SATreeCraft: Creating a Flexible Framework for the SAT Encoding of Optimal Decision Trees with Constraints

Author: Haris Rasul - 1005931716

Supervisor(s): Professor Eldan Cohen , Pouya Shati

Date: April 12th , 2024

B.A.Sc. Thesis

Division of Engineering Science
UNIVERSITY OF TORONTO

SATreeCraft: Creating a Flexible Framework for the SAT Encoding of Optimal Decision Trees with Constraints

Author: **Haris Rasul - 1005931716**
Supervisor(s): **Eldan Cohen**[1] **, Pouya Shati**[2]

A thesis submitted to the Department of Engineering Science, University of Toronto, Canada for the requirements of Bachelors of Applied Science and Engineering, Engineering Science.

University of Toronto
Ontario, Canada
April 12, 2024

[1]Department of Mechanical and Industrial Engineering, University of Toronto, Canada
[2]Department of Computer Science, University of Toronto, Canada

Abstract

This thesis presents a novel library for the construction of interpretable decision trees through the utilization of SAT encodings and solvers, making significant contribution to the intersection between machine learning and logical satisfiability. The newly developed Python library streamlines the process of constructing decision trees for both classification and clustering tasks. It offers practitioners a robust tool with customizable constraints and compatibility with popular data interface toolkits, significantly enhancing usability and flexibility.Empirical test results demonstrate that the library not only meets but also extends the capabilities of existing algorithms. With its adaptability and powerful features, the library is set to become a useful package for researchers and practitioners aiming to leverage exact optimization and optimal decision trees with complex datasets.

Acknowledgements

I would like to express my sincerest thanks to my thesis supervisor, Professor Eldan Cohen, for his constant support, guidance, and encouragement throughout the duration of my research. Professor Cohen's expertise in Optimization and his significant contributions to the field have been instrumental in shaping an effective thesis. I am sincerely grateful for his dedication to this project and his profound impact on the domains of logic and artificial intelligence.

I would also like to extend my thanks to Pouya Shati for his continuous assistance in developing this library and for his direct guidance on the most effective methodologies to apply. His insights have been invaluable in the construction of this thesis. His research work was the basis that helped form the thesis and raised the need to create a brand new toolkit that can empower practitioners with revolutionary optimization strategies.

Finally, I wish to express my appreciation to my friends and family for their endless support on this journey. This thesis is dedicated to them, as they have been my greatest inspiration.

Contents

List of Figures

List of Tables

1 Introduction

1.1 Background

Decision trees are highly used machine learning models due to their interpretability and performance, making them a popular decision making tool [1]. As datasets grow in complexity and size, the objective of optimizing decision trees for high accuracy while minimizing generalization error becomes more difficult, evidenced by its classification as NP-Complete [2]. Moreover, integrating constraints into decision trees such as adding constraints to limit tree depth for interpretability or adding instance-level constraints to enhance solution quality – can boost model performance [3]. Given these problems, boolean satisfiability-based (SAT) learning methods have been useful for exact optimization of decision trees [4].

Optimal decision trees serve as pivotal tools in addressing two classical machine learning problems: *classification* and *clustering*. In classification, the aim is to optimize decision trees such that they classify training data accurately while preventing overfitting. Two major problems within this domain that are solved using exact optimization methods include: (1) Crafting a tree that categorizes all training data with correct labels with the minimum depth possible, and (2) when constrained to a fixed depth d, determining a tree that maximizes the number of accurately assigned labels [5]. Transitioning from classification, another critical domain of utilizing optimal trees involves constrained interpretable clustering. This type of problem involves utilizing a tree's branching nodes to determine clustering assignments that optimize objectives such as *Maximum Diameter* [6].

Recent studies [5,6] have highlighted the efficacy of employing SAT models for optimizing decision trees. These works demonstrate the potential of unique SAT encodings in enhancing decision tree performance and generalization. However, given the wide range of available SAT software to build and solve problems, coupled with the potential to incorporate various domain-specific constraints being somewhat difficult, presents a need in developing a user-friendly and adaptable framework.

1.2 Objectives & Significance

The main goal of this project is to develop a modular framework in the form of an open-source Python library package designed to construct and solve SAT encodings of optimal classification and clustering trees. The package is designed to offer advanced, optimization-based algorithms, previously inaccessible to many practitioners, that aligns with modern AI and data science software commonly used. This framework aims to significantly accelerate research in this domain and enhance practical applications of decision tree models. To ensure the project's success, we have devised three primary objectives:

1. **Development of a Modular Framework:** The objective is to develop an accessible, Python-based software framework that allows users to effortlessly generate and solve SAT encodings for the different types of decision trees. Emphasizing a user-centric design, the framework will adopt an object-oriented approach, enabling users to specify tree types, objectives, datasets, and constraints, which it will dynamically accommodate. Integration with established data science libraries such as Scikit-learn will enrich the framework, providing versatile data preprocessing, tree estimation and visualizations, and a set of commonly used evaluation metrics within this domain, including k-fold cross-validation, to ensure a user-centric approach.

2. **Implementation of Classification and Clustering Objectives:** Utilize the framework to implement and solve the two major problem types of classification trees as well as clustering tree objectives as outlined in Section 1.1.

3. **Incorporation of User-specific Constraints:** We would like to extend the capabilities of the framework by integrating new cardinality-based and pairwise constraints. This goal serves as a test of our framework's adaptability and performance. It aims to provide evidence that our toolkit allows users to seamlessly tailor a model to their own specific needs.

1.3 Thesis Overview

The remainder of this thesis is organized as follows: Chapter 2 is a literature review of the domain regarding decision tree use cases as well as popular criteria for its construction for classification and clustering. Chapter 3 provides a detailed description of preliminary knowledge including rigorous definitions of decision tree structures and their optimization techniques, the application of SAT methodologies, and an overview of existing SAT solvers employed to address these computational challenges. Chapter 4 describes the structural encodings and algorithms presented in [5] for classification tasks, and design decisions that lay the foundation for the development of our software library. Chapter 5 describes additional encodings required for clustering objectives [6]. Chapter 6 outlines the proposed software architecture, detailing the methods and functions available for users to construct and solve their decision tree problems. Chapter 7 explains the testing procedures that were conducted in order to validate our implementation and design choices, and Chapter 8 highlights the test results. Finally, Chapter 9 concludes with a summary of the project and presents potential future work.

2 Literature Review

2.1 Search-based Decision Tree Algorithms

Search-based algorithms including CART [7] and ID3[8] are widely known methods for how trees should be constructed. These methods build trees from top to bottom using a greedy strategy by selecting the best split of the data based on the attempts to minimize or maximize some heuristic such as the gini impurity [7] at each level of the tree. This type of metric evaluates the purity of each node in a decision tree that reflects the likelihood of an incorrect classification of a new instance of a datapoint[7]. Many popular data science libraries adopt these methods for tree construction that a user can fit to their own dataset. This includes popular library frameworks such as Scikit-learn, XGboost,and DecisionTree. However, while these types of algorithms are intuitive in maximizing the accuracy of a decision tree, there is a chance it can lead to sub-optimal trees. Due to the heuristic nature, a disadvantage of the CART algorithm and similar methods, is that they cannot guarantee a global optimum solution and as result it may lead to overfitting if a tree structure is allowed to grow too deep [7]. Other issues include producing results that are too unstable due to the complexity of some datasets that are sensitive to small changes or noise [7].

