Universitat Oberta de Catalunya (UOC)

Master in Data Science (*Data Science*)

MASTER THESIS

Area: 4

# Optimal decision trees using optimization techniques

Author: Josep Alòs Pascual

Course instructor (UOC): Raúl Parada Medina

Area Responsible Professor (UOC): Jordi Casas Roma

External supervisor (UdL): Carlos Ansotegui

Balaguer, June 16, 2020

# Credits/Copyright

# FITXA DEL TREBALL FINAL

| | |
|---|---|
| Títol del treball: | Optimal decision trees using optimization techniques |
| Nom de l'autor: | Josep Alòs Pascual |
| Nom del colaborador/a docent: | Raul Parada Medina |
| Nom del PRA: | Jordi Casas Roma |
| Data d'entrega (mm/aaaa): | 06/2020 |
| Titulació o programa: | Màster de Ciència de Dades |
| Àrea del Treball Final: | Àrea 4 |
| Idioma del treball: | Anglès |
| Paraules clau | arbres de decisió, optimització |

# Acknowledgements

I would like to thank all the people that have helped me during this project. First of all, I want to thank both my course instructor at Universitat Oberta de Catalunya and my external supervisor at the Universitat de Lleida for their guiding during this project, their constant support and the time they spent resolving my doubts. Secondly, I want to thank my family that did not let me down even during the turbulent times we have experienced during the quarantine, and to my friends for the same exact reason, being as close as possible during those months.

# Abstract

The rising need of having a way to understand and explain the decisions produced by the artificial intelligence algorithms, used in a broad set of fields, led to the apparition of the concept of explainable artificial intelligence. One of the most simple, although powerful, algorithms are the decision trees. This project focuses on studying the algorithms that allow the creation of such trees, while ensuring that the tree is optimal, as smaller trees are usually easier to explain. The project presents a Python package whose purpose is to act as a barrier remover for the users that don't have the means to implement those algorithms, allowing them to use the implementations proposed by different authors while leveraging the implementation of both the algorithms and the interaction with the solving tools to the package.

In this report, the design of such tool is presented, as well as the technical considerations on which solving tools are used. Also, benchmarking on different datasets used in the bibliography is done to assess that the package accomplishes its main task, and to compare the different approaches implemented.

***Keywords***— optimal decision trees, optimization

# Contents

**Abstract** **vii**

**Table of contents** **ix**

**List of figures** **xi**

**List of tables** **xiii**

**List of code listings** **1**

**1 Introduction** **3**

1.1 Description of the project . . . . . . . . . . 4
1.2 Personal motivation . . . . . . . . . . 5
1.3 Objectives . . . . . . . . . . 6
1.4 Methodology of the project development description . . . . . . . . . . 6
1.5 Planning of the project . . . . . . . . . . 7
1.6 Document structure . . . . . . . . . . 11

**2 State of the art** **13**

2.1 Decision trees algorithms . . . . . . . . . . 13
2.1.1 Heuristic greedy algorithms . . . . . . . . . . 14
2.1.2 Mathematical optimization algorithms . . . . . . . . . . 15
2.2 Tools used . . . . . . . . . . 17
2.2.1 Python . . . . . . . . . . 17
2.2.2 Minizinc and Flatzinc . . . . . . . . . . 17
2.2.3 Mathematical solvers . . . . . . . . . . 18

**3 Library design** **21**

3.1 Library overview and design . . . . . . . . . . 21
3.2 Interaction with external components . . . . . . . . . . 22

**4 Library implementation** **25**
4.1 Overview . . . 25
4.2 Implementation . . . 26
4.2.1 CNF module . . . 27
4.2.2 Models module . . . 27
4.2.3 SAT models . . . 30
4.2.4 Cross validate . . . 33
4.2.5 Trees . . . 34
4.2.6 Data parsing . . . 39
4.3 Dependencies . . . 40
4.4 Installation and usage . . . 41

**5 Experimental results** **43**
5.1 Results discussion . . . 46

**6 Conclusions and future work** **49**
6.1 Conclusions . . . 49
6.2 Future work . . . 50

**Glossary** **51**

**Acronyms** **53**

**Appendices** **55**

**A API reference** **57**
A.1 decisiontrees.cnf.CNF . . . 57
A.2 decisiontrees.cnf_components . . . 60
A.3 decisiontrees.cnf.utils . . . 65
A.4 decisiontrees.data_parsing . . . 66
A.5 decisiontrees.trees . . . 67
A.6 decisiontrees.cross_validate . . . 70
A.7 decisiontrees.models . . . 72

**B Encodings implementations** **77**
B.1 SAT APPROACH . . . 77
B.2 MINIZINC APPROACH . . . 81
B.3 InferDT . . . 84
B.3.1 Inferring tree with a maximum depth . . . 84
B.3.2 Inferring a tree with a maximum number of nodes . . . 87

# List of Figures

1 Example of a decision tree . . . . . . . . . . 4
2 Project's planning (part 1/3) . . . . . . . . . . 8
2 Project's planning (part 2/3) . . . . . . . . . . 9
2 Project's planning (part 3/3) . . . . . . . . . . 10
3 Layout of the package interacting with the external components . . . . . . . . . . 24
4 Package structure . . . . . . . . . . 26
5 Models hierarchy . . . . . . . . . . 28
6 Solve method for MinizincModel . . . . . . . . . . 29

# List of Tables

1 Planned tasks and its timing . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 7
2 Correspondence of each model with the factory method used . . . . . . . . . . . . . . 37
3 Dataset list . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 44
4 Results for the small datasets . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 44
5 Results for the reduced datasets . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 45
6 Decision trees found by **Scikit-Learn** . . . . . . . . . . . . . . . . . . . . . . . . . . 46

# List of code Listings

1 Example of the encoding of a Knapsack problem in Minizinc . . . . . . . . . . . . . . . . 18
2 SklearnModel.find_tree method . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 30
3 CrossValidationResult class . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 33
4 cross_validate method . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 34
5 Optimal tree for the “irish” dataset (see table 3) . . . . . . . . . . . . . . . . . . . . . . . 35
6 EncoderPipeline class. The auxiliar methods are not added for the sake of brevity . . 39
7 Example of how to use the package . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 41
8 Weather dataset . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 42

# Chapter 1

# Introduction

Machine learning is a scientific discipline whose objective is to provide algorithms that can automatically detect patterns in data, with the aim of be able to exploit those patterns to predict future data[1].

When it comes to classify machine learning algorithms, we have two main blocks: (1) supervised learning and (2) unsupervised learning. Supervised algorithms are the ones that the training of a model is based on having both the input and the expected output for the model. On the other hand, unsupervised algorithms rely only in using the input data to find similarities and interpretations[2].

One of the main applications of supervised machine learning is to solve classification problems. A classification problem is the problem of assigning one label for an item according to some of its features (p.e. in a bank each client (item) could be classified as potential customer for a new product or not (label)). We name observation to the set of features for a specific item, and class to the specific label for this observation. The goal of the classification algorithms is to be able to predict correctly the class of unseen observations (i.e. observations not used in the training process) after the training.

In this project we will focus in one of the earliest and most common classification algorithm: the decision trees. A decision tree is a tool based on creating a tree-like structure of decisions based on the features, and paths based on the answers of those decisions. Starting from the root node, the path follows a list of decisions until it reaches a leaf. This leaf contains the class which will be applied to the observation.

With the increasing usage of artificial intelligence in day to day operations, the need of being able to explain how the algorithms take decisions has lead to a new discipline in this field called eXplainable Artificial Intelligence (XAI)[3]. The decision trees provide a substantial advantage over other algorithms that make them specially interesting nowadays: it is incredibly easy to visualize those decisions, making them one of the most interpretable models of machine learning.

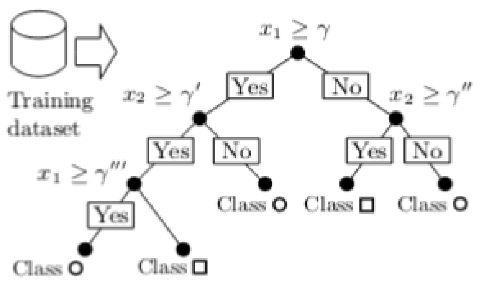


Figure 1: Example of a decision tree

**Source:** Explainable Artificial Intelligence (XAI): Concepts, Taxonomies, Opportunities and Challenges toward Responsible AI [3]

In [4] an optimal decision tree is defined as the tree with the minimum size (node count, depth…) that classifies correctly all the observations. In other approaches some percentage error is allowed, since perfect classification leads in general to overfitting, i.e., classifiers with high accuracy on the training set but do not generalize well.

In [5] it was stated that finding an optimal binary decision tree with a fixed depth is an NP-complete problem[1]. From that point, lots of approaches based on greedy heuristics or mathematical optimization techniques have appeared to overcome this problem. In this project, we will focus on the latest.

## 1.1 Description of the project

In this project we will explore the feasibility of using constraint programming techniques to find optimal decision trees. Lots of studies can be found along the line of creating new algorithms and encodings for solving the problem of finding those optimal trees using boolean SATisfiability problem (SAT)[7] [8][9], Linear Programming (LP)[10], or Constraint Programming (CP)[11].

Those studies are based on solving a sequence of decision problems where each decision problem is feasible (has a yes answer) if a tree of a given size $k$ and structural constraints can be built. By iteratively refining $k$, the optimal size is found when decision problem for size $k$ has a yes answer and for $k-1$ a no answer.

These decision problems can be encoded through the suitable formalism so that they can be solved by the application of mathematical or constraint programming solvers. The main advantage of this approach is that the user can leverage the advancements in the solvers without any modification to the encodings.

In order to assess those techniques, we will develop a library that implements some of those approaches,

[1]NP-Complete is a non deterministic problem that can be reduced in a polynomial time to the satisfiability problem[6]

from encoding the problem to finding the solution using the proposed algorithms. This library will allow the user to select between the different implementations, providing some degree of flexibility for different user cases. We will also conduct an extensive experimental study for all the approaches implemented.

This work is part of a project of the Logic&Optimization Group (LOG) at Universitat de Lleida (UdL)[2]. This group brings together researchers with different background from UdL and Universitat Politècnica de Catalunya (UPC). It focuses on the design of efficient solving techniques for industrial combinatorial optimization an machine learning problems, and on the design of automatic configuration and selection techniques for a wide range of applications. On the more theoretical side, this group specializes on computational complexity, proof complexity and complex networks analysis. As a member of the LOG group, during the development of this work we will have access to an extensive experience of its members in the optimization field, as well as having access to the high performance computation cluster required to run the benchmarks for the algorithms.

## 1.2 Personal motivation

With the rising usage of machine learning in day to day tasks, people can start to feel threatened by those algorithms that "take decisions for me that I can not understand". In order to be able to use safely those algorithms, we must consider two things: (1) the capacity to understand how they work, and (2) the robustness of those algorithms. As was stated before, decision trees are extremely easy to understand when it comes to evaluate why such a decision was taken (we only have to follow the path). But because of this simplicity, those trees might not be enough powerful to compete against algorithms such neural networks. Having the capacity to obtain optimal decision trees with reasonable resources in the real world, though, can provide those algorithms a huge boost in this "competition".

As a member of a research group where its main topic is mathematical optimization, being able to combine those two lines is an incredible opportunity to expand to potential unexplored ideas, thus being able to advance towards better machine learning algorithms and its underlying knowledge.

[2] http://ulog.udl.cat/

## 1.3 Objectives

The main goal of this project is to explore the methods of finding optimal decision trees using optimization techniques. Those techniques will be evaluated and compared using well-known benchmarks, thus assessing its feasibility in the real world problems.

To achieve this, we define those goals:

1. Create a library that implements the process of loading an instance, encoding it using a specific encoding, and retrieve its solution.
2. Use the Minizinc[3] modeling language to create parametizable encodings for each approach.
3. Implement the interface between the library and Minizinc.
4. Benchmark each approach implemented against well-known datasets[4].
5. Design the library in a way that can be easily extended with new encodings and/or techniques.
6. Implement at least 4 different approaches for obtaining optimal decision trees in the library.

## 1.4 Methodology of the project development description

While it might seem redundant to implement some existing ideas in this field, the goal of this project is to integrate all those ideas in a common library. Thus, the strategy this project follows is to research about existing algorithms with decent performance results, and integrate them in the extensible library.

After selecting which algorithms we want to integrate, and in order to keep the library as open to new ones as possible, we will first implement one of those algorithms and check the main structure of the library. Once this basic structure works, it will be the time to do the refactoring to make it extensible to the other algorithms selected.

Although during the implementation the algorithms will be tested for its correctness, once all the selected ones are integrated the benchmarking will be performed. This benchmarking will be executed in a high performance cluster to allow us to compare the algorithms with the results in the original papers.

[3] https://www.minizinc.org/

[4] Those can be either datasets used in the original papers or datasets used commonly in the bibliography.

| Task | Task | Start | End |
|---|---|---|---|
| 1 | Project definition | 19/02 | 01/03 |
| 2 | State of the art | 02/03 | 22/03 |
| 2.1 | Research for the state of the art of decision trees | 02/03 | 12/03 |
| 2.2 | Research for the algorithm 1 | 08/03 | 18/03 |
| 2.3 | Research for the algorithm 2 | 08/03 | 18/03 |
| 2.4 | Research for the algorithm 3 | 08/03 | 18/03 |
| 2.5 | Research for the algorithm 4 | 08/03 | 18/03 |
| 2.6 | Write the start of the art of the used tools (python, minizinc...) | 18/03 | 22/03 |
| 2.7 | Research for additional algorithms | 20/03 | 22/03 |
| 3 | Design and implementation | 23/03 | 23/05 |
| 3.1 | Get familiar with the minizinc tool | 23/03 | 31/03 |
| 3.2 | First library implementation to test the components integration | 01/04 | 10/04 |
| 3.3 | Implement the algorithm #1 in the library | 05/04 | 15/04 |
| 3.4 | Refactor the library to allow the addition of new algorithms | 16/04 | 20/04 |
| 3.5 | Implement the algorithm #2 in the library | 18/04 | 28/04 |
| 3.6 | Implement the algorithm #3 in the library | 29/04 | 08/05 |
| 3.7 | Implement the algorithm #4 in the library | 05/05 | 15/05 |
| 3.8 | Design the experiments | 16/05 | 17/05 |
| 3.9 | Benchmarking | 18/05 | 23/05 |
| 4 | Project report writing | 24/05 | 10/06 |
| 5 | Thesis delivery and defense | 11/06 | 24/06 |

Table 1: Planned tasks and its timing

## 1.5 Planning of the project

Next the Gantt diagram is presented with the planning of the project. The main restriction for this project is the time constraints set by the deliveries of the different parts of the project's memory. Thus, we divide the tasks to be done in the 5 deliveries, with those deliveries as milestones.

The tasks to be done for each delivery are listed in the table 1, and shown in the figure 2.

