Assignment of master's thesis

Title:	Standard Decision Trees in Machine Learning
Student:	Bc. Yuliia Syzon
Supervisor:	Ing. Adam Valenta
Study program:	Informatics
Branch / specialization:	Knowledge Engineering
Department:	Department of Applied Mathematics
Validity:	until the end of summer semester 2022/2023

Instructions

Describe the Standard Decision Tree (SDT) algorithm. Focus on explaining the approaches to SDTs creation. Explain why SDTs are essential for the explainability of models in Machine Learning (ML). Select one appropriate method to construct SDT and implement it into the H2O-3 Machine Learning Platform. Test your implementation with a suitable dataset and compare it with another alternative ML model.

Electronically approved by Ing. Magda Friedjungová, Ph.D. on 21 April 2022 in Prague.

Master’s thesis

STANDARD DECISION TREES IN MACHINE LEARNING

Bc. Yuliia Syzon

Faculty of Information Technology
Department of Applied Mathematics
Supervisor: Ing. Adam Valenta
December 27, 2022

Czech Technical University in Prague
Faculty of Information Technology
© 2023 Bc. Yuliia Syzon. All rights reserved.
This thesis is school work as defined by Copyright Act of the Czech Republic. It has been submitted at Czech Technical University in Prague, Faculty of Information Technology. The thesis is protected by the Copyright Act and its usage without author's permission is prohibited (with exceptions defined by the Copyright Act).

Citation of this thesis: Syzon Yuliia. *Standard Decision Trees in Machine Learning.* Master's thesis. Czech Technical University in Prague, Faculty of Information Technology, 2023.

Contents

Acknowledgments vi

Declaration vii

Abstract viii

Introduction 1

1 Research 3
- 1.1 Preliminaries . . . 3
  - 1.1.1 Machine Learning . . . 3
  - 1.1.2 Decision Trees . . . 6
  - 1.1.3 H2O-3 Machine Learning platform . . . 13
- 1.2 Approaches to Decision Tree creation . . . 16
  - 1.2.1 Standard Decision Tree (SDT) - Greedy approach . . . 18
  - 1.2.2 Oblique decision trees . . . 22
  - 1.2.3 Evolutionary approach - genetic algorithms . . . 24
  - 1.2.4 SAT-based Decision Tree Learning . . . 28
  - 1.2.5 Incremental approach . . . 30
- 1.3 ML models explainability with decision trees . . . 32
  - 1.3.1 Explanations via Surrogate Models . . . 34
  - 1.3.2 Neural-Backed Decision Trees . . . 36

2 Design of Single Decision Tree for H2O-3 39
- 2.1 Requirements . . . 39
  - 2.1.1 Algorithm (R1 requirement) . . . 40
  - 2.1.2 Splitting rules (R2-R5 Requirements) . . . 40
  - 2.1.3 Termination rules . . . 41
  - 2.1.4 Assigning decision values to the leaves (R3, R6 requirements) . . . 41
  - 2.1.5 Optimisations for huge datasets (R7 requirement) . . . 41
  - 2.1.6 Distributed computation (R8 requirement) . . . 42
  - 2.1.7 Presentability (R9 requirement) . . . 42
- 2.2 Implementation notes . . . 43

3 Evaluation of DT implementation 45
- 3.1 Datasets . . . 45
- 3.2 Comparing the results on data . . . 46
- 3.3 Scalability . . . 47
- 3.4 Explainability . . . 49
- 3.5 Drawbacks and future improvements . . . 50

Conclusion 51

Contents of enclosed CD 57

List of Figures

1.1 Major types of data in ML 4
1.2 Types of ML algorithms. [4] 5
1.3 Example of binary DT 6
1.4 Decision Tree hierarchy. [6] 6
1.5 Splitting in DT. Blue and yellow dots are samples of two different classes. 7
1.6 Impurity of nodes while splitting. Split A has lower purity, split B has higher purity. Each node is labeled by the split name and the place in the tree hierarchy: **AP** - split A, parent; **AL** - split A, left child; **AR** - split A, right child; **BP** - split B, parent; **BL** - split B, left child; **BR** - split B, right child. 8
1.7 DT as segment space. Case (a) corresponds to the example of DT on images 1.3 and 1.5. 11
1.8 An example of a key-value store. [12] 14
1.9 Hadoop MapReduce framework. [15] 15
1.10 Example of DT before (left) and after (right) pruning. 21
1.11 Example of oblique decision tree represented as segmented space (left) and tree structure (right). 22
1.12 Oblique decision tree 23
1.13 Standard Decision Tree 23
1.14 Genetic algorithm cycle. [28] 24
1.15 Crossover. [29] 26
1.16 Mutation. [29] 27
1.17 Illustration of nodes indexing in a binary tree. [35] 29
1.18 Replacing original subtree with newly generated one while local improvement. [37] 30
1.19 Incremental learning flow. [40] 31
1.20 The big picture of explainable machine learning. [45] 32
1.21 Black-box ML model. 33
1.22 Explainability vs. performance tradeoff. [49] 34
1.23 Neural-Based Decision Tree example. [52] 36
1.24 The training and fine-tuning process for a Neural-Backed Decision Tree. [50] . . 37
1.25 Building Induced Hierarchies. [50] 37
1.26 The prediction process for a Neural-Backed Decision Tree. [50] 38

3.1 Evaluation of models on the *prostate* dataset. 46
3.2 Evaluation of models on the *airlines* dataset. 47
3.3 Training and prediction time for a *prostate* dataset. 48
3.4 Training and prediction time for *airlines* dataset. 49

List of Tables

List of code listings

In the first row, I would like to thank my supervisor for the support and technical help during the whole process of work. I also thank H2O.ai for giving me the opportunity to work on the real problem and to have the code in Open Source, and for allowing me to use the office, which helped me be productive and more in touch with the team. This cooperation with H2O.ai was possible thanks to FIT's "Cooperation with Industry" program.
Moreover, of course, I am very thankful to my friends and family for the enormous support during searching and working on this thesis. Your patience and invested time are priceless.

Declaration

I hereby declare that the presented thesis is my own work and that I have cited all sources of information in accordance with the Guideline for adhering to ethical principles when elaborating an academic final thesis. I acknowledge that my thesis is subject to the rights and obligations stipulated by the Act No. 121/2000 Coll., the Copyright Act, as amended, in particular that the Czech Technical University in Prague has the right to conclude a license agreement on the utilization of this thesis as a school work under the provisions of Article 60 (1) of the Act.

In Prague on December 27, 2022 ....

Abstract

This thesis analyzes approaches for Decision Tree creation and the importance of the Decision Tree algorithm for Machine Learning Explainability. Several traditional and novel approaches are explained and analyzed in terms of usability for Big Data and distributed systems. The practical part of this work consists of designing and implementing the Standard Decision Tree for the H2O-3 Machine Learning Platform applicable to Big Data and optimized for distributed computations. The standard greedy approach is implemented. The evaluation of the implemented solution contains scalability and performance comparisons with other solutions on selected benchmark datasets.

Keywords H2O.ai, H2O-3 Machine Learning Platform, Decision Tree, Distributed Decision Tree, Machine Learning Explainability

Abstrakt

Tato práce analyzuje přístupy vytváření rozhodovacího stromu a význam rozhodovacího stromu jakožto algoritmu pro vysvětlitelnost modelů strojového učení. Je představeno několik tradičních i nových přístupů z hlediska použitelnosti pro velké objemy dat a distribuované systémy. Praktická část této práce spočívá v návrhu a implementaci standardního rozhodovacího stromu pro open-source platformu H2O-3, optimalizovaného pro distribuované výpočty. Je implementován standardní hladový přístup. Hodnocení implementovaného řešení obsahuje srovnání škálovatelnosti v porovnání s jinými implementacemi na vybraných testovacích datových sadách.

Klíčová slova H2O.ai, H2O-3 Machine Learning Platform, Rozhodovací strom, Distribuovaný rozhodovací strom, Vysvětlitelnost modelů strojového učení

Introduction

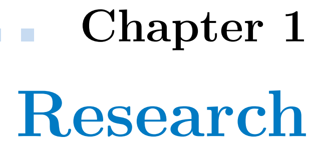
In the world of automation, we tend to search for the easiest and more effective solutions to make our lives easier or at least to avoid tedious manual work. Many usual things are now performed by Machine Learning algorithms instead of humans - detecting anomalies, approximating profit and expenses, image recognition, transforming a voice into text, and much more. Algorithms also assist us with making decisions, finding optimal solutions, and the system's diagnosis (including human organism) and give us all sorts of predictions. One of those types of work is also answering questions based on the given base of already answered questions. This task is called supervised learning, which lies in finding patterns in given data so that patterns can lead us to the correct answer to previously unseen questions. The core principle of supervised learning is having inputs and desired outputs to get a system that can give the correct output to new input.

Several approaches and algorithms were developed to solve the supervised learning problem, and each is more suitable for different use cases. A Decision Tree (DT) is one of those approaches. It serves as a weak learner as well as the standalone model. With a long development history and having been used for several problems and types of input data, decision trees have more building strategies and may look different. The resulting model is clearly explainable as the specific output is given by one specific branch in the tree and can be understood as a chain of conditions put on the set of attributes. Another advantage of DTs is that most building strategies can work with both categorical and numeric attributes and do not require deep data cleaning and transformations. DT can also operate with missing values, and different ranges of attributes, as each node is designed independently with the suitable condition for the attribute's datatype. However, there are types of trees when the features are combined in one node (see Oblique trees). DTs for sure have disadvantages like relatively low complexity as well as easy changing model structure as the result of small changes in data (instability). Also, finding an optimal DT for the given problem is hard (NP-complete).

This work is done in cooperation with the H2O.ai company that delivers the open-source H2O-3 Machine Learning Platform. The practical part of the work is the contribution to this platform - the implementation of the Single Decision Tree algorithm that will become available in the H2O-3 platform in the future release. The motivation for this cooperation is H2O.ai's need to implement a versatile and competitive decision tree algorithm that will be used as a standalone predictor as well as the explanation mechanism for other Machine Learning models.

The aim of this thesis is to research and analyze two main aspects of Decision Trees - their building approaches and usage for Machine Learning explainability. The theoretical part of this work is supplemented with the practical part that analyzes, designs, implements, and evaluates the Single Decision Tree algorithm into the H2O-3 Machine Learning Platform.

Text starts with a research chapter where I present some critical preliminaries for the topic, describe existing approaches for DT building, and finally, the usage of DT for ML explainability. In the second chapter I write about the practical task of this thesis - I analyze the requirements and present the design and the implementation of the *SingleDecisionTree* algorithm for the H2O-3 Machine Learning Platform. In the last chapter, I evaluate the *SingleDecisionTree* implementation. Finally, in the conclusion I summarize achieved results and further plans.



Chapter 1 Research

As the Decision Tree (DT) is an old and frequently used algorithm, there is a long research history on it. I have explored some classical resources as well as modern articles offering novel approaches and analyses. The results of the research are presented in this chapter.

1.1 Preliminaries

I am assuming that the reader is already familiar with ML and DT. However, I formally introduce the main principles of DT algorithms and additional practical knowledge essential to understanding the following chapters.

1.1.1 Machine Learning

Machine Learning (ML) was defined in the 1950s by AI pioneer Arthur Samuel:

> *"The field of study that gives computers the ability to learn without explicitly being programmed." [1]*

ML is a branch of Artificial Intelligence (AI) that operates with algorithms that can learn from and make predictions on data. It is valuable as it can find dependencies in a large amount of data at a speed and scale. It focuses on using data and algorithms to imitate how humans learn, gradually improving its accuracy.

The data is one of the most important parts of ML. The mandatory input for all the ML algorithms is the dataset.

▶ **Definition 1.1.1** (Dataset)**.** A dataset is a concrete collection of data. Usually, the dataset is a subset (selection) of all possible inputs and outputs for the specific problem.

Data can be **structured** (tables) or **unstructured** (text, media). In this work, I only work with structured data, but any unstructured data can be converted to structured by applying some embedding.

Structured data are represented by the table. A single row of structured data is called an **instance** or **data sample**. All instances share common attributes (columns) that are called **features**. A dimension of the table is $N \times M$, where N is a number of rows/instances/samples and M is a number of features. A single instance is a vector of M features:

$$(x_1, x_2, ..., x_m).$$

Two major types of data are **numerical/quantitative** – measurable data that have defined sorting order and mathematical operations – and **categorical** – represent some limited categories (see image 1.1). Numerical data can be **continuous** or **discrete**. Categorical data can be **ordinal** (categories have an order) and **nominal** (categories are independent without an order). The special data type is **binary/boolean** - it is a categorical type with only two values - true and false (0 and 1). Other types of data are **time-series** – data that are sorted in time – and **text data** – free text values that do not represent categories.

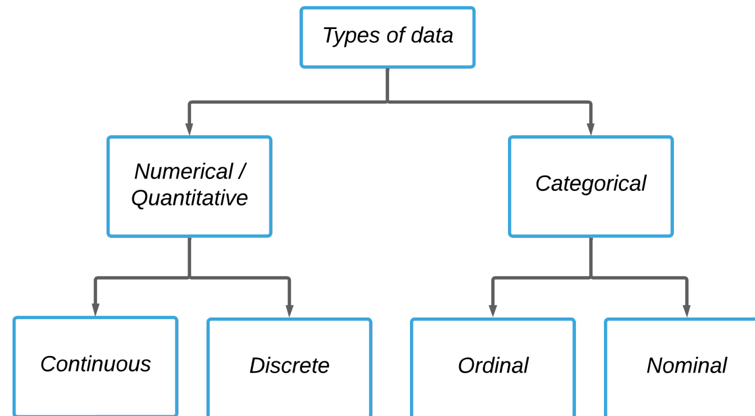


Figure 1.1 Major types of data in ML

Depending on the problem and available data, different types of ML algorithms are determined [2, 3]. This thesis deals with **Supervised Learning** only.

▶ **Definition 1.1.2** (Target variable)**.** The **target (output) variable** (usually Y) is the feature of a dataset that needs to be predicted or explained using the rest of the dataset (usually X). It can be of any data type.

Supervised Learning - each data point is labeled with the expected value of the target variable, and a dataset T of size N with M features and target variable Y can be represented as follows:

$$\{(x_{1,1}, x_{1,2}, ...x_{1,m}, y_1), .., (x_{n,1}, x_{n,2}, ..., x_{n,m}, y_n)\},$$

where $x_{:,i}$, $i \in \{1, ..., m\}$ are features, and y_i, $i \in \{1, ..., n\}$ is the target variable.

The task of Supervised Learning is to find a function $g : X- > Y$, where X is the input space and Y is the output space. There are different metrics that measure how good the function is, so it is possible to select the best function based on the selected metric. Some supervised learning algorithms are Decision Tree (DT), Generalized Linear Models (GLM), Neural Network (NN), etc.

Other types of ML algorithms are:

- **Unsupervised Learning** - the dataset consists of unlabelled examples only. These algorithms discover hidden patterns in data without the need to provide labels. The ability of unsupervised learning to discover similarities and differences in information makes it the ideal solution for exploratory data analysis. The tasks are typically clustering, dimension reduction, etc.

- **Semi-Supervised Learning** - the combination of supervised and unsupervised learning when some data points are labeled, and the rest are not. The amount of labeled data is much more significant than that of unlabeled data.
- **Reinforcement Learning** - enables an agent to learn in an interactive environment by trial and error using feedback. Both supervised, and reinforcement learning relies on labels. Unlike supervised learning, where the label is a correct set of actions for performing a task, reinforcement learning uses rewards and punishments as signals for positive and negative behavior.

Image 1.2 shows types of ML algorithms and typical use cases of each type.

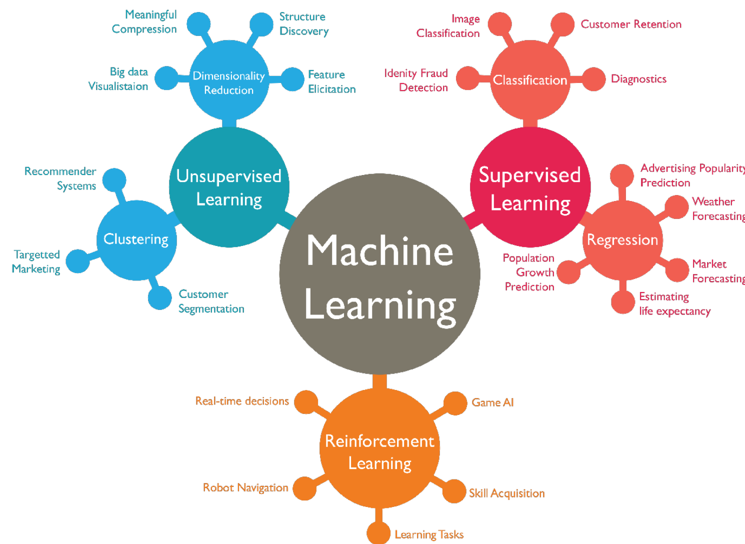


Figure 1.2 Types of ML algorithms. [4]

Depending on the datatype of the target variable, different types of Supervised Learning problems are defined:

▶ **Definition 1.1.3** (Classification)**.** For a categorical target variable, the task of its prediction is called **classification**, and each unique value of the target variable is called a class.

$$Y = \{y \mid y \in \{c_0, ..., c_k\}\}$$

▶ **Definition 1.1.4** (Binary classification)**.** Classification problems with two class labels are referred to as binary classification. In most binary classification problems, the classes are *True* and *False* or 0 and 1.

$$Y = \{y \mid y \in \{c_0, c_1\}\}$$

▶ **Definition 1.1.5** (Regression)**.** For a numerical target variable, the task of its prediction is called **regression**.

$$Y = \{y \mid y \in R\}$$

1.1.2 Decision Trees

A Decision Tree (DT) is a supervised learning algorithm for classification and regression tasks [5]. Image 1.3 shows an example of simple DT that uses two numerical features - x_1 and x_2 and a binary target variable Y with values 0 and 1.