2.2 Challenges in Decision Tree Optimization for Classification

The optimization of decision trees can become increasingly difficult with larger datasets and more features. A primary challenge is to balance accuracy while constraining the depth of the tree. A deep tree may perfectly classify all training data correctly however may fail to generalize and have poor accuracy on a test set of data. This highlights the need for a methodological approach to decision tree optimization such that it is able to find solutions that are consistent with the principle of Occams' Razor, that states the simplest working hypothesis is preferred[9]. Data science libraries have been developed to address these concerns - however each have their own advantages and drawbacks.

2.3 Exact Optimization Methodology for Decision Trees Using SAT

SAT modelling is a method of determining if there exists a set of variable assignments that satisfy a boolean formula. It is the process of finding a solution to a prepositional logic problem that makes the entire formula true. These approaches are considered interpretable because they provide exact, logically coherent solutions that meet specified constraints, and allow for easier understanding of model creation. Research described in [10] has shown that

using SAT for modelling decision trees has been beneficial due to the exactness of the solution to a precise definition of tree construction criteria with logical constraints that ensures trees strictly adhere to specified conditions. It also shown to provide scalable solutions over large datasets [5,10].

2.4 Constrained Clustering

Unlike classification, clustering tasks are unsupervised learning tasks that wishes to partition data into groupings without assigning it any labels. Datapoints are grouped by similarity that is assessed based on features or metrics. Constrained clustering is a semi-supervised learning variant that integrates domain knowledge by the addition of constraints a user may wish to enforce on their dataset. As such they have optimization criteria established in order to improve the quality of their clustering solution [6]. There have been many methods to implement constrained clustering.

There are many constrained clustering algorithms that have grown in popularity in the past decade such as [11] that proposes the constrained K-means algorithm (MPCK-Means) , a constrained variant of the popular clustering algorithm. This variant incorporates penalties for constraint violations and utilizes metric learning to adapt the distance measure to better respect pairwise constraints. These constraints include must-link and cannot-link sets of points that determine which points can and cannot be clustered together. It is important to note that popular algorithms like this are also heuristic in nature, signifying that there is no guarantee of finding a global optimal solution. This is due to factors such as dependence on initial conditions, selection of parameters, and scalability factors.

2.5 Exact Optimization in Constrained Clustering using SAT

Numerous optimization techniques have been proposed with the goal of achieving exact solutions for clustering problems. SAT methods have achieved state-of-the-art results evidenced by work in [12]. Their work presents a SAT based framework designed to handle cluster constraints with efficient robust performance. By translating clustering constraints into SAT problems, the framework ensures adherence to specified pairwise constraints, therefore optimizing the clustering process with an exact method. This methodology presents an advantage over heuristic methods to find globally optimal solutions [12]. Work seen in [6] has expanded on these types of methods to produce interpretable decision tree approaches that use SAT to encode tree structures and constraints for clustering problem.

2.6 Classification and Clustering Tools

The current landscape of decision tree libraries predominantly utilizes greedy heuristic methods, as outlined in Section 2.1. Among the optimal decision tree libraries available in Python, some other popular libraries recently produce include ODTLearn [13], which utilizes mixed integer programming to construct fair classification trees and optimize trees according to distribution shifts. However, it does not provide a tools for clustering. Another well known library that has been created is PyDL8.5 [14], that provides both clustering and classification objectives using heuristics and dynamic programming. Although these libraries enable constraints on trees to optimize for generalization they do not have the ability to always find exact solutions. The limited number of available libraries that always ensure exact solutions in interpretable formats presents the opportunity to build one that leverages SAT solvers to fill this gap. Due to the robust performance and wide range of use cases demonstrated by the SAT encodings for both clustering and classification as described in [5,6], we aim to develop our software based on the models outlined in these references.

3 Preliminaries

3.1 Decision Tree Structures

In order to understand how decision trees are constructed and how they are able to classify and cluster samples of data correctly from a dataset X, it is imperative to understand their structure. To ensure a systematic approach in developing our software, we base our design on the definitions provided by [5]. These baseline concepts provide the groundwork for a modular software architecture regarding tree constructions:

$$\mathcal{T} = (\mathcal{T}_B, \mathcal{T}_L, \delta, p, l, r) \tag{1}$$

A decision tree follows a tree structure $\mathcal{T}$ that is a tuple of sets and functions. $\mathcal{T}_B$ and $\mathcal{T}_L$ are a finite set of branch and leaf nodes respectively, p is the parent function to find the parent node of some node t, l and r are the left and right child functions to find the respective child nodes of some node $t \in \mathcal{T}_B$, and $\delta \in \mathcal{T}_B$ is the root node of the tree. In the scope of this toolkit for classification, we focus on complete binary tree structures. In this scenario we define a complete binary tree as a tree structure $\mathcal{T}$ where every branch node $t \in \mathcal{T}_B$ has both a left and right child node, and every leaf node $t \in \mathcal{T}_L$ is of the same depth. We use the following definition of depth of a node from [5]:

$$depth(t) = depth(p(t)) + 1,\ t \in \mathcal{T}_B \cup \mathcal{T}_L \tag{2}$$

Where depth of each node is recursively defined by each of its parent nodes and the root node has $depth(\delta) = 0$. We consider the depth of the tree $depth(\mathcal{T})$ to be the depth of its leaf nodes. A classification decision tree assigns labels from a set C to each of its data points $x_i \in X$ based on a sample point's values for each feature in some set of features F that describes the dataset. Starting at the root, the tree splits the data at each branch node by selecting the some feature $j \in F$ to partition the dataset into subsets. The feature selected at each branch node is based on some form of optimization criteria, guiding the input data through the branches until a leaf node is reached.For each branch node feature selected, there is a threshold value that determines the direction each point is guided based on the set of feature values $dom(j)$ that reach that node. The leaf node provides the class label for the input data, effectively classifying it according to the patterns learned during the tree's construction in training. The model is constructed and trained on some dataset X with a known set of classification labels for each of its points $\gamma(x_i)$. Considering the formal structure defined in Eq. 1 and the process to classify x_i outlined above, our modular software framework adopts a rigorous definition of a decision tree as presented in [5], which

helps structure the architecture:

$$\mathcal{D} = (\mathcal{T}, \beta, \alpha, \theta) \tag{3}$$

A decision tree is a tuple D that follows a tree structure $\mathcal{T}$, a feature selection function β such that for a given node $t \in \mathcal{T}_B$ a feature $j \in F$ is assigned to it, a threshold function α that determines the decision rule at each branch node, dictating whether a data point x_i with feature value $x_i[j]$ should be guided to the left child $l(t)$ or the right child $r(t)$ in the tree, and a labelling function θ that assigns a label from C to each leaf node. The final tree constructed is a predictor function Θ, that predicts a label for x_i.

Upon its construction, a model's performance can be measured by its accuracy, which is the fraction of correct label assignments predicted where $\Theta(x_i) = \gamma(x_i)$ for some X.

3.2 SAT Modelling

A common way of modelling a SAT problem is to design a Conjunctive Normal Form (CNF) formula. A CNF formula consists of an AND of clauses, each clause is an OR of literals, where each literal is a boolean variable that can be in its original form or its negation [15]. We construct an example below as a visualization of the form using an arbitrary set of boolean variables p,q,r:

$$(p \vee \neg q) \wedge (\neg r \vee q) \wedge (p \vee \neg r \vee q) \tag{4}$$

Where a clause is evaluated as true if at least 1 of its literals is evaluated as true, and the solution to the formula is one that finds an assigned value (1 or 0) of each literal that ensures all clauses are evaluated as true [15]. Decision tree constructions can be modelled using CNF formulae, where a solution to the formula indicates how the tree is structured from the features selected at each branching node to the labels assigned to each leaf node. There are a wide variety of SAT solvers that have differing approaches to finding optimal literal assignments that can result in varying times to find solutions.

