A Novel Approach to Evaluating Decision Tree Learning Algorithms

by

Tianqi Xiao

A thesis submitted to the
Department of Computer Science
in conformity with the requirements for
the degree of Master of Science

Bishop's University
Canada
August 2020

Abstract

Decision tree learning algorithms performance evaluation is essential for building or selecting the optimal algorithm to solve classification problems. Generally, research studies use datasets provided by universities, reputable companies and organizations, or collected by the research teams themselves. However, there are two major problems with this approach. First, the number of datasets used in these studies are generally limited (usually less than 25 datasets are used), which suggests the results may be to a large degree dependent on a specific dataset. Secondly, many traditional metrics rely on cross-validation to measure the correctness of the classification. In this case, the evaluation process is done by estimating how well the inferred model classifies the data examples in the test set without knowing what the actual model is. We recognize these problems and propose a new approach that evaluates the performance of decision tree learning algorithms generically. The underlying idea of this approach is paths tracing and comparing. We assess various decision tree algorithms with our new framework and compare their performance. We also investigate the relation between the inferred trees and the properties of the training datasets.

Acknowledgments

This work would not have been possible without the support, patience, and guidance provided by many people.

First of all, I would like to thank my supervisor, **Stefan D. Bruda**, for his assistance and support. He has been very patient with me whenever I have questions. I also thank him for giving me the freedom to explore various research projects during my degree, and encouraging me to settle into the work I am most passionate about.

Next, I would like to thank **Omer Nguena Timo** for hiring me as a research intern at CRIM (Computer Research Institute of Montreal). During my time there, he and **Florent Avellaneda** provided me with lots of interesting ideas and helped me tackle obstacles in my research project. I learned so much from them and the internship has been rewarding beyond my expectations.

I also thank **Yasir Malik** for his inspiration and advice. He introduced an exciting project to me when I was exploring different research ideas, which leads the work I am presenting in this paper. Moreover, he guided me through many difficulties along the way.

Finally, I want to express my gratitude to my family for their love and support. They always believe in me and support me unconditionally.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

Binary classification is a problem studied in supervised machine learning, where the task is to classify the samples of a given dataset into two predefined subgroups based on some classification rules. This is an important topic in machine learning with significant applications in many fields including medical diagnosis, spam detection, and malware recognition. There are many existing algorithms that are commonly used for binary classification. The use of decision trees is among the most popular choices mainly because they are easy to understand and visualize.

Traditionally, decision tree learning algorithms use heuristic functions to find the best combination of features and their values to construct a tree-like structure representing the classification rule set. Some of the most used decision tree learning algorithms, including ID3 [23], C4.5 [25], and CART [6], minimize the information entropy of different classes guided by some heuristic functions. Even though in general the decision tree learning algorithms infer models of great accuracy, they often fail to find the global optimal solution due to their greedy nature. Moreover, their performance also depends on the properties of the dataset on which the decision tree algorithms are trained, including the size of the dataset, the bias of class labels, the duplication of data examples, and noisy samples. In recent years, exact model inferences are getting increased attention. Algorithms like InferDT [3] are aiming to find the optimal decision trees consistent with the learning datasets. They are known for their ability to produce very accurate models, but very few research studies have been done comparing them to the heuristic-based decision tree algorithms. Therefore, evaluating the performance of the decision trees is crucial for classification tasks. A good evaluation metric helps us better understand the behavior of decision tree learning algorithms.

We propose in our study a new approach that evaluates the quality of decision trees using statistic-based metrics. Decision tree learning algorithms are attempting to infer a decision tree that can best represent a dataset, so our idea is to randomly generate a decision tree that acts as the oracle and produce sets of data

examples from it. Then after training decision tree learning algorithms on the generated dataset, we compare the trained models with the oracle model. We define a degree of equivalence (DOE) between the learned trees and the oracle to describe the similarity between the two models. We use this new approach to assess various decision tree algorithms and compare their performance. We also investigate the relation between the inferred trees and the properties of the training datasets.

The reminder of this thesis is organized as follows. We first address in Chapter 2 the motivation of our research by discussing some common evaluation metrics and their disadvantages. We also present the hypothesis and expectations in practical decision tree algorithm development. We introduce in Chapter 3 our novel evaluation method by explaining the design of the decision tree generators and introducing the concept of equivalence tests between the oracle and the inferred model. We then present our empirical experimental results on evaluating several decision tree learning algorithms (Chapter 4). We conclude the study in Chapter 5.

Chapter 2

Motivations and Hypothesis

2.1 Standard Evaluation Metrics

Traditionally, the evaluation metrics in binary classification problems are developed based on a 2x2 confusion matrix with the row of the table representing the predicted class labels and the column representing the actual class labels. As shown in Table 2.1, the true positive (tp) indicates the number of positive labeled data examples that are correctly classified. Similarly, the true negative (tn) indicates the number of negative labeled data examples that are correctly classified. The false positive (fp) and false negative (fn) denote the number of wrongly classified positive and negative data examples, respectively.

From the confusion matrix, some numerical evaluation metrics such as accuracy, sensitivity, specificity, precision, and recall can be calculated to discriminate the performance of the decision trees (Table 2.2). According to previous studies [9, 12], accuracy is the most chosen metric when optimizing decision trees due to its simplicity. However, it has some major disadvantages, especially when dealing with imbalanced data. Under these special circumstances the minority class has less impact on the accuracy score, whereas the majority class has an overwhelming impact [11]. Similarly, precision, recall, and the F1 Score all suffer from biased performance as they overlook the importance of correct negative predictions [22].

One alternative metric that mitigates the problems arising from evaluating imbalanced datasets is the Area under ROC Curve (AUC) [5]. ROC, which stands for Receiver Operating Characteristics, is plotted as True Positive Rate (Sensitivity) on

	Positive Class	Negative Class
Positive Prediction	tp	fp
Negative Predication	fn	tn

Table 2.1: Confusion Matrix for Binary Classification

Metric	Formula	Description
Accuracy	$\frac{tp+tn}{tp+fp+tn+fn}$	The proportion of correctly classified cases from the total number of cases
Precision	$\frac{tp}{tp+fp}$	The proportion of true positive cases from the cases that are predicted as positive
Recall	$\frac{tp}{tp+fn}$	The proportion of true positive cases from the cases that are actually positive
F1 Score	$2 * \frac{\text{precision}*\text{recall}}{\text{precision}+\text{recall}}$	The harmonic mean between precision and recall

Table 2.2: Common Evaluation Metrics for Binary Classification Based on Confusion Matrix

the y-axis against False Positive Rate ($1 -$ specificity) on the x-axis. The area between the ROC curve and the x-axis indicates how good the classification model is at distinguishing examples belonging to different classes. AUC is well-recognized as an excellent metric when discriminating and comparing decision trees [7, 31].

A number of comparative studies on decision trees exist [1, 13, 16, 18, 26]. Nevertheless, to this day scientists are still in search of a better way to generically compare the performance between different decision tree algorithms. It is particularly important to evaluate the decision trees generically when developing a new decision tree learning algorithm or aiming to improve an existing one. Previous comparisons were mostly performed using datasets provided by universities, reputable companies, and organizations, or collected by the research teams themselves [17]. There are two major problems with this approach. First, the number of datasets used in these studies are generally limited (usually under 25 datasets are used). When training on such a small number of datasets, the obtained results may be more dependent on a specific dataset. Secondly, many traditional metrics rely on cross-validation to measure the correctness of the classification. In this case, the evaluation process is done by estimating how well the inferred model classifies the data examples in the test set without knowing what the actual model is. To avoid these problems when discriminating decision tree learning algorithms, we propose a path tracing and comparing evaluation method which directly tests the equivalence between trained models and oracle models.