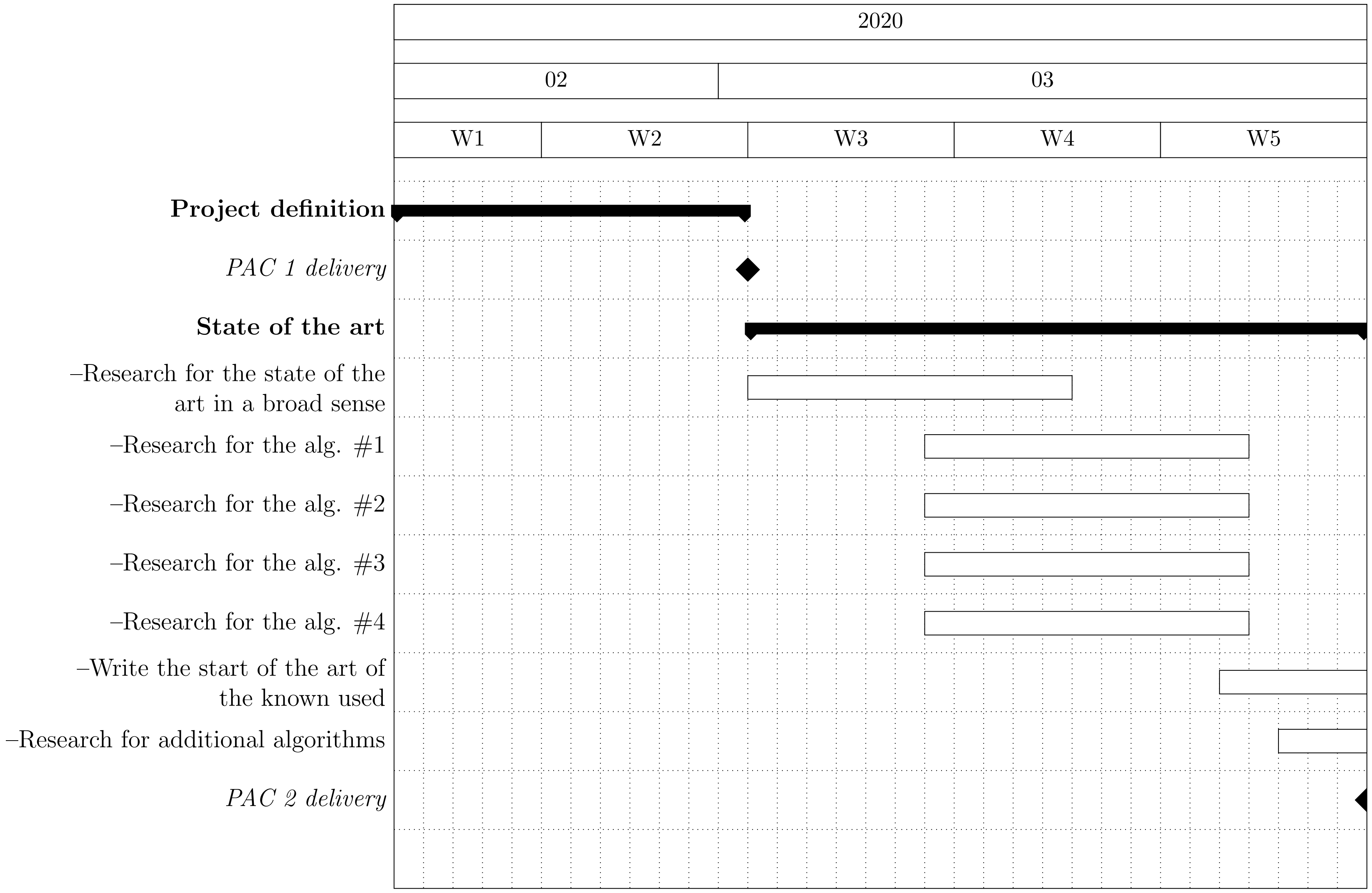


Figure 2: Project's planning (part 1/3)

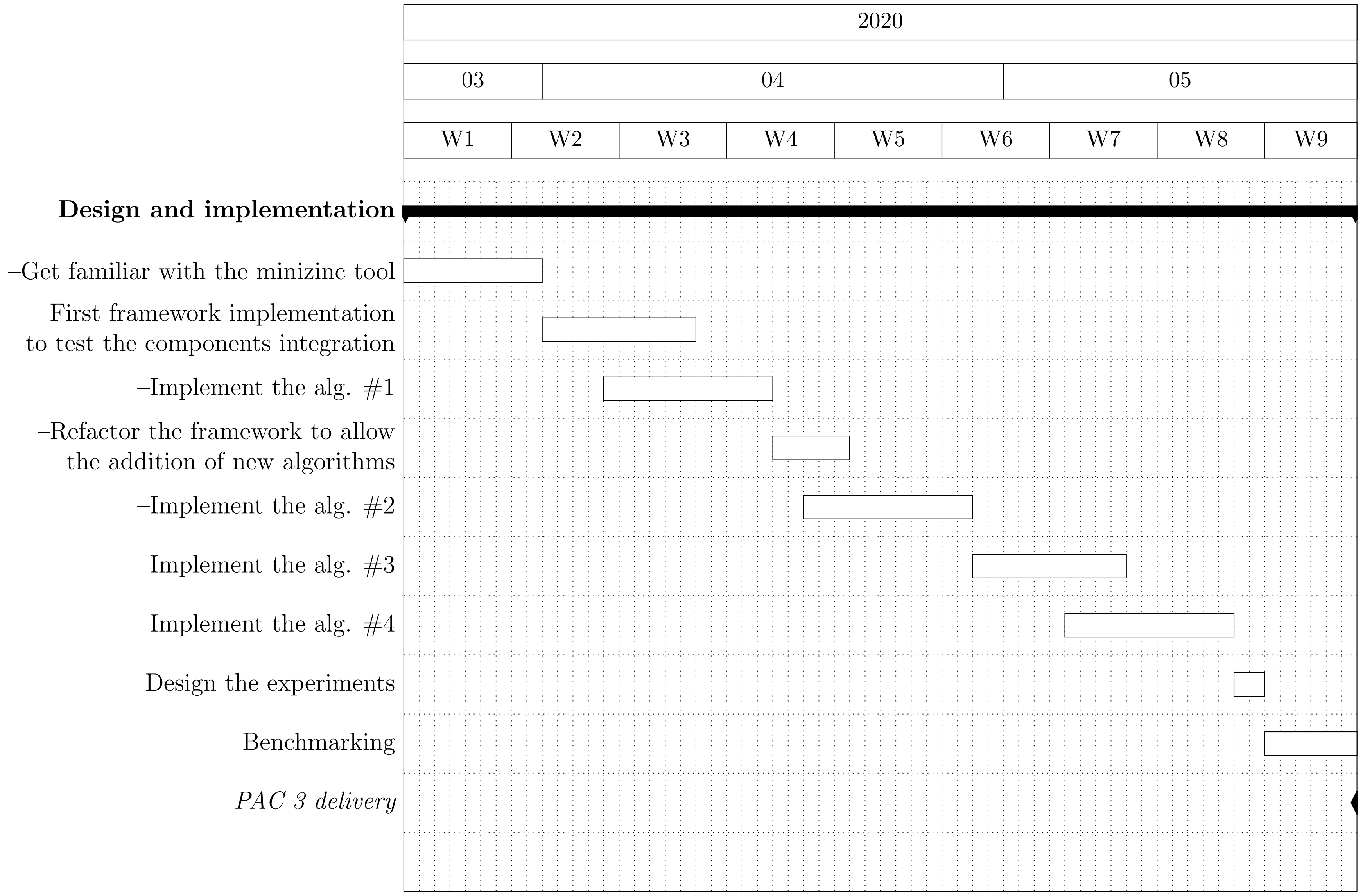


Figure 2: Project's planning (part 2/3)

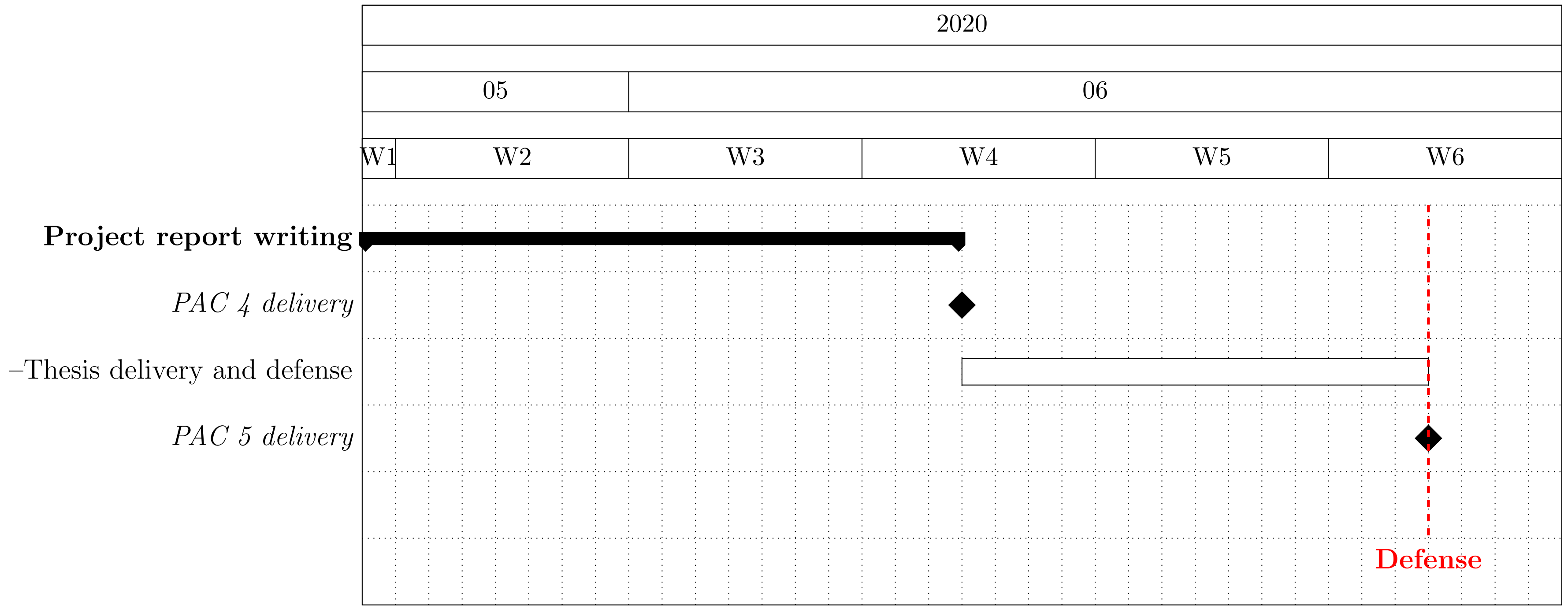


Figure 2: Project's planning (part 3/3)

## 1.6 Document structure

This report is divided in 5 chapters, explained below:

- Introduction: (The current chapter) presents the project and the motivation behind it.
- State of the art: In this chapter the previous work and knowledge relevant to the technologies and techniques used in the project is presented.
- Library design: This chapter shows the design and its considerations of the library that implements the selected algorithms.
- Library implementation: Complementary to the previous chapter, this chapter focuses on explaining how the specific algorithms are implemented in the project.
- Experimental results: This chapter shows the performance results of the different algorithms against a set of benchmark, as well as a comparison between those algorithms.
- Conclusions and future work: Here, the conclusions of the project are shown. Also, the potential research works to follow are explained.

# Chapter 2

# State of the art

This chapter provides an analysis of the state of the art for the concepts and tools that will be used in this project. It is structured in 3 main blocks: (1) the decision tree concept and algorithms, (2) the tools used for the formulation of the problems, and (3) the mathematical solvers used to solve those formulations.

## 2.1 Decision trees algorithms

Starting from 1963 with the publication of the first regression tree algorithm[12], the decision trees field has been extensively researched[13]. In those years, research has focused on multiple facets of those trees: from improving heuristics (for example, Classification and Regression Trees (CART)[14] (see below at 2.1.1) uses the Gini impurity to select which feature to use in a node), mining rulesets, pruning techniques…

In this section, we will review the state of the art of the approaches for creating those trees. Despite the fact that (as said before) decision trees can also be applied to regression problems, we will focus only in the usage in classification problems, as the output/prediction/classification when applying them to the regression problems can be a continuous value.

## 2.1.1 Heuristic greedy algorithms

When it comes to tackle classification problems, the usual strategy is to use greedy strategies. Those greedy strategies rely on exploring the near region of the search space to find locally optimal values, thus obtaining an approximation of the solution faster than strategies that ensure the answer is optimal. By definition, when NP problems increase linearly in size, the time requirements for finding a solution grows exponentially. Is in those cases that, in the real world, it might be better to find an approximate solution faster than finding the exact solution with an unknown delivery time. As the heuristics improve, the approximate solutions or the required time to find the tree also improve.

Is this benefit of having acceptable solutions faster that has led those heuristic greedy algorithms to have the usage share they have currently. Following, we present the greedy algorithms that are most used or perform better in real case scenarios.

#### 2.1.1.1 CART

Classification and Regression Trees (CART) was presented in [14], and it is a greedy algorithm that relies on constructing binary trees by, given the data that reaches a specific node, selects the best feature to split the data. This selection is based on the Gini impurity, a measure that represents the statistical error rate of labeling a random element of the set.

#### 2.1.1.2 ID3, C4.5, C5.0/See5

Ross Quinlan first developed Iterative Dichotomiser 3 (ID3)[15], an algorithm that creates a tree by splitting the training observations in each node based on the entropy or the information gain measure of each unused feature. The entropy[16] is the measure of impurity, disorder or uncertainty in a set. The information gain of a feature measures how much the entropy is reduced when the set is split based on this feature.

Seven years later, Ross Quinlan presented a new iteration of this algorithm, called C4.5[17]. The new features of this algorithm include accepting both continuous and discrete features, handling incomplete data points, using pruning to avoid overfitting, and accepting weights for the features.

Quinlan went further, and presented the tools called C5.0 on Linux and See5 on Windows[18]. Those tools are an implementation of the C4.5 algorithm. The version Quinlan's released is significantly faster and efficient in terms of memory, and also allows boosting[1].

[1] An ensemble method for improving the model predictions.

#### 2.1.1.3 CHAID

Gordon V. Kass presented Chi-square Automatic Interaction Detection (CHAID)[19], an heuristic approach to create decision trees based on adjusted significance testing. It is a tool to discover how the variables can be merged to explain the outcome in the given dependent variable using the chi-square test, that allows to measure the independence of two variables (in this context features) $x$ and $y$.

#### 2.1.1.4 ID5R, ITI

ID5R is an incremental algorithm created by Paul E. Utgoff[20] for inducing trees similarly to those created by ID3 algorithm; with the main difference that the instances are presented serially instead of giving all the training data in batch. In 1997, a new algorithm based on the same incremental procedure was developed[21], called Iterative Tree Induction (ITI).

Those approaches are suitable for online training, i.e. training the tree while predicting instead of pretraining the model using a batch of training data; as they are intended to construct the trees in an iterative way.

Although being from 1984 and 1993 respectively, the most used heuristic algorithms to find decision trees currently are CART and C4.5. For example, tools like **Scikit Learn** implement a variant of CART[22]. Although they do not find optimal decision trees, their good results are well known. Thus, we can use them to compare the state of the art of the mathematical optimization approaches with the heuristic approaches.

Those heuristic algorithms can be used to obtain upper bounds for the mathematical optimization approaches, as done by [8] with the Iterative Tree Induction (ITI) algorithm.

### 2.1.2 Mathematical optimization algorithms

Here, we present the approaches that in this project will be implemented in the library. In the implementation chapter those algorithms will be explained more extensively.

#### 2.1.2.1 Optimal decision trees with SAT

Multiple approaches have been presented to find optimal decision trees using SAT solvers [7][8][9]. In this project, two of those approaches will be implemented.

The first approach that will be implemented is presented in [8]. It shows a new SAT formulation of this problem that encodes a tree with a given size $N$. Using a SAT solver, it verifies the existence of a tree with this size that can encode perfectly the given train dataset. The optimal tree is found when, for a satisfiable tree with size $N$, the encoding of the trees with size $N' < N$ is unsatisfiable.

A similar strategy used in this paper can be found in [7], but the former achieves to create smaller encodings. Despite this, the encodings are still large; so to compare to other algorithm it uses an approximation using only a percentage of the available data in the encoding.

As said before, in order to search for the optimal tree, the ITI algorithm is used to set an upper bound. Then, using an iterative approach, the tree with $N = UB$ is encoded, and solved. If the problem is satisfiable, the size of the tree is reduced by one, and solved another time until $N'$ is found to be unsatisfiable. Then, any solution for the solved problem $N' + 1$ is an optimal tree.

The second approach is the one developed in [9]. It also uses a SAT solver as an oracle to decide whether the current tree layout is satisfiable or not. The main difference is that, while the former approach creates the encoding for all the tree and the entire training dataset, the latter is based on generating the feature constraints and class constraints, and then using incremental inference adding rows in the dataset that makes the current tree inconsistent in order to refine it.

This approach allows to specify the maximum of nodes (or leave it undefined). The algorithm will search for the smallest tree consistent with all the observations limited (or not, if undefined) by the $MaxNodes$ constraint.

#### 2.1.2.2 Sub-optimal decision trees using the cover size constraint

In [11], a constraint programming approach is presented. Using a branch-and-bound algorithm in a CP solver, it eliminates parts of the search space in which no solutions can be found. The encoding of the problem is based on a modification of the CoverSize[2] constraint.

This approach is focused on finding the decision tree that, with a limited depth, minimizes the classification error. Thus, this algorithm by default does not find the optimal tree.

[2]The cover size is the problem of linking an itemset to the number of transactions containing the itemset[23] [24]

#### 2.1.2.3 BinOCT

BinOCT is a binary linear programming formulation presented in [10]. Given a depth $K$, this encoding finds the tree that minimizes the classification error. This approach focuses on reducing the dependency of the problem size with the dataset size: the number of variables depends of the maximum number of unique combinations of the features values.

BinOCT allows the usage of non-binary feature values, splitting the values of the features in sets according to the label; thus it is not required to do a discretization nor a one hot encoding pre-processing step.

## 2.2 Tools used

This project is intended to be developed as a **Python** package. The implementation will also need some sort of tools that will enable the solving of the mathematical optimization approaches.