MaxSAT is a variant problem that is less strict where the objective is to find an assignment such that it maximizes the number of clauses to be evaluated as true. An extension of this variant is Partial MaxSAT that specifies clauses as *hard* or *soft*, that constrains a solution by specifying which clauses *must* be satisfied or *should* be satisfied in order to maximize the number of clauses evaluated as true [16].

3.3 Optimizing Classification Objectives

Finding a decision tree of minimum height and one of maximum accuracy given of some fixed depth d are the two classification objectives our solutions aims to optimize. This section formally defines both optimization problems and discusses works and methodologies that have been used to solve the problems.

3.3.1 Minimum Height Tree Problem

We begin by examining the minimum height problem description from [5]. We use the formal definition to structure our design approach:

$$\begin{aligned} \min \quad & depth(\mathcal{T}) \\ \text{s.t.} \quad & \sum_{x_i \in X} I(\Theta(x_i), \gamma(x_i)) = |X|, \\ & \forall t_i, t_j \in \mathcal{T}_L, depth(t_i) = depth(t_j) \end{aligned} \tag{5}$$

Where I is the indicator function. Eq. 5 states that a tree of a minimum height should be found for a decision tree's tree structure such that all training labels should be correctly classified.

There are several reasons for wanting to create a minimum height tree which includes greater interpetability, faster computation on larger datasets for classification, as well as reducing the chances of overfitting a model to its training set. Many solutions to this problem have included SAT modelling [5,9,17] with various different clauses. The various solutions can differ based on how they encode the thresholds for features selected at branch nodes as well as other clauses that enforce path traversals or label assignments to leaf nodes.

3.3.2 Maximum Accuracy Given Fixed Depth Problem

The maximum accuracy objective defined by Eq. 6 attempts to maximize the number of correct labels on a training dataset given by the decision tree predictor for any fixed depth[5]:

$$\begin{aligned} \max \quad & \sum_{x_i \in X} I(\Theta(x_i), \gamma(x_i)) \\ \text{s.t.} \quad & \forall t_i \in \mathcal{T}_L, depth(t_i) = d \end{aligned} \tag{6}$$

Research work seen in [18] describes the motivation for optimizing accuracy of a given depth. As datasets tend to become more complex with many different features and large samples,

ensuring high accuracy with high generalization may become increasingly difficult [18]. SAT approaches described in [18,19,20] propose methodologies that balance interpetability with generalization with a fixed depth constraint. The limiting depth constraint also aligns with the practical needs of users of the package who may require a model to be built within their own set of set of operational parameters and requirements. Research works [5,18] use the MaxSAT variant to solve this type of problem described in Section 3.2.

3.4 Decision Tree Clustering

Decision trees are able to produce interpretable clustering solutions due to its hierarchical structure. Upon construction of a tree, each leaf node represents a cluster for data points that are guided there [21]. This approach provides a natural and intuitive way to group data. In our software currently under development, we are harnessing this capability of decision trees for adding functionalities for constrained clustering. Our focus is on implementing well-known clustering objectives while incorporating specific constraints. This improves the robustness and utility of our designs by being able to create both interpretable and strong quality solutions. This rest of this section rigorously defines clustering tree objectives and constraints that we use from [6] to help our design our software.

3.4.1 Clustering Tree Assignment Process

Given a pre-determined number of clusters K, a decision tree $\mathcal{D}$, each data point $x_i \in X$ is assigned a cluster. The cluster assignment, denoted as $\Theta_{\mathcal{D}}(x_i)$, is determined by the leaf node x_i was directed to by $\mathcal{D}$. Each cluster has at least one data point in it and a cluster can span over multiple leaf nodes.

3.4.2 Clustering Constraints

We establish two fundamental pairwise constraints essential for the implementation of constrained clustering. The first constraint is the must-link constraint ML that we define as [6]:

$$\forall (x_1, x_2) \in ML : \Theta_{\mathcal{D}}(x_1) = \Theta_{\mathcal{D}}(x_2) \tag{7}$$

Eq. 7 states that for every pair of points that is part of a must-link set defined by a user, the clustering tree must assign them to the same cluster.

A user also has the ability to constrain a model by stating a cannot link constraint CL, defined in Eq. 8 , that enforces a condition that no two points part of the same CL set can

be in the same cluster [6]:

$$\forall (x_1, x_2) \in CL : \Theta_{\mathcal{D}}(x_1) \neq \Theta_{\mathcal{D}}(x_2) \tag{8}$$

3.4.3 Optimization Criteria

There are two distinct metrics that can be optimized to ensure high-quality clustering solutions based on the user-selected clustering objective/problem. The first metric is the *minimum split* metric that states the following [6]:

$$MS_{\mathcal{D}} = \min\{\|x_1 - x_2\| \text{ s.t. } \Theta_{\mathcal{D}}(x_1) \neq \Theta_{\mathcal{D}}(x_2)\} \tag{9}$$

Which measures the minimum distance between any two points in X that are part of separate clusters. Please note the distance term we use is representative of any distance measurement that user may wish to select such as euclidean, manhattan, or cosine distance.

The second optimization metric is the *maximum diameter* which measures the largest distance between any two points in the same cluster [6]:

$$MD_{\mathcal{D}} = \max\{\|x_1 - x_2\| \text{ s.t. } \Theta_D(x_1) = \Theta_D(x_2)\} \tag{10}$$

3.5 Decision Tree Clustering Problems

Upon understanding the clustering solutions decision trees can provide as well as their optimization criteria, users may encounter two prominent clustering challenges described in [6], for which they seek high-quality, interpretable solutions. The problem selected by the user will guide the determination of the specific metrics that need to be optimized.

3.5.1 Minimizing MD

In this problem we wish to find a complete tree of a user given depth of d such that we minimize a single criterion that is the maximum diameter of $\Theta_{\mathcal{D}}$ for a given dataset X and number of clusters K. We must abide by the following constraints [6]:

1. $\Theta_{\mathcal{D}}$ **respects all** ML and CL sets present
2. $MD_{\mathcal{D}} \leq MD_{\mathcal{D}^*} + \varepsilon$

Where the second constraint indicates that the maximum diameter produced should not be greater than the optimal solution's minimum possible maximum diameter $\mathcal{D}^*$ by some

bound approximation parameter ε defined by a user. Setting $\varepsilon = 0$ indicates one is looking for the optimal solution [6] with the smallest diameter possible and cannot minimize it anymore without violating the clustering constraints. Adding ε enables a solution to have more flexibility - this may be useful for more complex datasets when a user must balance computation with interpetability as it becomes increasingly difficult to find an exact optimal solution.

3.5.2 Bi-criteria: Maximize MS while Minimizing MD

For the second problem we now have a 2-variable optimization criteria to maximize minimum split and minimize maximum diameter. In this case we would like to find a complete tree of some fixed depth d for dataset X and number of clusters K such that we do not violate any of the constraints as we optimize for these metrics:

1. $\Theta_{\mathcal{D}}$ **respects all** ML and CL sets present
2. $MS_{\mathcal{D}} \geq MS_{\mathcal{D}^*} + \varepsilon$
3. $MD_{\mathcal{D}} \leq MD_{\mathcal{D}^*} + \varepsilon$

In this case $\mathcal{D}^*$ is a Pareto solution, which indicates in multi-objective optimization, we can no longer optimize one objective without the other worsening.