2.2 Hypothesis

In this study, we plan to examine the behavior of the decision trees by directly comparing the trained model to an oracle model, where the training dataset is randomly generated from the oracle. Hence, we are expecting to observe the relation between the size of the dataset and the correctness of the trained model. Our hypothesis is that with an increased number of data examples the decision tree learning algorithms produce more accurate models. The idea behind this hypothesis is that for a random dataset, more information can possibly be included in the resulting decision tree when the size of the dataset grows.

Additionally, we expect the accuracy of the "best" decision tree learning algorithms to increase more aggressively than the others. In this case, we define the best learning algorithm as the one that produces an accurate model with the least number of data examples. We are expecting the exact decision tree inference algorithm (InferDT) to have the best overall performance.

With our new approach, we also intend to investigate in the effect of the depth of the oracle on the performance of decision tree learning algorithms with the same number of data examples in the dataset. We predict that the learning algorithms need more data examples in the training dataset to infer an accurate model if the oracle grows deeper.

Chapter 3

Evaluation by Path Tracing and Comparing

As mentioned previously, our approach to evaluating the decision trees consists of generating a random oracle tree, generating a random training dataset, and performing an equivalence test between the inferred tree and the oracle. The first two stages occur before the training process, and the tree equivalence testing occurs after the decision tree learning algorithm finishes inferring the model.

3.1 Definitions

We first define the notations used in the following sections. For a given dataset, we identify $F = \{f_0, f_1, ..., f_{n-1}\}$ as the feature set, where the domain of the ith feature is denoted by D_i. We then use S for the set of training data examples and $C = \{c_0, c_1\}$ for the two labels in the target class; each example s_r is a vector in the feature space $\prod_{i=1}^{n} D_i$.

Since a multi-variable feature can be easily converted to multiple binary-valued (Boolean) features, we only consider Boolean features in this study. As shown in Table 3.1, a feature f_i which contains v different variables can be converted to v Boolean feature with each feature representing one possible variable of feature f_i. Consequently, the domain of each new feature contains only Boolean values, *false* and *true*.

We use $\mathcal{T}$ to denote the decision tree model inferred based on the training set S. Each path P in the tree consists of some nodes t and a single leaf lf. Each node in the tree represents a feature, and each leaf represents a class. We denote the depth of each path by k and the number of total paths as p. If all paths in the tree have the same depth k and ever node in the tree contains two children, then we call this tree a *perfect tree* [3]. Assuming we have a dataset containing the set of features from Table 3.1b, one example of a possible perfect tree is shown in Figure 3.1.

Features	f_0	f_1
Values	0, 1, 2	0, 2, 3, 5

(a)

Features	f_2	f_3	f_4	f_5	f_6
Meanings	$f_0 < 2$?	$f_0 < 1$?	$f_1 < 5$?	$f_1 < 3$?	$f_1 < 2$?
Values	*false*, *true*	*false*, *true*	*false*, *true*	*false*, *true*	*false*, *true*

(b)

Table 3.1: Conversion from Multi-variable Features to Binary Features

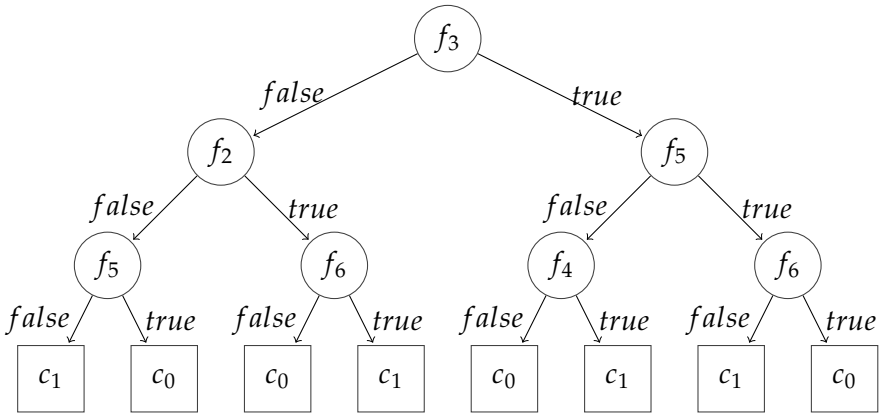

Figure 3.1: An example of perfect tree using the feature set in Table 3.1b

A decision tree is essentially a set of non-contradicting rules, where every path in a decision tree represents one unique combination of feature values. Taken the right-most paths in Figure 3.1 as an example, it can be described as the following rule: for a data instance, if its values for f_3, f_5, and f_6 are all *true*, then this instance should be classified as c_0. An oracle is a decision tree on which all the examples in the dataset can be mapped correctly. We use O to denote the oracle, which is essentially a decision tree. The oracle is randomly produced by a generator, then used as a reference when creating training datasets and comparing them with the inferred models. The design of the oracle generator is explained in the next section.

3.2 The Random Oracle Generator

To produce the oracle tree the generator first takes inputs from the user, specifying the desired number of features n, the depth of the paths k, and the number of variables v for each feature. As we explained in the previous section, because a feature with multiple variables can be translated to numerous binary features, the default

Algorithm 1 Oracle Tree Generator

Input: n, k, v
Output: O

Create F with n features $\{f_0, f_1, ..., f_{n-1}\}$
for each f_i in F **do**
    Create D_i with v values
end for
Create root node t_0 in O
if $|F| \neq 0$ **then**
    $lvl = 0$ ▷ lvl denotes depth of O
    EXPANDTREE(t_0, F, lvl)
end if
return O

procedure EXPANDTREE(t, F_s, lvl) ▷ $F_s \subseteq F$
    if $|F_s| \neq 0$ **and** depth(O) $< k$ **then**
        random $(0, n) \rightarrow i$ ▷ choose random from F_s
        feature(t) $\leftarrow f_i$
        for each d_{il} in D_i **do**
            create $t_l \rightarrow$ children(t)
            EXPANDTREE(t_l, $F_s \setminus f_i$, $lvl + 1$) ▷ "$\setminus$" denotes exclusion
        end for
    else if depth(O) $= k$ **then**
        for each j in $|C|$ **do**
            create $t_j \rightarrow$ children(t)
        end for
    end if
end procedure

number of variables for each feature is set to 2. To better demonstrate the usage of the generator, we design the generator to always produce perfect trees, which means that all the rules described by the tree will have an identical number of features. Note that though the length (depth) of each path is the same, if the depth k is smaller than the size of feature sets n, then each path may contain different feature subsets. An example of this situation is shown in Figure 3.1. Though the depth of the left-most path and the right-most path is the same (3), the feature subset of the left-most path (f_2, f_3, f_5) is different than the feature subset of the right-most path (f_3, f_5, f_6).

The generator starts by initializing the feature set and assigns values to each feature. The feature set is an array of feature pointers containing the name of each

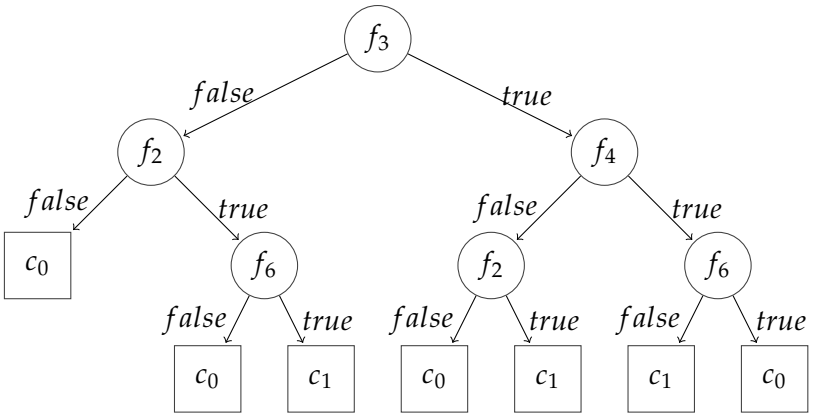

Figure 3.2: An example of a imperfect tree which does not meet the depth requirement

feature, an array of values for each feature, and the number of values for each feature. Then, the generator creates the root node of the tree and randomly assigns a feature to it. The tree expands from the root node by adding nodes recursively and choosing features in a random fashion until the prescribed depth k is reached. Each node is labeled with one feature which has not yet been selected by its ancestors, for indeed duplicate features in a single path will cause a contradiction. To avoid duplication, a number array is carried with the node indicating the availability of features at each stage. Moreover, every internal node links to its parent and its children, and it contains an index number that represents the chosen value for the feature that its parent labeled with.