In this section, the different tools related to the implementation of the project are explained. First, the programming language intended to be used will be introduced. Then, there is the description of the tools that will be used to implement the different optimization algorithms.

### 2.2.1 Python

**Python**[3] is a scripting language, that gained popularity due to its high-level language properties that allow the fast prototyping of algorithms; specially in the artificial intelligence field.

It is a multi-platform, general purpose language, interpreted during its execution. As this language resembles pseudo-code, it is one of the most used languages when it comes to researchers that do not have the deep knowledge of the low-level details of the system.

### 2.2.2 Minizinc and Flatzinc

**Minizinc**[4] is an open source constraint modeling language. It allows to define formally mathematical optimization problems in this language in a language that resembles mathematical expressions and constraints, and then connect to a set of mathematical solvers to obtain the solution.

[3] https://www.python.org/
[4] https://www.minizinc.org/

An example of how to express the Knapsack problem[5] is shown in the listing 1.

```
% From Minizinc examples
int: n; % number of objects
set of int: OBJ = 1..n;
int: capacity;
array[OBJ] of int: profit;
array[OBJ] of int: size;

array[OBJ] of var int: x; % how many of each object

constraint (x[i] >= 0 | i in OBJ);
constraint sum(i in OBJ)(size[i] * x[i]) <= capacity;
solve maximize sum(i in OBJ)(profit[i] * x[i]);

output ["x = ", show(x), "\n"];
```

Listing 1: Example of the encoding of a Knapsack problem in Minizinc

**Flatzinc** is the counterpart of this language, and its the responsible of compiling the minizinc mathematical language into the specific encoding accepted by the solver used, for example converting the `a->b` (a implies b) expression into the linear programming ".lp" format.

### 2.2.3 Mathematical solvers

#### 2.2.3.1 SAT solvers

The boolean SATisfiability problem (SAT) is one of the most studied NP hard problems. This problem is defined as, given a set of clauses (formula), composed by the disjunction of boolean literals, it is possible to find an assignation for the variables that compose those literals such as all the clauses are satisfied. So, in those problems we have:

**Boolean variables** Variables that can be assigned to either True or False.

**Literals** Minimal unit of the problem, it is a variable with its sign (i.e. a variable negated or not).

**Clause** A clause is a disjunction (boolean or operation) of literals.

**Formula** A conjunction (boolean and operation) of clauses.

[5]https://en.wikipedia.org/wiki/Knapsack_problem

As SAT problems are one of the most studied NP hard problems, it is common to encode the NP hard problems in this formulation (as done with the decision trees), and then solve it using the extensively developed techniques.

To find the best solvers for this problems, we can use the yearly international SAT competition[6]. The best solvers in 2019 were:

- Maple
- CaDiCaL
- Plingelin

#### 2.2.3.2 Linear programming solvers

The modern era of optimization starts when Dantzig developed simplex method in the late 1940s [25]. This method allowed to solve in a systematic and efficient way large models of linear programming. Linear programming is the application of an algorithm to maximize or minimize a given objective linear function, subjecting it to a set of linear equality and inequality constraints [26]. A linear function is a function whose constraints are of the first order (i.e. does not contain variables raise to any power other than one or variables multiplied by each other) [27].

A linear problem can be expressed in the following way:

$$maximize\ c^T x, \text{ subject to } Ax <= b, x >= 0 \tag{2.1}$$

where $c$ and $x$ are vectors in $\mathbb{R}^n$, $b$ is a vector in $\mathbb{R}^m$, and $A$ is an $m \times n$ matrix. $x$ represents the vector of unknown variables, $c$ and $b$ represent the vectors of known coefficients, $A$ is a known matrix of coefficients, and $^T$ is the matrix transpose. The expression $c^T x$ is the called objective function, and can be maximized or minimized.

The importance of the linear programming techniques in fields such management, economics, finance, and engineering since 1950; lead to a high investment in improving the solvers used. Thus, the maturity level of some of those solvers and its maintainers guarantees both a high quality software and bleeding edge techniques.

Lots of solvers exist for those problems, and because LP has multiple variants (integer programming, mixed integer programming. . . ), some solvers excel in a specific subset of those.

Following, we show three LP solvers:

[6] http://satcompetition.org/

- **CPLEX**[7] IBM's solver for linear programming, mixed integer programming, and quadratic programming.
- **Gurobi**[8] Gurobi is a mature solver for mathematical optimization problems. It is also one of the most competitive solver in terms of performance.
- **CLP**[9] CLP is the open source solver for linear programming from the Coin-OR research community.

[7]https://www.ibm.com/analytics/cplex-optimizer
[8]https://www.gurobi.com/
[9]https://projects.coin-or.org/Clp

# Chapter 3

# Library design

One of the most important goals of this project is to provide an extensible library that will allow its users to use different approaches to find optimal decision trees using constraint programming without requiring to mess with the implementation and knowledge details of each approach. In this chapter, the design of such library is presented.

## 3.1 Library overview and design

The library should provide two main functionalities: (1) find an optimal valid decision tree representation using any of the implemented methods, and (2) offer a class that represents this decision tree which can be used to classify observations.

In the context of finding decision trees using CP, a valid decision tree is an assignation of the variables of a model for a specific encoding where all the constraints of this model are satisfied.

All the algorithms perform a search on the set that contains all the valid decision trees for a dataset, but the specific details of how this search is done are different for each one. Thus, the library should abstract those details and proceed as shown in the algorithm 1.

The usage of such an algorithm relies on the premise that each implementation will deal with the entire search space and return either an optimal decision tree or *"No solution"*. This will come back later in the implementation for each method. Moreover, as the solution is "encoding specific", the algorithm should not return the raw solution, but the solution converted to the said class that represents a decision tree.

Those two responsibilities are the two functionalities that are mentioned in the begin of the chapter,

**Algorithm 1:** Searching for the optimal decision tree

```
input: A dataset data as a matrix of $n\_feats + 1 \times n\_rows$
search_space ← SOMETHING ;                          //specific of each encoding
while more_params_available(search_space) do
    params ← next_params(search_space);
    model ← encode(data, params);
    solution ← solve(model);
    if solution is optimal then
        return solution
    end
end
return “No solution”
```

and according to the single-responsibility principle[28] the library should have two different components that deal with the details of those, although they are interconnected as each tree creation must be according to the specific solution provided.

Another functionality not required but desirable is that the package provides a way to prepare the data for those algorithms. As this project focuses on the creation of decision trees for datasets with boolean features, a third component is created to prepare the data to a common format expected by the algorithms, so the implementation of the different methods can assume a clean dataset ready to be encoded.

In the following sections the different external components used are explained.

## 3.2 Interaction with external components

This library is not self-contained, as it relies on external tools to solve the CP models to find the decision tree. Thus, here those components are listed, and how the interaction with them is designed.

#### Library user source code

The most important external component is the projects that will use this library, otherwise this entire project will have no purpose. The library must hide as much as possible of the different implementation details to the end users. Thus, the user should be able to choose between one algorithm or another one without requiring to change much code as the goal of the algorithms is common (i.e. finding a decision tree). To achieve this, the different algorithms and the decision trees that represent their solutions must have a common interface that allow the following basic actions:

- Algorithm implementation:
  - Create the model for a dataset
  - Find the tree
- Decision tree
  - Classify an instance
  - Get the accuracy of the tree for a dataset
  - Print the tree

### Minizinc and solvers

Most of the algorithms will be encoded using the **Minizinc** language. This is a high-level mathematical language that cannot be used directly against the CP solvers. Thus, the library relies on the **Minizinc** official **Python** package[1] to input the model encoded in this high-level language to a solver that has an interface for this tool. This abstraction will allow the user to select between a set of different solvers, providing the flexibility of choosing between commercial implementations or open-source / freeware solvers; broadening the target users that otherwise could be limited by the restriction of having to buy a license to use this library.

### Glucose

Although the final goal is to have a **Minizinc**-like encoding for each of the algorithms, some of them rely on SAT solvers. At the moment of writing this project, no available interface was found to SAT solvers such as **Minisat** or **Glucose**. Thus, some algorithms have both implementations, in **Minizinc** and in pure SAT. This will allow us to compare the performance of the other solvers tested, determining if such interface should be developed to call SAT solvers from **Minizinc** in future iterations of the project.

[1] https://pypi.org/project/minizinc/

### Scikit-Learn

The last component that this library will interact with is the **Scikit-Learn** package[2]. This package will provide the algorithm for finding decision trees using a modified version of CART. Although this method is greedy, it will serve as a comparison point to assess the advantages of using CP approaches to find optimal decision trees.

The figure 3 shows the entire layout of the project taking into account the external components.

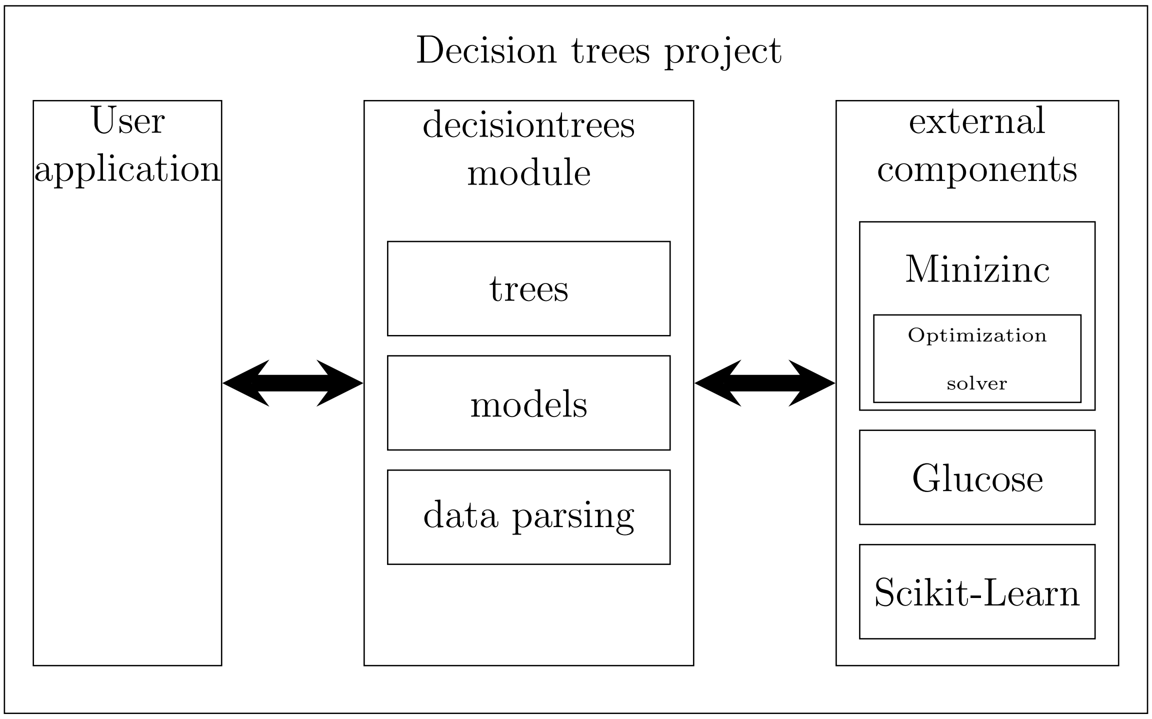


Figure 3: Layout of the package interacting with the external components

[2]https://scikit-learn.org/stable/

# Chapter 4

# Library implementation

Having discussed the design of the library in the previous chapter, here the implementation of the project is presented. First, the main overview of the solution is shown, as well as a minimal explanation about the technologies related. Then, the details of how the package is implemented are shown. Finally, there is a brief explanation about how to install the package and how to use it.

## 4.1 Overview

This project is implemented in the form of a **Python 3.6** package, called `decisiontrees`. This package is developed in the context of the LOG Group at UdL, so it is not available to the public using the official **pip** repositories[1]. Despite this, for the sake of this project some parts are disclosed to explain the implementation details.

The highlights of this package are:

- Provides different algorithms (some of them with multiple implementations) to find optimal decision trees using CP.
- Solve the encodings using the **Minizinc Python**'s Application Programming Interface (API) to delegate the interaction with a large set of optimization solvers.
- Parsing the encoding specific solution values to create a common object representing a decision tree that can be used to classify observations.
- Integration with **Scikit-Learn**'s `sklearn.DecisionTreeClassifier` tool, implemented with the same interface as the other algorithms to have a common usage.

[1]To obtain a copy of the package, please contact with the author of this project

Apart from **Python 3.6** and the packages listed in the package requirements (installed automatically thanks to the **pip** package manager), this package depends on a working installation of **Minizinc** as well as all the solvers that the user wants to use. Note that the default installation of **Minizinc** contains bundled solvers[29], so this step is not necessary but highly recommended as other solvers (such as **Gurobi**[2] or **CPlex**[3]) that have an interface with **Minizinc** can solve the models faster. The comparison of all the available solvers is out of the scope of this project.

## 4.2 Implementation

The package has the following structure (figure 4):

```
decisiontrees
├─ cnf
│   ├─ cnf.py
│   ├─ cnf_components.py
│   └─ utils.py
├─ models
│   ├─ base.py
│   ├─ sklearn.py
│   └─ sat_models.py
├─ cross_validate.py
├─ trees.py
└─ data_parsing.py
```

Figure 4: Package structure

In the design chapter (see 3), it was stated that this package has two main responsibilities. Those correspond to the components shown in the package structure as the `models/` and `trees.py` components. It also provides the methods to parse the data and to perform cross validation over the datasets.

In the sections below each of the components is described in more detail.

[2]https://www.gurobi.com/
[3]https://www.ibm.com/analytics/cplex-optimizer

### 4.2.1 CNF module

The first component in the hierarchy is the `cnf/` module. Its name stands for Conjunctive Normal Form. It is included because one of the algorithms provided (as seen below in 4.2.3.1) is an implementation that works with files in *.cnf* format, and solves them using the SAT solver **Glucose**[30].

Its main purpose is to serve as a high-level module to encode a SAT formula into the *.cnf* format using declarations that resemble the boolean logic language such as "A implies B"[4], while keeping track of the variables by name instead of its numeric representation. It is also the responsible of hiding the interaction with the solver.

The API reference for this module can be found in the appendix A.

### 4.2.2 Models module

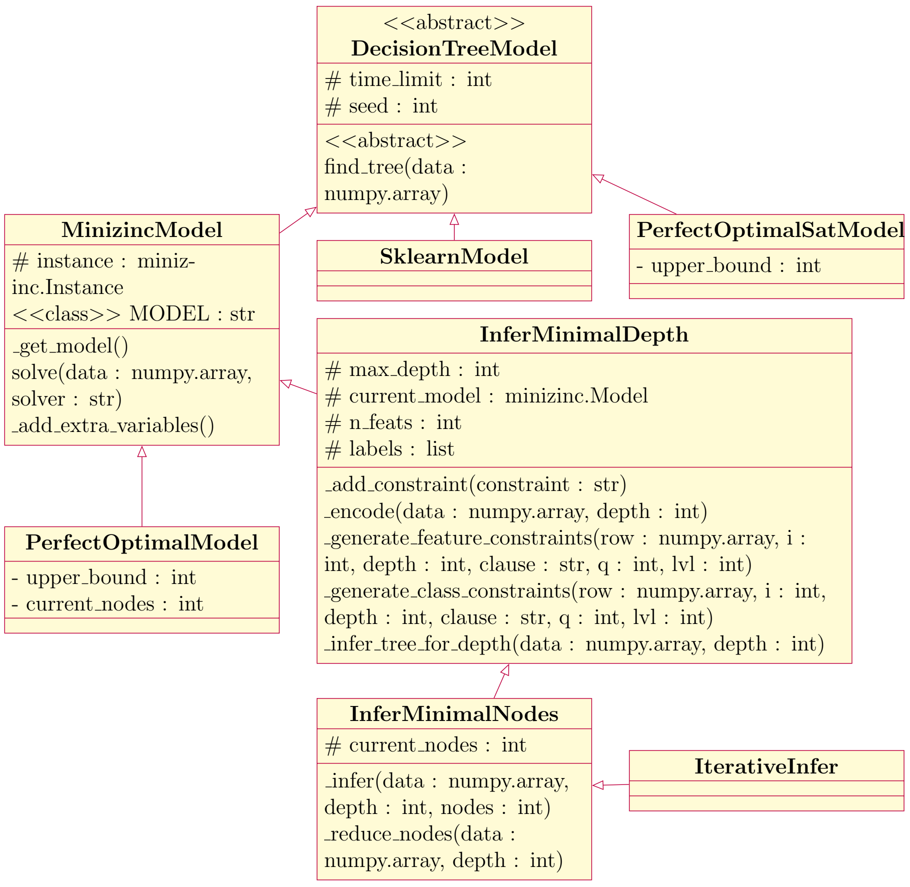
This component has the specific implementation for each type of algorithm that this project seeks to compare. The entire module is based on offering a single interface: `DecisionTreeModel`[5] This interface provides the method `find_tree`, that receives a bi-dimensional **Numpy** array, whose columns are the features plus the label in the last column, and the rows are the observations of the train dataset; and returns a tuple containing both the found decision tree and the accuracy of the tree in the train dataset (as most methods find the optimal tree for the provided dataset, this is usually 1.0 (or 100%)), but it is added for future extension. In this section only the parts related to the encoding and solving algorithm are explained. How each algorithm's variables are converted to a decision tree is explained in the section 4.2.5.