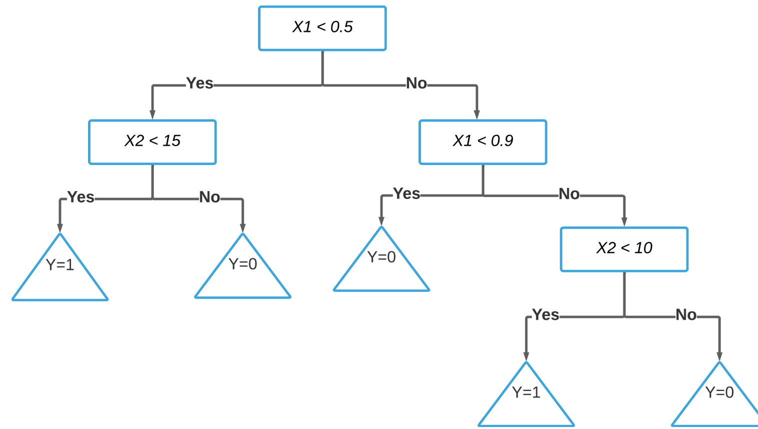


Figure 1.3 Example of binary DT

The main component of the DT algorithm is a **node**. There are two types of nodes - an **internal** node and a **leaf** (also called a terminal node). The **root** is the first internal node that holds the whole initial dataset and does not have a parent.

Internal nodes split data by specific **splitting criteria** into two or more mutually exclusive subsets. Splitting criteria are based on the feature's values and target variable and depend on the building algorithm and the data types. The most common are binary splitting rules that lead to binary trees.

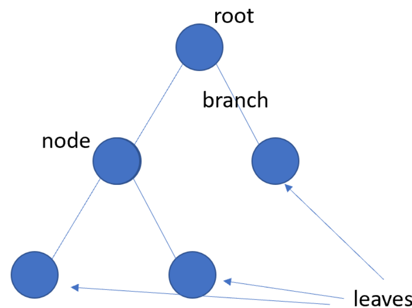


Figure 1.4 Decision Tree hierarchy. [6]

Leaf nodes represent a value of the target variable (**decision value**) - class in the case of classification and real value in the case of regression. All data samples that belong to the specific leaf have an identical value of the target variable given by this node. The decision value in the node does not have to be unique, and there can be several leaves predicting the same value of the target variable for the different subsets of data.

A DT is formed by a hierarchy of nodes (see image 1.4). Each internal node has a parent (except the root node) and two or more children. A subset of data in the parent node is the union of subsets in children nodes. In such a way, the initial dataset is distributed into leaves by the chain of the splitting rules. The chain of nodes leading to the specific leaf is called a **branch**. A branch can be expressed as a chain of rules from the root to the leaf.

Splitting rules are specific to different approaches. However, the main objective always stays the same - the subset of data has to be split into two or more **mutually exclusive and collectively exhaustive subsets** based on the values of one or more features in such a way that **minimizes the impurity**(see Definition 1.1.6) of the children nodes (see image 1.5).

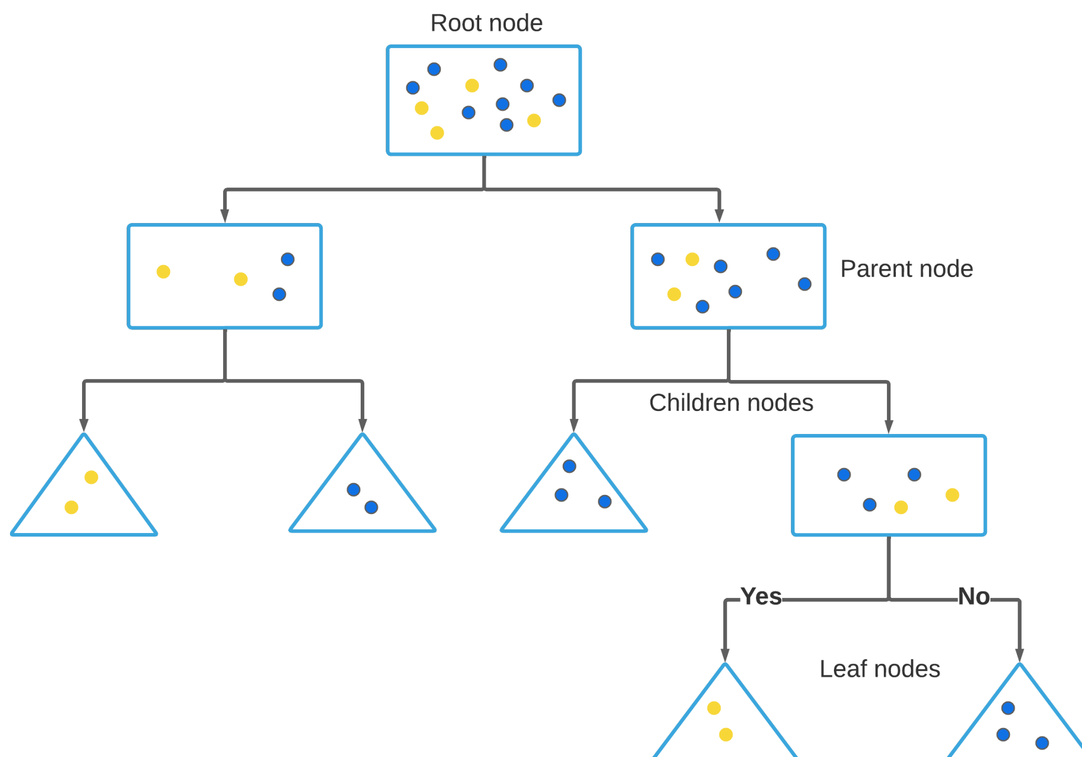


Figure 1.5 Splitting in DT. Blue and yellow dots are samples of two different classes.

The splitting rule can be represented as the function that assigns the child node to each data sample based on its features' values. Each data sample has to belong to exactly one child node to guarantee that the split is valid.

Common types of splitting rules are:

- **Binary splitting by one feature -** the easiest type of splitting rule. The value of the single selected feature is used to decide whether the data samples belong to the left or the right child. Boolean features are simply split by the values *true* and *false* into two subsets. Numerical features are split by the defined threshold (parameter). Categorical values are split based on the given subsets of their values (parameters). In rare cases, the numerical values can be split as categorical values using more complex rules than comparing values to the single threshold.
- **Multi-splitting by one feature -** uses the single feature value but splits into three or more subsets. The decision tree is not binary anymore once this type of splitting rule is used.

- **Multi-feature split** - uses more than one feature for a split. The splitting rule can be any function that takes features' values as input, combines them in any way, and assigns the child node for the specific data sample. All types of features can be combined in a single split. This type of splitting rule can also be used to reduce the number of splits by combining conditions for several features in one node.
- **Similarity splitting** - rarely used type of splitting when the data children nodes are assigned to the data samples based on the similarity of the sample and some given parameter. Usually, all features are used for computing similarity. This type is a special type of a multi-feature split as the similarity is also a function combining several features that can be used for splitting in a multi-feature split.

Most of the introduced types of splitting rules require the initialization of one or more parameters (threshold, set of categorical values, number of splits, etc.). The value of the parameter influences the output of the split. The parameter needs to be selected in such a way that minimizes the impurity of children nodes and so guarantees that the split is the most optimal.

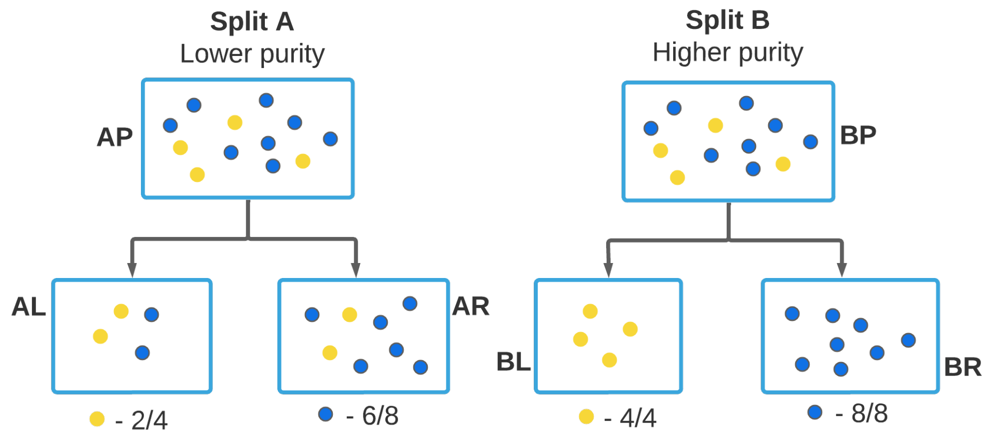


Figure 1.6 Impurity of nodes while splitting. Split A has lower purity, split B has higher purity. Each node is labeled by the split name and the place in the tree hierarchy: **AP** - split A, parent; **AL** - split A, left child; **AR** - split A, right child; **BP** - split B, parent; **BL** - split B, left child; **BR** - split B, right child.

▶ **Definition 1.1.6** (Node impurity)**. Impurity** is a measure of the homogeneity of the labels at the node.

A node is 100% pure when all of its data belongs to a single class or have the same value of the target variable. The more diverse the labels of data samples in the node, the more impure (the less pure) this node (see image 1.6).

▶ **Definition 1.1.7** (Information gain)**. Information Gain (IG)** represents the reduction in impurity between parent and children nodes. IG is calculated by comparing the impurity of the dataset before and after a transformation (splitting). The impurity of the children layer is measured as the average impurity of the child nodes. [7]

$$IG(T) = Impurity(parent) - Impurity(children).[7]$$

Concrete impurity metrics for the classification and regression tasks given the dataset T (see definition of a dataset for Supervised Learning) are:

- **Gini index** [8] - metric for the classification problem. It is related to the misclassification probability of a random sample. Assuming the dataset T contains k classes, its Gini index, $Gini(T)$, is defined as:

$$Gini(T) = 1 - \sum_{i=1}^{k} {p_i}^2,$$

  where $p_i, i \in (1, k)$ is the relative frequency of class i in T. Gini index may result in values inside the interval $[0, 0.5]$. The higher the gini impurity, the more impure the node is.
  Example (see image1.6):

  $Gini(AP) = Gini(BP) = 1 - ((4/12)^2 + (8/12)^2) = 1 - 1/9 - 4/9 = 4/9 = 0.44$

  $Gini(AL) = 1 - ((2/4)^2 + (2/4)^2) = 1 - 1/4 - 1/4 = 2/4 = 0.5$

  $Gini(AR) = 1 - ((2/8)^2 + (6/8)^2) = 1 - 1/16 - 9/16 = 6/16 = 0.375$

  $Gini(BL) = Gini(BR) = 1 - ((1)^2 + (0)^2) = 1 - 1 = 0$

  $Gini(AL, AR) = 0.5 * 4/12 + 0.375 * 8/12 = 0.42$

  $Gini(BL, BR) = 0 * 4/12 + 0 * 8/12 = 0$

  $IG(A) = 0.44 - 0.42 = 0.02$

  $IG(B) = 0.44 - 0 = 0.44$

- **Entropy** [8] - metric for the classification problem. It is a measure of information. Assuming that the dataset T associated with a node contains examples from k classes, its entropy $E(T)$ is:

$$E(T) = \sum_{i=1}^{k} -p_i \log_2 p_i,$$

  where $p_i, i \in (1, k)$ is the relative frequency of class i in T. Entropy takes values from $[0, 1]$. The higher the entropy, the more impure the node is. **Example** (see image1.6):

  $E(AP) = E(BP) = -(4/12) * log_2(4/12) - (8/12) * log_2(8/12) = 0.53 + 0.39 = 0.92$

  $E(AL) = -(2/4) * log_2(2/4) - (2/4) * log_2(2/4) = -0.5 * (-1) - 0.5 * (-1) = 1$

  $E(AR) = -(2/8) * log_2(2/8) - (6/8) * log_2(6/8) = -1/4 * (-2) - 3/4 * (-0.4) = 0.5 + 0.3 = 0.8$

  $E(BL) = E(BR) = -1 * log_2(1) - 0 = 0$

  $E(AL, AR) = 1 * 4/12 + 0.81 * 8/12 = 0.87$

  $E(BL, BR) = 0 * 4/12 + 0 * 8/12 = 0$

  $IG(A) = 0.92 - 0.87 = 0.05$

  $IG(B) = 0.92 - 0 = 0.92$

- **Chi-square** [8] - metric for the classification problem. It works on the statistical significance of differences between the parent and child nodes. It is measured as the sum of squared standardized differences between observed and expected counts of samples of each class for each node and is calculated using this formula:

$$chi\text{-}square(T) = \sum_{i=1}^{k} \sqrt{\frac{(n_i - n_{i_expected})^2}{n_{i_expected}}}$$

Here, the $f_{i_expected}, i \in (1,k)$ is the expected number of samples for a class i in a child node based on the distribution of classes in the parent node, and $f_i, i \in (1,k)$ is the actual number of samples of a class i in a child node. As the chi-square criterium represents the difference between the parent and root, it is not reasonable to calculate IG from the chi-square.

Example (see image1.6):

$n_{y_expected}(AL) = n_{y_expected}(BL) = 4/12 * 4 = 1.33$

$n_{b_expected}(AL) = n_{b_expected}(BL) = 8/12 * 4 = 2.66$

$n_{y_expected}(AR) = n_{y_expected}(BR) = 4/12 * 8 = 2.66$

$n_{b_expected}(AR) = n_{b_expected}(BR) = 8/12 * 8 = 5.33$

$chi\text{-}square(AL) = \sqrt{\frac{(2-1.33)^2}{1.33}} + \sqrt{\frac{(2-2.66)^2}{2.66}} = 0.33 + 0.16 = 0.49$

$chi\text{-}square(AR) = \sqrt{\frac{(2-2.66)^2}{2.66}} + \sqrt{\frac{(6-5.33)^2}{5.33}} = 0.16 + 0.08 = 0.24$

$chi\text{-}square(BL) = \sqrt{\frac{(4-1.33)^2}{1.33}} + \sqrt{\frac{(0-2.66)^2}{2.66}} = 5.36 + 2.66 = 8.02$

$chi\text{-}square(BR) = \sqrt{\frac{(0-2.66)^2}{2.66}} + \sqrt{\frac{(8-5.33)^2}{(5.33}} = 2.66 + 1.33 = 4$

- **Sum of Squared Error (SSE) [9]** and **Variance [8]** - metrics for a regression problem. Defines how broadly data labels are distributed relative to the average of the target variable in the node. The formula of SSE for the dataset T is:

$$SSE(T) = \sum_{i \in T} (y_i - \overline{y})^2,$$

where y_i are values of the target variable for data point i and $\overline{y}$ is the average of the target variable on the dataset T. The Variance is an SSE normalized by a number of data samples in the node ($|T|$):

$$Variance(T) = \frac{\sum_{i \in T}(y_i - \overline{y})^2}{|T|} = \frac{SSE}{|T|}$$

If a node is entirely homogeneous, then the variance and SSE are zero. The higher the variance and SSE, the more impure the node is.

Example: let's assume the regression problem and two nodes consisting of data points with following values of target variable: $y_{T1} = \{3, 2, 5.5, 5, 18, 4, 3, 3.5, 4, 5\}$ $y_{T2} = \{2.7, 3, 2, 2.2\}$.

$\overline{y_{T1}} = 5.3$

$\overline{y_{T2}} = 2.475$

$SEE(T1) = (3-5.3)^2 + (2-5.3)^2 + (5.5-5.3)^2 + (5-5.3)^2 + (18-5.3)^2 + (4-5.3)^2 + (3-5.3)^2 + (3.5-5.3)^2 + (4-5.3)^2 + (5-5.3)^2 =$
$5.29 + 10.89 + 0.04 + 0.09 + 161.29 + 1.69 + 5.29 + 3.24 + 1.69 + 0.09 = 189.6$

$SEE(T2) = (2.7-2.475)^2 + (3-2.475)^2 + (2-2.475)^2 + (2.2-2.475)^2 =$
$0.05 + 0.27 + 0.22 + 0.07 = 0.61$

$Variance(T1) = 189.6/10 = 18.96$

$Variance(T2) = 0.61/4 = 0.15$

- **Misclassification error [10]** - rarely used metric for the classification problem. It is based on the most probable class label and represents how uncertain is the decision of assigning this label as a value of the target variable for the node. The formula is

$$ME(T) = 1 - max\{p_i\}.$$

Example (see image1.6):
$ME(AL) = 1 - 0.5 = 0.5$
$ME(AR) = 1 - 0.75 = 0.25$
$ME(BL) = ME(BR) = 1 - 1 = 0$

The impurity metrics allow the selection of the best (the most optimal) splitting rule based on the resulting subsets of data samples. The parameter can be tuned together with the type of the split to achieve better results.

Training the DT model means constructing the tree from the training dataset. Approaches for DT construction are a key problem of this work and will be detailed in the section Approaches to DTs creation. As the result of training, each list of the tree will be assigned with the concrete value of the target variable based on the instances in that leaf.

Predictions for the new data sample are obtained by passing it through the tree based on its features' values only. After the data sample is assigned to the specific leaf, the leaf defines the value of the target variable.

DT can also be illustrated as segmented space (see image 1.7). The sample space of all possible data points is subdivided into mutually exclusive segments corresponding to the leaves ($R_i, i \in \{1, 7\}$). Each data sample belongs to exactly one segment of space based on its features' values.

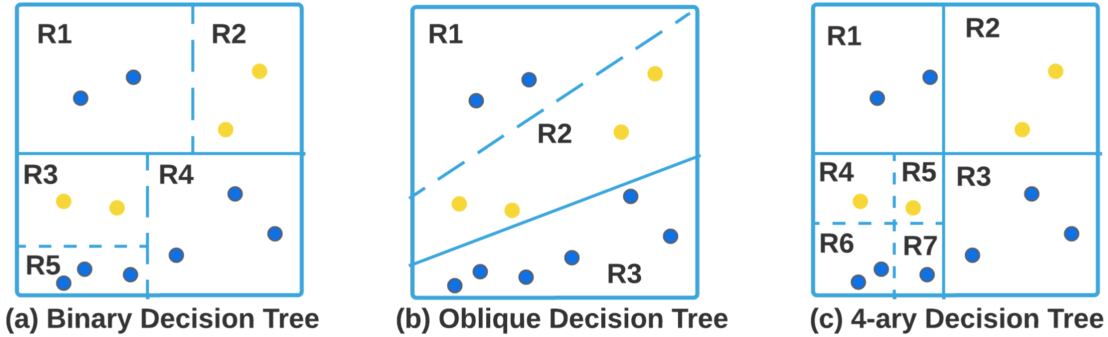


Figure 1.7 DT as segment space. Case (a) corresponds to the example of DT on images 1.3 and 1.5.