When a path reaches the depth of k the generator creates a leaf node and attaches it to the last internal node. The class label of the leaf is also randomly assigned, but for multiple leaves attaching to the same node at least one leaf has a different class label. Taken Figure 3.1 again as an example, if the left-most path has a leaf labeled c_0, then f_5 would become useless in the tree and so we would be able to re-plot the tree as in Figure 3.2. It is obvious that the left-most path in this new tree does not satisfy the depth requirement.

Once the oracle tree is fully constructed, the generator will output the tree, which can now act as a reference when producing training sets and evaluating decision trees. Pseudo-code demonstrating the rule set generation algorithm is presented in Algorithm 1.

3.3 The Random Dataset Generator

After obtaining an oracle tree from the oracle generator, our next goal is to create a training dataset generator. This generator follows the rules from the oracle

and randomly produces a dataset with a pre-defined number of instances. In our study, the generated training dataset S is *consistent* with the oracle, meaning every example $s_r \in S$ can be correctly mapped to the oracle trees.

To better understand possible behaviors of the generator, it is necessary to first investigate the relations between the number of features, the depth, and the number of unique data instances. We assume that we have a feature set with n features and every feature has a binary-valued domain. Since each data example is a vector in the feature space and the total number of unique combinations of feature values is 2^n, the number of unique data instances in the training set is 2^n.

If the oracle has a depth of $k = n$, because the oracle is a perfect tree and each path contains all n features, then each data example represents one path in the tree. However, if the prescribed depth $k < n$, then each path in the oracle has only k features, which means those features not present in this path do not impact the classification result. These features are said to be *free attributes*. Note that even though a feature is free in a specific path, it still cannot be classified as irrelevant. Indeed, different paths contain different feature subsets, and the free attributes of one path may well be essential components of other paths. For each path P, the number of free attributes is $n - k$, whereas the number of paths in the oracle is k. While the total number of unique data instances remains the same, each path is now represented by 2^{n-k} unique data examples.

We propose two different design methodologies for the dataset generator: generating completely random datasets and generating uniquely random datasets.

3.3.1 Completely Random Dataset

To generate completely random datasets the generator takes an input specifying how many data examples m to produce. Because the uniqueness of data instances is not considered in this type of dataset, the only restriction that can be imposed on m is that $m \in \mathbb{N}$. The generator then creates an empty 2D array with every row referring to one data instance, and every column representing a feature or a class. Meanwhile, the generator reads the oracle tree into the program and maps all the information to a tree structure. This step can be viewed as a reverse process of the outputting step of the oracle generator.

After having the tree and the plain dataset ready, the generator starts implementing a recursive random walk. For each data example in the dataset starting from the root, the generator randomly chooses one of its children. Because each node is labeled by a feature and indexed by the value of its parent's features, the value of this feature is updated accordingly for the current instances. The generator continues to walk through the tree from the top to the bottom until a leaf node is reached. Once the class label is updated for the current data example, the generator then moves to the next data example and repeats the random walk process to fill the entire dataset. The recursive process is formally presented in Algorithm 2.

Algorithm 2 Completely Random Dataset Generator

Input: m, O, F
Output: S

Create S with n features $\{f_0, f_1, ..., f_{n-1}\}$ and m data examples
for each s_r in S **do**
    RANDOMWALK(s_r, F, t_0) ▷ t_0 denotes the root of O
    for each $s_r[i]$ in s_r **do**
        if $s_r[i]$ is empty **then**
            random(0, $|D_i|$) $\rightarrow l$ ▷ choose random in D_i
            $s_r[i] \leftarrow d_{il}$
        end if
    end for
end for
return S

procedure RANDOMWALK(s_r, F, t)
    if feature(t)= f_i **then**
        random(0, $|D_i|$) $\rightarrow l$
        $s_r[f_i] \leftarrow d_{il}$
        RANDOMWALK(s_r, F, t_l) ▷ t_l is a child of t
    else if label(t) $= c_0$ **then**
        label(s_r) $\leftarrow c_0$
    else if label(t) $= c_1$ **then**
        label(s_r) $\leftarrow c_1$
    end if
end procedure

If the oracle's depth k is smaller than the number of features n, then some values in the dataset would not be updated by the recursive random walk simply because those free attributes do not exist in certain paths. In such a case, for each instance, the generator stochastically chooses a possible value of the free attributes and inserts it into the corresponding position. Finally, the completely random dataset will be outputted as CSV file.

The completely random dataset simulates a real-life dataset, in which many data examples are duplicated. Depending on the number of instances, it is also possible that some paths have no representation in the dataset. With this type of dataset, we can observe the performance of different decision trees when inferring models with incomplete information.

3.3.2 Uniquely Random Dataset

Similar to generating completely random datasets, to produce a uniquely random dataset the generator takes input m as the size of the dataset and then creates an empty 2D array for the dataset. However guaranteeing the uniqueness of data examples is now a critical task, so the number of instances m cannot exceed the total number of distinct examples 2^n. That is, $m \in \mathbb{N}$ and $m < 2^n$. The empty 2D array the generator create is also a bit different than the previous one. Because producing a uniquely random dataset requires the full knowledge of the oracle, instead of using recursive a random walk, the generator needs to allocate 2^n rows in the dataset to record the entire feature space. Each data example is labeled with the id of their associating path, denotes by $p_i d$.

After reading the oracle into the program and transforming it to a tree structure, the generator walks through the paths, one by one and in order. Proceeding from the left-most path, the generator follows the path and updates the value of each feature in the first data example accordingly. When a leaf node is reached the generator updates the class label and checks if the data example contains missing values. If the data example is complete (which means $k = n$), then the generator records the index of the path in the index array and moves on to the next path and updates the next instances. Because the oracle is set to be a perfect tree, all paths have the same depth and the generator is not required to check for missing values again for the remainder of the tree. In contrast, if some missing values are found in the first instances (which means $k < n$), then the generator copies the values of the first data instance to the next $2^{n-k} - 1$ instances. After inserting all possible combinations of values for the free attributes, these 2^{n-k} instances then complete the data representations of the first path. All the indexes referring to these instances get updated with the index of the first path. Then the generator repeats the process until all the paths are visited. The algorithm for finding the dataset with all distinct examples is illustrated in Algorithm 3.

When outputting the dataset the generator first compares the number of data examples m and the number of paths b. If $m < b$, then the generator randomly selects one instance of each path and stochastically outputs m of them into a CSV file. If $m = b$, then the generator randomly selects one instance of every path and outputs all the selected data into a CSV file. Finally, if $m > b$, the first b instances will be select similarly as in the case where $m = b$; then the remaining $m - b$ instances will be selected randomly throughout the full dataset. Note that each instance can only be select once to avoid duplicates. The pseudo-code of the uniquely random dataset algorithm is given in Algorithm 4.

The goal of generating a uniquely random dataset is to provide as much information in the dataset as possible without having redundant data examples. With such a dataset, we are able to discover the effect of having different representations of the same path on the accuracy of the classification.