The class hierarchy is shown in the figure 5. It can be seen that the `DecisionTreeModel` has three descendants:

- `MinizincModel`. It serves as a base for the models implemented using **Minizinc**. It abstracts the creation of the **Minizinc** related objects (model, instance, solver), adds the dataset and the number of features to the instance as variables, and then calls the `solve` method. The implementations of the specific algorithms inherit from this class.
- `PerfectOptimalSatModel` This class model (explained below, see 4.2.3.1) provides the implementation that encodes a model directly against SAT, instead of using **Minizinc**. Thus, it inherits from the base class `DecisionTreeModel`.

[4]In *.cnf* this should be encoded such as "-1 2"; where 1 correspond to the A variable and 2 correspond to the B variable.

[5]Note that in **Python** the concept of interface separated from class does not exist.



<<abstract>>
DecisionTreeModel
# time_limit : int
# seed : int
<<abstract>>
find_tree(data : numpy.array)
MinizincModel
# instance : minizinc.Instance
<<class>> MODEL : str
_get_model()
solve(data : numpy.array, solver : str)
_add_extra_variables()
SklearnModel
PerfectOptimalSatModel
- upper_bound : int
InferMinimalDepth
# max_depth : int
# current_model : minizinc.Model
# n_feats : int
# labels : list
_add_constraint(constraint : str)
_encode(data : numpy.array, depth : int)
_generate_feature_constraints(row : numpy.array, i : int, depth : int, clause : str, q : int, lvl : int)
_generate_class_constraints(row : numpy.array, i : int, depth : int, clause : str, q : int, lvl : int)
_infer_tree_for_depth(data : numpy.array, depth : int)
PerfectOptimalModel
- upper_bound : int
- current_nodes : int
InferMinimalNodes
# current_nodes : int
_infer(data : numpy.array, depth : int, nodes : int)
_reduce_nodes(data : numpy.array, depth : int)
IterativeInfer

Figure 5: Models hierarchy

- `SklearnModel` The last model that inherits directly from this class is the model that uses **Scikit-Learn** to find the decision tree using a greedy approach: CART.

## MinizincModel

This class provides the method `solve`, that can be called from its descendants to solve the model using **Minizinc**. It has two arguments: `data`, a **Numpy** array representing the dataset; and `solver`, a string that will be used for the solver lookup. It proceeds as following:

1. It creates a `minizinc.Model` object using the `_get_model()` method.
2. Retrieve a `minizinc.Solver` instance using the `minizinc.Solver.lookup` method, feeding it with the `solver` argument received.
3. It creates a `minizinc.Instance` object with the solver and the model, and sets two variables in those models:
    - `array [int, int] of int: data`: a bidimensional list representing the dataset.

- `int: n_feats`: the number of columns that represent a feature.

  Those two variables must be declared in the model.

4. Adds the extra variables (if any) using the `_add_extra_variables` method.

5. Solves the **Minizinc** instance and returns its result.

The two auxiliary methods that the `solve` relies are:

- `_get_model()` Class method that returns a `minizinc.Model` object. It has a default implementation that creates the model from a file using the class variable `MODEL`. The descendants must either set the class variable or override this method.

- `_add_extra_variables`. The descendants should implement this method if additional variables should be set in the instance.

The `solve` method is represented in the sequence diagram 6.

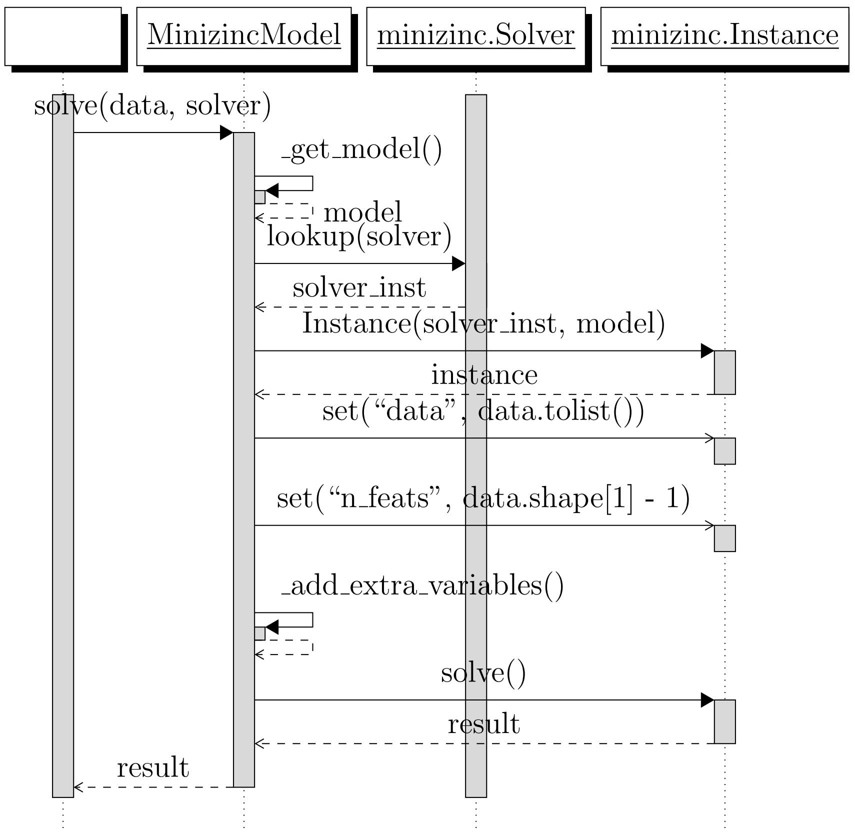


Figure 6: Solve method for MinizincModel

### SklearnModel

This model implements the `find_tree` method using a greedy approach. Its implementation is pretty straightforward, and is shown in the listing 2.

```python
def find_tree(self, data: np.array) -> Tuple[SklearnDT, float]:
    train_x = data[:, :-1]
    train_y = data[:, -1]

    model = DecisionTreeClassifier(random_state=self.seed) \
                .fit(train_x, train_y)
    train_score = model.score(train_x, train_y)

    tree = SklearnDT.from_sklearn(model)

    return tree, train_score
```

Listing 2: SklearnModel.find_tree method

Once the main idea of how the models work has been explained, the next sections detail how each algorithm is implemented.

### 4.2.3 SAT models

The first type of algorithms implemented in this project are the SAT approaches explained in 2.1.2.1. We have two different algorithms:

#### 4.2.3.1 PerfectDecisionTrees

The first approach is the one developed in [8]. It is implemented using two different methods: a pure SAT solver approach, and using **Minizinc**. This algorithm provides an encoding that, for a given number of nodes $N$ and a dataset, determines if a decision tree that can classify all the observations of the dataset exists. To find the optimal tree, it starts using a value of $N$ set to the provided upper bound, and reduces this value by two each iteration until this tree does not exist (i.e. the model is unsatisfiable). Thus, the last tree found is guaranteed to be optimal.

Although both approaches use a different method for asserting the existence of the said tree for each number of nodes, the rest of the algorithm is similar. Thus, the `find_tree` method is almost the same, with the difference being in how the tree is obtained and the specific line that checks if the solver reported that the model is satisfiable or not.

### Pure SAT approach

The SAT approach relies on the `cnf/` package to create the different constraints and solving the resulting *.cnf* file. The implementation of the algorithm is divided in two classes: the `PerfectOptimalSatModel` class, which implements the `DecisionTreeModel` interface; and the `SatDecisionTreeEncoder` class, that encapsulates the encoding provided for a specific number of nodes and a dataset.

The `SatDecisionTreeEncoder` receives in its constructor an integer representing the maximum number of nodes, and a **Numpy** array representing the dataset; and has three main methods:

- `encode` This method performs the creation of the constraints into the internal `cnf.CNF` object. The implementation of the entire method is not included in this report as it is pretty straightforward. In the appendix B (see B.1) it is shown how each constraint is created.
- `solve` Once the model is encoded, it can be solved using this method. It is simply a passthrough method, as it only calls the `solve` method of the internal `cnf.CNF` object.
- `get_tree` It creates a `DecisionTree` instance using the `TreeFactory` (see 4.2.5).

### Minizinc approach

The **Minizinc** approach relies on a model file created in advance, that represents the entire encoding. This file, shown in the appendix B.2, is expanded when calling the `solve` method of the `MinizincModel` parent class automatically when sending the model to the solver by **Minizinc**. Thus, the only required step each time is to set the value for the variable that represents the number of nodes in this model ($N$), as the data is added by the parent class. Because of this, the class `PerfectOptimalModel` only implements the iterative part of the tree search, and calls solve each time with a different value for the variable $N$.

#### 4.2.3.2 InferDT

The approach proposed in [9] is composed by three phases. As the objective of this project is to compare the different algorithms, three different models are implemented. Though, each model actually extends the previous step, so the inheritance is a strong advantage in this approach. The specific details of how the constraints are implemented are shown in the appendix B.3.

##### Inferring the tree with an optimal depth

The first step is to find the decision tree that minimizes its depth. The main difference with the perfect decision tree approach explained above, because this algorithm relies on generating the constraints in an iterative way the entire model has to be constructed dynamically.

The `find_tree` method is a loop that iterates through the depths in range from 2 (the minimum depth) to a maximum depth specified. In each iteration, it searches for a satisfiable tree using the method `_infer_tree_for_depth` that receives both the data and the depth. If a tree can be inferred, then the algorithm stops and returns this tree as the tree with minimal depth (note that because it searches from a small depth to the maximum depth, if the optimal depth is $d$, all the depths $d'$ such that $d' < d$ the solver will find the model unsatisfiable).

The method `_infer_tree_for_depth` creates the new model using the `_encode` method, and solves it using the `MinizincModel.solve` method.

##### Inferring the tree with an optimal number of nodes

The next step consists on, having found the minimal depth, try to minimize the number of nodes that are used. This is done using a dichotomic search in the `_reduce_nodes`, once the satisfiable depth is found. To allow the next step to be implemented, both when searching the minimal depth and the minimum number of nodes, it uses the `_infer` that solves the model for a given depth and a specific number of nodes. It also extends the `_encode` method to add the new constraints that encode the maximum number of nodes.

### Iterative inferring

The last step of the proposed algorithm is to reduce the number of instances that are used to create the tree, as the main problem of those methods is that the number of constraints depends on the number of observations used to create the tree. To do this reduction, the `_infer` is modified to start inferring the tree using only a subset of rows, the enough to have at least two different classes in this subset. Once a tree is found for this subset, the remaining rows are classified using the tree, and if a row is classified incorrectly, the row is added to the subset and the tree is inferred another time, until all the rows are classified correctly (or the model is reported to be unsatisfiable).

## 4.2.4 Cross validate

This component provides a method to perform cross-validation given a model class, a number of folds, the dataset and the arguments for the model class. It also contains a class called `CrossValidationResult`, shown in the listing 3. This class is used to accumulate the results for each fold split during the cross validation, and then apply a specific metric (mean, median...) to the results all at once.

```python
class CrossValidationResult:
    def __init__(self):
        self.train_accuracies = list()
        self.test_accuracies = list()
        self.sizes = list()
        self.solved = 0

    def add_result(self, train_accuracy, test_accuracy, size):
        self.train_accuracies.append(train_accuracy)
        self.test_accuracies.append(test_accuracy)
        self.sizes.append(size)
        self.solved += 1

    def get_results(self, metric):
        return (metric(self.train_accuracies),
                metric(self.test_accuracies),
                metric(self.sizes),
                self.solved)
```

Listing 3: CrossValidationResult class

The cross validation uses the **Scikit-Learn** `KFold` utility to split the data, and then searches for the decision tree using the model passed as an argument. It records both the train and the validation accuracy, as well as the size of the tree found. The cross validation method is shown in the listing 4.

```python
def cross_validate(model_class, n_folds, data, time_limit, seed,
    *args, **kwargs):
    kf = KFold(n_folds, random_state=seed, shuffle=True)

    result = CrossValidationResult()

    model = model_class(time_limit, seed, *args, **kwargs)

    for i, (train_index, test_index) in enumerate(kf.split(data)):
        fold_data = data[train_index]
        tree, train_accuracy = model.find_tree(fold_data)

        if tree:
            test_accuracy = tree.get_accuracy(data[test_index])

            result.add_result(
                train_accuracy,
                test_accuracy,
                tree.size()
            )
    return result
```

Listing 4: cross_validate method

### 4.2.5 Trees

Having the algorithms only fulfills half of the objective of make those new approaches user-friendly. Once the result is obtained, it has to be converted to a class that can be used without knowing the specific details of each encoding. This problem is dealt in the `trees/` module. This module provides two main classes: `DecisionTree` and `TreeFactory`.

The class `DecisionTree` is the class that represents the trees found during the solving processes, ready to be used. They serve as a fully functional decision tree model, able to classify new observations based on the learnt patterns during the training. It offers the following methods:

- `classify` This method implements the most important feature for a decision tree. This is classifying observations based on the previous knowledge.
- `print_tree` Following the topic of XAI, the tree can be printed to the console in an intuitive structure, so it is easy to inspect the resulting tree after the training process. An example of this tree printed is shown in the listing 5.
- `get_accuracy` One of the common tasks in machine learning is to compare the models using not the accuracy during the training step, but with a test set (i.e. rows that are never seen during the training of this model). This method allows to compute the accuracy of the trained decision tree for a given dataset.
- `size` To assess that the algorithms find the optimal value for the known data, this method is implemented. It gives the number of nodes (decision nodes and leaves) that compose the tree.

```
- Split by feature 71
| - Split by feature 70
|   | - Split by feature 69
|   |   | - Leaf has class (-)
|   |   | - Leaf has class (+)
|   | - Leaf has class (+)
| - Leaf has class (+)
```

Listing 5: Optimal tree for the “irish” dataset (see table 3)

As many of the implementations for tree-like data structures, this implementation follows a recursive pattern to represent the entire structure. Thus, the `DecisionTree` class is not only the entire tree but also each node and leaf, as the left and right child of a node can be seen as independent sub-trees. This recursivity eases the implementation of the methods that need to iterate the tree, and conveniently all the methods of this class require this iteration (one could think that the `get_accuracy` is called for a single tree, not on sub-trees, but this method is only a fancy wrapper for the `classify` method applied to an entire set of rows).

### classify

This method starts in the root node, and keeps calling recursively the left or right child according to the observation’s value in the feature chosen for each node. Once a leaf is reached, it returns the value assigned to this leaf.

### print_tree

This method prints the current node with a given prefix (that will represent the characters that "draw" the tree along with the correct indentation). Then, it appends the next level of indentation to this prefix, and calls the same method for the left child and then the right child. If one of the printed nodes is a leaf, it simply prints the value assigned to the leaf, and stops the recursive calls.

### get_accuracy

To get the accuracy for an entire set of observations, this method first applies the `classify` method to each of the observations given, and then compares the result of the label assigned by the tree with the real labels provided. Thus, the accuracy is computed as[31]:

$$1 - \frac{errors}{cardinality(observations)}$$

### size

This is the easiest method to implement for a tree. Defining the size of a node as the sum of the sizes of the sub-trees hanging from this node plus one (the node itself), the method just returns:

- 1 if the node is a leaf
- 1 + `left_child.size()` + `right_child.size()`

Thus, starting from the root node we can find the size of the tree.

It is important to mention that all the algorithms implemented in this library will use this class to represent the tree with the exception of the model that uses **Scikit-Learn**. This model has its own decision tree class, that implements the method to get the accuracy and the classify method as wrappers for the internal `sklearn.DecisionTreeClassifier` model used. Those methods will not follow the recursive strategy as they can be called directly to the model. This class has a problem, and it is that inspecting the resulting model it is possible to find which feature is used each node, but not the class assigned to the leaves. But as this class is only different internally, the class is private, and when it is returned it can be used as a `DecisionTree` object.