3.6 Harnessing PySAT Software for Development

Prior to exploring the detailed architecture of our library, it's crucial to acknowledge the foundational software utilized within this framework—a key component in the Python ecosystem tailored for addressing Boolean satisfiability problems. PySAT [22][1], released in 2018, has become a popular API for building SAT problems in Python. It includes methods for creating CNF formulae as well as other variants that we utilize within our own package. It also offers a wide variety of different solvers that can be used to solve CNF formulae. It also provides methods to handle Partial MaxSAT problems through a weighted CNF method that allows users to indicate hard clauses that *must* be satisfied as well as soft clauses that *should* be satisfied. PySAT enables the use of external solvers as it has the ability to export CNF problems in the form of DIMAC file types to a user's storage, a universal SAT solver file type that can be passed into many other SAT solver technologies to get solutions. Its flexibility of user input and wide range of SAT tools has been useful in designing our own package for the problems of focus.

[1]For more details of the library, visit `https://pysathq.github.io`

4 Classification Methodologies and Set-Up

The software we have designed handles both major classification problems described in Section 3.3. The library is based off the encodings in [5] due to its uniqueness for handling a wider range of different types of features with high effectiveness, its capabilities to find smaller depth trees for many different datasets compared other works, as well as the time to find a solution has shown to be quicker than others [5,9].

4.1 Features in Data

Using the framework provided in [5], features and can be categorized into two different domains: F_C *categorical features*, F_N *Numerical features*, where binary features with the values of 1 or 0, can be considered numerical features. These two types of features have different SAT encodings as seen in Section 4.3.

The concept of power set branching is used, as suggested in [5], is employed to construct decision trees that must handle categorical features. Unlike numeric features, categorical features lack a natural order, meaning that we cannot directly apply a threshold-based approach that separates data points into two groups where data points above or below a threshold are directed in opposite directions. Numeric features can be easily ordered and split at a node based on whether they are greater or less than a given threshold, inherently organizing the dataset into a hierarchy that reflects the natural ordering of the numbers. We apply power set branching to select a group of categorical features to be directed left of a branch node that would be solely determined by our SAT solution to the problem encodings [5].

4.2 Tree Variables

In modelling our SAT problem we use the same variables for the literals as seen in [5]. Defining these variables facilitates a more intuitive understanding of clause structures and the interpretation of solution assignments:

$a_{t,j}$: Represents whether feature j is chosen for the split at branching node t.

$s_{i,t}$: Represents whether point i is directed towards the left child, if it passes through branching node t.

$z_{i,t}$: Represents whether point i ends up at leaf node t.

$g_{t,c}$: Represents whether label c is assigned to leaf node t.

p_i: Represents whether point x_i is correctly classified.

A set of these variables are defined for each depth of a tree. Please note that that p_i is only used in the max accuracy problem.

4.3 Encoding General Trees Structures

Our software follows the encodings of clauses seen in [5]. General tree structure encodings involve enforcing conditions such as ensuring a feature is selected at a branch node, enforcing conditions on path validity of data points, as well as label assignment conditions at leaf nodes.

For datasets that only include binary or numeric features we utilize the clauses from [5] which we will refer to as *numerical-feature tree clauses* . However, for datasets that include categorical features we must utilize clauses the other clauses provided in [5], which we will refer to as *categorical-feature tree clauses*, as it implements the power set branching rules and natural ordering rules for categorical and numerical features respectively.

4.4 Solving Minimum Height Vs. Maximum Accuracy

Upon encoding the general tree structure with the correct clauses based on whether or not categorical features are present, there are two different sets of clauses that must be applied based on the classification objective one wishes to solve. There are also slightly different approaches we must implement in order to handle each objective.

If one wishes to solve the minimum height objective our software will add clause a from [5] that is unique to this problem represented by Eq. 11. This clause enforces that the training labels of the data points match the labels given by the leaf node they reach:

$$(\neg z_{i,t}, g_{t,\gamma(x_i)}) \quad t \in \mathcal{T}_L, x_i \in X \tag{11}$$

Our solution-finding approach is iterative, as recommended in [5]. We construct the the set of literals and SAT clauses starting with a tree depth of 1 and attempt to find a solution. If the problem is unsolvable at this initial depth, we increment the depth by one and reattempt to solve it with a new set of literals and clauses. This process is repeated until a solution is identified.

The maximum accuracy problem is a partial MaxSAT problem in order to maximize the number of correct labels for a given depth specified by a user. For this problem we have a set of hard and soft clauses that need to be added and specified. The clauses that define general

tree structure are all tagged as hard clauses. We then proceed to add two sets of clauses from [5] that are unique to this objective, represented by Eq. 12 and Eq. 13 respectively:

$$(\neg p_i, \neg z_{i,t}, g_{t,\gamma(x_i)}) \quad t \in \mathcal{T}_L, x_i \in X \tag{12}$$

$$(p_i) \tag{13}$$

Eq. 12 is tagged as hard clauses that ensures that p_i is only true if the label given to a training data point by the tree as it arrives to its respective leaf node matches its ground truth label. Eq. 13 are soft clauses that are added to the SAT formula which a solver will try to evaluate as many possible as true for a given depth in order to maximize accuracy. These clauses represent the count of data points correctly labeled by the decision tree. The cost of a solution is determined by the number of these soft clauses that are not satisfied.

4.5 Additional User Constraints for Classification

In addition to the fixed depth constraint that is required for maximum accuracy classification problems, one may be required to incorporate various other constraints to develop a decision tree model that is most optimal for their specific dataset. Integrating such constraints can allow tree models to generalize more effectively on out-of-sample data and prevent overfitting. This section highlights these additional constraints, which can be broadly categorized into pruning constraints and tree structure constraints. We describe how these constraints can be encoded as SAT formulae, adding them onto a user's existing optimization problem. For pruning constraints, we focus on two key constraints as outlined in [5]: *Minimum Support* and *Minimum Margin*. Regarding tree structure constraints, we explore the *Oblivious Tree* constraint, a novel addition derived from the foundational papers guiding our implementation.

As suggested in [5], while it is possible to overlay these constraints on both the minimum height and maximum accuracy problems, it is better to focus on enforcing these constraints on the maximum accuracy problem. Applying these constraints to the minimum height tree problem could result in the infeasibility of finding a solution where all training data points are correctly classified. It is also important to note that the addition of these constraints to the maximum accuracy problem can lead to longer solution times or increased cost of solutions [5].

4.5.1 Minimum Support Constraints

The minimum support constraint a user may wish to apply guarantees that each leaf node has a certain amount of data points S directed towards it. This constraint can prevent a tree from creating highly specific rules for splitting that only apply well to a very small number of samples and scenarios. To impose this condition onto a classification problem we use the following definition from [5]:

$$\forall i \in \mathcal{T}_L : \sum_i \#z_{i,t} \geq S \tag{14}$$

where $\#z_{i,t}$ is the numerical indicator of the $z_{i,t}$ literal. Eq. 14 is a cardinality constraint that states at least S of those literals must be evaluated as true, which is equivalent to having minimum of S data points at every leaf node. Encoding cardinality constraints is not trivial, however we employ a sequential cardinality encoder as suggested in [5,23], which effectively transforms the constraint into a series of logical clauses that can be interpreted by SAT solvers, ensuring that the decision trees follow the specified minimum support.

4.5.2 Minimum Margin Constraints

The minimum margin constraint that can be added guarantees a minimum number of points M should be directed both left and right of each branch node. One may choose to enforce this constraint to ensure that each split at a branch node is more balanced and prevents one split form having very few points, which could potentially contribute to the robustness of a model. We used the encodings from [5] that a user has the option to enforce:

$$(\neg a_{t,j}, s^{M}_{\#_j,t}) \quad t \in \mathcal{T}_B, j \in F \tag{15}$$

$$(\neg a_{t,j}, s^{|X|-M+1}_{\#_j,t}) \quad t \in \mathcal{T}_B, j \in F \tag{16}$$

As mentioned in [5], based on Eq. 15 and Eq. 16, this constraint only works for numerical feature nodes because categorical feature branching is not ordered by the baseline encodings.