Algorithm 3 Full Dataset with All Possible Data Examples

Input: O, F, n, k
Output: S', p_{id}

Create S' with n features $\{f_0, f_1, ..., f_{n-1}\}$, and 2^n data examples
$p_{id} \leftarrow 0$ ▷ p_{id} denotes the ID of current path
ALLEXAMPLES(S', F, t_0, p_{id}) ▷ t_0 denotes the root of O
p_{id}− ▷ avoid over increment
return S', p_{id}

procedure ALLEXAMPLES(S', F, t)
    if feature(t)= f_i **then**
        for each d_{il} in D_i **do**
            $s_r[i] \leftarrow d_{il}$
            ALLEXAMPLES(S', F, t_l) ▷ t_l is a child of t
        end for
    else if label(t) = c_0 or label(t) = c_1 **then** ▷ check if a leaf is reached
        label(s_r) ← label(t)
        path(s_r) = p_{id} ▷ assign s_r to path p_{id}
        COMBINATION(S', F, t)
        p_{id}++
    end if
end procedure

procedure COMBINATION(S', F, t)
    $ct \leftarrow 0$ ▷ ct denotes a counter
    $ms \leftarrow |n - k|$ ▷ ms denotes total num of free variables
    $ps \leftarrow 0$ ▷ ps denotes the num of free variables encountered
    while $ct < pow(2, ms)$ **do**
        for each $s_r[i]$ in s_r **do**
            if $s_r[i]$ is not empty **then**
                $s_{r+ct}[i] \leftarrow s_r[i]$
            else if $s_r[i]$ is empty **then**
                $ct\%(int)(pow(2, ps)) \rightarrow l$
                $s_{r+ct}[i] \leftarrow d_{il}$) ▷ % denotes modulo operation
                ps++
            end if
        end for
        label(s_{r+ct}) ← label(s_r)
        path(s_{r+ct}) ← path(s_r)
        ct++
    end while
    $r = r + ct$
end procedure

Algorithm 4 Uniquely Random Dataset Generator

Input: m, F, S', p_{id}
Output: S

Create S with n features $\{f_0, f_1, ..., f_{n-1}\}$ and m data examples
$pct \leftarrow 0$
while $pct <= p_{id}$ **do**
    random(0, $|S'|$) $\rightarrow r$ if $path(s_r) = pct$
                                ▷ choose random in S' with path label pct
    add(s_r, S)
    remove(s_r, S')   ▷ avoid duplication
    pct++
end while
if $m > b_{id}$ **then**
    $tmp \leftarrow m - b_{id}$
    while $tmp > 0$ **do**
        random(0, $|S'|$) $\rightarrow s_r$
        add(s_r, S)
        remove(s_r, S')
        tmp–
    end while
else if $m < b_{id}$ **then**
    $tmp \leftarrow b_{id} - m$
    while $tmp > 0$ **do**
        random(0, $|S|$) $\rightarrow s_r$
        remove(s_r, S)
        tmp–
    end while
end if
return S

3.4 Equivalence Test

Now that we have a training dataset generated based on the oracle and trained the decision trees on the dataset, we need to evaluate the correctness of the trained model. Adopting the tracing technique from model-based testing in formal verification [29], we design a path tracing equivalence tester E. We say that a path in the decision tree T is *equivalent* to the path in the oracle O if the rule set they are representing do not contradict each other. By the same logic, the tree T is considered *equivalent* to the oracle O if the decision tree classifies all the examples generated by the oracle correctly.

The equivalence tester E consist of two pointers, with one of the pointer ptr_1

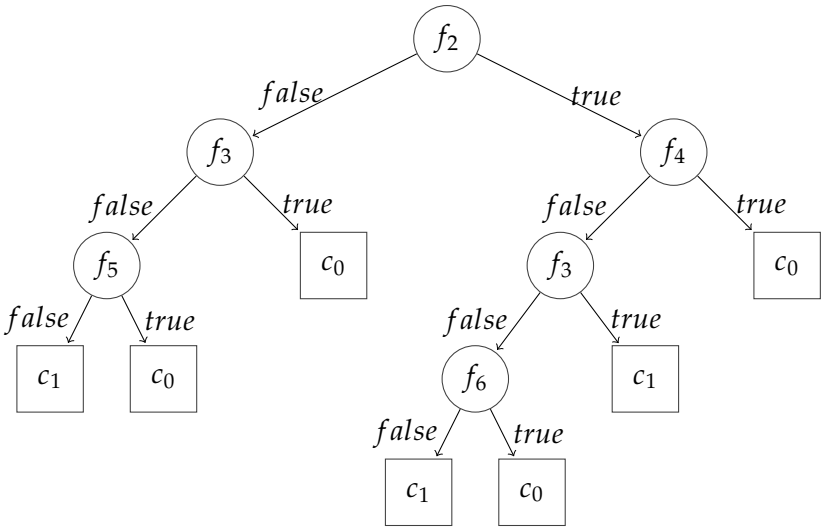

Figure 3.3: An example of a inferred decision tree

tracing the nodes in oracle O and the other pointer ptr_2 tracing the nodes in the inferred model. In addition, a cache of values *cache* stores the features and values visited by the pointer in the oracle tree. The tracing starts at the root node and follows the oracle node by node, and path by path. While tracing, we record the total number of paths visited *total* and the number of consistent paths *succ*. We define the new evaluation metric *degree of Equivalence* (DOE) as the ratio of consistent paths over the total number of paths, that is:

$$DOE = \frac{succ}{total} \tag{3.1}$$

DOE ranges from 0 to 1, with a higher number indicating a better inference.

We first place one pointer ptr_1 at the root of the oracle, and the other pointer ptr_2 at the root of the inferred model. The value cache *cache* is empty because none of the nodes has been visited yet. The total number of paths visited *total* and the number of consistent paths *succ* are both 0. The equivalence test is a recursive procedure and is shown in Algorithm 5.

This recursive process guarantees that no duplication occurs when counting the paths, because we are using the oracle as reference and always trace the inferred model accordingly. Once the process is complete we calculate the degree of equivalence (DOE) of the decision tree by dividing the number of consistent paths by the total number of paths.

To demonstrate the evaluation process we consider the following example. Assume that a dataset S is generated based on the tree in Figure 3.1, which is the oracle O in our example. Let some decision tree learning algorithm produce the model T shown in Figure 3.3 after training on S.

Algorithm 5 Equivalence Test

Input: O, T
Output: DOE

```
1: ptr1 ← o0, ptr2 ← t0        ▷ ptr1 and ptr2 points to root of O and T respectively
2: total ← 0, succ ← 0
3: ScanTree(ptr1, ptr2, cache, total, succ)
4: DOE ← succ/total
5: return DOE
6:
7: procedure ScanTree(ptr1, ptr2, cache, total, succ)
8:     if ptr1 not leaf and ptr2 not leaf then
9:         for each vl in D_(feature(ptr1)) do
10:            ptr1 ← o_vl                          ▷ move ptr1 to the child node o_vl
11:            add((feature(ptr1), vl), cache)      ▷ add the feature-value pair to cache
12:            if feature(ptr2) in cache then
13:                vf ← cache.get(feature(ptr2))    ▷ get the value of feature(ptr2)
14:                ptr2 ← t_vf                      ▷ move ptr2 to the child node t_vf
15:            end if
16:            ScanTree(ptr1, ptr2, cache, total, succ)
17:        end for
18:    else if ptr1 is leaf and ptr2 not leaf then
19:        if feature(ptr2) in cache then
20:            vf ← cache.get(feature(ptr2))
21:            ptr2 ← t_vf
22:            ScanTree(ptr1, ptr2, cache, total, succ)
23:        else if feature(ptr2) not in cache then
24:            for each vf in D_(feature(ptr2)) do
25:                ptr2 ← t_vf
26:                ScanTree(ptr1, ptr2, cache, total, succ)
27:            end for
28:        end if
29:    else if ptr1 not leaf and ptr2 is leaf then
30:        for each vl in D_(feature(ptr1)) do
31:            ptr1 ← o_vl
32:            add((feature(ptr1), vl), cache)
33:            ScanTree(ptr1, ptr2, cache, total, succ)
34:        end for
35:    else if ptr1 is leaf and ptr2 is leaf then
36:        total+ = 1                               ▷ add this path to total
37:        if label(ptr1) = label(ptr2) then succ+ = 1   ▷ add this path to succ
38:        end if
39:    end if
40: end procedure
```

In what follows we represent the three values manipulated by the algorithm as a triplet $\langle prt_1, ptr_2, cache \rangle$ and we refer to this triplet as the *scanner* data structure. We then initialize ptr_1 to the root of the oracle, ptr_2 to the the root of the decision tree, and an empty *cache* stores values when we start to visit nodes. The *scanner* structure is thus initialized as $\langle f_3, f_2, \{\} \rangle$. We also initialize both *total* and *succ* to 0.

