1082. ISSN: 1573-0565. DOI: 10.1007/s10994-017-5633-9. URL: https://doi.org/10.1007/s10994-017-5633-9.

[3] Dimitris Bertsimas and Romy Shioda. *Classification and Regression via Integer Optimization.* Apr. 2007. DOI: 10.1287/opre.1060.0360.

[4] Rafael Blanquero et al. "Optimal Randomized Classification Trees". In: *Computers Operations Research* 132 (Mar. 2021), p. 105281. DOI: 10.1016/j.cor.2021.105281.

[5] Leo Breiman et al. *Classification and Regression Trees.* Cole Statistics/Probability Series, 1984.

[6] Emir Demirovic et al. *MurTree: Optimal Classification Trees via Dynamic Programming and Search.* 2020. arXiv: 2007.12652. URL: https://arxiv.org/abs/2007.12652.

[7] Élisa Fromont and siegfried nijssen siegfried. "Mining Optimal Decision Trees from Itemset Lattices". In: Aug. 2007. DOI: 10.1145/1281192.1281250.

[8] Laurent Hyafil and Ronald L. Rivest. "Constructing optimal binary decision trees is NP-complete". In: *Information Processing Letters* 5.1 (1976), pp. 15–17. ISSN: 0020-0190. DOI: https://doi.org/10.1016/0020-0190(76)90095-8. URL: https://www.sciencedirect.com/science/article/pii/0020019076900958.

[9] Asdrúbal López-Chau et al. "Fisher's decision tree". In: *Expert Systems with Applications* 40.16 (2013), pp. 6283–6291. ISSN: 0957-4174. DOI: https://doi.org/10.1016/j.eswa.2013.05.044. URL: https://www.sciencedirect.com/science/article/pii/S0957417413003424.

[10] Ross Quinlan. *C4.5: Programs for Machine Learning.* Kaufmann, 1993.

[11] Jiawei Su, Danilo Vasconcellos Vargas, and Kouichi Sakurai. "One pixel attack for fooling deep neural networks". In: *CoRR* abs/1710.08864 (2017). arXiv: 1710.08864. URL: http://arxiv.org/abs/1710.08864.