4.5.3 Oblivious Tree Constraints

We have developed an innovative encoding method to impose oblivious tree structures onto the existing tree structure encodings presented in [5]. The oblivious tree structure we have successfully implemented is one such that for every branch node on the same level, the same

feature is chosen for splitting criteria [24]. The encodings are defined as the following:

$$(\neg a_{t_1,j}, a_{t_2,j}, a_{t_1,j}, \neg a_{t_2,j}), \forall t_1, t_2 \in \mathcal{T}_B, \forall t_1, t_2 \in \mathcal{L}(d_r), \forall d_r \in \{0, \ldots, d-1\}, j \in F \tag{17}$$

Where $\mathcal{L}(d_r)$ is the set of branch nodes at some depth d_r within our tree structure of depth d. It is important to note Eq. 17 only enforces that the feature chosen for all branch nodes of the same depth level is consistent, however the threshold α at each of those nodes may vary. A simpler visualization of the difference between the standard complete and oblivious trees we employ can be seen in Appendix A.

5 Clustering Methodologies

Currently, our framework has been implemented to optimize clustering problems with a primary focus on the maximum diameter objective, as mentioned in Section 3.5. This feature serves as evidence of the versatility of our framework by being able solve both classification and clustering tasks. The current version lays the groundwork for a solid foundation for accommodating a broader range of clustering objectives such as bi-criteria optimization tasks. The rest of this section describes the clustering techniques and encodings we have developed, based on the highly effective methods proposed in [6]. The approaches proposed within this research is the first instance of leveraging SAT for decision tree clustering, allowing us to create a library that yields interpretable and strong constrained clustering results. Clustering optimization in our library is only supported for datasets with strictly numerical features.

5.1 ε–Optimal Tree Clustering and Distance Classes

As mentioned in Section 3.5, ε is a parameter the user sets to define the flexibility of the solution deduced. By defining ε-optimality, we allow for clusters whose maximum diameters exceed the absolute minimum by no more than the user set threshold. This is particularly useful when finding exact optimal solutions become very difficult or expensive to find due to a dataset's inherent variability. The notion of distance we have implemented in our library is a euclidean distance between points, however, any measure of distance is applicable for minimizing maximum diameter.

As mentioned in [6] to operationalize and encode the ε-approximation we use the concept of distance classes. We divide all the pairs of data points in X into μ non-overlapping intervals $D = \{D_1, ..., D_\mu\}$ such that the largest and smallest distance of pairs of points within each class is at most ϵ apart. All pairs of points are sorted in order from smallest to largest diameter and are then added to each respective distance class, guaranteeing the distances within each class are within the user defined threshold. All pairs of points in the same distance class are then treated similarly whether or not they are clustered together. Using this new data structure, we proceed to define the index λ^- that defines the distance class containing the maximum allowable distances that we are willing to accept within a single cluster. If a pair of points are in a distance class smaller than the index , they are able to be clustered together, however any two pair of points belonging to a distance class that has an index greater than λ^-, they cannot be clustered together as summarized in Eq. 18 :

$$((x_1, x_2) \in D_w, w > \lambda^-) \rightarrow \Theta_{\mathcal{D}}(x_1) \neq \Theta_{\mathcal{D}}(x_2) \tag{18}$$

By optimizing λ^- index, we are effectively minimizing the maximum diameter objective

previously defined in Eq. 10. Appendix B provides a more detailed example of the impact of the ε-approximation and how it can lead to sub-optimal solutions.

5.2 Clustering Variables

Similarly to classification, [6] defines several new literals and redefines other literals from [5] to now handle clustering problems. To model the SAT problem within our library, we also define these literals for a more intuitive understanding of clause structure and decoded solutions. Listed below are several new literals required for clustering, as well as a few literals whose definitions have been modified from their original use in classification as outlined in [6]:

$g_{t,c}$: The cluster assigned to leaf t is or comes after cluster c, $c \in K$.

$x_{i,c}$: The cluster assigned to point x_i is or comes after cluster c.

b_w^-: The negation of whether the pairs in distance class w should be clustered separately

5.3 Encoding Cluster Trees Structures

The general tree structure encodings regarding feature selection, path validity, and label assignment utilized in classification problems [5] are also employed in solving clustering problems. However, [6] adds few more general clustering SAT clauses, as well as specific clauses required for the maximum diameter problem. These are all hard clauses that must be solved regardless of the optimization criteria that we have implemented from [6] that we will refer to as the *general clustering constraints*. It is important to note that the clustering encodings is dependant on the depth of the tree, and needs to be specified by the user as the number of leaf nodes should be enough such that there are enough clusters. The depth of the tree should be at least $\lceil \log_2(K) \rceil$ for K clusters defined by the user.

5.4 Solving Maximum Diameter Problem

In order to minimize the maximum diameter problem we would like to minimize the λ^- index. This index value is equivalent to the number of b_w^- literals evaluated as true expressed by the following equation:

$$\lambda^- = \sum_w I(b_w^- = \text{true}) \tag{19}$$

To minimize this summation and the value of the index, we transform the problem into a partial MaxSAT problem and add the following soft clauses that are strictly added for solving a maximum diameter problem:

$$(\neg b_w^-) \quad w \in W \tag{20}$$

5.5 Encoding Additional User Constraints for Clustering

As mentioned prior, our library would like to extend to the semi-supervised learning tasks of constrained clustering. As such our design provides encodings for *Must-link* pair and *Cannot-link* pair set constraints if applicable that can be enforced on top of the maximum diameter problem.

If a user has a set of ML pairs for their problem, the following encodings are integrated into the maximum diameter SAT formula:

$$(\neg x_{i,c}, x_{i',c}) \quad \forall (i,i') \in ML, c \in [1..k-1] \tag{21}$$

$$(x_{i,c}, \neg x_{i',c}) \quad \forall (i,i') \in ML, c \in [1..k-1] \tag{22}$$

Eq. 21 and Eq 22 ensure that pairs of of points within the set are assigned to the same cluster c of k clusters. If a user has pairs of points that cannot link together , they are able to enforce the following encodings into the SAT model to integrate CL constraints:

$$(x_{i,1}, x_{i',1}) \quad \forall (i,i') \in CL \tag{23}$$

$$(\neg x_{i,k-1}, \neg x_{i',k-1}) \quad \forall (i,i') \in CL \tag{24}$$

$$(\neg x_{i,c}, \neg x_{i',c}, x_{i,c+1}, x_{i',c+1}) \quad \forall (i,i') \in CL, c \in [1..k-2] \tag{25}$$

6 Software Design for SATreeCraft

6.1 Software Architecture

This section discusses the software architecture of the library tree for various objectives. It also walks through critical logic of the implementation as proof of its modularity.The full logic and current library in-development can be found at the following link: `https://github.com/HarisRasul12/ESC499-Thesis-SAT-Trees`

The overall software architecture has been represented by Figure 6.1 below:

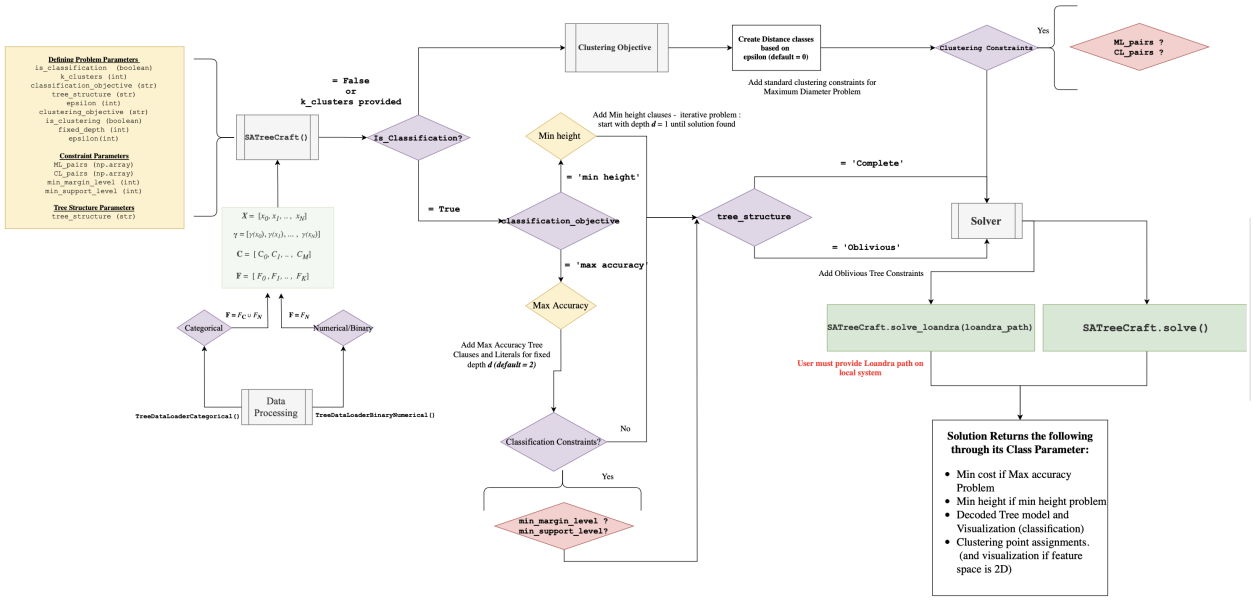

Figure 6.1: Software pipeline for classification tree objectives

The library has been structured to include 4 main components:

1. **Dataloaders**: Used to both parse raw dataset file types as well as transform datasets such that it is able to be efficiently encode feature types and labels into SAT formulae.

2. **SATreeCraft()**: The main module interface that allows user to specify problem type, model parameters, add constraints they may have, and then solve the problem.

3. **SATreeClassifier()**: Upon solving a classification problem, the user can input the returned tree model data structure into this interface to make predictions on out of sample data. Scikit learn methods and metrics have been fully integrated within this interface.

4. **Utilities**: We integrated many tools within this library from Scikit-learn to evaluate model performance such as k-fold testing, confusion matrices, and other metrics.

The architecture of the software, as illustrated in Figure 6.2, showcases a modular framework with a distinct file structure. We have created a modular structure such that every problem is first encoded with integral tree structure encodings, then based on the type of data present and the problem selected, the correct module will be called in the backend to help solve and decode a model for the user. This design principle ensures that each type of problem and associated user constraint is managed by a dedicated module. The library's structure enables seamless operation and integration, allowing for the handling of various tasks without the need for manual configuration or interaction for building SAT clauses.

Figure 6.2: File structure and Software Architecture of Library

6.1.1 Data Processing

The Python library we have created has functions that simultaneously handle both categorical and numerical features both in data pre-processing and building clauses.The following classes have been developed to handle datasets that contain the different types of features. Two different data loaders have been created for numerical/binary and categorical feature datasets respectively. A data loader requires the file path of the data, a delimiter for how features are separated, as well as the index position of the labels in the data format provided - it has been designed to handle .txt, .csv, and .xslx files based on the dataset formats that were tested and obtained from the UCI repository [25]. The data loader function processes the input dataset and returns a structured array of data points (X), each with their respective feature values. It also generates a list of features (F), numerically labeled based on their index positions within the data file using a label encoding method from the scikit-learn library. In addition, the function outputs the target labels ($\gamma_{(x_i)}$) for each data point and the unique set of labels (C) present in the dataset, which aids in creating the literals necessary for SAT encodings. The features array is crafted to act as a numerical identifier for the features, enhancing their traceability during subsequent data processing stages. The loader further offers the capability to exclude specific features from processing by accepting column indices that correspond to these features in the dataset, providing greater control over data preprocessing and manipulation.

```
from SATreeCraft.utils import *
file_path_to_test = 'Datasets/wine/wine.data'
X = TreeDataLoaderBinaryNumerical(file_path=file_path_to_test, delimiter=',', label_position= -1)
print(X.dataset)
print(X.features)
print(X.labels)
print(X.true_labels_for_points)
```

Code Excerpt 6.1: Example usage of numerical/binary feature dataset loader

The data loader for categorical features was designed similar to the the numerical feature exclusive data loader with slight different due to the formats that were provided for categorical datasets used from [25]. Our data loader is designed to accommodate datasets with categorical features that may or may not be separated by delimiters. Users can specify their data's format to ensure accurate processing. When a dataset contains both categorical and numerical features, the user should indicate the column indexes of the numerical features to ensure their correct extraction. The data loader outputs the standard structured array of

data points, feature values, and target labels, along with two additional arrays. One array identifies the column indexes of categorical features, and the other identifies the indexes for binary/numerical features, representing the sets F_C and F_N respectively.

```
from utils import *
# Example usage:
file_path = 'Datasets/credit+approval/crx.data'
label_index = -1
categorical_feature_index = None
numerical_indices = np.array([1, 2, 7, 10, 13,14])
data_loader = TreeDataLoaderWithCategorical(file_path= file_path,
    ↪ label_index= label_index,numerical_indices=numerical_indices,
    ↪ categorical_feature_index=categorical_feature_index)
# Accessing the processed data
print("Features:", data_loader.features, data_loader.features.shape)
print("Categorical Features:", data_loader.features_categorical,
    ↪ data_loader.features_categorical.shape)
print("Numerical Features:", data_loader.features_numerical, data_loader.
    ↪ features_numerical.shape)
print("Labels:", data_loader.labels, data_loader.labels.shape)
print("True Labels for Points:", data_loader.true_labels_for_points,
    ↪ data_loader.true_labels_for_points.shape)
print("Dataset:\n", data_loader.dataset,data_loader.dataset.shape)
```

Code Excerpt 6.2: Example usage of categorical feature dataset loader

Both data loaders' output, comprising a structured array of data points along with associated feature values, target labels, and a unique set of labels, facilitates the construction of SAT clauses by aligning with the sets identified in [5]. This not only simplifies the clause-building process but also streamlines the iteration over data points and the subsequent generation of proper clauses. By providing an organized format that mirrors the sets defined in the referenced study, the data loader supports the traceability of features throughout the data processing stages, thereby contributing to a more efficient SAT encoding process. Renaming features to integer indexes as well as labels helps us stay consistent with the tree variables described in Chapter 4.