The algorithm first proceeds on the left-most path of O by choosing the value $false$ for f_3 and moving the pointer to the child with f_2 as the label. We update *scanner* with the new feature and the selected value of the visited feature, which gives $\langle f_2, f_2, \{f_3 = false\} \rangle$. Because the feature label f_2 of the node that ptr_2 points to does not exist in *cache* yet, ptr_2 stays at the same position. We then move the pointer ptr_1 to the next node on this path, which is f_5, by choosing the value $false$ for f_2. The updated *scanner* is $\langle f_5, f_2, \{f_3 = false, f_2 = false\} \rangle$. We notice that f_2 is present in *cache* now, so ptr_2 can move to the child satisfying $f_2 = false$, which is the node labeled f_3. Again since f_3 exist in *cache*, ptr_2 can move to the node labeled f_5. The new *scanner* becoming $\langle f_5, f_5, \{f_3 = false, f_2 = false\} \rangle$. After moving ptr_1 to the left child of the current node and updating *cache* and ptr_2 accordingly, *scanner* becomes $\langle c_1, c_1, \{f_3 = false, f_2 = false, f_5 = false\} \rangle$. Since both pointers reach the leaves on their respective tree and the resulting class labels are the same, we add this path to both *total* and *succ*. We repeat this process path-by-path and so we are able to scan both trees thoroughly without duplication.

Chapter 4

Empirical Evaluation of Decision Tree

To test the effectiveness of our novel evaluation metrics, we now select a few popular decision tree learning algorithms and compare their performance in terms of DOE using our equivalence test.

4.1 Objective Decision Tree Learning Algorithms

We selected four heuristic-based learning algorithms namely, ID3, J48, simpleCART, and RandomTree, and one exact model inference algorithm, InferDT.

4.1.1 ID3

ID3, which stands for Iterative Dichotomiser 3, is a basic yet powerful decision tree learning algorithm invented in 1986 by Ross Quinlan [23]. The idea of ID3 is to construct a decision tree by using a heuristic-based greedy search algorithm to test each feature in the data subset at each node. Starting with the root node, with the input of the entire training set S, the learning algorithm selects the *best* feature to split S into subsets $S_0, S_1, \ldots$. Each child node will have one data subset attach to it. The splitting process repeats for all successive nodes until all data examples in the training set are correctly classified or a stopping criterion is satisfied.

The selection of the *best* feature at each node is critical in decision trees. In information theory, *entropy* is a measure of uncertainty of the outcome when a selection choice is made [27]. In the context of classification, entropy can be defined as the measure of information impurity in a collection of data examples. For a feature f_i with v values and the target class C, the entropy of class C is:

$$H(C) = \sum_{j}^{|C|} -P_C(c_j) log_2 P_C(c_j) \tag{4.1}$$

where $P_C(c_j)$ is the probability that a randomly picked example belongs to class c_j, which equals to the proportion of examples belonging to class c_j in S.

If the feature f_i has v possible values, then the dataset S can be divided into v subsets using f_i. Let S_x be the subset of the of S in which all data examples have value x for f_i. Let S_{xj} denotes the number of examples in S_j labelled with c_j. The expected entropy after the partition by f_i is then:

$$H(C_A) = \sum_{x}^{v} H(C_x) \frac{\sum_{j=1}^{|C|} S_{xj}}{S} \tag{4.2}$$

where $\frac{\sum_{j=1}^{|C|} s_{xj}}{S}$ is the weight of the x^{th} subset in S.

Entropy ranges from 0 to 1. If all examples in the data sample belong to the same class, the entropy value is 0. Conversely, if the proportion of examples belonging to each distinct class are equal, then the entropy value is 1.

Based on the entropy of the current dataset and expected entropy after selecting f_i, we can calculate the information gain as follows:

$$InfoGain(f_i) = H(C) - H(C_A) \tag{4.3}$$

ID3 considers the feature with the largest information gain to be the *best* feature, and this feature will be used to split the current node. This approach tries to minimize the size of the tree; however, due to its greedy nature, the results may not be optimal.

As an early invention in the decision tree family ID3 does not support pruning, so the tree expands fully until all examples in the data subset have the same class label. Moreover, ID3 can only handle features with nominal values.

4.1.2 J48

J48 is a Java implementation of the C4.5 algorithm [25] in WEKA [30]. C4.5 is developed based on the ID3 learning algorithm so it has a similar design which also employs the concept of information entropy when constructing the decision tree. However, instead of using information gain, which tends to favor features with larger value sets, C4.5 uses the *Information Gain Ratio* [24] to mitigate this problem. The idea behind Information Gain Ratio is normalizing the information gain by the entropy of feature values when partitioning the sample dataset, which can be formally presented as follows:

$$GainRatio(f_i) = \frac{InfoGain(f_i)}{\sum_{x=1}^{v} -\frac{S_x}{S} log_2 \frac{S_x}{S}} \tag{4.4}$$

C4.5 selects the feature that yields the highest information gain ratio to split the current node. As the successor of ID3, C4.5 can handle features with both nominal and continuous values, and also data examples with missing information.

In contrast to ID3, the standard C4.5 algorithm features a pruning process. Pruning, or *post-pruning*, is a technique used in decision tree learning algorithms to reduce the effect of data uncertainty by removing parts of the inferred tree that are statistically trivial [19]. Because in our approach the datasets generated by the oracle are deterministic and contain no noise, we are going to evaluate the performance of J48 without the pruning stage.

4.1.3 simpleCART

simpleCART is a Java implementation of the Classification and Regression Trees (CART) algorithm [6] in WEKA. CART was first introduced in 1984 and it has been a popular choice of classification tool ever since. Like ID3 and C4.5, CART uses heuristic functions for sample impurity calculation to split the nodes and so grow the tree. However, instead of using entropy, CART embraces another definition of impurity namely, *Gini impurity*. In this definition, the level of impurity is measured as the error rate of a randomly selected class label at each node [8]. The impurity score obtained from this measure is called *Gini index*, and is computed as follows:

$$GINI(t) = 1 - \sum_{j=1}^{|C|} P^2(c_j) \tag{4.5}$$

where t denotes the objective node and P_C denotes the probability of a randomly selected data example belonging to class c_j.

Similar to C4.5, the standard CART algorithm can handle features with nominal, numerical, and missing values. It also includes a pruning process. Again, because our generated datasets are consistent with the oracle, the simpleCART with no pruning stage is used in our experiments.

4.1.4 RandomTree

RandomTree is an ensemble algorithm implemented in WEKA. It combines the idea of single tree models with random forests [10]. When inferring a tree model, it investigates a feature sub-space with K randomly chosen attributes at each node. Then each node is split using the best feature of the chosen subset at that node [15]. Usually, the number of features K is defined as follows:

$$K = int(log_2(\#features) + 1) \tag{4.6}$$

The RandomTree learning algorithm can be trained on datasets with both nominal and numerical values, and it does not support pruning.