In the context of segmented space, predictions with DT can be viewed as a function that takes a vector of features as input and returns the target variable value estimation based on the corresponding segment:

$$\hat{y}(x) = \sum_{m=1}^{M} c_m I\{x \in R_m\},$$

where $R_m, m \in \{1, M\}$ are segments of the sample space defined by leaves and $c_m, m \in \{1, M\}$ are values of the target variable corresponding to those segments.

As with any ML algorithm, DT is also striving for optimality.

▶ **Definition 1.1.8** (Optimality)**. Optimality** is a trade-off between the model's complexity and its performance.

In the case of DT, the complexity measure is a tree's depth. Performance can be measured by any metric suitable for a particular problem.

▶ **Definition 1.1.9** (Full (complete) tree)**.** A tree, when each branch has the same length N is a **full tree** of depth N.

▶ **Definition 1.1.10** (Min-Depth Optimal Decision Tree)**.** Given a specific training dataset, **Min-Depth Optimal Decision Tree** is a minimum-depth complete tree that correctly predicts all training examples.

▶ **Definition 1.1.11** (Max-Accuracy Optimal Decision Tree)**.** Given the training dataset and specific depth d, **Max-Accuracy Optimal Decision Tree** is a complete tree of the chosen depth d, that maximizes the number of training examples that are correctly predicted.

1.1.3 H2O-3 Machine Learning platform

> *"H2O-3 is a fully open source, distributed in-memory machine learning platform with linear scalability." [11]*

The H2O-3 platform enables a user to develop and run his ML systems with a wide choice of supported technologies. The input data can be directly loaded from HDFS, Spark, S3, Azure Data Lake, or any other data source into H2O-3's in-memory **Distributed Key-Value store (DKV)** (see Definition 1.1.12). The computations can be done in Cloud or On-Premise, depending on the user's choice and the task size.

Key features of H2O-3 are [11]:

- **Wide range of ready-to-use algorithms.** Several ML algorithms, both supervised and unsupervised, are implemented in H2O-3 in a distributed manner and are optimized to work with big data. Some of them are GLM, GBM, XGBoost, Word2Vec, Random Forest, and many more.
- **Interfaces for different programming languages and graphical notebook-based interface H2O Flow.** Code can be written in Python, R, or Java, depending on the user's preferences, as the H2O-3 offers Python and R interfaces, and its engine is written in Java. The H2O Flow interface is interactive and does not require any coding.
- **AutoML.** H2O's AutoML allows users to compare different methods and compare which performs better on their specific problem. It is the possibility to automate ML workflow.
- **Distributed, In-Memory Processing.** The principal privilege of the H2O-3 is significantly speeding up the computations without degradations in the model's accuracy thanks to the distributed computations.
- **Simple Deployment.** The H2O-3 platform is easy to deploy for personal or business usage and offers Enterprise Support.

Data in H2O-3 are stored in **Frames**, which is distributed $2D$ data structure that consists of **Vectors**. Vector is also distributed data structure that contains values. In the context of the Frame, each element of the Vector is the row element. Both Frame and Vector have DKVs. The correct and optimal way of operating with Frame is passing it to the **MapReduce (MR)** task (see Definition 1.1.13). The Frame is split into several parts by rows and passed to each node as the array of Chunks, where each Chunk contains segments of the original columns of the Frame. As the result of the MR task, another Frame can be obtained along with a limited count of constant metrics.

Key-Value store

▶ **Definition 1.1.12** (Key-Value store)**.** A **key-value store**, or **key-value database**, is a type of data storage that stores data as a set of unique identifiers (**keys**), each of which has an associated **value**. This data pairing is known as a "key-value pair." The value can be the data being identified or the location of that data. [12]

The key can be of any data type that is not restricted by the database software, and the value can be anything, including a list of other key-value pairs. See the example of a key-value store in image 1.8.

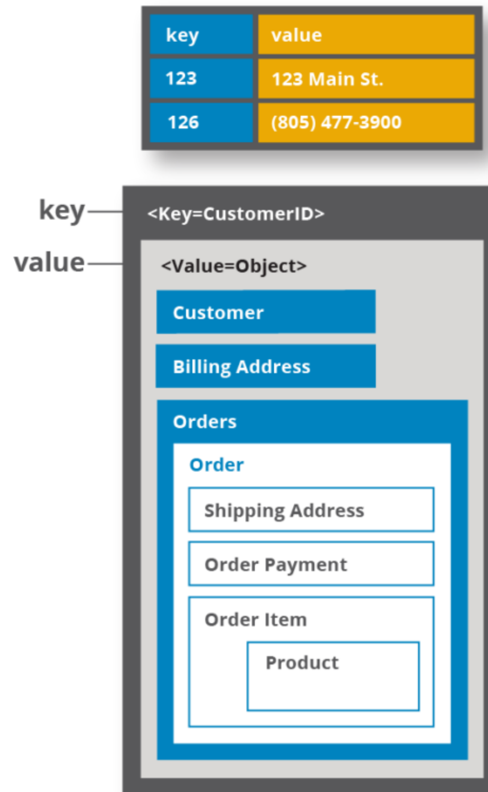


Figure 1.8 An example of a key-value store. [12]

There are a few advantages that a key-value store provides over traditional row-column-based databases. A key-value store can be very fast for read and write operations. Moreover, key-value stores are flexible and do not force a unified data structure. Also, key-value stores do not require placeholders such as "null" for optional values, so they may have more minor storage requirements and often scale better.

A **distributed** key-value store is built to run on multiple computing nodes working together, which allows working with larger datasets or doing the same job for less time. Also, the risk of losing data becomes lower if the data are duplicated across the nodes (**replication**).

MapReduce

▶ **Definition 1.1.13** (MapReduce)**. MapReduce** is a programming model and an associated implementation for processing and generating big data sets with a parallel, distributed algorithm on a cluster. [13]

A MapReduce program is composed of a **map** procedure, which performs filtering and sorting, and a **reduce** method, which performs a summary operation. The model is a specialization of the **split-apply-combine** strategy for data analysis. [14]

See MapReduce scheme on image 1.9.

It is inspired by the map and reduce functions commonly used in functional programming, although their purpose in the MapReduce framework is not the same as in their original forms.

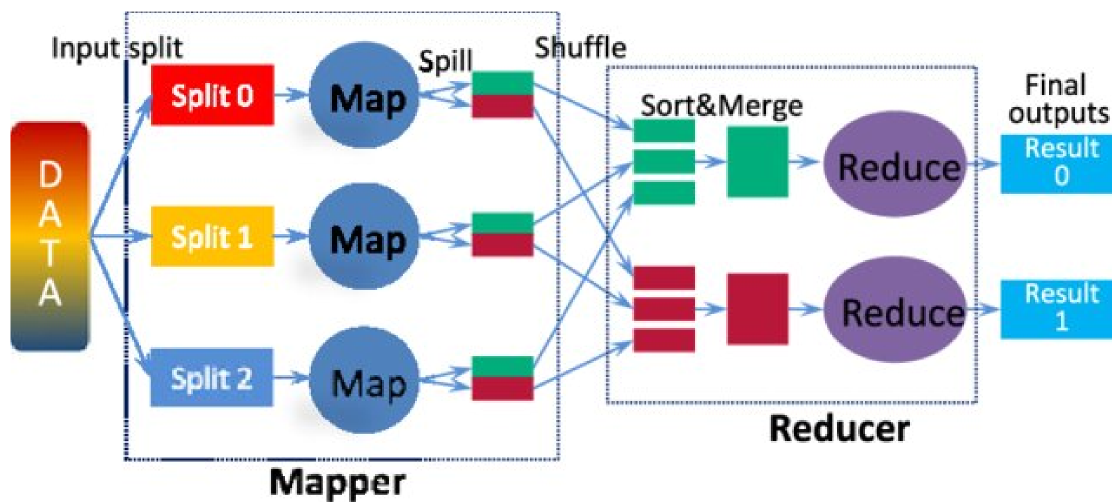


Figure 1.9 Hadoop MapReduce framework. [15]

The key contributions of the MapReduce framework are the **scalability** and **fault-tolerance** achieved for a variety of applications by optimizing the execution engine. A single-threaded implementation of MapReduce is usually not faster than a traditional (non-MapReduce) implementation; the benefits of the framework come with multi-threaded implementations on multi-processor hardware. [13]

This model is beneficial only when the optimized distributed **shuffle** operation (which reduces network communication cost) and fault tolerance features of the MapReduce framework are used. Optimizing the communication cost is essential to a good MapReduce algorithm.

1.2 Approaches to Decision Tree creation

In this section, I present the main approaches to building DTs. I start with the most widely used classical approaches and introduce novel or experimental approaches at the end of the chapter.

While working on this thesis, I decided to slightly change the terminology in the assignment of this thesis based on the knowledge I gained. In the context of this work, I consider the Standard Decision Tree (SDT) to be the concrete building algorithm, while the Decision Tree (DT) or Single Decision Tree (no abbreviation) means the algorithm in general, not depending on the building strategy. For clearance, consider the DT instead of the SDT in the assignment to this thesis.

It is essential to mention that in this section, I present general approaches for building DTs, not the specific algorithms that are based on those approaches. Specific algorithms and their implementations might combine several approaches and define their operators in such a way that optimizes selected objectives and are more suitable for specific domain or type of problems. Depending on the implementation, some algorithms may be used for classification and regression problems and different types of features. I am also reviewing possible implementations of approaches' components that can be used for designing a specific algorithm. Some approaches or implementations are relatively young and less frequently used, and others are historical and might be outdated, but still, all of them are attractive as the concept. [16]

The two main objectives for DTs are:

- better **modeling of the training data** - maximizing selected evaluation metric on the training data;
- **generalization** - maximizing the evaluation metric on the previously unseen data;

Mentioned objectives are generally common for all ML models and are relevant for the DT in particular.

Another aspect applicable to DTs is the three **optimality** (see Definition 1.1.8). Optimality is a tradeoff between the tree size and its prediction power on the training dataset. The less the size of the tree, the more optimal it is. The tree size can be measured by the depth, count of nodes, or another metric relevant to the approach and problem. Optimal trees are usually less overfitted and so more performant on new data. The problem of designing an optimal DT[1] is an NP-complete problem, as was shown by [17].

As for all NP-complete problems, several **heuristics** were introduced and became a basis for some DT's building approaches. Existing heuristic approaches can roughly be divided into two main categories [16]:

- **Bottom-Up** - the tree is constructed directly from the train data samples.

  Using some distance measure, the nearest data samples are merged together. Obtained clusters can be merged with other clusters or with the data points. The clustering continues until one single cluster containing all the training data is obtained. In this way, the most discriminative splits are done near the root, and more detailed splits appear near the leaves. The splitting rules in this approach are defined by the hyperplanes between the corresponding clusters.

  The leaves are all the initial nodes - one node per each data sample, but for the optimization, any of the obtained clusters can become a leaf and have a decision value assigned.

[1] Optimal in the sense of minimizing the expected number of tests required to classify an unknown sample

- **Top-Down** - the tree is constructed recursively from the root by splitting train dataset into two or more subsets based on selected splitting criteria.

  On each level, the best splitting rule is obtained by measuring the impurity of children nodes. The process is stopped when the defined depth is reached or when some stopping criteria are satisfied.

  This approach offers better control over data splits and performance throughout the tree growing.

In the following subsections, I will review several classical as well as novel approaches that belong to the mentioned categories. Each of the approaches led to several modifications and improvements that became specific algorithms. I will substantially describe and analyze approaches instead of concrete algorithms but will mention some essential implementation details when appropriate.

The design of the DT consists of the following **design steps** that may be defined differently for different approaches:

- **DS1:** Designing the optimal tree structure - depth and maximal children count for each node.
- **DS2:** Selecting the splitting rule or formula to be used in each node for splitting data between children nodes. Feature selection is part of this step.
- **DS3:** Deciding which nodes are leaves and assigning decision values to those nodes.

An introduced list of steps is typical for almost all approaches, yet each approach resolves them differently, giving more weight to any of them. A specific implementation of these decisions varies even within individual approaches making DTs flexible for different problems and data.

1.2.1 Standard Decision Tree (SDT) - Greedy approach

The most classic approach for DT creation is a **greedy top-down strategy** that is based on the **information-theoretic approach**, like a node's impurity. The history of this approach formally started with the paper of Ronald Fisher, in which he developed "Linear Discriminant Analysis" [18].

Some of the most common SDT algorithms still used today are CART (1984) [19], ID3 (1986) [20], C4.5 (1993) [21], and CHAID (1980) [22]. These algorithms have different complexities and performances, but all are based on the same greedy approach. All algorithms are actively evolving with the development of computation systems and technologies and may be flexible for different problems and datasets.

SDT utilizes the idea of a **recursive splitting** dataset by selecting optimal splitting rules that minimize the impurity of nodes from the root to the nodes - how impure in the sense of the target variable the children nodes are (see preliminaries for more details).

As mentioned above in the introduction to this section, three design steps should be made to design the DT. In the context of the SDT, these steps are the following:

- **DS1:** Tree structure is usually binary with limited depth. The depth is defined by the user based on the problem's complexity and available resources.
- **DS2:** Selection of the splitting rule is performed for each node separately by finding the most optimal split among all available features and splits by the selected impurity criteria.
- **DS3:** The tree growth is usually stopped by the termination condition, and the decision values are assigned by the selected strategy depending on the values of the target variable in the specific leaf.

The algorithm recursively splits data samples in nodes by the given splitting rule, starting from the root, which contains all the training samples. The splitting of each branch stops when the termination condition is fulfilled. Then the node is declared as a leaf, and the decision value is assigned to it based on the data samples in it.

See the pseudocode for the SDT creation in Algorithm 1.

Algorithm 1 $growTree(DatasetT)$

1: **if** $stopping_conditions(T)$ **then**
2: $selectLeafLabel$
3: $createLeaf$
4: **return** $leaf$ (recursion - termination condition)
5: **end if**
6: $findBestSplit$
7: $createNodeWithSplittingRule$
8: **for** each possible outcome of splitting rule **do**
9: $selectDataSubsetForTheSplittingRuleResult$
10: $growTree(subset)$ (recursion)
11: $connectChildSubtreeToCurrentNode$
12: **end for**
13: **return** $root$ (recursion)

Splitting rules

The task of a splitting rule is to split the input set of data samples into two or more mutually exclusive and collectively exhaustive subsets in such a way that minimizes the impurity of the output subsets.

The binary trees are mostly preferred because of the boolean nature of most conditions, but that is not a rule. Based on the feature type and values range, the node can have any number of children.

Binary trees often contain the splitting rule in the form of single-feature true/false of less/greater condition when the samples responding as true/greater are going to the right split, and the rest is going to the left split. For the categorical features, some set of values is given as a parameter, and the splitting rule checks if the feature value of the current data sample belongs to the given set or not (mapping feature values to the output subsets). Most conditions contain parameter like threshold or mapping for categorical values.

The main objective in choosing any splitting criteria is to make the data in the children nodes purer. To achieve this, the impurity function has to be defined and minimized (see preliminaries for definitions of the specific impurity metrics). The **Gini impurity** and **Entropy** are mostly used for computing the impurity criterion in classification tasks. In the case of regression problem, the common criterion to minimize is **Variance** (reduction in variance).

The ”locally optimal” splitting rule is defined by the minimum value of average impurity criteria for all children relative to the number of samples in each child node. The Information Gain (IG) is also frequently computed as the difference between the current and the children's impurity. The IG shows how significant the improvement in impurity criteria is in the children nodes relative to the parent node. It can also be calculated between any levels of the tree. Calculated between the root and leaves, it shows a general IG of a tree and can be used for evaluating the goodness of a tree. The bigger it is, the better the split/tree.

Usually, only one feature is used for splitting in each node, so the sample space is divided by hyperplanes parallel to the feature axes. Using more than one feature makes the DT building much more complex, and the count of possible splits grows exponentially. In this case, the subset of features has to be selected as well as the operation and condition that may combine selected features in any mathematical way. The advantage of this approach is that the relationships between features can be taken into account, unlike the standard single-feature approach. Such trees (Oblique trees) are usually smaller and have better performance.

Binning

For the numerical features, the task is usually to find the suitable threshold for splitting samples into two (possibly more) splits - left (less or equal) and right (greater). For categorical features, the split can be made by subsetting the values into two (or more) sets. For binary trees, all potential thresholds should be tried, and the split impurity should be measured to find a suitable threshold that provides the most optimal split. As the categorical features do not have order, all combinations of values should be tested to find the optimal mapping of feature values and splits.

To optimize searching for the best split when the training set is vast, the **histogram/binning approach** of working with a dataset can be used. The idea is to aggregate the numbers of data samples with the same attribute values (or the value from a specific range) and operate with these statistics instead of the original dataset.

The feature range is divided into bins; for each bin, the counts of data samples and their decision values are accumulated. Any other statistics can be calculated for the obtained bins. Accumulating values allows saving them in a bin and reusing them while selecting the best split. Also, the amount of potential candidates for the threshold, in the case of the numerical feature, is reduced to the number of bins.

Possible types of histograms are:

- **Equal-width** - all bins are defined by an equal range of feature values, and the feature range is equally divided into bins' ranges independently of the data distribution.

  This type of binning is fast, but data samples are not distributed evenly between bins.

- **Equal-height** - bins are defined by the equal amount of data samples belonging to them.

  Defining bins' ranges is more complicated in this case than in the equal-width binning, but all obtained bins contain approximately the same amount of data samples.

- **Any bins defined by the user** - flexible definition of bins' ranges allows the user to apply some external knowledge of the data to effectively split them into the bins.

Binning needs to be performed for each feature independently as the features are usually independent, and the amount of data samples with specific feature values does not depend on the values of other features. Also, the statistics and features' ranges change after splitting the dataset, so in each node, the histogram has to be updated before the next splitting.