6.1.2 Main Class Module: SATreeCraft

SATreeCraft serves as the central module in the algorithm suite where the user will be able to feed their transform dataset and adjust parameters according to the problem they wish to

solve. In general, the user should always provide the dataset, features , the list of numerical features, and the list of categorical features (if present) that would be prepared by the data loader. Upon selecting the problem as well the appropriate parameters for the task at hand, the correct SAT encodings will be built for solving. To optimize for a particular problem, users should specify the following essential parameters within the module:

- **labels**: For any classification task, the user must provide the unique set of labels present in the dataset domain, which is provided by our dataloader. For clustering no label information is required. The default parameter value is set to NULL.
- **true labels for points**: For classification tasks only - user must provide the true label for each data point $x_i \in X$.
- **is classification?**: This is a boolean parameter that states that the user wants to solve a classification problem. If False, will assume the user would like to solve a clustering problem and the user must provide the proper set of clustering parameters. The default parameter is set to True.
- **classification objective**: If the user wishes to solve a classification objective, they must then specify the problem to either be minimum height or maximum accuracy. To initialize the minimum height problem the user should input '*min_height*', and for max accuracy '*max_accuracy*'. The default parameter is set to solve the minimum height problem.
- **fixed depth**: This parameter is only relevant to the max accuracy and clustering problems. This defines the height of the tree to be built and solved for. In clustering settings, the depth of the tree should be specified such that there is enough leaf nodes for the number of clusters the user would specify.
- **tree structure**: This parameter is determines whether or not to add oblivious tree constraints to a problem. To specify an oblivious tree structure the user must provide the input '*Oblivious*'. The default is set to the complete tree structure.
- **min support**: This parameter is determines whether or not to add minimum support SAT encodings to the problem or not. The user should provide an integer for the minimum support. The default is set to 0.
- **min margin**: This parameter determines whether or not to add minimum margin SAT encodings to the problem or not. The user should provide an integer for the minimum margin. The default is set to 1.

- **k clusters**: This parameter is only relevant to clustering problems. The user should provide an integer for the number of clusters they want on their dataset. Providing a value instantiates a clustering objective immediately. Default parameter value is set to NULL.
- **clustering objective**: This parameter determines the clustering objective the user would like to solve. The current version only supports the default problem maximum diameter.
- **epsilon**: This parameter controls the distance class optimization and flexibility of the optimality of clustering solutions. The default parameter is set to 0, where the problem is encoded to find the global optimal solution.
- **CL pairs**: This parameter controls the cannot-link pair encodings if a user has a set of points that cannot be clustered together. User should provide an array of pairs to initialize encodings for all points that cannot be clustered together.
- **ML pairs**: This parameter controls the must-link pair encodings if a user has a set of points that must be clustered together. The parameter default is an empty set.

```
SATreeCraft(dataset,
            features,
            labels = None,
            true_labels_for_points = None,
            features_numerical = None,
            features_categorical = None,
            is_classification = True,
            classification_objective = 'min_height',
            fixed_depth = None,
            tree_structure = 'Complete',
            min_support = 0,
            min_margin = 1,
            k_clusters = None,
            clustering_objective = 'max_diameter',
            is_clustering = False,
            epsilon = 0,
            CL_pairs = np.array([]),
            ML_pairs = np.array([])
            )
```

Code Excerpt 6.3: SATreeCraft main module to instantiate optimization problem

Proper error handling as well informative error messages are provided to the user should they provide incorrect parameters for the problem they wish to solve.

6.1.3 Backend SAT Formulation for Problems

As outlined in Chapter 4 and Chapter 5, the specific clauses required and the SAT implementation vary depending on the objective and the types of features present in the user's dataset. To accommodate these variations, our framework is designed to adapt its construction of SAT problems, allowing for both MaxSAT and standard SAT formulations. To address partial MaxSAT problems like maximum accuracy and clustering, our framework employs PySAT's weighted CNF method, that allows us to assign unit weights to certain clauses to indicate they are soft clauses. We assign unit weights to each soft clause in this case in order to formulate the cost of our solution as it would match the number data points misclassified or not clustered together if the clause is not satisfied. Conversely, for the minimum height problem, we utilize PySAT's standard CNF method. As specified in Chapter 4, we apply the *categorical-feature tree clauses* for datasets with categorical features, while for datasets containing only numerical and binary features, the *numerical-feature tree clauses* are employed to model the general tree structure. The following functions listed below are called by SATreecraft in the backend to build the required literals for the problem, the correct clauses unique to the problem, add any constraint encodings provided by the user, and ultimately prepare the CNF or WCNF object from PySAT to be solved.

```
## Strictly nuemerical dataset handling:

# min height tree problem F_N only
# SAT Problem - utilized PySAT CNF object
find_min_depth_tree_problem(self, features, labels, true_labels_for_points
    ↪ , dataset)

# max accuracy tree problem F_N only
# MaxSAT Problem - utilized PySAT WCNF object with unit weights on soft
    ↪ clauses
find_fixed_depth_tree_problem(self, features, labels,
    ↪ true_labels_for_points, dataset,depth)

## Categorical dataset handling:

# min height tree problem F_C present
# SAT Problem - utilized PySAT CNF object
find_min_depth_tree_categorical_problem(self, features,
    ↪ features_categorical, features_numerical, labels,
    ↪ true_labels_for_points, dataset)

# max accuracy tree problem F_C  present
# MaxSAT Problem - utilized PySAT WCNF object with unit weights on soft
    ↪ clauses
find_fixed_depth_tree_categorical_problem(self, features,
    ↪ features_categorical, features_numerical, labels,
    ↪ true_labels_for_points, dataset, depth)

## Clustering Problems:
# MaxSAT Problem - utilized PySAT WCNF object with unit weights on soft
    ↪ clauses
solve_clustering_problem_max_diameter(self, dataset,features,k_clusters,
    ↪ depth, epsilon, CL_pairs, ML_pairs)
```

Code Excerpt 6.4: Internal SAT clause formulation functions

This framework allows for different data types and optimization objectives, allowing users to fine-tune the SAT clauses to their specific requirements. These methods not only sheds light on the underlying mechanics of SAT problem formulation but also allows users to engage with and adapt the process according to their needs.

6.1.4 Solving SAT Problems

Once the correct CNF representations are developed for the dataset, they are ready to be solved by SAT Solvers. We have built two different methods a user can call from SATreeCraft to get the solution to their problem:

- **SATreeCraft.solve()**: CNF formula is solved by PySAT's default solvers. Currently, the default solvers being MiniSAT22 for CNF and Glucose3 for WCNF forms.
- **SATreeCraft.solve_loandra(***LoandraPath***)**: CNF/WCNF solved using the state of the art C++ based LOANDRA MaxSAT solver [26][1].

LOANDRA is an open source MaxSAT solver that is externally provided from PySAT. The solver has seen strong performance in solving partial MaxSAT problems compared to default PySAT solvers. This is due to different SAT solving algorithms, as well as being based in C++. Default Solvers from PySAT that have been interfaced with Python, an interpreted language, may also suffer from slower execution times due to the overhead of the interpreter. For these reasons, we wanted to add this functionality to further improve the usability and customization of our library. LOANDRA is implemented by calling our *loandra.py* module as seen in Figure 6.2. In this module, we export the user's CNF formula into a DIMAC file type using a PySAT method that LOANDRA can easily interpret. We then call operating systems commands using Python's sub-process library to locally execute the the LOANDRA command to solve the problem. We then parse and save the outputs from the OS terminal that detail the solution results. It is important to note that LOANDRA must be installed on the user's device to use this method and the installation directory must be provided as an argument into the respective solve method.

Both solver methods produce an output that includes a SAT solution array, marking the assignment of variables as either true (1) or false (0). In cases where the objective is to determine the minimum height, the system returns the smallest depth identified that satisfied the SAT formulae modelled after the dataset through iterative processing, whereas the max accuracy modules provide the minimum cost for a fixed depth given by the user. Additionally for classification problems, we generate fully decoded tree structure objects detailing the feature selections, threshold values at branching nodes, assigned labels at the leaves, and a visual representation of the tree is also outputted for improved interpretability. Classification model outputs are designed to be compatible with scikit-learn such that it can be passed in to their methods for measuring metrics such as model accuracy and other

[1]For more details of the LOANDRA solver, visit `https://github.com/jezberg/loandra`

reports. For clustering problems, we return a dictionary of clustering groups that each data point belongs to, as well as an auxiliary dictionary that lists the maximum diameter per clustering group. All problems also return a literal map, that provides a dictionary of the realized truth values mapped to the variables discussed in Section 4.2 and Section 5.2.