4.1.5 InferDT

Recently, several exact algorithm to infer decision tree have been introduced, such as InferDT [3] and DL8.5 [21]. These algorithms infer optimal decision trees consistent with the set of learning dataset. Several definitions of optimality have been used, but the main optimality criteria used are: a tree with a minimum number of nodes [4, 20]; a tree with a minimum depth [3]; or a tree with a minimum number of nodes among the trees with a minimum depth [3].

The problem of inference of optimal decision trees is known to be NP-complete [14] and it is also hard to approximate up to any constant factor under the assumption $P \neq NP$ [28]. Despite the complexity of the problem, recent approaches can infer an optimal decision tree in a reasonable time for small decision trees [2, 3]. Moreover, from a theoretical point of view, and in accordance with the principle of parsimony, optimal decision trees should be more accurate than non-optimal decision trees.

In our experiments we evaluate the quality of these optimal decision trees by considering only the maximum depth as a criterion of optimality. We evaluate the performance of InferDT, but we expect that similar results are obtained for any other exact algorithm.

4.2 Experiments and Results

We are aiming to answer the following questions during our empirical experiments:

Question 1 With the same number of features and depth in the oracle, what is the relation between number of data instances in the training set and the correctness of the model inferred by the learning algorithms?

Question 2 With the same number of features, how does the depth in the oracle impact the performance of decision tree learning algorithms in terms of DOE?

Question 3 With the same number of features and depth in the oracle and same number of instances in the training set, which decision tree learning algorithm infers the most accurate model?

Question 4 With the same number of features and depth in the oracle, what is the difference when training on completely random dataset and on uniquely random dataset?

To answer the above questions we perform a number of atomic experiments. Each such an experiment takes three parameters as inputs: the number n of features, the maximal depth k of an oracle, with $1 \leq k \leq n$, and the size m of the

training dataset. An experiment proceeds in four steps. First, we generate an oracle with Algorithm 1. A generated oracle uses n features and the length of each of its path is k. Two paths in the same oracle can use different features and the order of occurrence of the features on two distinct paths can be different. In the second step, we randomly generate a training dataset with m data examples from the oracle. Third, for each generated training dataset we use the the L-th learning algorithm under evaluation to generate the L-th learned tree. In the fourth step, we determine the DOE of the generated learned trees by performing an equivalence test for each learned tree against the oracle.

For each set of input values (n, k, m, L), we perform a number tl of experiments and average the DOE values obtained from the equivalent tests. The purpose of calculating average DOE is to minimize the potential performance bias when the respective algorithm trains on a specific dataset. In our experiment, we set tl to 100 when evaluating the performance of ID3, J48, simpleCART, and RandomTree; on the other hand we set tl to 20 when inferring tree models using InferDT because the results from optimal tree inference are more stable.

Figure 4.1 uses line plots to illustrates the performance difference between decision tree learning algorithms for different input depth, where the number of features n is fixed to 10, and the training is done on completely random datasets. In these plots, each point (x, y) represents the average DOE values y obtained from the equivalence tests between the oracle and the tree constructed by the learning algorithm trained on a dataset with the given size x. The depth input ranges from 5 to 8, aiming to observe the performance difference of each decision tree learning algorithm when the oracle tree grows deeper.

As shown in the plots, when the number of features and depth is fixed the DOE score increases as the size of the training set gets larger (**Question 1**). For the heuristic-based decision trees, the line plots show a logarithmic-like increase, whereas the DOE scores of InferDT increase near-linearly with a very steep slope and exceed 99% when the number of data examples in the datasets are relatively small.

We also observe that, with the same number of features and as the depth of the oracle increases, the learning algorithms need larger training sets to train on in order to achieve the same DOE score as before (**Question 2**). To take ID3 as an example, it requires 1200 random data examples to infer a model with a 90% DOE score when depth is 5. However, when the depth is set to 8, a dataset with 1800 data examples is needed to train a model with the same DOE score. J48 and simpleCART are more sensitive to depth increase. When the oracle deepens from 5 to 8, J48 and simpleCART need training sets with nearly double the size (from 1800 to 3000) to produce a model with 90% DOE score. It is also noticeable that the curves are flattened when the depth escalates.

Based on the graphs, InferDT clearly outperforms all the heuristic-based learning algorithms because it requires fewer data examples to infer an accurate model

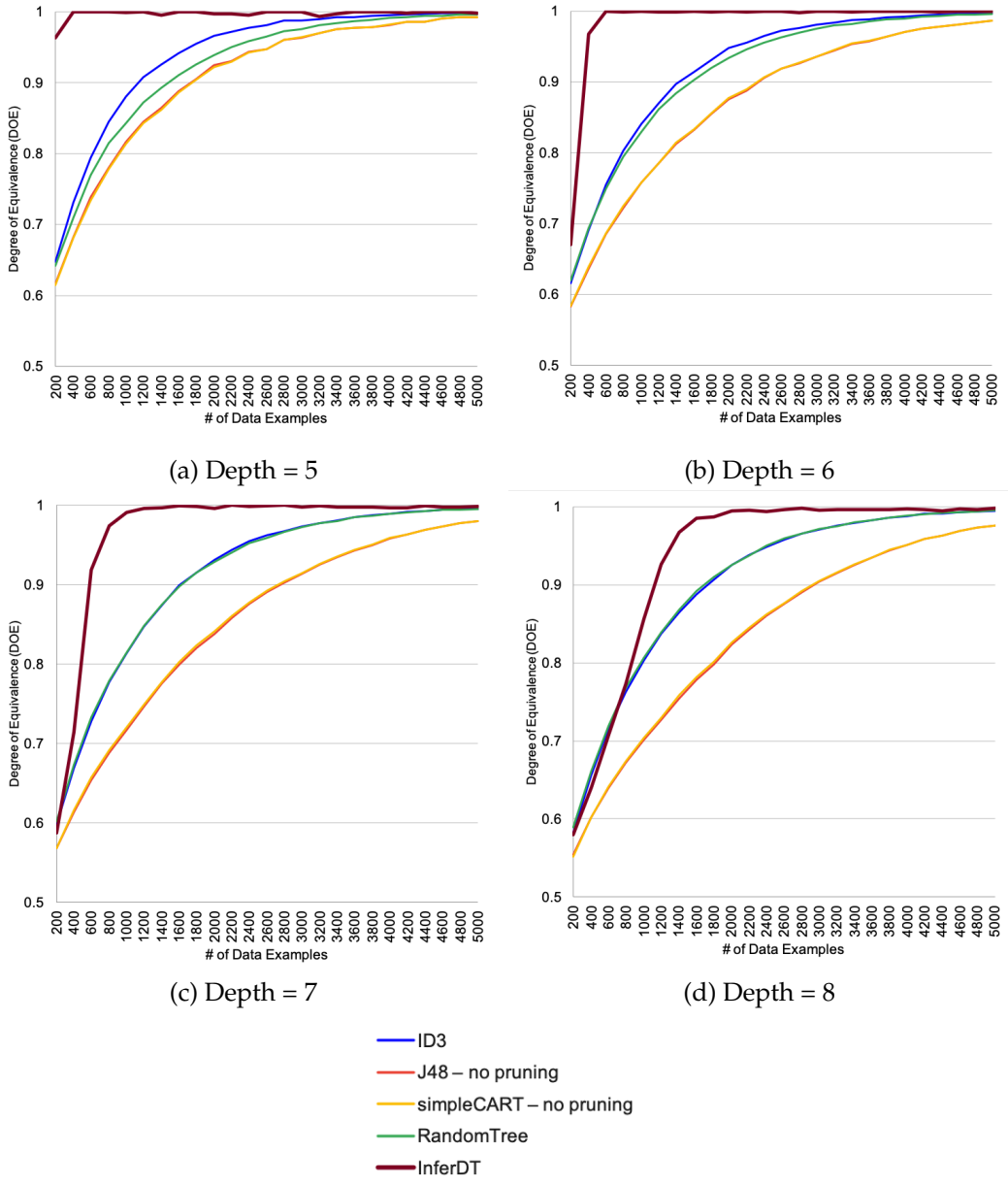

Figure 4.1: DOE comparison for decision tree learning algorithms trained on completely random datasets with 10 features and binary values

(**Question 3**). However, because the computational time increases exponentially as the oracle tree gets deeper, this algorithm takes much longer to produce a tree model. ID3 shows significantly better results when compared to J48 and simpleCART despite it being the earliest member of the decision tree algorithm family. It

also performs better than RandomTree when the depth of oracle is small; however, because ID3 is more sensitive to the depth increase, the performance difference between the two algorithms becomes very small when the depth grows. The performance of J48 and simpleCART are very similar since we observe that their curves overlap each other in every plot.