Termination rules (stopping conditions) and pruning

Greedy recursion algorithms need a stopping condition (also called pre-pruning or early stopping) to create a finite tree.

The algorithm itself generates some trivial conditions:

- **Absolute purity of the node**, so there is no reason to split the data.

- **Inability to split data** in the node by any feature because the values of all features are equal through all the data samples (data redundancy).

These conditions, though, lead to enormous **overfitted** trees. To avoid this, two strategies could be used - defining relevant **stopping conditions** and **tree pruning**.

Stopping conditions:

- **Impurity level** - if all the labels in the subset are the same (the node is pure), there is no sense in splitting this node further.

  Also, the user can set an acceptable impurity level (threshold) to avoid overfitting.

- **Max depth** - the depth of the tree can be manually limited to avoid excess levels.

  The deeper the tree is, the more complex it is, so the optimal maximum depth depends on the complexity of the problem and data.

- **Information gain** - likewise the purity condition, the information gain condition uses a measure of impurity to stop expanding the tree.

  The difference is that the improvement of node purity is considered in this case. If the information gain is too low for all possible splits, it is profitable to stop expanding as it no more helps the model.

- **The number of samples in the leaf -** the customization of trivial condition with setting the minimum number of samples in the node to allow split.

  It can also be interpreted as a minimal number of data samples in the child node, so if no split with a suitable amount of data samples in each child node exists, then the current node becomes the list.

The other way of building an optimal-depth tree is to apply pruning (specifically **post-pruning**). Pruning simplifies a full decision tree[1] by removing the **weakest rules** - rules that can be removed without a significant decrease in model performance. This technique allows first to build the tree that predicts the training set perfectly and then to prune it to optimal size.

See the example of DT before and after pruning in image 1.10.

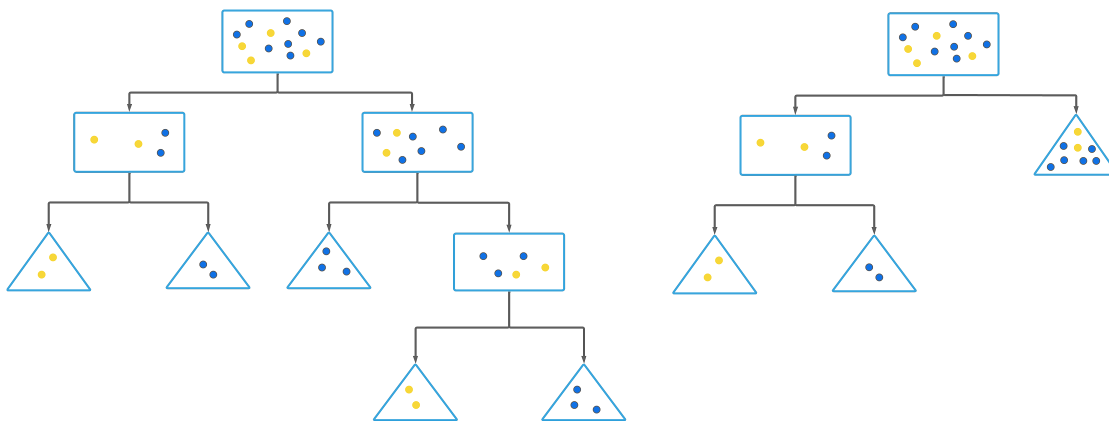


Figure 1.10 Example of DT before (left) and after (right) pruning.

Pruning approaches [23]:

- **Minimal Cost-Complexity Pruning** (Breiman et al., 1984 [19], Prodromidis, 1999 [24]).

  The algorithm defines the cost-complexity measure $R_\alpha(T) = R(T) + \alpha|T|$ where $\alpha(\geq 0)$ is a parameter and $|T|$ is the number of lists in T and $R(T)$ is the value of selected loss function for the tree T. The bigger the parameter alpha, the more branches are pruned, and the less complex the tree is. The model's generalization improves, but the train loss function is increasing. The task is then to find a trade-off between complexity and impurity.

- **Critical Value Pruning** (Mingers', 1987 [25]).

  The method uses the goodness of split calculated while tree building and specifies a critical value for this measure. The value of the measure reflects how important and strong the given node is. The nodes are then pruned if they do not reach the specified critical value unless a node further along the branch does reach it.

- **Pessimistic Error Pruning** (Quinlan, 1986 [20]).

  The algorithm aims to enable the absence of a separate test set. The misclassification rates (or, in general, the loss function) on the training data are too optimistic, and the pruning that is based on the train set does not simplify the tree effectively. This algorithm uses the continuity correction for the binomial distribution to obtain a more realistic estimate of the misclassification rate.

[1] The tree with only trivial stopping conditions - absolute purity or features equality

Pruning reduces the complexity of the final tree and, as a result, reduces overfitting. Pruned trees are shorter, simpler, and easier to explain. Post-pruning is effective when finding the best split is inexpensive; otherwise, the full tree creation would be insufficient.

Assigning decision value to the leaf

Assigning a decision value for the leaf is a straightforward step. The decision value in the leaf has to represent the training data samples belonging to this leaf. For the classification task, the major class of the training samples belonging to the leaf is usually assigned to this leaf. For the regression task, the mean or weighted average of values of the target variable is used.
Different techniques can be selected instead of the mentioned classical ones:

- Mode - the value that appears most frequently in a set of data samples;
- Median - the value separating the higher half from the lower half of a data samples;
- Closest value to the mean - the actual value that appears in the set of data samples in the leaf and is closest to the calculated mean of the values of the target variable in that leaf;
- Any custom rule relevant to the data and the problem being solved.

1.2.2 Oblique decision trees

Standard decision trees split the feature space by only one feature at a time, so the splitting hyperplanes are parallel to the axes of the features. Oblique decision trees consider combinations of the features' values, making it possible to utilize dependencies between different features and allow any splitting hyperplanes. Feature combinations also allow to achieve a better separation with a smaller number of splits, so the tree is shallower. See example of oblique decision tree in image 1.11 and the comparison of Oblique Decision Tree and SDT on images 1.12 and 1.13.

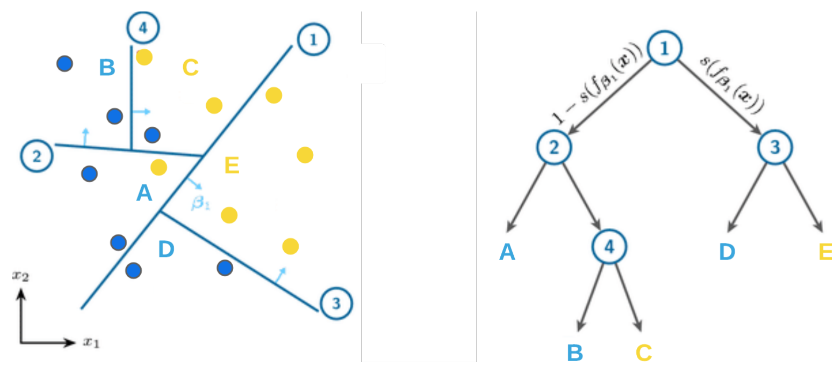


Figure 1.11 Example of oblique decision tree represented as segmented space (left) and tree structure (right).

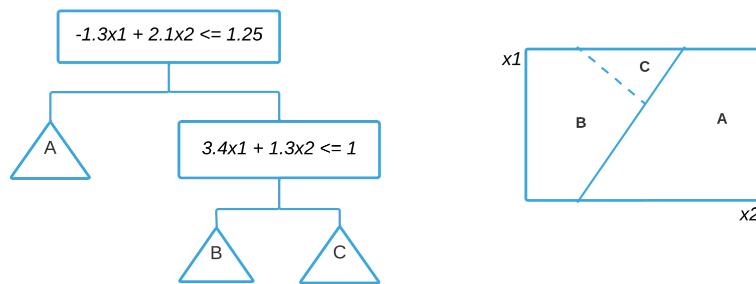


Figure 1.12 Oblique decision tree

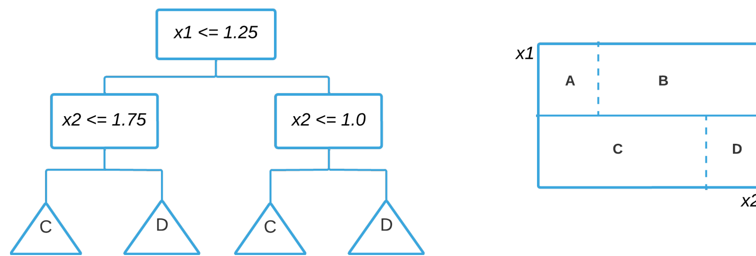


Figure 1.13 Standard Decision Tree

Three main design steps that need to be made for any building strategy are defined for oblique decision trees as follows:

- **DS1:** Designing the optimal tree structure - depth and maximal children count for each node can be set by the user. The count of children nodes can also depend on the selected splitting rules;
- **DS2:** Selecting splitting rules, including features selection - the most optimal split is selected among all possible splits. For oblique DTs, all possible combinations of features and operations with them should be considered to find an optimal split.
- **DS3:** Deciding which nodes are leaves and assigning decision values can be done in any way specific to the selected building approach. This step can be similar to the SDT.

Almost any building strategy can be used to build the oblique decision tree instead of the SDT [26]. The only difference is that the splitting rule contains a combination of features, so in each node, the three aspects have to be defined/selected:

- **The subset of features** for the current node.
- **The relations between each feature.**

  The relation can be linear, comparative, logical, or any other. Even the features of different data types can be combined in one node with a suitable function. The problem is that there is an uncountable amount of such operations that make this step more complex.
- **Splitting condition.**

  If the relation between features results in some value, then that value can be interpreted as a new, categorical, or numerical feature. The subset of values or the threshold needs to be defined to be able to split data to the children nodes.

Compared to SDT, oblique trees are often smaller and more accurate as the same operations can be much more compact. In addition, the relationships between features can bring a significant improvement that could not be reached by separating the space only by hyperplanes parallel to the features' axes.

1.2.3 Evolutionary approach - genetic algorithms

The Genetic Algorithm (GA) can be used for building optimal decision trees as the DT is a complex structure, and the problem of finding the optimal instance is a hard optimization problem.

There are multiple ways to apply GA to the DT construction; each has plenty of modifications and can be customized to the specific problem [27]. The evolutionary approach is inspired by nature and utilizes the rules and processes of natural evolution.

The GA itself creates a **population** of **individuals** (instances of solution) and, in iterations (**epochs**), performs genetic operators - **crossover**, **mutation**, and **selection**. The result is one or more solutions that are the most optimal by defined criteria (**fitness** value).
The **encoding** and **decoding** must be defined for the specific problem to apply the genetic operators mentioned above. All the operators - crossover, mutation, and selection must also be defined together with necessary functions and constants such as fitness function, size of the population, number of epochs, etc. See genetic algorithm cycle on image 1.14 and pseudocode in Algorithm 2.

Algorithm 2 *EvolutionaryDecisionTrees*

1: Generate an initial population of trees
2: Compute the fitness of each individual
3: **repeat**
4: Select individuals for crossover based on their fitness
5: Apply crossover and mutation to create new trees
6: Compute the fitness of each new tree
7: Update the current population with new individuals
8: **until** a stopping criterion is met

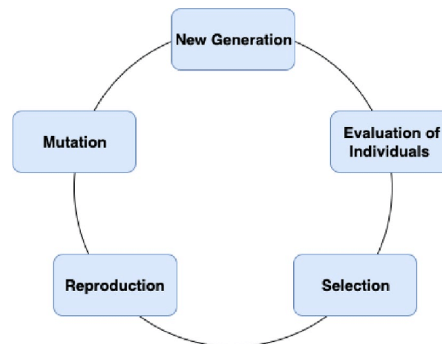


Figure 1.14 Genetic algorithm cycle. [28]

In this section, I will describe specific definitions for DT and analyze the specifics of the GA for DT construction.

Three designed steps are defined for the evolutionary approach as follows:

- **DS1:** Designing the optimal tree structure - depth and maximal children count for each node can be set by the user. The count of children nodes can also depend on the selected splitting rules. Moreover, the size of a tree might be changed while crossover and mutation which can be allowed or denied by the tree validation.

- **DS2:** Selecting splitting rules, including features selection. Unlike previous approaches, the evolutionary approach does not aim to select the most optimal splitting rule right away. Depending on the implementation of the initialization step, splitting rules together with the selections of features are either random or defined by another building approach like SDT. In the process of crossover and mutation, initial splitting rules can be changed multiple times.

- **DS3:** Deciding which nodes are leaves and assigning decision values are also performed during the initialization of the population. Leaves can be created when the termination condition is fulfilled or in a random way. Assigning decision values can be either random or based on the data in the leaf. Unlike the greedy approach, there is no subset of data in a leaf while initialization the tree, so if the decision value has to be assigned based on the data, an additional step of initialization is needed.

1.2.3.1 Encoding and decoding

Encoding/decoding of decision trees for GA is pretty straightforward. The individual is the DT itself; the structure is kept and is part of the problem being solved. As described in the following sections, the genetic operators manipulate with subtrees and nodes. So the tree structure has to be preserved while storing it in memory. The tree can be effectively implemented as an array that preserves parent-child relations so that children of the node on index n are on indexes $2n$ and $2n+1$. For optimization, the features and class labels can be represented by numbers (indexes), so it takes less space and complexity. The histogram approach (binning) can also be applied to initial data in the standard way (see SDT section for details).

1.2.3.2 Initialization of population

The starting/initial population is usually initialized randomly or with some explicitly given solutions that could have been created by other approaches or designed by the expert. Population size is the model's hyperparameter that needs to be tuned together with genetic operators. The initial population has to be diverse enough to avoid fast convergence to one type of individual.

1.2.3.3 Fitness function

Each individual in the population is evaluated by a fitness function that measures its quality as a solution. For a DT, the fitness function is usually the same metric measured while evaluating the model on test data. The same data set should be used to compute the fitness value for different individuals as it is nothing other than the model validation. Therefore for the classification trees, accuracy is one of the metrics frequently used as a fitness function, and for the regression problems, it is MSE. The tree complexity can also be considered to penalize too complex trees. Any combination of metrics can be used by defining weights and summarizing criteria:

$$Fitness = \alpha_1 f_1 + \alpha_2 f_2 + \dots,$$

where α_i is a weight and f_i is a metric.
The fitness function compares the solutions to sort them from the better to the worse to select the best part of the population. The fitness function should not be too complex in order not to be the model's bottleneck; it has to be fast and unambiguous.

1.2.3.4 Selection and crossover

The most special operator in GA is the crossover which can sometimes be replaced by mutation alone. It is the recombination mechanism: information from two (or any count) individuals is mixed and passed to their offspring(s). Several options exist when it comes to selecting the parents that will be used to create the next generation (selection step). The most common are the following approaches:

- **Roulette wheel** (Fitness proportionate selection): The higher the fitness value of the individual, the higher its probability of being selected for crossover. The selection process looks like the roulette wheel, where each individual has a wheel area proportional to its fitness value.
- **Tournament selection**: A subset of the individuals is randomly (with equal probability) selected from the population. The individuals with the higher fitness value within this set are selected for crossover.

If it is not prohibited by design, each individual can be selected multiple times, which enables most fit individuals to spread their genes to more children within one epoch.

Children (also called **offspring**) are obtained by combining selected pairs (or any other number - depending on the design) of parents from the current generation. The preferred definition of crossover for the decision trees is randomly selecting a node in each parent tree and swapping them in new tree instances. The new individuals (offspring) are thus formed by replacing the subtree of the first parent with the one from the second parent and vice versa. In some cases, the crossover might lead to invalid instances. For a decision tree, for example, the mixing subtrees might be a problem because it might create invalid trees in the sense of redundant attributes or even contradictions that will lead to empty data subsets in nodes. It can be resolved by pruning invalid branches or validating branches during crossover - crossover will be repeated until the offspring is valid. See example of crossover on image 1.15

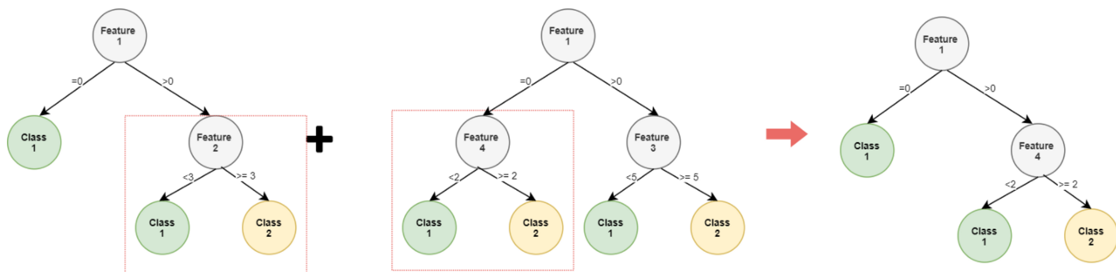


Figure 1.15 Crossover. [29]

Another approach is only to apply mutation on the parent(s) copy. It excludes mixing genes from different individuals but can be helpful when mixing is not possible due to the structure of the solution.

1.2.3.5 Mutation

After obtaining new offspring, the mutation step is performed naturally on new individuals. Other randomly generated values replace some randomly selected parts (depending on encoding) of the offspring. In the context of decision trees, the separate nodes are replaced by new ones. The whole subtree stays as it is; the only node itself is changed. Depending on the design, only the threshold / values mapping is changed, or the attribute in the node is also

changed. The other extension of mutation is turning selected internal nodes into lists by removing their subtrees, which can be interpreted as random pruning. See the example of a mutation in image 1.16

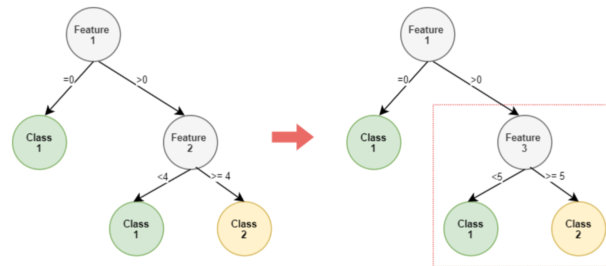


Figure 1.16 Mutation. [29]

The mutation helps with keeping and extending diversity in the population and enables searching a broader space by bringing previously unseen splitting hyperplanes. The mutated instances should also be validated, as the instance (DT) can become invalid after mutation. The mutation is then performed again until it is successful.