The other important component of this module is the `TreeFactory` class. that implements the factory pattern[32]. Its a class composed entirely by static methods that contain the logic to translate the

| Model | Factory method | arguments |
|---|---|---|
| SklearnModel | from_sklearn | DecisionTreeClassifier |
| PerfectOptimalSatModel | from_perfect_optimal_sat | SatDecisionTreeEncoder |
| PerfectOptimalModel | from_perfect_optimal | minizinc result |
| InferMinimalDepth | from_infer | minizinc result, depth |
| InferMinimalNodes | from_infer | minizinc result, depth |
| IterativeInfer | from_infer | minizinc result, depth |

Table 2: Correspondence of each model with the factory method used

specific result of each algorithm to an instance of the `DecisionTree` class. It is designed to be used exclusively from the `models/` module, as the implementation details of how the problem and the solution are encoded should be contained inside the package, and those methods only translate the solution encoding to an instance of a decision tree.

Currently it has 4 methods corresponding to the models as shown in the table 2. The conversion of each encoding is explained below.

### from_sklearn

To convert the sklearn model to a decision tree the factory inspects the attribute `tree_` of the `sklearn.DecisionTreeClassifier` class. This attribute has the description of each node of the tree as arrays. This internal structure can be found in the **Scikit-Learn** official documentation[33].

The method has an identifier for the current node that it is inspecting, and calls recursively the creation of the sub-trees for the left and right children (stored in the `tree_.children_left` and `tree_.children_right` arrays). The internal structure of the model also assigns both the left child and the right child identifiers to the same value for leaves, thus it is possible to determine when this recursion has to stop. Finally, the feature used to split each node is found inspecting the `tree_.feature` array.

### from_perfect_optimal_sat and from_perfect_optimal

Those two methods are explained together, as the logic behind them is the same, with the difference being how the values of the variables are accessed.

The variables inspected for creating the tree are:

- $V_i$: True if the node $i$ is a leaf.
- $C_i$: True if the assigned class for node $i$ is 1.

- $L_{i,j}$: True if the node $j$ is the left child of the node $i$.
- $R_{i,j}$: Counterpart for right child of the $L$ variable.
- $A_{i,f}$: True if the node $i$ has the feature $j$ as the decision criterion.

For each node $i$ (starting from the root identified as $i = 0$), the method finds whether this node is a leaf, and if so creates the `DecisionTree` representing this leaf and the label assigned. Otherwise, it finds which nodes are the left and right children, and creates the sub-trees recursively. It also adds the feature of the current node.

### from_infer

To convert the inferred trees from the 3 approaches of InferDT the same method can be used, as the variables describing the found tree do not conflict between the three versions.

The procedure is the following: first, a list is created with enough space to hold the decision nodes and the leafs, according to the depth received as a parameter. The list will hold two values for each element: a boolean that represents if the node is a leaf, and the feature or label assigned to the node (if its a label or a feature is determined by the boolean). This list is created in two steps:
First, the variable $F_{i,f}$ is inspected to determine if the node $i$ has the feature $f$. If so, the element $i$[6] is set with the tuple $(False, f)$.
Then, the variable $C_{i,c}$ is inspected to determine if the leaf $i$ has the class $c$. If so, the element in the list $2^{depth} + i$ is set to $(True, c)$. This indexing is justified by the fact that if a perfect binary tree is traversed using the Breadth First Search (BFS) order, the leaves will be after all the decision nodes (which cardinality is $2^{depth}$). Note that not all the decision nodes will exist when using the two improvements `InferMinimalNodes` and `IterativeInfer`, but later the tree will be pruned.

Once the entire list is filled with one of the three possible values: (1) $(True, label)$ representing a leaf, (2) $(False, feature)$ representing a node, or (3) $(False, -1)$ (the default value) that represents a node that should be pruned; the `_inferred_list_to_tree` starts creating the tree recursively. Starting with the node 1, each element is inspected and three cases can happen:

1. The node should be pruned: it simply returns $None$.
2. The node is a leaf: create the `DecisionTree` with the class assigned to the leaf.
3. Otherwise: find the left and right sub-trees, following the property of the BFS ordering (left child is found at $i \times 2$ and the right child at $i \times 2 + 1$). If one of the sub-trees is found to be

[6]The first element will never be used as the list starts at 0 but the nodes start at 1. The implementation assumes all the list indexed at 1 to ease the calculations of the left and right child.

*None* (pruned) the other child's sub-tree will be returned as being the sub-tree for this node[7]. Otherwise, create the `DecisionTree` is created with the left and right children, and the feature for this node.

### 4.2.6 Data parsing

This component is mainly a helper tool. As most of the models expect binary features, and a binary class, the package also provides a way to convert an entire dataset to this format. It performs two basic transformations:

- One hot encoding[34] of the columns (i.e. features) that have that have more values than 2 (non-binary). This transformation creates multiple columns for each unique value $A$, and assigns a value of 1 if the row's value was $A$.
- Mapping of the columns with 2 unique values that are not 0 or 1 to those values.

Those two transformations allow the algorithms to assume the data has a value of 1 (0) corresponding to the *True* (*False*) value. As the transformation is automatic, the class used to encode the data (shown in listing 6) also offers the method `inverse_transform(data)` that converts the transformed data to the original one. Note that currently the library expects the label to be the last column of the dataset. In future iterations of this project this should be deprecated in favor of using a specific column identified by its name.

```
class EncoderPipeline:
    def __init__(self):
        self.columns_mappings = dict()
        self.columns_order = None

    def transform(self, data: pd.DataFrame) -> pd.DataFrame:
        transformed = data.copy()  # Do not modify the dataframe
           received

        self.columns_order = data.columns
        dummies = list()
        for col in transformed.columns:
            if len(transformed[col].unique()) > 2:
                dummies.append(col)
```

[7]Note that if both sub-trees are to be pruned, this node will also return *None*, thus being pruned in the parent.

```
                transformed[col] = transformed[col].astype(str)
            elif set(transformed[col].unique()) != {0, 1}:
                # Elif as the dummies function will use 0 and 1
                mapping, inverse_mapping =
                   _convert_to_binary_vals(transformed[col])
                self.columns_mappings[col] = (mapping,
                   inverse_mapping)
        return pd.get_dummies(transformed, columns=dummies)

    def inverse_transform(self, data: pd.DataFrame) -> pd.DataFrame:
        inverse = _reverse_dummy(data)
        for column, (mapping, inverse_mapping) in
           self.columns_mappings.items():
            inverse[column].replace(inverse_mapping, inplace=True)

        return inverse[self.columns_order]
```

Listing 6: EncoderPipeline class. The auxiliar methods are not added for the sake of brevity

## 4.3 Dependencies

In this section the dependencies of this package are listed, as well as how and when they are used.

- **Minizinc (0.2.3)** Provides all the functionality to solve the models using the optimization solvers.
- **Pandas (1.0.3)** This package is used during the load of the datasets. It is used mainly for data manipulating such as the one hot encoding performed during the data loading (see 4.2.6) as well as the mapping of the values.
- **Scikit-learn (0.23.0)** This library provides both the model used to create decision trees using the greedy approach, but also it is used in the cross validation utility, as it provides a convenient tool called `KFold`, used to split the data into different folds.
- **Numpy (1.18.4)** Although the data load is performed using **Pandas**, the algorithms work with multidimensional arrays that represent the observations, where each row is an observation and each column is a feature (or the label). **Numpy** is a framework that allows to work with those multidimensional arrays by rows and columns easier than using pure **Python** nested lists.

## 4.4 Installation and usage

Assuming the provided python package is called decisiontrees.whl, and located in the directory `WHEEL_DIR`

```
$ python -m venv venv
$ source venv/bin/activate
$ pip install ${WHEEL_DIR}/decisiontrees.whl
```

An example of the usage is shown in the listing 7. Running this file, will result in creating the decision tree for the dataset named *"weather.csv"*, shown in the listing 8. This file is one benchmark file from the paper [7].

```
>>> from decisiontrees.models import PerfectOptimalModel
>>> from decisiontrees import data_parsing
>>>
>>> dataset = "weather.csv"
>>> encoder, data = data_parsing.load_data(dataset)
>>>
>>> timeout = 600
>>> seed = 42
>>>
>>> model = PerfectOptimalModel(timeout, seed, 15)
>>> tree, accuracy = model.find_tree(data.to_numpy())
[......]
Time: 3.24725341796875 seconds
>>> print(tree.size())
5
>>> print(tree.get_accuracy(data.to_numpy()))
1.0
>>> print(tree.print_tree())
- Split by feature 9
  | - Split by feature 2
  |   | - Leaf has class (+)
  |   | - Leaf has class (-)
  | - Leaf has class (-)
```

Listing 7: Example of how to use the package

```
outlook:sunny,outlook:rain,outlook:overcast, temp:mild, temp:hot,
   temp: cool, humidity, windy, classes
0,0,1,0,1,0, high, false, +
0,1,0,1,0,0, high, false, +
0,1,0,0,0,1, normal, false, +
0,0,1,0,0,1, normal, true, +
1,0,0,0,0,1, normal, false, +
0,1,0,1,0,0, normal, false, +
1,0,0,1,0,0, normal, true, +
0,0,1,1,0,0, high, true, +
0,0,1,0,1,0, normal, false, +
0,1,0,1,0,0, high, true, -
1,0,0,0,1,0, high, false, -
1,0,0,0,1,0, high, true, -
0,1,0,0,0,1, normal, true, -
1,0,0,1,0,0, high, false, -
```

Listing 8: Weather dataset

# Chapter 5

# Experimental results

This chapter presents the experimental results performed in this project, according to the objective 4 (see chapter 1). The benchmarking of the different approaches will allow to assess whether the project is a suitable tool for finding optimal decision trees or not.

The datasets are divided in 3 classes: (1) small, (2) reduced, and (3) big. They are presented in the table 3. Note that for each "big" dataset, the "reduced" counterpart also exists. Those reduced datasets use a subset of the features for each observation.

The datasets selected are the ones used in [8], the paper that presents the first approach implemented; although in this paper it is explained that those datasets originally appear in [7], whose author kindly shared with us those datasets.

The experiments were executed on the computation cluster managed by LOG. The computation cluster used is (for reproducibility purposes) composed by 12 nodes with two Intel Xeon 4110 scalable and 96 GiB of main memory each. The problems were launched to the computation cluster with a timeout of 4 hours for each solving process.

Each experiment was solved using the 5 different approaches implemented: (1) inferring the minimal depth, (2) inferring the minimal nodes used, (3) iterative inferring, (4) perfect optimal decision trees (using **Minizinc**), and (5) perfect optimal decision trees (using a pure SAT implementation). The four first approaches use **Minizinc**, and the configured solver for those experiments is **Gurobi 8.1.1**, while the pure SAT approach uses **Glucose 4.0**[30].

For those experiments, both the solving time and the size of the tree is shown. Also, in the infer minimal depth experiment the depth column is added.

The results for the "small" datasets are shown in the table 4, and the ones for the "reduced" dataset are shown in the table 5. In some rows, the label "OOT" appears in the time column. This is due to

| Set class | Dataset name |
|---|---|
| Small | Mouse |
| Small | Weather |
| Small | Irish |
| Small | Corral |
| Small | Mux6 |

| Set class | Dataset name |
|---|---|
| Reduced | Appendicitis-un |
| Reduced | Australian-un |
| Reduced | Backache-un |
| Reduced | Cancer-un |
| Reduced | Car-un |
| Reduced | Cleve-un |
| Reduced | Colic-un |
| Reduced | Haberman-un |
| Reduced | Heart-statlog-un |
| Reduced | Hepatitis-un |
| Reduced | HouseVotes-un |
| Reduced | Hungarian-un |
| Reduced | Promoters-un |
| Reduced | Shuttle-un |
| Reduced | Spect-un |
| Reduced | New-thyroid-un |

| Set class | Dataset name |
|---|---|
| Big | Appendicitis |
| Big | Australian |
| Big | Backache |
| Big | Cancer |
| Big | Car |
| Big | Cleve |
| Big | Colic |
| Big | Haberman |
| Big | Heart-statlog |
| Big | Hepatitis |
| Big | HouseVotes |
| Big | Hungarian |
| Big | Promoters |
| Big | Shuttle |
| Big | Spect |
| Big | New-Thyroid |

Table 3: Dataset list

the instance exceeding the time limit imposed.

| Dataset | Model[1] | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|
| | IMD | | | IN | | II | | POM | | POSM | |
| | size | depth | time (s) | size | time (s) | size | time (s) | size | time (s) | size | time (s) |
| Corral | 31 | 5 | 32.71 | 13 | 48.21 | 13 | 37.84 | 13 | 11.12 | 13 | 1.01 |
| Irish | 15 | 4 | 1123.36 | 7 | 524.23 | 7 | 604.96 | 7 | 421.98 | 7 | 22.79 |
| Wheather | 9 | 3.32 | 2.41 | 9 | 5.2 | 9 | 6.33 | 9 | 11.86 | 9 | 1.06 |
| Mouse | 25 | 4.70 | 127.71 | 15 | 600.17 | 15 | 469.92 | 15 | 1090.52 | 15 | 14.2 |
| Mux6 | 15 | 4 | 10.26 | 15 | 15.56 | 15 | 18.25 | 15 | 326.24 | 15 | 16.43 |

[1] The names of the columns are:

- IMD: InferMinimalDepth
- IN: InferNodes
- II: IterativeInfer
- POM: PerfectOptimalModel
- POSM: PerfectOptimalSatModel

Table 4: Results for the small datasets

The results of the “big” dataset are not presented as no instance was solved given the system restrictions of memory and time.

| Dataset[1] | Model[2] | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|
| | IMD | | | IN | | II | | POM | | POSM | |
| | size | depth | time (s) | size | time (s) | size | time (s) | size | time (s) | size | time (s) |
| Append. | 31 | 5.00 | 473.39 | 25 | 428.2 | 25 | 971.8 | 25 | 9714.45 | 25 | 394.04 |
| Austr. | 27 | 4.81 | 1450.22 | 19 | 1145.33 | 19 | 1228.7 | 19 | 4070.56 | 19 | 723.02 |
| Backache | 63 | 6.00 | 225.4 | 23 | 356.1 | 23 | 2350.7 | 23 | 8125.72 | 23 | 104 |
| Cancer | 117 | 6.88 | 3622.29 | 35 | 4457.11 | | OOT | | OOT | 35 | 2354 |
| Car | 37 | 5.25 | 1938.04 | 19 | 2582.92 | 19 | 902.5 | 19 | 12078 | 19 | 744 |
| Cleve | 31 | 5.00 | 56.79 | 15 | 4461.62 | 15 | 112.2 | 15 | 8546.88 | 15 | 4136 |
| Colic | 25 | 4.70 | 104.34 | 19 | 167.23 | 19 | 125.84 | 19 | 705.67 | 19 | 437.98 |
| Haberman | | | OOT | | OOT | | OOT | | OOT | 25 | 4297.33 |
| Heart. | 61 | 5.95 | 176.11 | 25 | 309.58 | 25 | 356.87 | 25 | 9320.14 | 25 | 427.13 |
| Hepatitis | 7 | 3.00 | 3.68 | 7 | 55.76 | 7 | 52.11 | 7 | 1096.63 | 7 | 62 |
| House. | 63 | 6.00 | 215.93 | 27 | 454.1 | 27 | 467.28 | 27 | 211.35 | 27 | 144.62 |
| Hungarian | 55 | 5.81 | 207.51 | 19 | 306.74 | 19 | 390.73 | 19 | 446.37 | 19 | 501.17 |
| Promoters | 63 | 6.00 | 67.48 | 23 | 240.12 | 23 | 186.3 | 23 | 3732.07 | 23 | 3.65 |
| Shuttle | 15 | 4.00 | 1029.16 | 7 | 10748.64 | 7 | 12568.69 | | OOT | 7 | 700.69 |
| Spect | | | OOT | 23 | 13256.84 | 23 | 7055.65 | 23 | 2147.01 | 23 | 822.09 |
| Thyroid. | 7 | 3.00 | 2.74 | 5 | 16.66 | 5 | 20.48 | 5 | 1612.18 | 5 | 548.69 |

[1] Some names were abbreviated, indicated with the dot at the end of the name

[2] The names of the columns are:

- IMD: InferMinimalDepth
- IN: InferNodes
- II: IterativeInfer
- POM: PerfectOptimalModel
- POSM: PerfectOptimalSatModel

Table 5: Results for the reduced datasets

For comparison, the **Scikit-Learn** module was also used to solve the same datasets. In the table 6 it is shown the size obtained by the greedy approaches, with the optimal known size for each dataset.

| Dataset | Size (known optimal) | time |
|---|---|---|
| Appendicitis-un (reduced) | 25 (25) | 2.81ms |
| Australian-un (reduced) | 25 (19) | 2.86ms |
| Backache-un (reduced) | 23 (23) | 2.48ms |
| Cancer-un (reduced) | 49 (35) | 2.73ms |
| Car-un (reduced) | 25 (19) | 2.85ms |
| Cleve-un (reduced) | 15 (15) | 3.09ms |
| Colic-un (reduced) | 19 (19) | 2.46ms |
| Corral | 27 (13) | 3.10ms |
| Heart-statlog-un (reduced) | 29 (25) | 2.67ms |
| Hepatitis-un (reduced) | 7 (7) | 2.80ms |
| House-votes-84-un (reduced) | 29 (27) | 3.21ms |
| Hungarian-un (reduced) | 21 (19) | 2.69ms |
| Irish | 7 (7) | 3.74ms |
| Wheather | 13 (9) | 2.85ms |
| Mouse | 15 (15) | 5.74ms |
| Mux6 | 47 (15) | 5.47ms |
| Promoters-un (reduced) | 23 (23) | 2.54ms |
| ShuttleM-un (reduced) | 19 (7) | 6.28ms |
| Spect-un (reduced) | 33 (23) | 2.55ms |
| New-thyroid-un (reduced) | 5 (5) | 2.48ms |

Table 6: Decision trees found by **Scikit-Learn**

## 5.1 Results discussion

As we can see from the results, all the approaches managed to find smaller decision trees than the ones found by the greedy approach implemented in **Scikit-Learn** in 11 of the datasets, at the cost of requiring a significantly greater amount of time. This is the expected behaviour, as those approaches obviously try to reduce the time required, but it is not the key feature of those algorithms.
Note that, although some datasets were also solved to the optimal size using the greedy approach, it is worth noticing that it is only in the easy datasets. Once the rows grow in complexity, the sizes differences become larger.