To answer **Question 4**, we also performed a series of experiments using uniquely random datasets. Similar to the above-mentioned experiments, we evaluate the objective decision tree algorithms by training them on the generated training sets. We then compare the DOE score computed by the equivalence tests and plot line graphs to illustrate the performance difference for inputs of various depth. Figure 4.2 shows the results of these experiments. The number of features is again set to 10 for comparable results.

One major difference between the results obtained from these two sets of experiments is the size of the dataset. For 10 features with binary values, the total number of unique data examples is 1024 (2^{10}). Hence, it is impossible for the number of data examples in the uniquely random datasets to be more than 1024. On the other hand, completely random datasets contain redundant data examples, which means a larger dataset is required to represent the same amount of information as in the uniquely random dataset.

Another observation from comparing the two sets of results is the difference in the shape of the curves in the line plots. In contrast to the logarithmic-like curves, when the heuristic-based decision tree learning algorithms train on uniquely random datasets, the line plots are approximately linear. The curve of InferDT shapes distinctively. When the datasets are small, the curves are roughly straight and the slopes are similar to the slopes of ID3 and RandomTree curves; however, when the size the datasets passes a critical number (i.e., 150 when the depth is set to 6, 300 when the depth is 7, or 500 when the depth is 8), the increase of the DOE values accelerates and the shape of the curves becomes logarithmic-like.

It may seem odd that InferDT does not have advantages over heuristic-based algorithms when training on datasets with a small number of examples. Yet, the reason is rather simple: small datasets do not have enough data examples to fully represent the entire oracle model. For an oracle with depth of k, at least 2^k data examples are needed in the datasets to represent every path in the oracle. As the a depth k gets larger, the minimum number of examples for full oracle representation grows exponentially. Without enough data providing information about the oracle, the exact model inferred by InferDT would not be equivalent to the oracle model. Note that when the size of the datasets grows over a critical number, the performance of InferDT improves drastically.

Overall, InferDT shows the best performance among the learning algorithms under test (**Question3**). Even if InferDT produces similar DOE values as ID3 and RandomTree when the training sets contain a limited number of instances, it quickly surpasses the other learning algorithms as the size of the datasets grows.

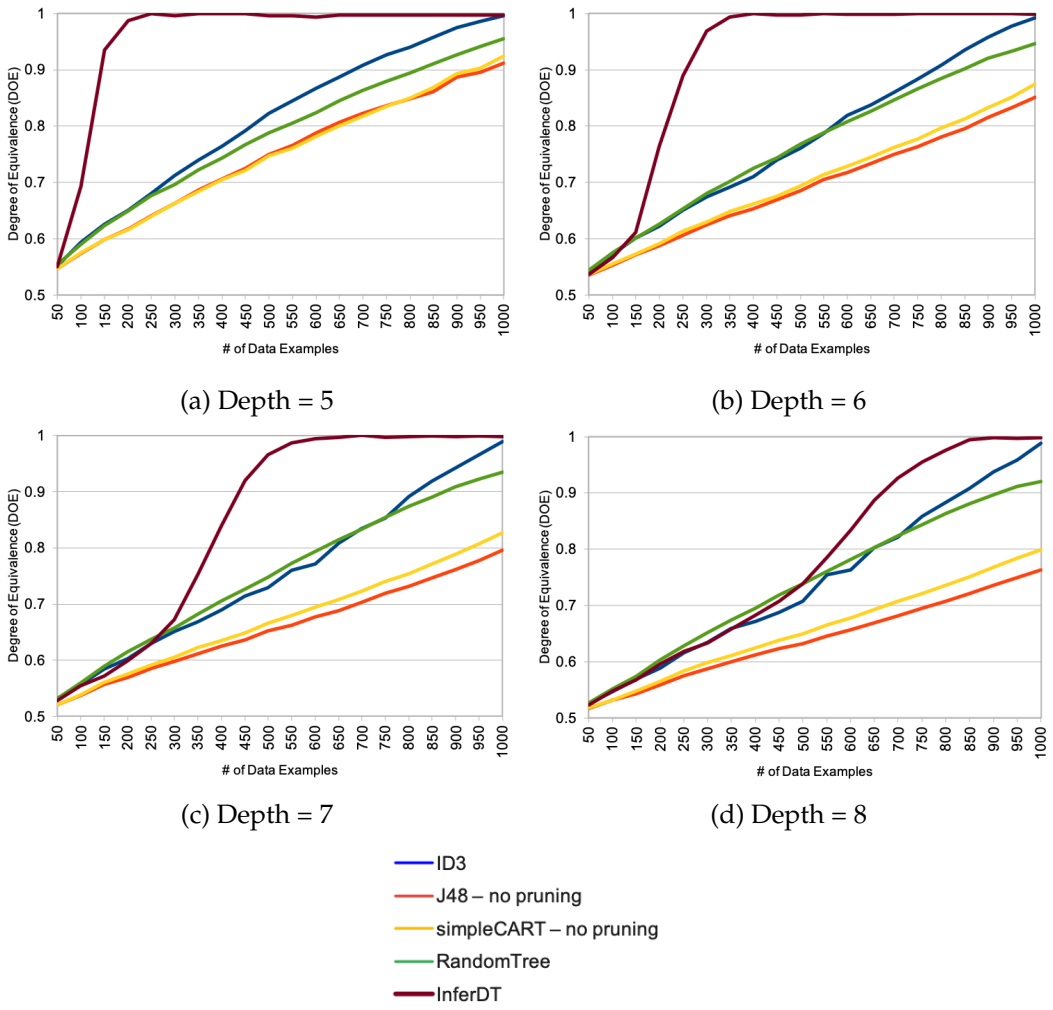

Figure 4.2: DOE comparison for decision tree learning algorithms trained on uniquely random datasets with 10 features and binary values

ID3 is better than the other heuristic-based learning algorithms, especially when training on large datasets. RandomTree also infers very accurate models in general. It even defeats ID3 when dealing with small datasets generated by deep oracle models. J48 and simpleCART again produce similar results, but simpleCART performs slightly better when the oracle tree is deeper.

Chapter 5

Conclusion and Future Work

We propose a novel approach to evaluating decision tree learning algorithms. This approach consists of generating data from reference trees playing the role of oracles, use the generated data to produce learned trees using existing learning algorithms, and determining the correctness of the learned trees by comparing them with the oracles. The correctness of the learned trees is measured by the degree of equivalence (DOE), which is calculated based on the percentage of correctly labeled paths.