1.2.3.6 Updating the population

In order to enable converging and avoid complexity growth, the population is kept of the same size in each generation. It is done by removing the oldest individuals and the individuals with the lowest fitness by some selected strategy.
Some approaches just replace the original generation with the newly generated offsets. It may not be the optimal approach as we are keeping in population the individuals with low fitness value. Another way that does not consider the individual's age is to sort all individuals and take the fittest ones to the next generation.
Elitism: Individuals with the highest fitness value are directly carried over to the next generation, ensuring the most successful individuals in the population. The rest of the population is filled with new offspring by any previously described strategy.

1.2.3.7 Stopping criteria

A few main stopping criteria used for GA are:

- **Fitness no more improves with new generations**, although the population is diverse enough. The small diversity of the population can be the reason for early converging. To avoid this, a more active mutation step should be applied, or elitism should be limited.
- **The required fitness value was reached**. If the required fitness is defined in the beginning, we can stop iterating once the solution is found.
- **The defined number of epochs exceeded**. The maximal number of epochs is defined to ensure that the algorithm will stop. If the result is unsatisfactory, then the population might not be diverse enough, or the number of epochs is too small.

The GA on decision trees enables searching the space that might not be searched by greedy building strategy. The GA is one of the classic optimization approaches for complex structures, and the DT is one of those.

1.2.4 SAT-based Decision Tree Learning

Another optimization approach as GA is Boolean Satisfiability (SAT). [30] This approach encodes the problem of DT building into SAT and uses SAT solvers to find an optimal solution. The SAT solver is used in a black-box fashion; any existing SAT solver can be used to solve encoding problems.

▶ **Definition 1.2.1** (SAT)**. Boolean Satisfiability Problem** or **Satisfiability Problem (SAT)** is a problem of assigning values to Boolean variables to satisfy a given Boolean formula or to decide that formula is unsatisfiable. [31]

▶ **Definition 1.2.2** (Satisfiability)**.** The formula is **satisfiable** if it is true under some assignment of values to its variables. A formula is **unsatisfiable** if there is no assignment to its variables that makes it true. [31]

SAT formulas are represented in Conjunctive Normal Form (CNF) and are defined over a set of Boolean variables - variables that can be either true or false (0 or 1). A SAT formula is a conjunction of clauses, each clause is a disjunction of literals, and each literal is either a Boolean variable or its negation. An assignment of the Boolean variables satisfies a clause if at least one of its literals is true. The SAT problem consists of finding an assignment of the variables that satisfies all clauses in a formula [31].

The **MaxSAT** problem is the optimization variant of the SAT problem and consists of finding an assignment of the variables that maximizes the number of satisfied clauses.

Currently, the SAT problems can be effectively solved by existing SAT solvers.

▶ **Definition 1.2.3** (SAT solver)**.** A **SAT solver** is a computer program that aims to solve the Boolean satisfiability problem. It takes a formula over Boolean variables as the input and outputs, whether the formula is satisfiable or unsatisfiable.

Boolean satisfiability is an **NP-complete** problem in general [32, 33], which means that only algorithms with exponential complexity are known. Despite this, efficient and scalable algorithms for SAT were developed. SAT solvers often begin by converting a formula to CNF. They are often based on core algorithms such as the DPLL algorithm [34], but incorporate a number of extensions and features that make the solver efficient.

To find the optimal DT for the given dataset, an SAT solver is called iteratively with different sizes of the problem; in the case of the DT, the size of the problem is maximal tree depth and/or count of nodes/leaves.

The design steps for the SAT approach are defined as follows:

- **DS1:** Designing the optimal tree structure - depth and maximal children count can be defined by the user or can be part of the task. The SAT-solver is then run for each selected structure.
- **DS2:** Selecting the splitting rule. Boolean variables are represented by true and false values of a given boolean variable. All other data types are encoded to binary variables.
- **DS3:** Deciding which nodes are leaves and assigning decision values to those nodes. The leaves assignment is expressed in boolean variables that are tuned while solving SAT. The values assignment is expressed in SAT clauses.

To be able to apply an SAT solver to the problem, the problem needs to be encoded to SAT. Different approaches to encoding DT to SAT were presented:

- **Node-based encoding** [35, 36] An encoding is based on the way the nodes are indexed - the root node corresponds to the node $V[1]$, and for each node $V[i]$, the left child corresponds to $V[i\text{Ö}2]$ and the right child to $V[i\text{Ö}2+1]$. For example, if the binary coding of index i is 1011, then the node $V[i]$ is reached by taking the right branch of the root, then the left branch, and finally the right branch twice. See example of nodes indexing on image 1.17. [35]

  The idea of this encoding consists of assigning features to the nodes and arranging the training examples in the leaves of the tree. All training examples placed in the same leaf must belong to the same class. Moreover, the left child of a node corresponds to the feature value "*False*" and the right child corresponds to the feature value "*True*".

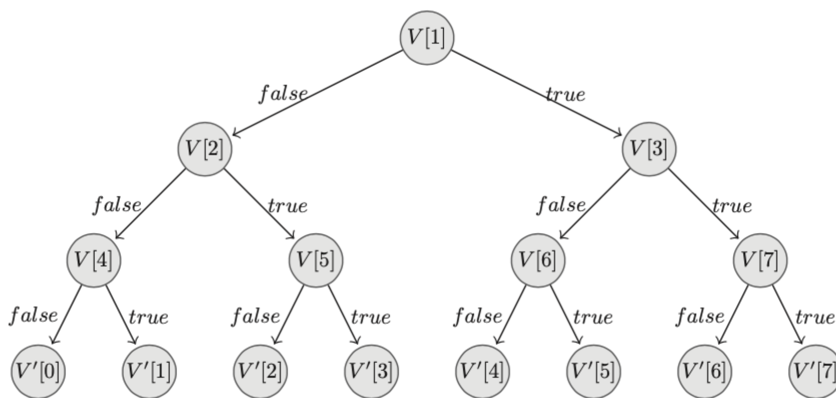


Figure 1.17 Illustration of nodes indexing in a binary tree. [35]

  Several types of propositional variables are used in clauses to formulate constraints that a perfect decision tree of a given depth should satisfy:

  - Assigning specific data samples to a specific leaf. It also describes whether the leaf is a right or left descendant of a specific node.
  - Labeling specific nodes with a specific feature.
  - Labeling specific leaves with a specific class.

- **Partition-based encoding [37].** SAT-based Decision Tree Learning for Large Data Sets: The problem of finding a DT of a given depth for a classification instance I is formulated by partitioning the dataset. This partitioning definition is converted into a propositional CNF formula (I,d), which is satisfiable if and only if the DT it describes correctly classifies the instance *I*.

- **Path-based encoding [38]**. Each separate branch (path from the root to the leaf) is encoded in a way that enforces the relations between separate branches so that the final DT is a valid binary tree. Each path is encoded as a sequence of 0's and 1's so that 0 is a step to the left and 1 is a step to the right in each node. Then, the paths are considered in lexicographic order. Also, for each step, we need to remember if the path has already ended and what prefix is shared with the previous path.

Classical SAT-based approaches have limitations on the size of the problem - the depth of resulting DT. The currently best method (due to Avellaneda, 2020 [35]) can produce decision trees of a depth of up to 12. Potentially the tree depth is connected with the dataset size, so big datasets are usually problematic for processing. For the large datasets, a different method was proposed that combines the scalability of heuristic methods and the strength of encoding-based exact methods. A proposed approach for large datasets follows the principle of SAT-based Local Improvement (SLIM), which starts with a solution provided by a fast heuristic. After that, the solution is improved by repeatedly applying the SAT-based method locally. The main idea of local improvement is finding a more compact T subtree instead of an initial T by utilizing the local instance I, which corresponds to the subtree T in the initial DT T. The partitions-based encoding is used to encode subtrees and pass them to the SAT-solver. The resulting tree then contains newly generated subtrees, so it is more optimal than the one created by the initial heuristic. Image 1.18 shows an example of this process. [37]

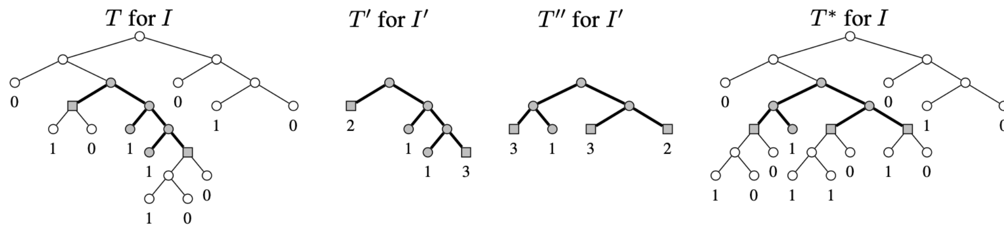


Figure 1.18 Replacing original subtree with newly generated one while local improvement. [37]

1.2.5 Incremental approach

All of the previously described algorithms are non-incremental - they create a DT from the given dataset and can not adapt the solution to the new data samples.

▶ **Definition 1.2.4** (Incremental learning task)**. Incremental learning task** - task when the new training samples appear after the model was already trained.

For incremental learning tasks, an incremental algorithm is preferred as it is more efficient to review and validate existing hypotheses than to build a new instance each time new data samples become available [39]. The cost of building a new DT might be too high to apply a non-incremental method to an incremental learning task. See the incremental learning flow on image 1.19.

The design steps for the incremental learning approach are defined as follows:

- **DS1:** Designing the optimal tree structure - depth and maximal children are defined by the user. Important part of the tree structure is keeping statistical data about splits in the nodes.
- **DS2:** Selecting the splitting rule. The splitting rule and the features selection can be defined in any way. If multiple features are selected, the tree becomes Oblique DT.
- **DS3:** Deciding which nodes are leaves and assigning decision values to those nodes. The leaves are assigned based on the termination criteria and the decision values are calculated based on the data samples in the leafe.

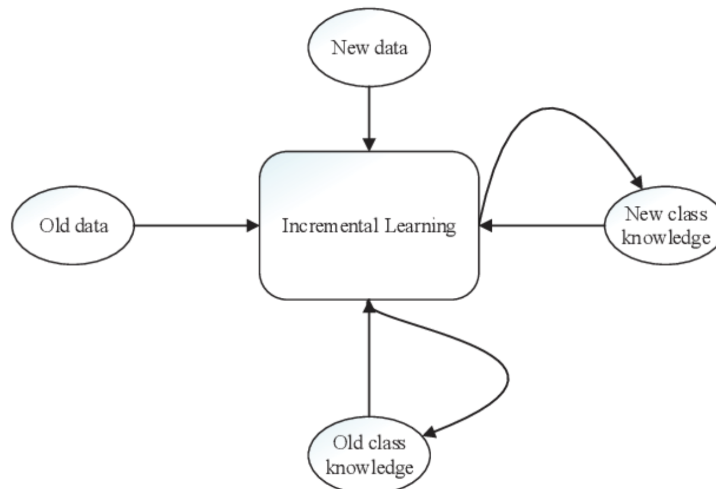


Figure 1.19 Incremental learning flow. [40]

The classic incremental algorithm for DT building is the incremental analogy of the classic ***ID3*** algorithm (see subsection:sdt) - an algorithm named ID4 (Schlimmer and Fisher, 1986 [41]). The ID4 algorithm updates the decision tree once a new training instance appears. The structural specialty of this algorithm is that all information needed to compute impurity is kept at that node.

In the case of the ID4 algorithm, the impurity is measured by the $E - score$ - the result of computing Quinlan's *expected information function* E of an attribute at a node (see [41] for a detailed definition). The information needed for computing the $E - score$ in a node consists of values counts of all attributes at each node. Selecting an attribute with a lowest $E - score$ is equivalent to selecting an attribute with a highest information gain [41].

Having all the needed information stored in the node, it is fast to determine which attribute has the lowest $E - score$ at any moment. If the current attribute in the node does not have the lowest $E - score$, then it is replaced by a non-test attribute with the lowest $E - score$. Because the counts are kept in the node, there is no need to use a train set each time. Refer to [41] for implementation of the ID4 algorithm.

However, it turned out that many concepts are not learnable by *ID4*, even though they are learnable by *ID3*. The ID4 algorithm builds the same tree as the basic ID3 only when there is a clear best choice on attribute in every node in terms of its $E - score$. When the order of features changes during the training, *ID4* discards all subtrees below that node, so certain concepts stay unlearnable by the tree. [42]

As the solution, a modified algorithm was presented - ***ID5R*** (Utgoff, 1989 [42]), that is guaranteed to build the exact same decision tree as ID3 for a given set of training instances. The ID5R algorithm is a successor of the ID5 algorithm (Utgoff, 1988 [43]) introduced earlier by the same author. The main difference of ID5R is that the subtree is not discarded but restructured in a way that the new attribute becomes a root. This process is called **pull-up**, which is the key to consistency with the training dataset and the DT built by the ID3 algorithm. For further information and a detailed description of the ID5 and ID5R algorithms, please refer to the original publications [43] and [42].

1.3 ML models explainability with decision trees

▶ **Definition 1.3.1** (Explainability)**. Explainability** is the concept of machine learning that allows humans or other algorithms to understand the model and its outputs in an acceptable way. [44]

Explanations are essential for the human user to trust the model and its outputs. In most use cases, using the model output has a real-world positive or negative impact, and mistakes might not be acceptable.

The real world goes through many layers while being analyzed by ML. It needs to be encoded for the model to be able to work with it, and then, it needs to be decoded for the human to see it as the real world again. The explanations must be done to allow the human user to see that the model output corresponds to the real world. Image 1.20 shows the big picture of explainable machine learning concerning the real world.

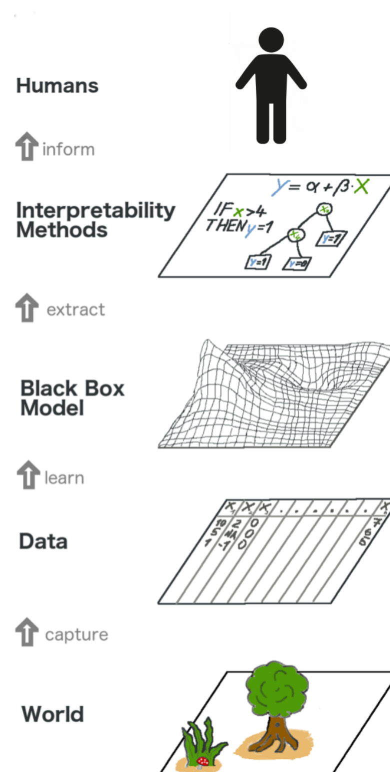


Figure 1.20 The big picture of explainable machine learning. [45]

Different ML algorithms have different internal structures and ways of making decisions that are more or less understandable by a human. Improving the ability to explain different ML systems is still an active research area. Some classical ML approaches based on rules, decision trees, or linear models are more understandable by humans as long as they are not very large (the number of rules, the size of the decision trees, or the number of explanatory attributes are adequately small). More complex systems can still be explained by the human who owns some knowledge about the system being modeled (domain expert). Having the domain expert in a loop can be one approach to ML explainability. However, it is not scalable as a human is not able to provide explanations for a high number of requested inputs.

▶ **Definition 1.3.2** (Black-box)**.** In science, computing, and engineering, a **black-box** is a device, system, or object which produces useful information without revealing any information about its internal functions. The explanations for its conclusions remain opaque or "black". [46]

In terms of ML, a black-box model is a sufficiently complex model that is not straightforwardly interpretable by humans (see image 1.21) [47]. Typical black-box models are neural networks - deep neural networks in particular. The structure of a black-box model might not be controlled by humans but by the training algorithm.

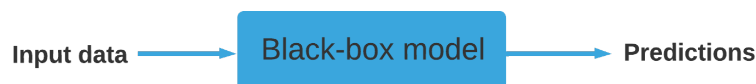


Figure 1.21 Black-box ML model.

The opposite of a black-box is a white-box. Its results and structure are transparent and known and can be easily analyzed by the user. A typical example of a white-box model is DT, as its structure is intuitive for humans, and each model's decision is evident.

There is no given threshold between black-box and white-box models in ML. Typical representatives of the white-box approach might become black-box if their size and complexity are large and vice versa.

The more complex the model is, the more it becomes a black-box for the end user and the more additional actions it needs to generate explanations. The deep learning models bring critical complexity for explainability, as being black-box, those make inexplicable decisions, and their inner structure is not as straightforward as previously. It is much easier to program an artificial neural network and train it than to understand why it gives a particular output for a particular input. With the popularity of deep learning models, their explainability has become a more significant challenge for modern research.

Referring to the previous observations, there are two types of ML explainability [48]:

- **Inherent:** Some models are self-explainable - can be explained out of the box thanks to their simple structure, limited size, and human-understandable logic. These algorithms include linear models, decision trees, rule sets, decision sets, generalized additive models, and case-based reasoning methods. In the case of a decision tree, it is possible to convert it to a set of rules and explain each data sample with the chain of conditions it satisfies.
- **Post-hoc:** When the model structure is not clear, or the model size is too big, we approach the model as a black-box and explain it based only on its outputs. This approach requires additional actions after the model is trained, so the model explainability becomes a more complex problem.

The fact that self-explainable models have a more straightforward structure than the black-box ones results in the observation that these models have less modeling power in the sense that they are less performant on more complex problems. This observation has led to the so-called explainability vs. performance tradeoff (see image 1.22 - the more performant a model is, the less explainable it is, and vice versa. This comparability applies to very complex problems that require more robust algorithms. For simple problems, this tradeoff may not be reasonable or can even be reversed as simpler models are suitable to model simpler problems, and the black-box model would only bring overhead.