Another important detail to notice is that, while the encoding is the same for the “PerfectOptimalModel” and the “PerfectOptimalSatModel”, the second has an advantage in two aspects: (1) it does not require the step of converting the **Minizinc** encoding to the SAT encoding, and (2) the solver used is specifically designed to solve those problems, while **Gurobi** (used as the backend solver for **Minizinc**) has a performance reduction when the amount of constraints is extremely large (as in those encodings). Thus, the obvious solution would be to repeat the experiments using **Minizinc** against one of the SAT solvers presented in , but unfortunately no interface between **Minizinc** and any of those solvers (including **Glucose**) was found.

The other result to point out is that the first version of the “infer” methods (the one inferring the minimal depth) finds trees that use more nodes that the optimal ones. This is because this first version does not optimize against the tree node count. Though, the versions that inherit from this first version manage to find the optimal trees. In some cases, for example in the car dataset, the iterative method manages to solve the instance much faster than the other versions also, indicating that probably it is using only a subset of the rows to find the optimal tree, thus reducing the required number of constraints sent to the solver.

# Chapter 6

# Conclusions and future work

In this chapter the conclusion of this project is presented, as well as some guidelines on how to improve the project as well as the potential branches of development that could follow this project.

## 6.1 Conclusions

This project presents a new tool that allows to find small decision trees that perform perfectly to the train dataset using a mathematical language to describe the dataset and the relations of the different nodes that compose the tree, without requiring to learn the intrinsic details of the different encodings, thus reducing the entry barrier for some users that do not have the knowledge (nor the time to acquire it) to implement such approaches to find the decision trees.

Using the library provided, one is now able to not only find those trees, but also experiment with different approaches and benchmark them to find the most suitable one for the specific needs of the user.

All the objectives have been achieved, as the library allows to solve instances using the **Minizinc** modeling language from start to end with the different approaches presented in the chapter 2. It implements all the steps required from start to end of the process, including pre-processing the data, encode the dataset and call the solver the amount of times required, and parse the solution once a solution is found (or detect when the solution cannot be found given the time or memory restrictions). This library is also easily extensible. This has been seen during the development, as the different approaches were implemented over the library "skeleton" requiring only small corrections or no changes at all. Those approaches have been evaluated against the benchmarks used in the literature related to the project.

As commented in the previous chapter (see 5.1), the main handicap of this project is the lack of a suitable modern SAT solver for **Minizinc**, reducing the performance of those algorithms to the point where some instances could not be solved in the required time. It is expected that, if an interface is developed to interact with any of the SAT solvers that occupy the highest places in the ranking, the solving process would be faster (although still slower than the greedy approaches, everything has a cost).

On a personal level, this project allowed me to deep dive in the field of describing problems using mathematical language, and discover the full potential of those generic descriptions, or encodings, to leverage the hardest part of the problem solving to bleeding edge solvers with the help of interfaces like **Minizinc**. It also indirectly introduced me to the XAI field, which I find really useful and necessary given the fast advancements that are done every day in the artificial intelligence and optimization fields.

It also served me to increase my **Python** programming skills, to learn some tips and tricks to use a computation cluster as the one used during the benchmarking process, as well as increasing my **bash** skills when developing all the wrappers needed to launch batches of experiments all at once, and parse their results.

## 6.2 Future work

The proposed future work for improving this project in the future iterations is:

- Implement an interface from **Minizinc** to a SAT solver.
- Implement more approaches that find not only perfect decision trees, but also another ones such as [11] or [10] that rely on finding sub-optimal decision trees while still guaranteeing the property of being the smallest one or best one given the restrictions of a maximum depth or a maximum error.
- Implement the pure SAT counterparts of all the algorithms.
- Improve the underlying SAT library used to interact with the **Glucose** solver to reduce its memory usage as well as allowing different solvers to be used.

# Glossary

**BinOCT**

From BINary Optimal Classification Trees, a method proposed in [10]. 17

**C4.5**

Extension of ID3 algorithm, presended in [17]. 14, 15

**C5.0**

Implementation of the C4.5 algorithm[18]. 14

**Gini impurity**

A measure that represents the statistical error rate of labeling a random element of the set. 13, 14

**ID5R**

An incremental algorithm created by Paul E. Utgoff[20]. 15

**NP**

Any problem that cannot be solved in a polynomial time respect the problem size. 14

**NP hard**

Any problem that is at least as hard as the hardest problems in NP. 18, 19

**NP-Complete**

Any problem that can be reduced in a polynomial time to the satisfiability problem (also called the hardest problem in NP) [6]. 4

**overfit**

Said from a machine learning model that performs excellent with the train dataset, but poorly on unseen data (test dataset, future data…). 4, 14

# Acronyms

**API**

Application Programming Interface. 25, 27

**BFS**

A traversing order for a tree-like data structure where the elements are traversed by depth levels, so before traversing any node in a depth $N$, the ordering has traversed all the nodes in the layers $N' < N$. 38

**CART**

Classification and Regression Trees. 13–15, 24, 28

**CHAID**

Chi-square Automatic Interaction Detection. 15

**CNF**

Conjunctive Normal Form. 27

**CP**

Constraint Programming. 4, 16, 21–25

**ID3**

Iterative Dichotomiser 3. 14, 15

**ITI**

Iterative Tree Induction. 15, 16

**LOG**

Logic&Optimization Group from Universitat de Lleida. 5, 25, 43

**LP**

Linear Programming. 4, 19

**SAT**

Boolean satisfiability problem (abbreviated SAT) is the problem of determining if there exist an interpretation that satisfies a boolean formula. 4, 16, 18, 19, 23, 27, 30, 31, 43, 46, 50

**UdL**

Universitat de Lleida. 5, 25

**UPC**

Universitat Politècnica de Catalunya. 5

**XAI**

eXplainable Artificial Intelligence. 3, 35, 50

# Appendices

# Appendix A

# API reference

## A.1 decisiontrees.cnf.CNF

`class decisiontrees.cnf.CNF(`*print_when_added: bool = False*`)`

Class that represents a Conjunctive Normal Formula.

**Parameters** `print_when_added` (*bool*) – Flag that allows verbosity when adding clauses to the formula.

`clauses`

List of clauses that compose this formula.

**Type** List[*OrClause*]

`variables`

Dictionary with the boolean variables that exist in this formula, identified by its encoding.

**Type** Dict[int, *Variable*]

`solve_status`

The current status of the formula. By default it is unsolved.

**Type** *SolveStatus*

`add(`*item: Union[*decisiontrees.cnf.cnf_components.Variable, decisiontrees.cnf.cnf_components.Term, decisiontrees.cnf.cnf_components.OrClause, decisiontrees.cnf.cnf_components.AndExpression*]*`)`

Adds an undetermined boolean item to the formula.

**Parameters** `item` (*Union[*`Variable`*,* `Term`*,* `OrClause`*,* `AndExpression`*]*) – This item will be converted to a corresponding clause and then added to the formula. If the item is an AndExpression (which represents multiples clauses), all of its clauses are added.

`add_clause`(*clause:* decisiontrees.cnf.cnf_components.OrClause)

Adds a clause to the formula.

If the print_when_added flag was set to true it also prints the clause to stdout.

**Parameters** `clause` (`OrClause`) – The clause to be added.

**Raises** `TypeError` – The argument is not an OrClause.

`add_clauses`(**clauses:* decisiontrees.cnf.cnf_components.OrClause)

Adds clauses in batch.

**Parameters** `*clauses` (`OrClause`) – A variable amount of clauses to be added.

`add_var`(*var_name: str*) → *decisiontrees.cnf.cnf_components.Variable*

Creates a new variable with a given name.

The var is created, with the name and the next encoding available. This encoding is an integer used to represent the variable in the .cnf file format.

**Parameters** `var_name` (*str*) – The name of the newly created variable.

**Returns** The variable created with the name and its encoding set.

**Return type** *Variable*

`add_vars`(*prefix: str*, **ranges: Iterable*) → *decisiontrees.cnf.utils.NestedDict*

Creates indexed variables.

Those variables are variables such as $V_{i,j,k}$

**Parameters**

- `prefix` (*str*) – The name of the variable without the indices.
- `*ranges` (*Iterable*) – Variable amount of ranges. Those ranges determine the amount of variables for each index.

**Returns** A nested dictionary that allows to access the variables for each index.

**Return type** *NestedDict*

`get_aux`() → *decisiontrees.cnf.cnf_components.Variable*

Create an auxiliary var.

The auxiliary vars are named by default $aux_i$, where $i$ is its a sequential identifier.

**Returns** The newly created variable.

**Return type** *Variable*

`get_variables`()

Obtain the list of variables that are in this formula.

`is_sat`() → bool

Return whether the formula is known to be satisfiable.

`is_solved()` → bool

Return whether the formula has been solved or not.

`print_header(`*out_file=None*`)`

Print the header.

This header is always expected to be in the first line of the .cnf files.

**Parameters** `out_file` – a file-like object (stream); defaults to the current sys.stdout.

`solve(`*print_output: bool = True*`)`

Solve the formula using a sat solver.

This method will create a temporary directory which will be the working directory for the solver. After dumping the formula in a cnf file, it will call the solver. Once the solver finishes, the status will be updated accordingly and the solution will be retrieved if found.

**Warning:** This is a blocking method, and it might take a **very** long time for it to return, depending on the formula. This is the expected behaviour

`to_cnf(`*filename: str*`)`

Dump the formula into a .cnf file.

Create the new file and dumps the entire formula in the cnf format to this file. It first creates the header with the variables and clauses counts, and then writes all the clauses using the correct encoding for the variables.

It also prints to stdout the size of the file in a human-readable format.

**Parameters** `filename` (*str*) – The name of the file to create.

## A.2 decisiontrees.cnf_components

`class decisiontrees.cnf.cnf_components.Variable(`*name: str, encoding: int*`)`

Represents a single boolean variable.

`name`

The variable name

**Type** str

`encoding`

The integer that represents this variable in a CNF formula.

**Type** int

`value`

If not None, this value represents the assignation of this variable in an satisfiable interpretation of the CNF formula it belongs.

**Type** Optional[bool]

`__add__(`*other: Union[Variable, Term, OrClause, AndExpression]*`)`

Perform the logical operation OR with another element.

`__eq__(`*other*`)`

Return self==value.

`__hash__()`

Return hash(self).

`__init__(`*name: str, encoding: int*`)`

Initialize self. See help(type(self)) for accurate signature.

`__mul__(`*other: Union[Variable, Term, OrClause, AndExpression]*`)`

Perform the logical operation AND with another element.

`__repr__()`

Return repr(self).

`__weakref__`

list of weak references to the object (if defined)

`get_encoding()` → str

String that represents the variable encoded.

`class decisiontrees.cnf.cnf_components.Term(`*var:* decisiontrees.cnf.cnf_components.Variable, *negated: bool = False*`)`

A term is a variable with an specified sign in a CNF formula.

`var`

The variable that composes this term.

**Type** *Variable*

`negated`

True if the variable is negated.

**Type** bool

`__add__`(*other: Union[Variable, Term, OrClause, AndExpression]*)

Perform the logical operation OR with another element.

`__eq__`(*other*)

Return self==value.

`__hash__`()

Return hash(self).

`__init__`(*var:* decisiontrees.cnf.cnf_components.Variable, *negated: bool = False*)

Initialize self. See help(type(self)) for accurate signature.

`__mul__`(*other: Union[Variable, Term, OrClause, AndExpression]*)

Perform the logical operation AND with another element.

`__repr__`()

Return repr(self).

`__weakref__`

list of weak references to the object (if defined)

`get_encoding`() → str

String that represents the term encoded.

This encoding is in the form of: "<sign><variable encoding>". The sign is omitted when it is positive.

`class decisiontrees.cnf.cnf_components.OrClause`(*terms: Iterable[Union[Variable, Term, OrClause, AndExpression]]*)

An or clause represents a sequence of elements joined by OR operations.

`terms`

The sequence of elements. They are represented as a set as:

$$A \vee A \vee B \equiv A \vee B$$

**Type** Set[CnfComponent]

`__add__`(*other: Union[Variable, Term, OrClause, AndExpression]*)

Perform the logical operation OR with another element.

`__eq__`(*other*)
: Return self==value.

`__hash__`()
: Return hash(self).

`__init__`(*terms: Iterable[Union[Variable, Term, OrClause, AndExpression]]*)
: Initialize self. See help(type(self)) for accurate signature.

`__mul__`(*other: Union[Variable, Term, OrClause, AndExpression]*)
: Perform the logical operation AND with another element.

`__repr__`()
: Return repr(self).

`__weakref__`
: list of weak references to the object (if defined)

`class decisiontrees.cnf.cnf_components.AndExpression(`*clauses*`)`

`__add__`(*other: Union[Variable, Term, OrClause, AndExpression]*)
: Perform the logical operation OR with another element.

This method expands the expression using the distributive property of the OR operation to ensure that the CNF is still a valid conjunction of clauses (i.e. any OR operation has no AND operation inside).

`__eq__`(*other*)
: Return self==value.

`__hash__`()
: Return hash(self).

`__init__`(*clauses*)
: Initialize self. See help(type(self)) for accurate signature.

`__mul__`(*other: Union[Variable, Term, OrClause, AndExpression]*)
: Perform the logical operation AND with another element.

`__repr__`()
: Return repr(self).

`__weakref__`
: list of weak references to the object (if defined)

`decisiontrees.cnf.cnf_components.Not(`*item: Union[Variable, Term, OrClause, AndExpression]*`)`

Negate a boolean expression.

**Parameters** `item` (*`CnfComponent`*) – The boolean expression to negate.

**Raises** `TypeError` – If the item's type is not a known class.

`decisiontrees.cnf.cnf_components.or_all`(*terms: Iterable[Union[Variable, Term, OrClause, AndExpression]], with_reification: bool = False, formula: CNF = None*) → Union[*decisiontrees.cnf.cnf_components.OrClause*, *decisiontrees.cnf.cnf_components.AndExpression*]

Creates a disjunction of all the terms. If reification is specified, the disjunction will keep creating auxiliary variables to the formula and the corresponding clause.

---

**Note:** This method can return either a single OR clause, or a conjunction of those. This is due to the expansion done when the disjunction is something like:

$$(A \wedge B) \vee C$$

as this expression will be expanded to:

$$(A \vee C) \wedge (B \vee C)$$

---

**Parameters**

- `terms` (*Iterable [ CnfComponent ]*) – The terms that will form the disjunction.
- `with_reification` (*bool*) – If the conjunction should be created using reification to avoid an explosion of clauses when expanding the OR operations.
- `formula` (`CNF`) – The formula where the reification clauses will be created. If reification is false this formula will not be used, thus it can be None.

`decisiontrees.cnf.cnf_components.and_all`(*clauses: Iterable[Union[Variable, Term, OrClause, AndExpression]]*) → *decisiontrees.cnf.cnf_components.AndExpression*

Creates a conjunction of the different clauses received.

**Parameters** `clauses` (*Iterable [ CnfComponent ]*) – The clauses that conform this conjunction.

`decisiontrees.cnf.cnf_components.at_least_one`(*terms: Iterable[Union[Variable, Term, OrClause, AndExpression]], with_reification: bool = False, formula: CNF = None*) → Union[*decisiontrees.cnf.cnf_components.OrClause*, *decisiontrees.cnf.cnf_components.AndExpression*]

Creates a clause that specifies that at least one of the terms should evaluate to true. It is created using the or_all method.