Using this new evaluation framework, we then assess five decision tree learning algorithms, namely ID3, J48, simpleCART, RandomTree, and InferDT. The first four algorithms under test are heuristic-based, whereas InferDT is an exact algorithm aiming to infer the optimal model. The preliminary evaluation results show that, when training on deterministic datasets, InferDT produces the most accurate models. In the family of heuristic-based decision trees, ID3 and RandomTree have the best performance, with ID3 performing slightly better than RandomTree. The results also show the effectiveness of our evaluation method. By using the DOE metric, our framework successfully distinguishes the performance difference between learning algorithms.

We believe that our approach can be improved in several ways, as follows: We plan on enhancing our framework to consider noisy data, which involves generating non-deterministic datasets. We are expecting J48 and simpleCART to show better performance in this context because of their pruning process. We also intend to apply this approach to evaluate feature selection techniques. Indeed, when the depth of an oracle is smaller than the number of available features, each path would have ”free attributes” that do not contribute to assigning class labels. These free attributes are *irrelevant* to this path. Based on these free attributes we are expecting then to be able to rank features based on their relevancy and correctly identify the ones that are irrelevant overall. We would also like to examine the relation between the size of the dataset and the performance of these feature selection techniques in terms of their ability to recognize trivial features.

Bibliography

[1] Gaël Aglin, Siegfried Nijssen, and Pierre Schaus. Learning optimal decision trees using caching branch-and-bound search. In *Proceedings of the Thirty-Fourth Conference on Artificial Intelligence (AAAI), New York, USA*, 2020.

[2] Gaël Aglin, Siegfried Nijssen, and Pierre Schaus. Learning optimal decision trees using caching branch-and-bound search. In *The Thirty-Fourth AAAI Conference on Artificial Intelligence, AAAI 2020, The Thirty-Second Innovative Applications of Artificial Intelligence Conference, IAAI 2020, The Tenth AAAI Symposium on Educational Advances in Artificial Intelligence, EAAI 2020, New York, NY, USA, February 7-12, 2020*, pages 3146–3153. AAAI Press, 2020.

[3] Florent Avellaneda. Efficient inference of optimal decision trees. In *The Thirty-Fourth AAAI Conference on Artificial Intelligence, AAAI 2020, The Thirty-Second Innovative Applications of Artificial Intelligence Conference, IAAI 2020, The Tenth AAAI Symposium on Educational Advances in Artificial Intelligence, EAAI 2020, New York, NY, USA, February 7-12, 2020*, pages 3195–3202. AAAI Press, 2020.

[4] Christian Bessiere, Emmanuel Hebrard, and Barry O'Sullivan. Minimising decision tree size as combinatorial optimisation. In *International Conference on Principles and Practice of Constraint Programming*, pages 173–187. Springer, 2009.

[5] Andrew P Bradley. The use of the area under the ROC curve in the evaluation of machine learning algorithms. *Pattern recognition*, 30(7):1145–1159, 1997.

[6] Leo Breiman, Jerome H. Friedman, Richard A. Olshen, and Charles J. Stone. *Classification and Regression Trees*. Wadsworth International Group, Belmont, California, 1984.

[7] Dariusz Brzezinski and Jerzy Stefanowski. Prequential AUC: Properties of the area under the ROC curve for data streams with concept drift. *Knowledge and Information Systems*, 52(2):531–562, 2017.

[8] Richard O Duda, Peter E Hart, and David G Stork. *Pattern classification*. John Wiley & Sons, 2012.

[9] Qiong Gu, Li Zhu, and Zhihua Cai. Evaluation measures of the classification performance of imbalanced data sets. In *International symposium on intelligence computation and applications*, pages 461–471. Springer, 2009.

[10] Tin Kam Ho. Random decision forests. In *Proceedings of 3rd international conference on document analysis and recognition*, volume 1, pages 278–282. IEEE, 1995.

[11] M Hossin, MN Sulaiman, A Mustapha, N Mustapha, and RW Rahmat. A hybrid evaluation metric for optimizing classifier. In *2011 3rd Conference on Data Mining and Optimization (DMO)*, pages 165–170. IEEE, 2011.

[12] Mohammad Hossin and MN Sulaiman. A review on evaluation metrics for data classification evaluations. *International Journal of Data Mining & Knowledge Management Process*, 5(2):1, 2015.

[13] Badr Hssina, Abdelkarim Merbouha, Hanane Ezzikouri, and Mohammed Erritali. A comparative study of decision tree ID3 and C4.5. *International Journal of Advanced Computer Science and Applications*, 4(2):13–19, 2014.

[14] Laurent Hyafil and Ronald L Rivest. Constructing optimal binary decision trees is NP-complete. *Information Processing Letters*, 5(1):15–17, 1976.

[15] Sushilkumar Kalmegh. Analysis of Weka data mining algorithm reptree, simple cart and randomtree for classification of Indian news. *International Journal of Innovative Science, Engineering & Technology*, 2(2):438–446, 2015.

[16] T Miranda Lakshmi, A Martin, R Mumtaj Begum, and V Prasanna Venkatesan. An analysis on performance of decision tree algorithms using student's qualitative data. *International journal of modern education and computer science*, 5(5):18, 2013.

[17] D Lavanya and K Usha Rani. Performance evaluation of decision tree classifiers on medical datasets. *International Journal of Computer Applications*, 26(4):1–4, 2011.

[18] William J Long, John L Griffith, Harry P Selker, and Ralph B D'agostino. A comparison of logistic regression to decision-tree induction in a medical domain. *Computers and Biomedical Research*, 26(1):74–97, 1993.

[19] John Mingers. An empirical comparison of pruning methods for decision tree induction. *Machine learning*, 4(2):227–243, 1989.

[20] Nina Narodytska, Alexey Ignatiev, Filipe Pereira, Joao Marques-Silva, and IS RAS. Learning optimal decision trees with SAT. In *IJCAI*, pages 1362–1368, 2018.

[21] Siegfried Nijssen, Pierre Schaus, et al. Learning optimal decision trees using caching branch-and-bound search. In *Thirty-Fourth AAAI Conference on Artificial Intelligence*, 2020.

[22] David Martin Powers. Evaluation: From precision, recall and F-measure to ROC, informedness, markedness and correlation. *Journal of Machine Learning Technologies*, 2(1):37–63, 2011.

[23] J. Ross Quinlan. Induction of decision trees. *Machine learning*, 1(1):81–106, 1986.

[24] J Ross Quinlan. *C4.5: Programs for machine learning*. Elsevier, 2014.

[25] Ross Quinlan. *C4.5: Programs for Machine Learning*. Morgan Kaufmann Publishers, San Mateo, CA, 1993.

[26] Shiju Sathyadevan and Remya R Nair. Comparative analysis of decision tree algorithms: ID3, C4.5 and random forest. In *Computational intelligence in data mining-volume 1*, pages 549–562. Springer, 2015.

[27] Claude E Shannon. A mathematical theory of communication. *Bell system technical journal*, 27(3):379–423, 1948.

[28] Detlef Sieling. Minimization of decision trees is hard to approximate. *Journal of Computer and System Sciences*, 74(3):394–403, 2008.

[29] Omer Nguena Timo, Alexandre Petrenko, and S Ramesh. Using imprecise test oracles modelled by FSM. In *2019 IEEE International Conference on Software Testing, Verification and Validation Workshops (ICSTW)*, pages 32–39. IEEE, 2019.

[30] Ian H Witten, Eibe Frank, Leonard E Trigg, Mark A Hall, Geoffrey Holmes, and Sally Jo Cunningham. Weka: Practical machine learning tools and techniques with Java implementations. Working paper 99/11, University of Waikato, Department of Computer Science, 1999.

[31] Zhiyong Yang, Taohong Zhang, Jingcheng Lu, Dezheng Zhang, and Dorothy Kalui. Optimizing area under the ROC curve via extreme learning machines. *Knowledge-Based Systems*, 130:74–89, 2017.