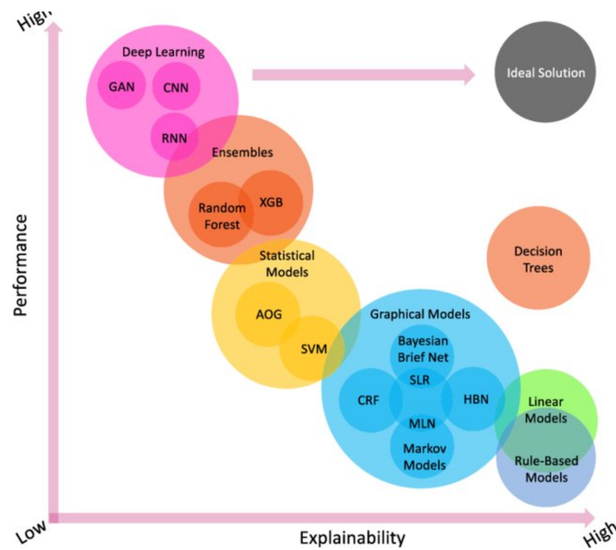


Figure 1.22 Explainability vs. performance tradeoff. [49]

1.3.1 Explanations via Surrogate Models

One of the approaches used for post-hoc explainability is building a surrogate model that would approximate the original black-model. [48] Training a surrogate model does not use the inner structure of the black-box model; instead, it uses the model's outputs (prediction function) to analyze the model, which makes the method model-agnostic.

The surrogate model takes the same input as the original black-box model and predicts the same target variable, though it is trained on the outputs of the black-box model instead of the original values of the given target variable. The purpose of (interpretable) surrogate models is to approximate the predictions of the underlying model as accurately as possible, but not ideally. The surrogate model always makes more mistakes than the original black-box model. As explained earlier, the self-explainable models are less performant on the highly-complex problems than the black-box ones. Naturally, the surrogate model used to explain the more complex black-box ML model will be less performant than the original black-box model (assuming the problem being modeled is complex enough). Otherwise, we do not need a black-box model and can use a simpler explainable model.

Though, the desired outcome of the surrogate model is not an output itself but the structure of the the model. As the surrogate model is self-explainable, each prediction can be straightforwardly explained by the model structure. In the case of the DT, each prediction corresponds to exactly one branch in the tree and can be represented as a chain/set of rules.

Model-agnostic interpretation methods can be distinguished into **global** and **local** methods. They differ in the size of the interpreted space, and their usage depends on the requirements. The local methods are more specific and detailed, while the global methods cover the whole sample space. Next two sections describe global and local methods with reference to the [48].

1.3.1.1 Global methods

Global methods (global surrogate models) approximate the average behavior of the original model. They are particularly useful for understanding the general mechanisms in the data.

Practically, the global surrogate model is a self-explainable model trained to approximate the predictions of a black-box model. The idea is to approximate the black-box prediction function f as closely as possible with the surrogate model prediction function g. For the function g any self-explainable model can be used (DT is one of those).

The algorithm for building a global surrogate model is following:

- Prepare a dataset X that is, ideally, the same dataset used for training the black-box model. Exclude the target variable from the dataset.
- Generate Y - the outputs of the black-box model for dataset X.
- Select the self-explainable model type (DT, linear model, etc.) and its structure.
- Train the selected model on the X and Y. Important step is to use Y as the reference target variable instead of the original values of the target variable in dataset X.
- Measure how well the surrogate model replicates the outputs of the original model. The final model is the approximation of the original one and can be used for its explainability.

1.3.1.2 Local methods

Local methods (local surrogate models) explain individual predictions or the groups (clusters) of the predictions of black-box ML models. One of the concrete implementations is Local Interpretable Model-agnostic Explanations (LIME). Instead of training a global surrogate model, LIME focuses on the smaller data clusters to explain individual predictions.

The practical idea is quite similar to the global methods (see algorithm above - add link), but the data are limited to the specific selection. The dataset X is not the whole train dataset but only the nearest surrounding of selected data samples. Having the instance of the interest and the black box model, LIME generates the model's outputs for the variations of the requested instance - nearest neighboring instances in the sample space. This can also be done by selecting the appropriate cluster of the original train space. The new dataset is created from the neighboring instances and their predictions from the black box model. It is used to train the self-explainable surrogate model, which covers only the selected (local) subspace. The surrogate model in LIME is often weighted by the proximity of the sampled instances to the instance of interest. The learned model is the approximation of the black box model locally on the selected subspace/cluster. Predictions generated by the local surrogate model can be straightforwardly explained through the model's structure. In the case of the DT, each prediction corresponds to the specific branch of the tree and can be represented as a chain/set of rules.

1.3.2 Neural-Backed Decision Trees

Neural-Backed Decision Tree (NBDT) is the approach of interpretable image classification that is based on replacing a neural network's final linear layer with a decision tree [50]. The approach builds a hierarchy from pre-trained Neural Network (NN) weights to reduce overfitting and then uses the hierarchical loss for training that improves high-level decisions.

Neural-Backed Decision Tree (NBDT) is usually characterized by a Convolutional Neural Network (CNN) that is integrated into the decision tree. NBDT is usually used for explainable image classification. For example, NBDT not only can identify a dog in image 1.23, but it can also specify intermediate decision steps. At first, NBDT identifies the image as an "animal". Then, it identifies the image as a "chordate". Next, it identifies the image as a "carnivore". Finally, it identifies the image as a "dog" with a probability of 99.88%. This inference method enhances the interpretability of the model. [51]

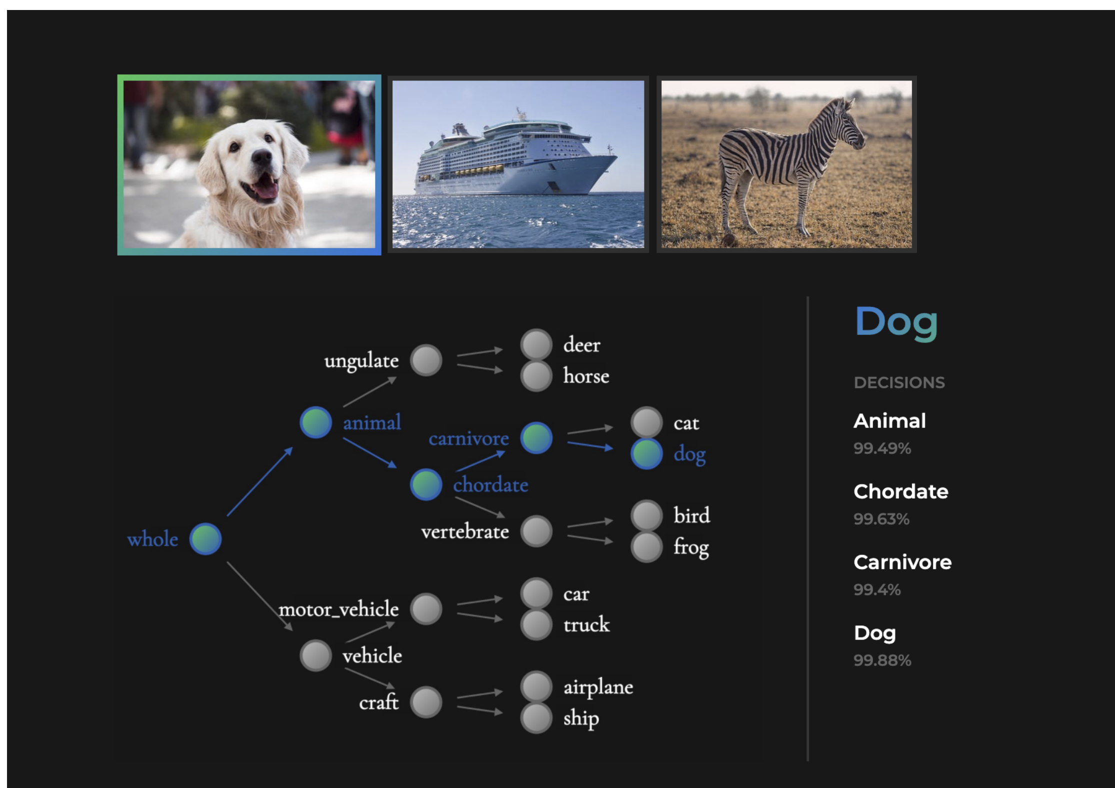


Figure 1.23 Neural-Based Decision Tree example. [52]

NBDT uses a pre-training + fine-tuning framework. The overall training process is roughly divided into the following three steps (see image 1.24):

- **Training step 1:** Pre-train a CNN model.

  Take the weights of its last layer as the hidden vector of each category. The last layer of such a CNN is usually a fully-connected layer.

- **Training step 2:** Form a hierarchical tree structure called the **Induced Hierarchy**.

  Use the hidden vectors of categories for hierarchical clustering and form the Induced Hierarchy. See image 1.25 for a detailed scheme of how the induced hierarchies are created. The induced hierarchies determine which sets of classes the NBDT must decide between.

- **Training step 3:** Fine-tune the network with a custom loss.

  Induced Hierarchy yields a particular loss function, which we call the Tree Supervision Loss. After the induced hierarchies are established, the complete model is no longer a CNN but CNN + DT. To enable the prediction of new samples based on the tree structure, the classification loss of the tree structure has to be added to the total loss, and after that, the model has to be fine-tuned.

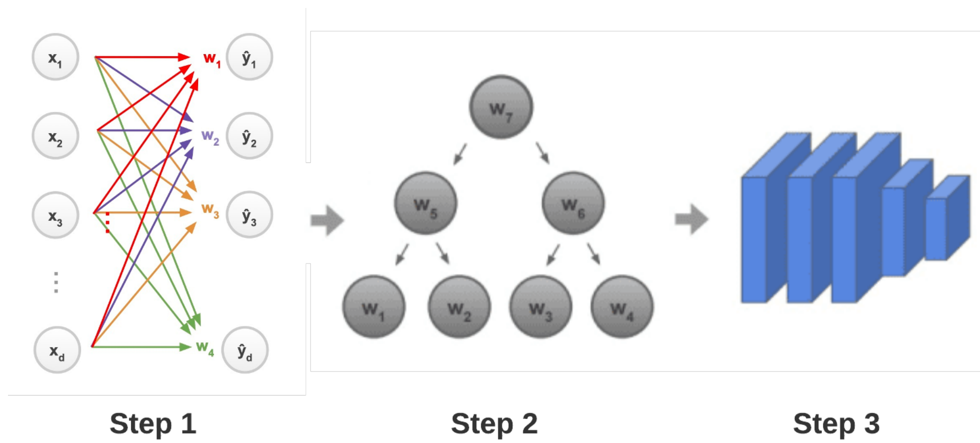


Figure 1.24 The training and fine-tuning process for a Neural-Backed Decision Tree. [50]

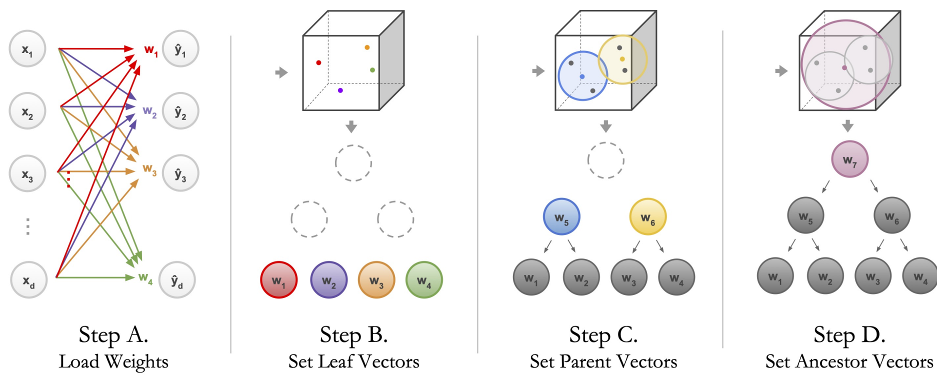


Figure 1.25 Building Induced Hierarchies. [50]

The prediction process of the NBDT can also be broken down into two steps (see image 1.26):

- **Prediction step 1:** Start predicting by passing the sample through the neural network backbone. The backbone is all neural network layers before the final fully-connected layer.
- **Prediction step 2:** Finish predicting by running the final fully-connected layer as a DT - **Embedded Decision Rules**. The final decision is obtained as for classic DT and can be explained by the branch leading rom the root to the leaf.

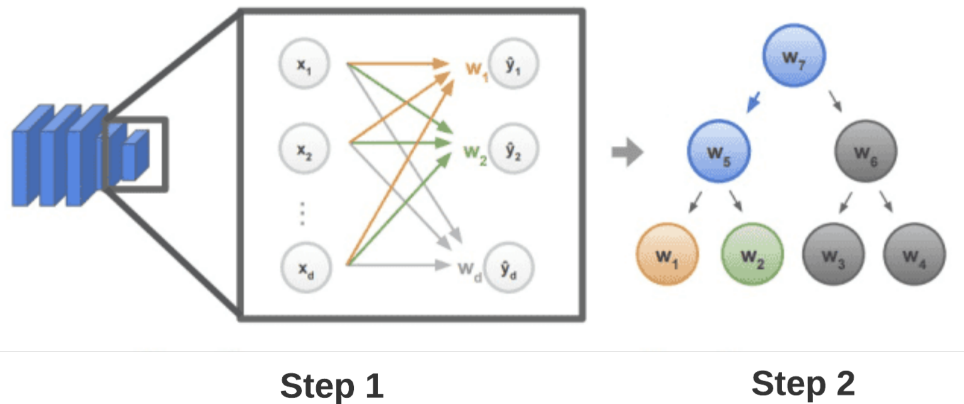


Figure 1.26 The prediction process for a Neural-Backed Decision Tree. [50]

NBDT research [50] aims to propose an interpretable method for classification in Computer Vision. However, the approach can be applied to any other scenario by replacing the CNN with another suitable NN.
For more details on NBDTs see [50].

Chapter 2

Design of Single Decision Tree for H2O-3

This work is done in cooperation with the H2O.ai company that delivers the Open-Source ML platform called H2O-3 (see subsection:prelim-h2o-3 for more details). The practical part of the work is the contribution to this platform - the implementation of the Standard Decision Tree (SDT) algorithm that will become available in the H2O-3 platform in the future release. The algorithm is designed and implemented within this work.

The classical greedy approach (SDT)for DT building was chosen. Several improvements and adaptations were identified during analysis and development.

2.1 Requirements

In this section, I introduce the main requirements of SDT for the H2O-3 Machine Learning Platform and their realization in the designed algorithm. I also point out the implementation specialties that appeared in the process.

- **R1:** Greedy approach
- **R2:** Support of numerical and categorical features
- **R3:** Support of classification and regression
- **R4:** Support of non-trivial splitting conditions in nodes
- **R5:** Allowance of missing values in the data
- **R6:** Providing decision probabilities in leaves
- **R7:** Optimisation for operating with huge datasets
- **R8:** Distributed computations and scalability
- **R9:** Presentability for the purposes of ML explainability

2.1.1 Algorithm (R1 requirement)

The greedy approach (**R1**) utilized in this design is to start in the tree root with the whole dataset and continue to the leaves by splitting the data through the tree branches.
The major specificity of the implementation is an iterative approach instead of a recursive one to avoid excessive stack load. The Breadth-first search (BFS) algorithm is used for building a tree. Only metadata of the split is stored in the queue, as it is unacceptable to store the real dataset in more than one place due to its size. More details about what is stored in the node and queue will be provided later in this section.
The tree structure is selected to be binary - each node is either a leaf or has two children. One of the reasons for the limitation on children count is that it allows optimal storing and operating of the tree.
As described in the SDT section, three decisions should be made to design the SDT:

- selection of a node splitting rule;
- defining termination rules - how to decide if the node is internal or terminal;
- assignment a decision value to each terminal node.

2.1.2 Splitting rules (R2-R5 Requirements)

As there is a requirement to support numerical and categorical features (**R2**) and non-trivial splitting conditions (**R5**), the splitting condition must be configurable depending on the data types or configuration (strategy pattern is used). It can use any number of features at one node as well as any operations with them. The only limitation for the splitting condition is that it has to be binary to ensure building a binary tree. Implemented types of splitting conditions are:

- Threshold for one numerical feature
- Splitting values of one categorical feature into two sets

The suitable splitting rule is selected automatically based on the feature's datatype. Both implemented conditions are based on the same idea - splitting values into two sets - left and right, but from the technical view, they differ in operating with values. For numeric features, the logical way to split values is to select the threshold that splits the sorted data into left and right sets. As the categorical values can not be sorted, there is the need to split all possible values into two sets artificially.

Both defined conditions contain the parameter - threshold and values split - that has to be unambiguous for the condition to be applicable as the splitting rule. To select the best parameter value, several criteria can be used (see section *Splitting Rules* in *Preliminaries*):

- Gini impurity (classification)
- Information gain / Entropy (classification)
- Variance Reduction / Sum of Squared Error (SSE) (?) (regression)
- Chi-square (classification)

For computing the quality of the split with a specific value of the parameters, the criterion for left and right splits is calculated, then weighed with the number of data samples in each split and summed. The final value is the value of the criteria for a split, and the most optimal value (minimum or maximum, depending on the selected criteria) corresponds to the most optimal split.

Another requirement that is very close to the current topic is dealing with missing values (**R7**). Technically, the missing value is a categorical value for both categorical and numerical features, so the natural solution is to consider it a separate value. For categorical variables, the missing value becomes an additional value as any other value. For numerical variables, we need to define the ordering of values - whether the missing values are the least or the greatest values possible. In this work, I decided to consider both options and implemented this selection as the parameter for the splitting rule.

2.1.3 Termination rules

For H2O-3's SDT, the pre-pruning approach was selected. The full tree is not built; instead of it, a set of termination conditions is applied to each node to define if the node is going to be a list:

- Purity - if all the labels in the subset are the same (the node is pure), there is no sense in splitting this node further. Also, the threshold of the impurity can be set by a user to avoid overfitting. For most datasets, if they are not significantly unbalanced, the purity threshold can be set as the ratio of classes in the given node.
- Max depth - the depth of the tree is manually limited by a default value or the value defined by the user. The deeper the tree is, the more complex it is, so the optimal maximum depth depends on the complexity of the problem and data.
- Information gain - likewise the purity condition, the information gain condition uses a measure of impurity to stop the tree expanding. The difference is that, in this case, the improvement of node purity is considered. If the information gain is too low for all possible splits, it is profitable to stop expanding as it no more helps the model.
- Number of samples in leaf - the customization of trivial condition with setting the minimum number of samples in node to allow split. It can also be interpreted as a minimal number of data samples in the child node, so if no split with a suitable count of samples in each child node exists, then the current node becomes a list.