**See also:**

*`or_all()`*

`decisiontrees.cnf.cnf_components.at_most_one`(*terms: Iterable[Union[Variable, Term, OrClause, AndExpression]]*) → *decisiontrees.cnf.cnf_components.AndExpression*

Creates a set of clauses specifying that at most one of those terms should evaluate to true.

**Parameters** `terms` (*`Iterable[CnfComponent]`*) – The list of components that at most one of them can evaluate to true.

`decisiontrees.cnf.cnf_components.exactly_one`(*terms: Iterable[Union[Variable, Term, OrClause, AndExpression]], with_reification: bool = False, formula: CNF = None*) → *decisiontrees.cnf.cnf_components.AndExpression*

Creates the set of clauses that represents that, from all the terms, only one can evaluate to true.

**See also:**

*`at_least_one()`*, *`at_most_one()`*

`class decisiontrees.cnf.cnf_components.SolveStatus`

Enumeration representing the status of a CNF formula.

## A.3 decisiontrees.cnf.utils

Utilities for the CNF module

`class decisiontrees.cnf.utils.NestedDict`

Dictionaries with arbitrary depth.

This class allows the creation of dictionaries that can accept the addition of a value at an arbitrary depth.

`nested_set_value`(*keys: List[Any]*, *value: Any*)

Sets a value for the dictionary given multiple keys.

**Parameters**

- `keys` (*List [Any ]*) – List of the keys that will identify the value. For example, [“a”, “b”, “c”] corresponds to dict[“a”][“b”][“c”]
- `value` (*Any*) – The value that will be set.

`decisiontrees.cnf.utils.sizeof_fmt`(*num: int*) → str

Converts the size in bytes to a human-readable form.

## A.4 decisiontrees.data_parsing

Utilities for abstracting the load and pre-processing of the datasets.

`class decisiontrees.data_parsing.EncoderPipeline`

Represents the pre-processing performed to a dataset.

`inverse_transform(`*data: pandas.core.frame.DataFrame*`)` → pandas.core.frame.DataFrame

Reverses the transformations that applies this processing step.

**Parameters** `data` (*pd.DataFrame*) – A transformed dataset.

**Returns** The data with the transformation reverted.

**Return type** pd.DataFrame

`transform(`*data: pandas.core.frame.DataFrame*`)` → pandas.core.frame.DataFrame

Applies this processing step to a dataset. The transformations applied are

- Convert the non-binary columns to binary columns using the One Hot Encoding method.
- Map the values in the binary columns to {0, 1}, if the are not already those.

**Parameters** `data` (*pd.DataFrame*) – The dataset that will be processed.

**Returns** The processed dataset.

**Return type** pd.DataFrame

`decisiontrees.data_parsing.load_data(`*csv_file: str*`)` → Tuple[*decisiontrees.data_parsing.EncoderPipeline*, pandas.core.frame.DataFrame]

Loads a dataset from a csv file, and performs the pre-processing step to this data.

**Parameters** `csv_file` (*str*) – The file to be loaded.

**Returns** Returns both the encoder pipeline used to process this dataset, and the dataset processed.

**Return type** Tuple[*EncoderPipeline*, pd.DataFrame]

## A.5 decisiontrees.trees

This module contains the logic related to create DecisionTrees from the results of the solvers.

`class decisiontrees.trees.DecisionTree`(*feat: Optional[int] = None, right: Optional[DecisionTree] = None, left: Optional[DecisionTree] = None, leaf: bool = False, label: Optional[int] = None*)

Class that represents a decision tree.

Technically, this class represents a node for the tree, with "pointers" to its childs. But as each node can be considered as a subtree, it is correct to represent the entire tree with the root node.

**Parameters**

- `feat` (*Optional [int]*) – If this node is a decision node, this parameter sets the current feature used to split.
- `right` (*Optional [DecisionTree]*) – The pointer to the right descendant. It represents the branch where the current feature takes a value of 1.
- `left` (*Optional [DecisionTree]*) – The pointer to the left descendant. It represents the branch where the current feature takes a value of 0.
- `leaf` (*bool*) – True if this node is a leaf, False if it is a decision node.
- `label` (*Optional [int]*) – If this node is a leaf, this parameter sets the label assigned to the observations that reach this leaf.

`classify`(*row: numpy.array*) → int

Assigns a label to an observation.

**Parameters** `row` (*np.array*) – A 1D array that represents the observation.

**Returns** The predicted label for the observation.

**Return type** int

`static from_array`(*str_array: List[str]*) → *decisiontrees.trees.DecisionTree*

Iterates a BFS representation of the tree to generate the corresponding decision tree.

**Parameters** `str_array` (*List [str]*) – The array that represents the tree. Each element is in the form of <node_id>/<content>

**See also:**

*to_array()*

**Returns** The decision tree represented by the array.

**Return type** *DecisionTree*

`get_accuracy`(*test: numpy.array*) → float

Calculates the accuracy of the current decision tree given a set of observations.

**Parameters** `test` (*np.array*) – 2D array where each row is an observation to be classified. Each row is assumed to contain the true label at the last element.

**Returns**

The accuracy of the decision tree, calculated as:

$$1 - \frac{misclassified\ rows}{total\ rows}$$

**Return type** float

`static load`(*filename*) → *decisiontrees.trees.DecisionTree*

Recover a decision tree from a file with its BFS representation.

`print_tree()`

Prints the tree with in a visual structure.

`save`(*filename: str*)

Store the tree in a file using the BFS representation

`size()` → int

Count the number of nodes hanging of this node.

`to_array()` → List[str]

Create an array representing this tree in BFS order. Each node is represented as a string in the form of <node_id>/<content>, where content is either the feature (label) used in this decision node (leaf).

**Returns** The tree in BFS order.

**Return type** List[str]

`class decisiontrees.trees.TreeFactory`

Class that implements the factory pattern to create decision trees from the results of each encoding.

`static from_infer`(*result: minizinc.result.Result, depth: int*) → *decisiontrees.trees.DecisionTree*

Creates the decision tree for the InferDT models.

**Parameters**

- `result` (*minizinc.Result*) – The result returned by the solver, containing the solution.
- `depth` (*int*) – The maximum depth of the tree inferred.

**Returns** The found decision tree.

**Return type** *DecisionTree*

`static from_perfect_optimal(`*result: minizinc.result.Result*, *node=0*`)` → *decisiontrees.trees.DecisionTree*

Creates the decision tree for the perfect optimal decision trees model (minizinc version).

**Parameters**

- `result` (*minizinc.Result*) – The result returned by the solver, containing the solution.
- `node` (*int*) – The index of the node that is being created. Used to create the tree by recursion.

**Returns** The found decision tree.

**Return type** *DecisionTree*

`static from_perfect_optimal_sat(`*model: SatDecisionTree*, *node: int = 1*`)` → *decisiontrees.trees.DecisionTree*

Creates the decision tree for the perfect optimal decision trees model (pure sat version).

**Parameters**

- `model` (*SatDecisionTree*) – The SatDecision tree found during the solving process.
- `node` (*int*) – The index of the node that is being created. Used to create the tree by recursion.

**Returns** The found decision tree.

**Return type** *DecisionTree*

`static from_sklearn(`*estimator: sklearn.tree._classes.DecisionTreeClassifier*, *node=0*`)` → *decisiontrees.trees.DecisionTree*

Creates the decision tree for the Scikit-Learn model.

**Parameters**

- `estimator` (*DecisionTreeClassifier*) – The decision tree object found by Scikit-Learn.
- `node` (*int*) – The index of the node that is being created. Used to create the tree by recursion.

**Returns** The found decision tree.

**Return type** *DecisionTree*

## A.6 decisiontrees.cross_validate

Compute cross-validation over a dataset.

`class decisiontrees.cross_validate.CrossValidationResult`

Store the results for each cross-validation step.

`add_result`(*train_accuracy: float*, *validation_accuracy: float*, *size: int*)

Add the result of a single cross-validation step in the results list.

**Parameters**

- `train_accuracy` (*float*) – The accuracy of the training set.
- `validation_accuracy` (*float*) – The accuracy of the validation set.
- `size` (*int*) – The node count of the tree found.

`get_results`(*metric: Callable[[List[Union[int, float]]], float]*) → Tuple[float, float, float, int]

Calculate the result of the cross validation according to an specific metric.

**Parameters** `metric` (*Callable[[List[Number]], float]*) – The metric function. It receives a list of numbers and applies the metric over all those values together. An example of this metric is np.mean.

**Returns** A tuple containing the summary of the cross-validation with the metric applied. This tuple contains (in order) the train accuracy, test accuracy, the size (all of those with the metric applied), and the number of steps that were solved during this cross-validation.

**Return type** Tuple[float, float, float, float]

`decisiontrees.cross_validate.cross_validate`(*model_class: type*, *n_folds: int*, *data: numpy.array*, *time_limit: int*, *seed: int*, **args*, ***kwargs*) → *decisiontrees.cross_validate.CrossValidationResult*

Performs the cross-validation over a dataset using the specified number of folds.

**Parameters**

- `model_class` (*type*) – The type of DecisionModel that will be used during the cross validation.
- `n_folds` (*int*) – The number of folds that will be used.
- `data` (*np.array*) – The dataset that will be used to obtain the training and validation sets.
- `time_limit` (*int*) – The time limit (in seconds) that will be requested the model when solving each step of the cross-validation.

- `seed` (*int*) – The random seed that will be used during both the split of the train-validate sets and during the solving process.
- `*args` – Additional arguments for the DecisionModel class.
- `**kwargs` – Additional named arguments for the DecisionModel class.

**Returns** Object containing the results of each step of the cross-validation.

**Return type** *CrossValidationResult*

## A.7 decisiontrees.models

`class decisiontrees.models.SklearnModel(`*time_limit: int*, *seed: int*`)`

Find a decision tree using a Scikit-Learn classifier.

`find_tree(`*data: numpy.array*`)` → Tuple[*decisiontrees.trees.DecisionTree*, float]

Solve the model for a set of observations, obtaining the decision tree if found.

**Warning:** To be implemented by the sub-classes.

**Parameters** `data` (*np.array*) – A bi-dimensional array containing one row per observation. Those observations are used to train the model to obtain the decision tree.

**Returns** A tuple containing the decision tree found and the accuracy of the model in the training set.

**Return type** Tuple[*DecisionTree*, float]

`stop_tree_find()`

Notify the process of finding a tree to stop.

`class decisiontrees.models.PerfectOptimalSatModel(`*time_limit: int*, *seed: int*, *upper_bound: int*`)`

Find the perfect decision tree using a pure SAT implementation.

`upper_bound`

The maximum nodes that the tree can have.

**Type** int

`find_tree(`*data: numpy.array*`)` → Tuple[*decisiontrees.trees.DecisionTree*, float]

Solve the model for a set of observations, obtaining the decision tree if found.

**Warning:** To be implemented by the sub-classes.

**Parameters** `data` (*np.array*) – A bi-dimensional array containing one row per observation. Those observations are used to train the model to obtain the decision tree.

**Returns** A tuple containing the decision tree found and the accuracy of the model in the training set.

**Return type** Tuple[*DecisionTree*, float] `stop_tree_find()`

Notify the process of finding a tree to stop.

`class decisiontrees.models.PerfectOptimalModel(`*time_limit: int*, *seed: int*, *upper_bound: int*`)`

Find the perfect decision tree using a Minizinc model.

`upper_bound`

The maximum nodes that the tree can have.

**Type** int

`find_tree(`*data: numpy.array*`)` → Tuple[*decisiontrees.trees.DecisionTree*, float]

Solve the model for a set of observations, obtaining the decision tree if found.

**Warning:** To be implemented by the sub-classes.

**Parameters** `data` (*np.array*) – A bi-dimensional array containing one row per observation. Those observations are used to train the model to obtain the decision tree.

**Returns** A tuple containing the decision tree found and the accuracy of the model in the training set.

**Return type** Tuple[*DecisionTree*, float]

`class decisiontrees.models.PerfectMinimalDecisionTreeMaxSat(`*time_limit: int*, *seed: int*, *max_nodes: int*`)`

`find_tree(`*data: numpy.array*`)` → Tuple[*decisiontrees.trees.DecisionTree*, float]

Solve the model for a set of observations, obtaining the decision tree if found.

**Warning:** To be implemented by the sub-classes.

**Parameters** `data` (*np.array*) – A bi-dimensional array containing one row per observation. Those observations are used to train the model to obtain the decision tree.

**Returns** A tuple containing the decision tree found and the accuracy of the model in the training set.

**Return type** Tuple[*DecisionTree*, float]

`class decisiontrees.models.DecisionTreeWithError(`*time_limit: int*, *seed: int*, *max_nodes: int*, *max_error: float*`)`

`find_tree`(*data: numpy.array*) → Tuple[*decisiontrees.trees.DecisionTree*, float]
Solve the model for a set of observations, obtaining the decision tree if found.

**Warning:** To be implemented by the sub-classes.

**Parameters** `data` (*np.array*) – A bi-dimensional array containing one row per observation. Those observations are used to train the model to obtain the decision tree.

**Returns** A tuple containing the decision tree found and the accuracy of the model in the training set.

**Return type** Tuple[*DecisionTree*, float]

`class decisiontrees.models.PenalizeDepthDT`(*time_limit: int*, *seed: int*, *max_nodes: int*)

# Python Module Index

## d

decisiontrees.cnf.utils, 65

decisiontrees.cross_validate, 70

decisiontrees.data_parsing, 66

decisiontrees.models, 72

decisiontrees.trees, 67

# Appendix B

# Encodings implementations

## B.1 SAT APPROACH

```python
# Constraint (1)
self.cnf.add(Not(v[1]))

# Constraint (2)
for i in range(1, self.n + 1):
    lr = self._LR(i)
    if not lr: # No possible childs -> node i is leaf
        self.cnf.add(v[i])
        continue
    for j in lr:
        self.cnf.add(Implies(v[i], Not(l[i][j])))

# Constraint (3)
for i in range(1, self.n + 1):
    lr = self._LR(i)
    if not lr:
        continue
    for j in lr:
        self.cnf.add(DoubleImplies(l[i][j], r[i][j+1]))

# Constraint (4)
for i in range(1, self.n + 1):
    rhs_terms = [l[i][j] for j in self._LR(i)]
    if rhs_terms:
        rhs = exactly_one(rhs_terms)
        self.cnf.add(Implies(Not(v[i]), rhs))

# Constraint (5)
for i in range(1, self.n + 1):
    lr = self._LR(i)
    rr = self._RR(i)
    if lr:
        for j in self._LR(i):
            self.cnf.add(DoubleImplies(p[j][i], l[i][j]))
    if rr:
        for j in self._RR(i):
            self.cnf.add(DoubleImplies(p[j][i], r[i][j]))

# Constraint (6)
for j in range(2, self.n + 1):
    terms = [p[j][i] for i in self._possible_parents(j)]
    self.cnf.add(exactly_one(terms))


self.cnf.add(and_all(
    [Not(d0[k][1]) for k in range(self.features)]
))

for j in range(2, self.n + 1):
    parents = self._possible_parents(j)
    for k in range(self.features):
        # d0 <-> V((p ^ d0) v (a ^ r))
        lhs = d0[k][j]
        rhs_elements = list()
        for i in parents:
            p1 = self.p[j][i] * d0[k][i]
            try:
                p2 = a[k][i] * self.r[i][j]
                rhs_elements.append(p1 + p2)
            except KeyError:
                rhs_elements.append(p1)

        if rhs_elements:
            rhs = or_all(rhs_elements, with_reification=True, formula=self.c
            self.cnf.add(DoubleImplies(lhs, rhs))

# Constraint (8)
self.cnf.add(and_all(
    [Not(self.d1[k][1]) for k in range(self.features)]
))

for j in range(2, self.n + 1):
    parents = self._possible_parents(j)
    for k in range(self.features):
        # d1 <-> V((p ^ d1) v (a ^ l))
        lhs = d1[k][j]
        rhs_elements = list()
        for i in parents:
            p1 = self.p[j][i] * d1[k][i]
            try:
                p2 = a[k][i] * self.l[i][j]
                rhs_elements.append(p1 + p2)
            except KeyError:
                rhs_elements.append(p1)
        if rhs_elements:
            rhs = or_all(rhs_elements, with_reification=True, formula=self.c
            self.cnf.add(DoubleImplies(lhs, rhs))

# Constraint (9)
# j = 1 (no parents, 9.1 not applicable)
for k in range(self.features):
    # 9.2 (without the OR for parents)
    self.cnf.add(DoubleImplies(u[k][1], a[k][1]))

# j > 1 (with parents)
for k, j in itertools.product(range(self.features), range(2, self.n + 1)):
    parents = self._possible_parents(j)
    # 9.1
    self.cnf.add(and_all(
        [Implies(u[k][i] * self.p[j][i], Not(a[k][j])) for i in parents]
    ))
    # 9.2
    lhs = u[k][j]
    rhs = a[k][j] + or_all([(u[k][i] * self.p[j][i]) for i in parents],
                            with_reification=True,
                            formula=self.cnf)
    self.cnf.add(DoubleImplies(lhs, rhs))

# Constraint (10)
for j in range(1, self.n + 1):
    lhs = Not(self.v[j])
    rhs = exactly_one(self.a[k][j] for k in range(self.features))
    self.cnf.add(Implies(lhs, rhs))

# Constraint (11)
for j in range(1, self.n + 1):
    lhs = self.v[j]
    # no feature is used == not(a_r0_j) ^ not(a_r1_j) ^ ....
    rhs = and_all(Not(a[k][j]) for k in range(self.features))
    self.cnf.add(Implies(lhs, rhs))

# Constraint (12) & (13)
for observation in self.data:
    if observation[-1] == 0:
        lhs = lambda j: self.v[j] * c[j]
    else:
        lhs = lambda j: self.v[j] * Not(c[j])

    for j in range(1, self.n + 1):
        rhs = or_all(d[j] for d in self._get_discriminators(observation))
        self.cnf.add(Implies(lhs(j), rhs))
```