2.1.4 Assigning decision values to the leaves (R3, R6 requirements)

Assigning a decision value for the list is a straightforward operation. For the classification task, the major class is assigned to the leaf. For the regression task, the average of values is assigned to the leaf.

Alongside with the decision value, algorithm has to provide decision probabilities for each leaf (**R6**). For the classification task the probability is calculated as ratio of number of samples belonging to the selected class to the number of all samples in class.
For the regression task the probability depends on the variance of the target variable in the leaf.

2.1.5 Optimisations for huge datasets (R7 requirement)

The main optimization for big data is binning, which is used while splitting the data into nodes. As working directly with the input data is time-consuming, the summarization of data is used. While splitting the node, the optimal threshold or data split have to be found. In the case of categorical features, binning only keeps the summary of data with a specific value of the feature - counts of each class or mean values of the final variable. In the case of the numerical

variable binning, in addition, it works as a discretization of the continuous variable to reduce the count of potential thresholds.
So binning helps reduce the count of candidate splits and keeps a summary of the dataset by specific values/ranges of the features to optimize computations.

2.1.6 Distributed computation (R8 requirement)

To allow scalability, the algorithm should contain parallelization parts that can be executed simultaneously. In the designed algorithm, the parallelizable part works with data as the dataset is usually huge, and any operation with it is very time-consuming.

As mentioned in the Preliminaries, section *H2O-3 Machine Learning Platform*, the H2O-3 utilizes MapReduce tasks to work with data. All operations with data are implemented with MapReduce tasks. In the SDT algorithm, the following tasks are implemented with MapReduce tasks:

- Binning. As described in a previous section, binning reduces the count of potential splits and keeps summary statistics about the subset of data with a particular value/range of features. The MapReduce task calculates statistics for the bins directly from the given data. Calculated statistics are then stored in each bin and are available for further computations. Specific calculated statistics are counts of data samples in a bin with a specific value of the target variable. This task has to be executed for each bin in each node independently, as all nodes correspond to different data subsets, so their statistics are not related.
- Discovering the data. For computing the minimum and maximum values of each feature, the MapReduce task is used. This step is done in the beginning to discover the data. While creating the inner nodes, a similar task is executed for updating the actual limits of the data, as the actual data limits can be more strict than the limits defined by the rules in the nodes. This step is an optimization of binning as it reduces potential empty bins around the data limits.
- Setting decision value? Is it needed? We already calculate the counts of classes while splitting.

2.1.7 Presentability (R9 requirement)

One of the use cases for designed SDT is usage for ML explainability either as a self-explainable model or a surrogate model (see section ML Explainability). For this use case, the resulting tree has to be presentable. Two approaches were designed to fulfill this requirement:

- Graphical tree structure scheme. The tree is represented by a scheme, remaining the original structure. This graphical way of presenting helps to easier orient in branches and see the hierarchy of splitting rules.
- Set of extracted rules. The tree is decomposed into a set of independent rules, each representing a branch leading from the root to the leaf. The number of rules is the same as the number of leaves, as each branch corresponds to exactly one leaf.

2.2 Implementation notes

SingleDecisionTree is implemented in Java. Python and R interfaces are also available. The algorithm is part of the H2O-3 infrastructure that unifies its usage. In this section, I am demonstrating the usage of the algorithm from the Python interface.

The H2O-3 Cluster needs to be initialized at the beginning. The Python client then connects to the running cluster and executes user code. The cluster can be initialized automatically with the default configuration if the *h2o.connect()* method is called.

```
h2o.init(nthreads=nthreads)
h2o.connect()
```

After connection, the models can be initialized. Also, the data loading step can be performed here. Data have to be loaded to the H2O Frame.

```
data_train_frame = h2o.H2OFrame('data_train.csv')
SDT_h2o = H2OSingleDecisionTreeEstimator(
            model_id="single_decision_tree.hex", max_depth=5)
```

The training step requires training data and performs training in-place.

```
SDT_h2o.train(training_frame=data_train_frame, y='label')
```

Prediction is made from the testing data that also need to be H2O Frame. The result of the prediction step is also an H2O Frame. The first column of the Frame is the prediction itself, and the second column is the probability of the predicted value.

```
predictions_sdt = SDT_h2o.predict(h2o.H2OFrame(data_predict))
```

After all manipulations, the cluster needs to be correctly shouted down:

```
h2o.cluster().shutdown()
```

Chapter 3

Evaluation of DT implementation

In this chapter I evaluate the results of implemented SDT from several perspectives. I explore its scalability for different problem sizes and different computational resources to prove that the distributivity brings a big improvement and that the SDT uses all the given computational resources effectively.

3.1 Datasets

For the evaluation of the solution, I use the following datasets:

- ***prostate*** [53] - real dataset for binary classification. The dataset was originally preprocessed and subsampled by H2O.ai for testing purposes.

  Available features (in preprocessed dataset): 8 including binary target variable (7 features used for training).

  Available rows (in preprocessed dataset): 382 (306 used for training, 76 used for predictions/testing). The dataset is randomly split into training and testing datasets.

  In preprocessed phase all features are numeric and the target variable (*CAPSULE*) is binary (0/1).

  Preview of the dataset:

CAPSULE	AGE	RACE	DPROS	DCAPS	PSA	VOL	GLEASON
0	1.09..	-0.30..	-1.12..	-0.38..	-0.21..	2.18..	0.77..
1	-0.99..	-0.30..	2.03..	-0.38..	0.48..	-0.85..	0.77..
0	-0.09..	-0.30..	-0.07..	-0.38..	-0.12..	0.10..	0.77..

  The dataset is imbalanced. In the training set: 133 samples of class 0 and 32 samples of class 1.

- ***credit card*** [54] - real dataset for binary classification. The dataset was originally preprocessed and subsampled by H2O.ai for testing purposes.

  Available features (in preprocessed dataset): 9 including binary target variable (8 features used for training).

  Available rows (in preprocessed dataset): 27114 (24422 used for training, 2692 used for predictions/testing). The dataset is randomly split into training and testing datasets.

In preprocessed phase all features are numeric and the target variable (*IsDepDelayed*) is binary (0/1).

Preview of the dataset:

IsDepDelayed	Year	Month	Day	DayOfWeek	Carrier	Origin	Dest	Distance
0	1991	1	26	6	3	50	70	402
0	1998	1	1	4	4	85	13	868
1	2000	1	28	5	8	88	13	872

The dataset is less imbalanced then the previous one. In the training set: 7772 samples of class 0, 11495 samples of class 1.

3.2 Comparing the results on data

In this section I evaluate the implemented *SingleDecisionTree* algorithm with common evaluation metrics for classification - **accuracy** and **f1-score**. I also compare the SDT with the benchmark solutions - scikit-learn Decision Tree and H2O's Distributed Random Forest with one tree.

All three algorithms are trained and evaluated on the same machine, same training datasets (see previous section) and with the same settings:

- Max depth - multiple values are used for the comparing models' performance but these values are changing simultaneously for all three models.
- Tree structure - all models use only binary splitting rules with a threshold.

The evaluation of all three models is shown in the images 3.1 and 3.2.

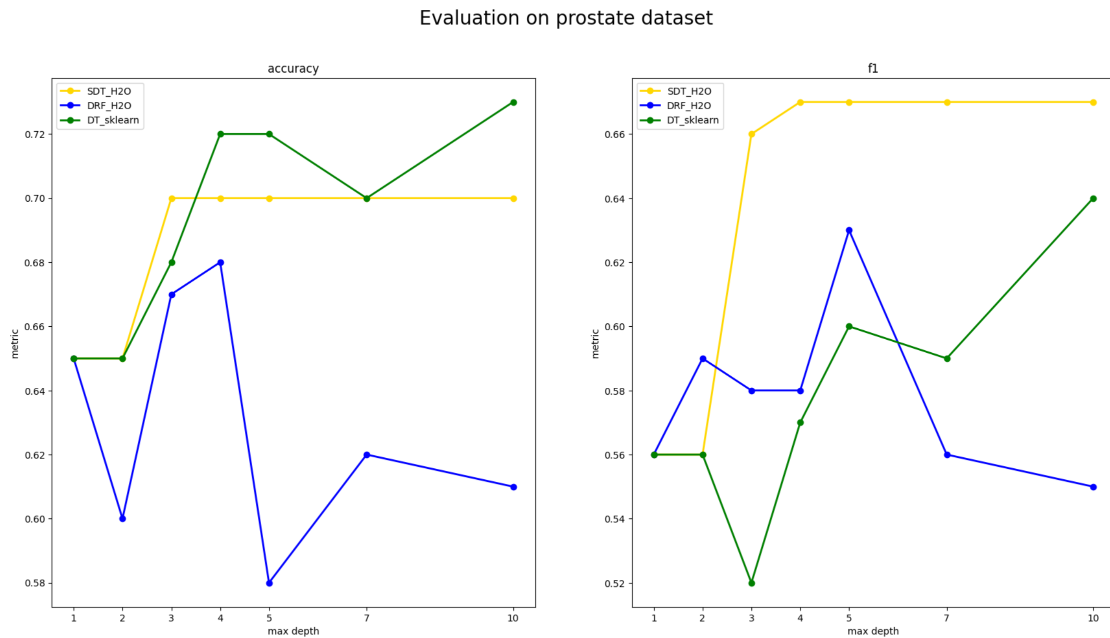


Figure 3.1 Evaluation of models on the *prostate* dataset.

For the *prostate* dataset, I choose the following values of the *max_depth* parameter: $[1, 2, 3, 4, 5, 7, 10]$. The dataset is small, so the tree should not be too deep as there can be insufficient data for training.

Diagram 3.1 shows that my implementation of *SingleDecisionTree* (labeled *SDT_H2O*) is competitive by both metrics. By the *accuracy* metric, my solution slightly outperforms H2O-3's benchmark solution *DistributedRandomForest* with one tree (labeled *DRF_H2O*) and slightly underperforms *sklearn* benchmark *DecisionTreeClassifier* (labeled *DT_sklearn*). By the *f1-score* metric, my solution outperforms both benchmark models.

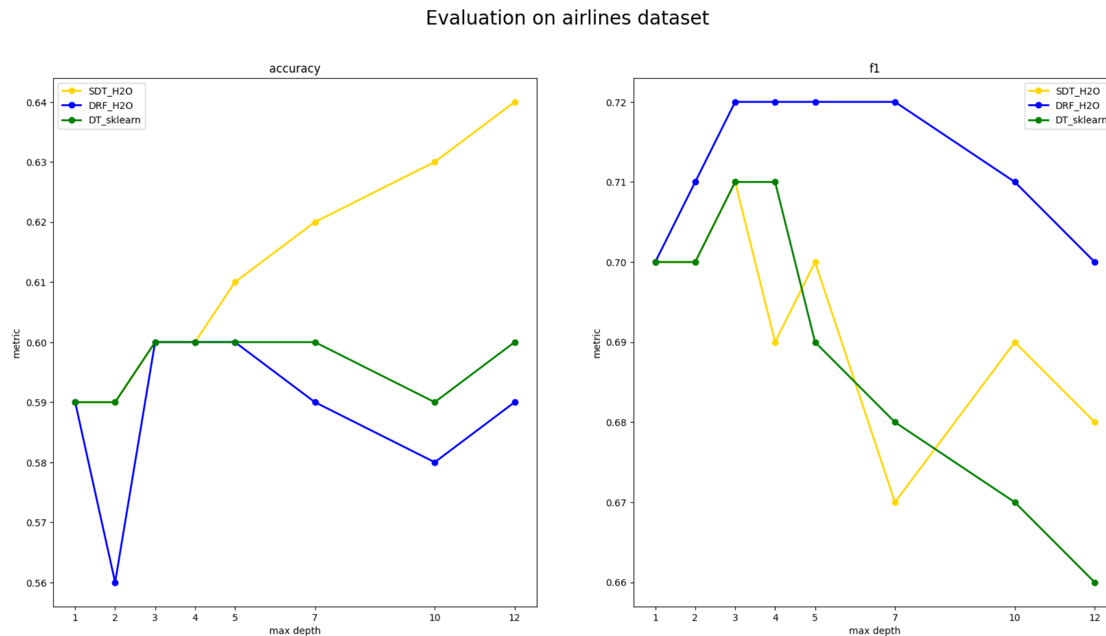


Figure 3.2 Evaluation of models on the *airlines* dataset.

For the *airlines* dataset, I choose the following values of the *max_depth* parameter: $[1, 2, 3, 4, 5, 7, 10, 12]$ as this dataset is bigger than the previous one, and deeper tree could show better performance.

Diagram 3.2 shows that the *SingleDecisionTree* (labeled *SDT_H2O*) has higher *accuracy* on the deeper tree but at the same time its *f1-score* is lower for the deeper tree. By *accuracy* metric, my solution outperforms both benchmark solutions on the high depth and has the same performance on the small depth. The trend of *F1-score* metric is the same for all three models. *SingleDecisionTree* does not significantly outperform any model by *f1-score* metric, but it corresponds to the trend and shows adequate results.

3.3 Scalability

As H2O-3 is a distributed platform, I am evaluating algorithm's scalability on multiple threads. Similarly, as in the previous section, I compare the implemented solution with the H2O-3's DRF with one tree. I do not compare my algorithm with *sklearn* solution as their infrastructures are not comparable.

Images 3.3 and 3.4 show the training (left) and prediction (right) times of both models on a different number of threads. I evaluated both models on both datasets on all numbers of threads from 1 to 12 (inclusive). The *max_depth* parameter was fixed for all runs with value 5 as the optimal value based on the evaluation section (see images 3.1 and 3.2). For each number of threads the training and prediction were executed multiple times (5 times each step) to ensure the stability of the measurement by averaging multiple results.

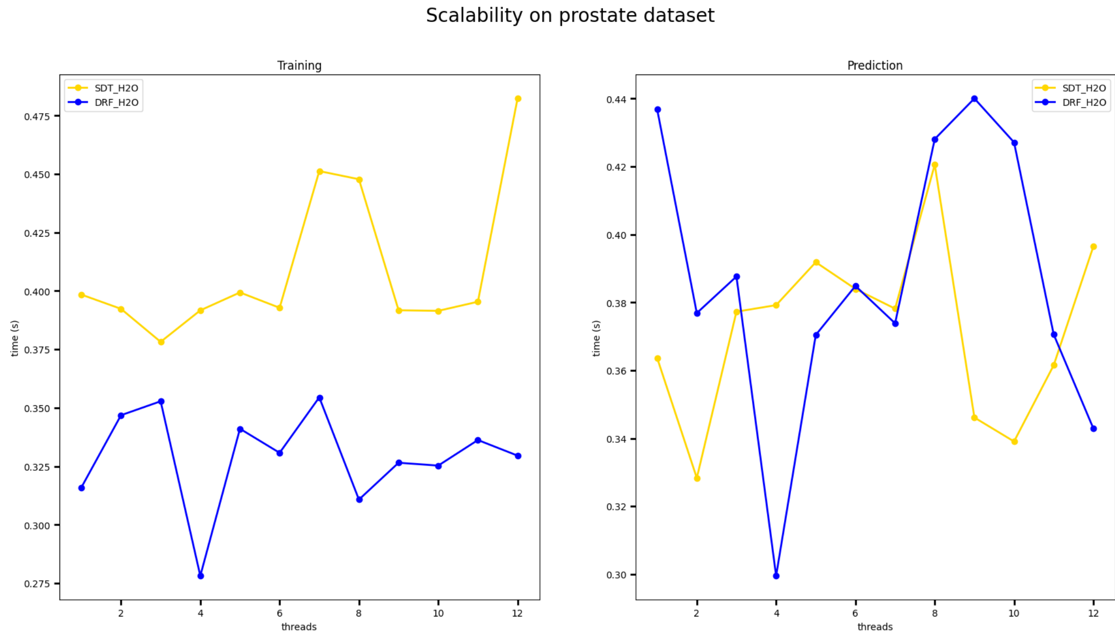


Figure 3.3 Training and prediction time for a *prostate* dataset.

On *prostate* dataset *SingleDecisionTree's* (labeled *SDT_H2O*) training and prediction processes are slightly slower then *DestributedRandomForest's* labeled *DRF_H2O*). The *DRF* is highly optimized to being trained fast and *SDT* can be optimized in the same way in the future (see the Drawbacks and future improvements section below).

The prediction step of *SingleDecisionTree*, on average, is comparably fast as the *DestributedRandomForest*.

The *airlines* dataset is bigger then the *prostate*, so the difference in time performance is more expressed. As the *SingleDecisionTree* is not optimized in a way *DestributedRandomForest* is, its training time is significantly higher, but still acceptable for the first version of the implementation.

The prediction step of the *SingleDecisionTree* is slightly faster than the *DestributedRandomForest*.

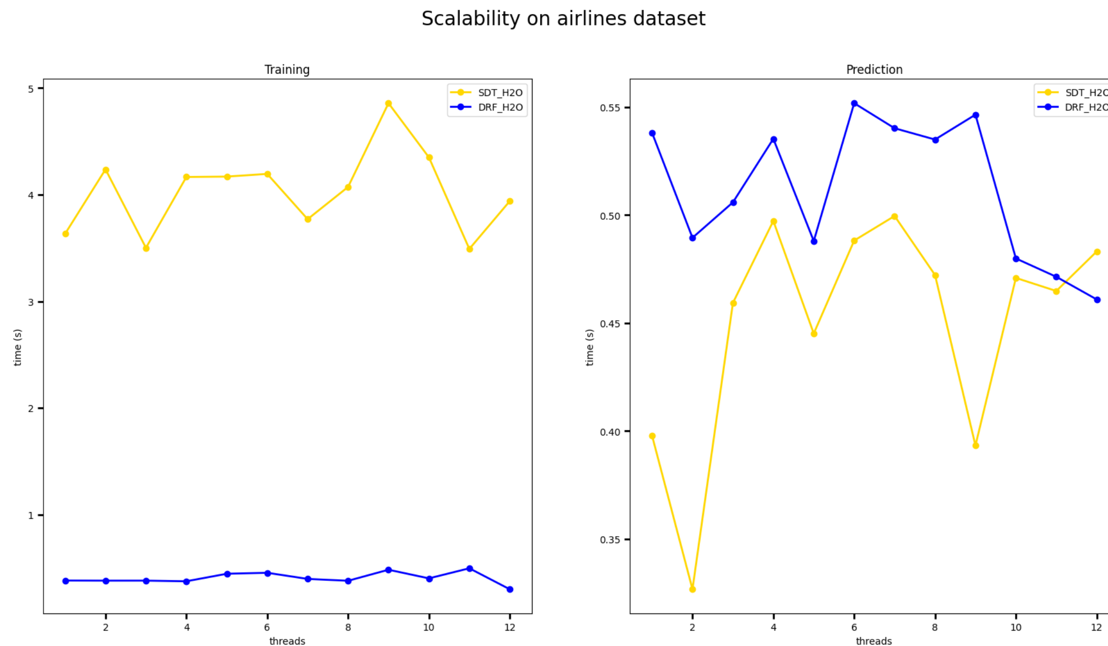


Figure 3.4 Training and prediction time for *airlines* dataset.