## B.2 MINIZINC APPROACH

```
% Given a max nodes "N", a dataset "data" with "n_feats" features
% determine if we can find a DT with 100% accuracy using "N" nodes

int: N;
int: n_feats;
int: rows = length(data) div (n_feats + 1);
array[int, int] of int: data;

test even(int: n) = n mod 2 = 0;
test odd(int: n) = n mod 2 = 1;

function var set of int: LR(int: i) = {j | j in (i+1)..(min ({2*i,
   N-1})) where even(j)};
function var set of int: RR(int: i) = {j | j in (i+2)..(min ({2*i +
   1, N})) where odd(j)};
function var set of int: possible_parents(int: j) = { j | j in (j div
   2)..(j-1)};

predicate atmostone(array[int] of var bool: x) =
    forall (i,j in index_set(x) where i < j)(
    (not x[i] \/ not x[j])
    );
predicate exactlyone(array[int] of var bool: x) = atmostone(x) /\
   exists(x);



array[1..N] of var bool: v;
array[1..N, 1..N] of var bool: l;
array[1..N, 1..N] of var bool: r;
array[2..N, 1..N] of var bool: p;
array[1..n_feats, 1..N] of var bool: a;
array[1..n_feats, 1..N] of var bool: u;
array[1..n_feats, 1..N] of var bool: d0;
array[1..n_feats, 1..N] of var bool: d1;
array[1..N] of var bool: c;
```

```

% ===================Tree shape constraints===================

% For all the non-possible left and right childs, set the vars to
   false
constraint forall (i, j in 1..N) (not (j in LR(i)) -> (not l[i,j]));
constraint forall (i, j in 1..N) (not (j in RR(i)) -> (not r[i,j]));

% For all non possible parents, set the vars to false
constraint forall (i in 1..N, j in 2..N) (not (j in LR(i) \/ j in
   RR(i)) -> (not p[j,i]));


% Constraint 1
constraint not v[1]; % Node 1 isn't a leaf

% Constraint 2
constraint forall (i in 1..N) (forall (j in LR(i)) (v[i] -> not
   l[i,j]));

% Constraint 3
constraint forall (i in 1..N) (forall (j in LR(i)) (l[i,j] <->
   r[i,j+1]));

% Constraint 4
constraint forall (i in 1..N) ((not v[i]) -> sum (j in LR(i))
   (l[i,j]) = 1);

% Constraint 5
constraint forall (i in 1..N)
                    (forall (j in LR(i))
                            (p[j,i] <-> l[i,j])
                    );
constraint forall (i in 1..N) (forall (j in RR(i)) (p[j,i] <->
   r[i,j]));

% Constraint 6
constraint forall (j in 2..N) ((sum (i in possible_parents(j))
   (p[j,i])) = 1);
```

```

% ==============================================================

% ==================Decision tree constraints=======================

% Constraint 7
constraint forall (k in 1..n_feats) (not d0[k,1]);
constraint forall (k in 1..n_feats, j in 2..N) (
    d0[k, j] <-> exists (i in possible_parents(j)) ( (p[j, i] /\
       d0[k, i]) \/ (a[k, i] /\ r[i, j]))
);

% Constraint 8
constraint forall (k in 1..n_feats) (not d1[k,1]);
constraint forall (k in 1..n_feats, j in 2..N) (
    d1[k, j] <-> exists (i in possible_parents(j)) ( (p[j, i] /\
       d1[k, i]) \/ (a[k, i] /\ l[i, j]))
);

% Constraint 9
constraint forall (k in 1..n_feats, j in 2..N) (
    forall (i in possible_parents(j)) (u[k,i] /\ p[j,i] -> not a[k,
       j])
);
constraint forall (k in 1..n_feats, j in 2..N) (
    u[k,j] <-> a[k,j] \/ ( forall (i in possible_parents(j)) (u[k,i]
       /\ p[j,i]) )
);

% Constraint 10
constraint forall (j in 1..N) (
    (not v[j]) -> exactlyone (k in 1..n_feats) (a[k,j])
);

% Constraint 11
constraint forall (j in 1..N) (
    v[j] -> not exists (k in 1..n_feats) (a[k,j])
);

```

```
% Constraint 12
constraint forall (row in 1..rows where data[row, n_feats + 1] = 1) (
    forall (j in 1..N) (
    v[j] /\ not c[j] -> exists (k in 1..n_feats) (if data[row, k] = 0
       then d0[k,j] else d1[k,j] endif)
    )
);

% Constraint 13
constraint forall (row in 1..rows where data[row, n_feats + 1] = 0) (
    forall (j in 1..N) (
    v[j] /\ c[j] -> exists (k in 1..n_feats) (if data[row, k] = 0
       then d0[k,j] else d1[k,j] endif)
    )
);

% ==================================================================

solve satisfy;
```

## B.3 InferDT

### B.3.1 Inferring tree with a maximum depth

```
# Create the variables
self._current_model = minizinc.Model()
self._current_model.add_string("int: n_feats;\n")
self._current_model.add_string("array[int, int] of int: data;\n")

n_nodes = 2**depth - 1
n_leaves = 2**(depth + 1)
n_rows = data.shape[0]

# variables
vars = f"array[0..{n_rows - 1}, 0..{depth - 1}] of var bool: X;\n" + \
        f"array[1..{n_nodes}, 0..(n_feats - 1)] of var bool: F;\n" + \
        f"array[0..{n_leaves - 1}, 0..(n_feats - 1)] of var bool:
           C;\n"
self._current_model.add_string(vars)

# Constraint 1 and 2
self._add_constraint(
    f"forall (i in 1..{n_nodes}) " +
    f"((sum (j in 0..(n_feats - 1)) (F[i, j])) = 1)"
)

# Constraint 3 and 4
def _generate_feature_constraints(self,
                                  row: np.array,
                                  i: int,
                                  depth: int,
                                  clause: str = "",
                                  q: int = 1,
                                  lvl: int = 0):
    if lvl == depth:
        return

    for f in range(0, self._n_feats):
        if row[f] == 1:
            new_constraint = _or_join([clause, f"X[{i},{lvl}]",
               f"not(F[{q}, {f}])"])
            self._add_constraint(new_constraint)

    self._generate_feature_constraints(
        row, i, depth, _or_join([clause, f"X[{i},{lvl}]"]),
        q * 2, lvl + 1
    )

    for f in range(0, self._n_feats):
        if row[f] == 0:
            new_constraint = _or_join([clause, f"not(X[{i},{lvl}])",
               f"not(F[{q}, {f}])"])
            self._add_constraint(new_constraint)

    self._generate_feature_constraints(
            row, i, depth, _or_join([clause, f"not(X[{i},{lvl}])"]),
            q * 2 + 1, lvl + 1
        )

    # Constraint 5 and 6
    def _generate_class_constraints(self,
                                    row: np.array,
                                    i: int,
                                    depth: int,
                                    clause: str = "",
                                    q: int = 0,
                                    lvl: int = 0):
        if lvl == depth:
            label = row[-1]
            label_idx = self._labels.index(label)

            constraint = _or_join([clause, f"C[{q}, {label_idx}]"])
            self._add_constraint(constraint)

            for a in range(0, len(self._labels)):
                if a == label_idx:
                    continue
                constraint = _or_join([clause, f"not(C[{q}, {a}])"])
                self._add_constraint(constraint)
            return

        self._generate_class_constraints(
            row, i, depth, _or_join([clause, f"X[{i}, {lvl}]"]),
            q * 2, lvl + 1
        )
        self._generate_class_constraints(
            row, i, depth, _or_join([clause, f"not(X[{i},{lvl}])"]),
            q * 2 + 1, lvl + 1
        )
```

### B.3.2 Inferring a tree with a maximum number of nodes

```
# New vars
self._current_model.add_string(
    f"array[0..{n_leaves - 1}] of var bool: U;\n" +
    f"array[0..{n_leaves}, 0..{n_leaves}] of var bool: H;\n"
)

# Constraint 7
self._add_constraint(
    f"forall (i in 0..{n_leaves - 1}, a in 0..{len(self._labels) -
        1}) "
    "(not(C[i, a]) \\/ U[i])"
)

# Constraint 8
self._add_constraint(
    f"forall (i in 0..{n_leaves - 1}, j in 0..i) "
    "(not(H[i,j]) \\/ H[i+1,j])"
)

# Constraint 9
self._add_constraint(
    f"forall (i in 0..{n_leaves - 1}, j in 0..i) "
    "(not(U[i]) \\/ not(H[i,j]) \\/ H[i+1,j+1])"
)

# Constraint 10
self._add_constraint("H[0,0]")
self._add_constraint(f"not(H[{2**(depth + 1)}, {self.current_nodes +
    1}])")
```

# Bibliography

[1] Murphy KP. Machine Learning: A Probabilistic Perspective. The MIT Press; 2012.

[2] Segaran T. Programming Collective Intelligence. O'Reilly Media, Inc.; 2008.

[3] Arrieta AB, Díaz-Rodríguez N, Del Ser J, Bennetot A, Tabik S, Barbado A, et al. Explainable Artificial Intelligence (XAI): Concepts, Taxonomies, Opportunities and Challenges toward Responsible AI. arXiv:191010045 [cs]. 2019 Dec;.

[4] Bertsimas D, Dunn J. Optimal Classification Trees. Machine Learning. 2017 Jul;106(7):1039–1082.

[5] Hyafil L, Rivest RL. Constructing Optimal Binary Decision Trees Is NP-Complete. Information Processing Letters. 1976;5(1):15–17.

[6] Garey MR, Johnson DS. Computers and Intractability; A Guide to the Theory of NP-Completeness. USA: W. H. Freeman & Co.; 1990.

[7] Bessiere C, Hebrard E, O'Sullivan B. Minimising Decision Tree Size as Combinatorial Optimisation. In: Principles and Practice of Constraint Programming - CP 2009. Berlin, Heidelberg: Springer Berlin Heidelberg; 2009. p. 173–187.

[8] Narodytska N, Ignatiev A, Pereira F, Marques-Silva J. Learning Optimal Decision Trees with SAT. In: Proceedings of the Twenty-Seventh International Joint Conference on Artificial Intelligence. Stockholm, Sweden: International Joint Conferences on Artificial Intelligence Organization; 2018. p. 1362–1368.

[9] Avellaneda F. Efficient Inference of Optimal Decision Trees. In: The Thirty-Fourth AAAI Conference on Artificial Intelligence (AAAI-20). New York; 2020. p. 8.

[10] Verwer S, Zhang Y. Learning Optimal Classification Trees Using a Binary Linear Program Formulation. Proceedings of the AAAI Conference on Artificial Intelligence. 2019 Jul;33:1625–1632.

[11] Verhaeghe H, Nijssen S, Pesant G, Quimper CG, Schaus P. Learning Optimal Decision Trees Using Constraint Programming. In: The 25th International Conference on Principles and Practice of Constraint Programming (CP2019); 2019. p. 17.

[12] Morgan JN, Sonquist JA. Problems in the Analysis of Survey Data, and a Proposal; 1963. .

[13] Loh WY. Fifty Years of Classification and Regression Trees 1; 2014. /paper/Fifty-Years-of-Classification-and-Regression-Trees-Loh/f1c3683cacc3dc7898f3603753af87565f8ad677.

[14] Breiman L, Friedman J, Stone CJ, Olshen RA. Classification and Regression Trees. Taylor and Francis; 1984.

[15] Quinlan JR. Induction of Decision Trees. Machine Learning. 1986 Mar;1(1):81–106.

[16] Shannon CE. A Mathematical Theory of Communication. Bell System Technical Journal. 1948;27(3):379–423. _eprint: https://onlinelibrary.wiley.com/doi/pdf/10.1002/j.1538-7305.1948.tb01338.x.

[17] Quinlan JR. C4.5: Programs for Machine Learning. San Mateo, Calif.: Morgan Kaufmann Publishers; 1993. OCLC: 26547590.

[18] RuleQuest. Information on See5/C5.0;. https://www.rulequest.com/see5-info.html.

[19] Kass GV. An Exploratory Technique for Investigating Large Quantities of Categorical Data. Journal of the Royal Statistical Society: Series C (Applied Statistics). 1980;29(2):119–127. _eprint: https://rss.onlinelibrary.wiley.com/doi/pdf/10.2307/2986296.

[20] Utgoff PE. Incremental Induction of Decision Trees. Machine Learning. 1989 Nov;4(2):161–186.

[21] Utgoff PE, Berkman NC, Clouse JA. Decision Tree Induction Based on Efficient Tree Restructuring. Machine Learning. 1997 Oct;29(1):5–44.

[22] Scikit-learn developers. Decision Trees (Scikit Documentation); 2020. https://scikit-learn.org/stable/modules/tree.html#tree-algorithms-id3-c4-5-c5-0-and-cart.

[23] Nijssen S, Fromont E. Mining Optimal Decision Trees from Itemset Lattices. In: Proceedings of the 13th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining - KDD '07. San Jose, California, USA: ACM Press; 2007. p. 530.

[24] Schaus P, Aoga JOR, Guns T. CoverSize: A Global Constraint for Frequency-Based Itemset Mining. In: Beck JC, editor. Principles and Practice of Constraint Programming. vol. 10416. Cham: Springer International Publishing; 2017. p. 529–546.

[25] Nocedal J, Wright S. Numerical Optimization. Springer Science & Business Media; 2006.

[26] Poe WA, Mokhatab S. Chapter 4 - Process Optimization. In: Poe WA, Mokhatab S, editors. Modeling, Control, and Optimization of Natural Gas Processing Plants. Boston: Gulf Professional Publishing; 2017. p. 173–213.

[27] Dantzig GB. Maximization of Linear Function of Variables Subject to Linear Inequalities. Activity Analysis of Production and Allocation Proceedings of the Conference on Linear Programming. 1951 Jan;.

[28] Martin RC, Rabaey JM, Chandrakasan AP, Nikolic B. Agile Software Development: Principles, Patterns, and Practices. Pearson Education; 2003.

[29] Peter J Stuckey, Guido Tack, Kim Marriott. 1.2. Installation — The MiniZinc Handbook 2.4.3; 2018. https://www.minizinc.org/doc-2.4.3/en/installation.html.

[30] Gilles Audermard, Laurent Simon. Glucose's Home Page; 2016. https://www.labri.fr/perso/lsimon/glucose/.

[31] Casas Roma J, Lozano Bagén T, Bosch Rué A. Deep learning: principios y fundamentos. Barcelona: Editorial UOC; 2019. OCLC: 1145244813. Available from: `http://digital.casalini.it/9788491806578`.

[32] Gamma E, Helm R, Johnson R, Vlissides J. Design Patterns: Elements of Reusable Object-Oriented Software. Edición: 1st ed., reprint ed. Reading, Mass: Addison Wesley; 1994.

[33] Scikit-learn developers. Understanding the Decision Tree Structure — Scikit-Learn 0.23.1 Documentation; 2020. https://scikit-learn.org/stable/auto_examples/tree/plot_unveil_tree_structure.html#sphx-glr-auto-examples-tree-plot-unveil-tree-structure-py.

[34] Harris D, Harris S. Digital Design and Computer Architecture. 2nd ed. Amsterdam: Morgan Kaufmann; 2012.