3.4 Explainability

As the *SingleDecisionTree* should be usable for the explanability of ML models, the presentability requirement is implemented (see section Requirements). In the first version of *SingleDecisionTree* only text representation of the tree structure is available in form of exported set of rules. See example of an exported rules for the airlines dataset:

```
(Carrier > 6.3) and (Distance > 919.0) and (Year > 1997.2) and
(Year > 1998.2) and (Origin <= 80.0) and (Origin > 59.0) and
(DayOfWeek <= 6.4) and (DayOfWeek > 3.0) and (DayOfWeek > 7.0) and
(DayOfWeek > 5.0) -> (IsDepDelayed: 1.0, Probability: 0.83..)


(Carrier > 6.3) and (Distance > 919.0) and (Year > 1997.2) and
(Year > 1998.2) and (Origin <= 80.0) and (Origin > 59.0) and
(DayOfWeek > 6.4) -> (IsDepDelayed: 1.0, Probability: 0.94..)


(Carrier > 6.3) and (Distance > 919.0) and (Year > 1997.2) and
(Year > 1998.2) and (Origin > 80.0) ->
(IsDepDelayed: 0.0, Probability: 0.82..)
```

3.5 Drawbacks and future improvements

The implemented *SingleDecisionTree* is planned to be part of the future release of H2O-3 Machine Learning Platform. Further improvements need to be implemented for the algorithm to be ready for production usage by customers:

- Support of categorical features - the splitting rules for categorical features need to be implemented.
- Support of missing values (NaN) values.
- Implement other types of histograms (now only Equal-Width binning is implemented).
- Support multinomial classification and regression - splitting rules and assigning values to the leaves need to be implemented.
- Support grid search.
- Enable parallel run of splitting rules.
- Support cross-validation.
- Support MOJO.
- Support ML Explainability functions: Shapley values, PDP plot etc.
- Add Python and R examples to the documentation.

Conclusion

This work aimed to research the approach of Decision Tree in Machine LearningIn particular - two main concepts of the Decision Tree - its building approaches and its usage for Machine Learning Explainability. The practical task of this work was to analyze, design and implement *SingleDecisionTree* into H2O-3 Machine Learning Platform.

In the first chapter, I researched the existing approaches for building decision trees - from the Standard Decision Tree with a greedy approach, through Oblique, Evolutionary, SAT-based decision trees to the Incremental approach. Main concepts and practical implementation notes were presented in the chapter. Later in the same chapter, I introduced the usage of the Decision Tree in Machine Learning explainability as a typical self-explainable model. I presented approaches such as Surrogate Models (local and global) and Neural-Backed Decision Trees.

In the next chapter, I designed the *SingleDecisionTree* for the H2O-3 Machine Learning Platform. I analyzed the requirements on the algorithm and the specifics of the platform that needed to be satisfied and designed the algorithm that is based on the greedy Standard Decision Tree approach. The algorithm is designed in such a way that allows extensibility and custom implementation of separate steps. The designed algorithm can be used as a stand-alone model as well as the surrogate model for ML explainability.

After implementing the designed algorithm, I evaluated it in the last chapter. The accuracy and f1-score metrics were used to evaluate the performance of the models on real datasets and to compare it with the H2O-3's *DistributedRandomForest* with one tree and sklearn's *DecisionTreeClassifier*. The training and prediction times were also measured and compared to the benchmark *DistributedRandomForest* with one tree. I also presented future steps that were left out of the scope of this work.

Bibliography

1. SAMUEL, A. L. Some studies in machine learning using the game of checkers. *IBM Journal of Research and Development*. 2000, vol. 44, no. 1.2, pp. 206–226. Available from DOI: 10.1147/rd.441.0206.
2. BURNS, E. *Definition. Machine learning* [online]. techtarget, Mar 2021. Available also from: https://www.techtarget.com/searchenterpriseai/definition/machine-learning-ML.
3. THAKAR, P. The math behind Machine Learning Algorithms. *Medium*. Jul 2020. Available also from: https://medium.com/towards-data-science/the-math-behind-machine-learning-algorithms-9c5e4c87fff.
4. JHA, Vishakha. *Machine Learning Algorithm - Backbone of emerging technologies*. techleer, [n.d.]. Available also from: https://www.techleer.com/articles/203-machine-learning-algorithm-backbone-of-emerging-technologies/.
5. SONG, Yan-Yan; LU, Ying. Decision tree methods: applications for classification and prediction. *Shanghai archives of psychiatry*. 2015, vol. 27, pp. 130–5. Available from DOI: 10.11919/j.issn.1002-0829.215044.
6. ALTO, Valentina. *Splitting Criteria for Decision Tree Algorithm — Part 1*. medium, [n.d.]. Available also from: https://medium.com/analytics-vidhya/splitting-criteria-for-decision-tree-algorithm-part-1-64e949aa2465.
7. OGOLA, Willies. Entropy and Information Gain to Build Decision Trees in Machine Learning. *section*. July 3, 2021. Available also from: https://www.section.io/engineering-education/entropy-information-gain-machine-learning/.
8. SHARMA, Abhishek. 4 Simple Ways to Split a Decision Tree in Machine Learning. *analyticsvidhya*. June 30, 2020. Available also from: https://www.analyticsvidhya.com/blog/2020/06/4-ways-split-decision-tree/.
9. HUI LIN, Ming Li. *Introduction to Data Science*. Independently published (October 21, 2022), 2022. Available also from: https://github.com/happyrabbit/IntroDataScience.
10. POOJATAMBE. Decision Tree Splitting: Entropy vs. Misclassification Error. *Medium*. Oct 26, 2022. Available also from: https://pub.towardsai.net/decision-tree-splitting-entropy-vs-misclassification-error-27fdf2f5e3bf.
11. *H2O. The 1 open-source machine learning platform for the enterprise*. H2O.ai, [n.d.]. Available also from: https://h2o.ai/platform/ai-cloud/make/h2o/.
12. *What is a key-value store?* [Online]. hazelcast, [n.d.]. Available also from: https://hazelcast.com/glossary/key-value-store/.

13. *MapReduce Tutorial* [online]. apache, [n.d.]. Available also from: `https://hadoop.apache.org/docs/stable/hadoop-mapreduce-client/hadoop-mapreduce-client-core/MapReduceTutorial.html`.
14. WICKHAM, Hadley. The Split-Apply-Combine Strategy for Data Analysis. *Journal of Statistical Software*. 2011, vol. 40, no. 1, pp. 1–29. Available from DOI: `10.18637/jss.v040.i01`.
15. PARK, Dongchul; WANG, Jianguo; KEE, Yang-Suk. In-Storage Computing for Hadoop MapReduce Framework: Challenges and Possibilities. *IEEE Transactions on Computers*. [N.d.].
16. SAFAVIAN, S.R.; LANDGREBE, D. A survey of decision tree classifier methodology. *IEEE Transactions on Systems, Man, and Cybernetics*. 1991, vol. 21, no. 3, pp. 660–674. Available from DOI: `10.1109/21.97458`.
17. HYAFIL, Laurent; RIVEST, Ronald L. Constructing optimal binary decision trees is NP-complete. *Information Processing Letters*. 1976, vol. 5, no. 1, pp. 15–17. Available from DOI: `10.1016/0020-0190(76)90095-8`.
18. FISHER, R. A. THE USE OF MULTIPLE MEASUREMENTS IN TAXONOMIC PROBLEMS. *Annals of Eugenics*. 1936, vol. 7, no. 2, pp. 179–188. Available from DOI: `https://doi.org/10.1111/j.1469-1809.1936.tb02137.x`.
19. BREIMAN, Leo; FRIEDMAN, Jerome; STONE, Charles J.; OLSHEN, R. A. *Classification and Regression Trees*. Taylor & Francis, 1984. ISBN 978-0-41204841-8.
20. QUINLAN, J. R. Induction of Decision Trees. *Mach. Learn.* 1986, vol. 1, no. 1, pp. 81–106. ISSN 0885-6125. Available from DOI: `10.1023/A:1022643204877`.
21. QUINLAN, J. Ross. *C4.5: Programs for Machine Learning*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 1993. ISBN 1558602380.
22. KASS, G. V. An Exploratory Technique for Investigating Large Quantities of Categorical Data. *Journal of the Royal Statistical Society: Series C (Applied Statistics)*. 1980, vol. 29, no. 2, pp. 119–127. Available from DOI: `https://doi.org/10.2307/2986296`.
23. MINGERS, John. An Empirical Comparison of Pruning Methods for Decision Tree Induction. *Machine Learning*. 1989, vol. 4, pp. 227–243.
24. PRODROMIDIS, Andreas L.; STOLFO, Salvatore J. Minimal Cost Complexity Pruning of Meta-Classifiers. In: *Proceedings of the Sixteenth National Conference on Artificial Intelligence and the Eleventh Innovative Applications of Artificial Intelligence Conference Innovative Applications of Artificial Intelligence*. USA: American Association for Artificial Intelligence, 1999, p. 979. AAAI '99/IAAI '99. ISBN 0262511061.
25. MINGERS, John. Expert Systems-Rule Induction with Statistical Data. *The Journal of the Operational Research Society* [online]. 1987, vol. 38, no. 1, pp. 39–47 [visited on 2022-11-04]. ISSN 01605682, ISSN 14769360. Available from: `http://www.jstor.org/stable/2582520`.
26. CHATURVEDI, Dr. SETU; PATIL, Sonal. Oblique Decision Tree Learning Approaches - A Critical Review. *International Journal of Computer Applications*. 2013, vol. 82, pp. 6–10. Available from DOI: `10.5120/14174-2023`.
27. CHA, Sung-Hyuk; TAPPERT, Charles. Constructing Binary Decision Trees using Genetic Algorithms. In: 2008, pp. 49–54.
28. BOURHNANE, Safae; ABID, Mohamed; LGHOUL, Rachid; ZINE-DINE, Khalid; EL KAMOUN, Najib; BENHADDOU, D. Machine learning for energy consumption prediction and scheduling in smart buildings. *SN Applied Sciences*. [N.d.].

29. FAIK, Lina. *Evolutionary Decision Trees: When Machine Learning draws its Inspiration from Biology*. medium, [n.d.]. Available also from: `https://towardsdatascience.com/evolutionary-decision-trees-when-machine-learning-draws-its-inspiration-from-biology-7d427fa7554b`.

30. SHATI, Pouya; COHEN, Eldan; MCILRAITH, Sheila. SAT-Based Approach for Learning Optimal Decision Trees with Non-Binary Features. In: MICHEL, Laurent D. (ed.). *27th International Conference on Principles and Practice of Constraint Programming (CP 2021)*. Dagstuhl, Germany: Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2021, vol. 210, 50:1–50:16. Leibniz International Proceedings in Informatics (LIPIcs). ISBN 978-3-95977-211-2. ISSN 1868-8969. Available from DOI: `10.4230/LIPIcs.CP.2021.50`.

31. SHATI, Pouya; COHEN, Eldan; MCILRAITH, Sheila. SAT-Based Approach for Learning Optimal Decision Trees with Non-Binary Features. In: MICHEL, Laurent D. (ed.). *27th International Conference on Principles and Practice of Constraint Programming (CP 2021)*. Dagstuhl, Germany: Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2021, vol. 210, 50:1–50:16. Leibniz International Proceedings in Informatics (LIPIcs). ISBN 978-3-95977-211-2. ISSN 1868-8969. Available from DOI: `10.4230/LIPIcs.CP.2021.50`.

32. COOK, Stephen A. The Complexity of Theorem-Proving Procedures. In: *Proceedings of the Third Annual ACM Symposium on Theory of Computing*. Shaker Heights, Ohio, USA: Association for Computing Machinery, 1971, pp. 151–158. STOC '71. ISBN 9781450374644. Available from DOI: `10.1145/800157.805047`.

33. LEVIN, Leonid A. Universal Sequential Search Problems. *Problems of Information Transmission*. 1973, vol. 9, no. 3.

34. DAVIS, Martin; LOGEMANN, George; LOVELAND, Donald. A Machine Program for Theorem-Proving. *Commun. ACM*. 1962, vol. 5, no. 7, pp. 394–397. ISSN 0001-0782. Available from DOI: `10.1145/368273.368557`.

35. AVELLANEDA, Florent. Efficient Inference of Optimal Decision Trees. In: 2020.

36. NARODYTSKA, Nina; IGNATIEV, Alexey; PEREIRA, Filipe; MARQUES-SILVA, Joao. Learning Optimal Decision Trees with SAT. In: *Proceedings of the Twenty-Seventh International Joint Conference on Artificial Intelligence, IJCAI-18*. International Joint Conferences on Artificial Intelligence Organization, 2018, pp. 1362–1368. Available from DOI: `10.24963/ijcai.2018/189`.

37. SCHIDLER, Andre; SZEIDER, Stefan. SAT-based Decision Tree Learning for Large Data Sets. *Proceedings of the AAAI Conference on Artificial Intelligence*. 2021, vol. 35, no. 5, pp. 3904–3912. Available from DOI: `10.1609/aaai.v35i5.16509`.

38. JANOTA, Mikoláš; MORGADO, António. SAT-Based Encodings for Optimal Decision Trees with Explicit Paths. In: Berlin, Heidelberg: Springer-Verlag, 2020. ISBN 978-3-030-51824-0. Available from DOI: `10.1007/978-3-030-51825-7_35`.

39. UTGOFF, Paul E. Incremental Induction of Decision Trees. *Mach. Learn.* 1989, vol. 4, no. 2, pp. 161–186. ISSN 0885-6125. Available from DOI: `10.1023/A:1022699900025`.

40. YANG, Qing; GU, Yudi; WU, Dongsheng. Survey of incremental learning. *2019 Chinese Control And Decision Conference (CCDC)*. [N.d.].

41. SCHLIMMER, Jeffrey; FISHER, Doug. A Case Study of Incremental Concept Induction. In: 1986, pp. 496–501.

42. UTGOFF, Paul E. Incremental Induction of Decision Trees. *Mach. Learn.* 1989, vol. 4, no. 2, pp. 161–186. ISSN 0885-6125. Available from DOI: `10.1023/A:1022699900025`.

43. UTGOFF, PAUL E. ID5: An Incremental ID3. In: LAIRD, John (ed.). *Machine Learning Proceedings 1988*. San Francisco (CA): Morgan Kaufmann, 1988, pp. 107–120. ISBN 978-0-934613-64-4. Available from DOI: `https://doi.org/10.1016/B978-0-934613-64-4.50017-7`.

44. *Machine Learning and Knowledge Extraction*. Springer-Verlag, 2018. ISBN 978-3-319-99739-1.

45. VIEIRA, Carla; DIGIAMPIETRI, Luciano. A study about Explainable Articial Intelligence: using decision tree to explain SVM. [N.d.].

46. KENTON, W. *What Is a Black Box Model? Definition, Uses, and Examples* [online]. investopedia, Mar 2022. Available also from: https://www.investopedia.com/terms/b/blackbox.asp.

47. Opening the Black Box: The Promise and Limitations of Explainable Machine Learning in Cardiology. *Canadian Journal of Cardiology*. 2022, vol. 38, no. 2, pp. 204–213. ISSN 0828-282X. Available from DOI: https://doi.org/10.1016/j.cjca.2021.09.004.

48. MOLNAR, Christoph. *Interpretable Machine Learning. A Guide for Making Black Box Models Explainable.* Independently published (February 28, 2022), 2022. ISBN 979-8411463330. Available also from: https://christophm.github.io/interpretable-ml-book/.

49. YANG, Guang; YE, Qinghao; XIA, Jun. Unbox the black-box for the medical explainable AI via multi-modal and multi-centre data fusion: A mini-review, two showcases and beyond. *Information Fusion*. [N.d.].

50. WAN, Alvin; DUNLAP, Lisa; HO, Daniel; YIN, Jihan; LEE, Scott; JIN, Henry; PETRYK, Suzanne; BARGAL, Sarah Adel; GONZALEZ, Joseph E. *NBDT: Neural-Backed Decision Trees*. arXiv, 2020. Available from DOI: 10.48550/ARXIV.2004.00221.

51. *Research on Neural-Backed Decision Trees Algorithms*. Alibaba Cloud, [n.d.]. Available also from: https://www.alibabacloud.com/blog/research-on-neural-backed-decision-trees-algorithms_597094.

52. WAN, Alvin. *Making Decision Trees Accurate Again: Explaining What Explainable AI Did Not*. bair.berkeley, [n.d.]. Available also from: https://bair.berkeley.edu/blog/2020/04/23/decisions/.

53. *Prostate Datasets*. National Cancer Institute, [n.d.]. Available also from: https://cdas.cancer.gov/datasets/plco/20/.

54. *Data Expo 2009 - Airline on-time performance*. ASA Section on Statistical Computing ASA Section on Statistical Graphics, [n.d.]. Available also from: https://community.amstat.org/jointscsg-section/dataexpo/dataexpo2009.

Contents of enclosed CD

```
readme.txt ............................ Description of medium content
src
├── impl .............................. Source code of the implementation
└── thesis ............................ Thesis text in LaTeX format
text
└── thesis.pdf ........................ Thesis text in PDF format
```