The code excerpt below demonstrates the general process for setting up and solving a problem. We use the maximum accuracy classification problem for numerical feature datasets as a demonstration of this approach.

```
## Example: Maximum accuracy problem with only numerical features
max_accuracy_numerical_problem = SATreeCraft(dataset=dataset,features=
    ↪ features,labels=labels,true_labels_for_points=true_labels_for_points
    ↪ ,classification_objective='max_accuracy',fixed_depth=2)

# OPTION 1 for solving: Solve using PySATs default Solver
max_accuracy_numerical_problem.solve()


# OPTION 2 for solving: Solve using LOANDRA
loandra_path = 'directory/to/LOANDRA/Installation'
max_accuracy_numerical_problem.solve_loandra(loandra_path= loandra_path)

print("Final Model: ", max_accuracy_numerical_problem.model)
print("Min cost found: ", max_accuracy_numerical_problem.min_cost)
```

Code Excerpt 6.5: Example: SAT problem construction and solving

The full outputs saved as attributes of the main module upon solving each type of problem are provided below:

- **SATreeCraft.sat_solution()**: Returns an array of literals that were assigned as true or false
- **SATreeCraft.final_literals()**: Returns a dictionary of the literals within the SAT solution mapped to tree variables.
- **SATreeCraft.model()**: Describes the tree structure model after solving a classification problem. Object that stores all tree nodes, thresholds and features selected per branch nodes, as well as labels assigned to leaf nodes.
- **SATreeCraft.min_depth()**: For the minimum height problem - returns the smallest depth found to perfectly classify all training datapoints

- **SATreeCraft.min_cost()**: For the maximum accuracy problem - returns the cost of the solution (number of misclassified points)
- **SATreeCraft.cluster_assignments()**: For clustering problems - returns a dictionary of the datapoints within each cluster group
- **SATreeCraft.cluster_diameters()**: For clustering problems - returns a dictionary of the maximum diameter within each cluster group

6.2 Decoding Solutions

6.2.1 Decoding Classification Tree Solutions

Our framework upon finding an optimal solution presents a user with tree visualizations for enhanced understanding of model reasoning. The images are automatically produced and saved in the *images* directory within the repository upon solving. There are two types of trees that can be produced based on whether or not categorical features are present. Figures 6.3 and 6.4 highlight these differences:

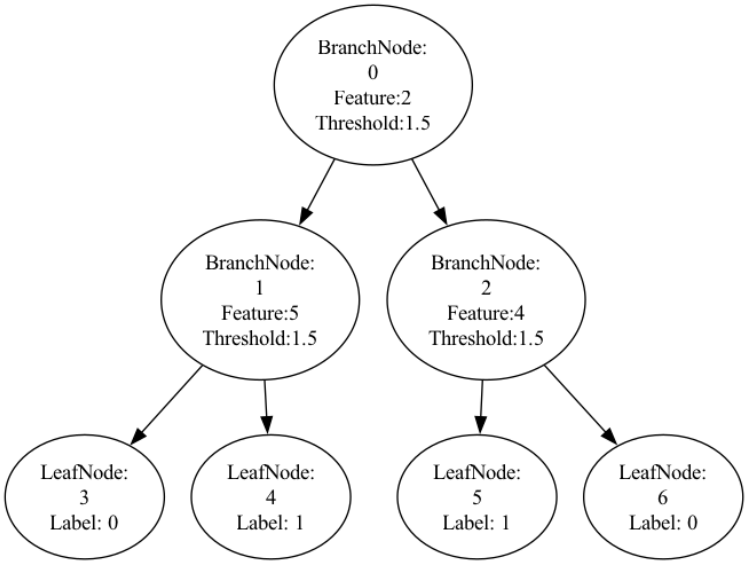

Figure 6.3: Decoded decision tree solution with numerical/binary features

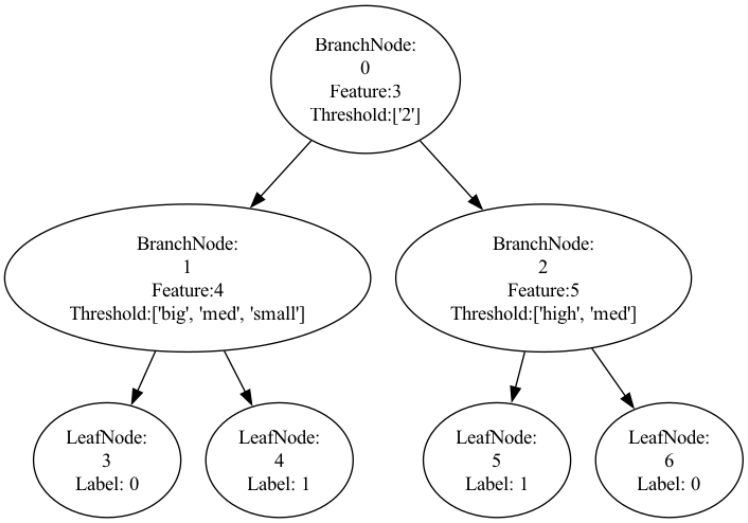

Figure 6.4: Decoded decision tree solution with categorical features

6.2.2 Decoding Feature Selection and Leaf Label Assignment

To decode the overall structure of the tree we use the SAT solution and literals data structure we had created prior when constructing the tree structure for a given depth:

1. Features selected at branch nodes can be determined by examining which $a_{t,j}$ variables evaluated as **true** in our SAT solution.

2. Labels assigned to leaf nodes can be determined by examining which $g_{t,c}$ variables evaluated as **true** in our SAT solution.

6.2.3 Decoding Thresholds for Numerical Features

To decode the threshold for branch nodes that have numerical features, we must observe the feature values for each data point as well as the solution assignments for all $s_{i,t}$ variables[5].To establish the threshold for numerical feature branch nodes, the data points are first sorted by their feature values in ascending order, and the $s_{i,t}$ index variables are arranged to match this order. The threshold is then determined by locating the point where $s_{i,t}$ changes from true to false and computing the average of the two adjacent feature values at this junction. This average becomes the branch node's splitting threshold, delineating the dataset according to the numerical feature.

6.2.4 Decoding Power Sets for Categorical Features

The power set branching method previously outlined simplifies the threshold determination for categorical features in a branch node [5]. The power set of a branch node consists of the training feature values that were directed left at the node. To determine these sets, we examine the $s_{i,t}$ variables that are true, indicating the data points that branched left of a node. For the selected categorical feature j_C at some node, we compile an array of the values part of $dom(j_C)$ that correspond to the left-directed data points.

6.2.5 Decoding Clustering Solutions

In the decoding process for clustering solutions, we use unary encoding with the literals $x_{i,c}$. Each data point x_i's cluster is represented by a sequence of binary values corresponding after solving , where c would represent that each data point x_i is in that cluster or comes after it. The sequence's pattern of 0s and 1s indicates the cluster for that data point. Data points are assigned to the same cluster if their sequences match.

For simpler explanation, consider the following example matrix for a set of literals after finding the SAT solution to some clustering problem:

$$
\begin{aligned}
x_{1,c}: &\quad 1 \quad 0 \quad 0 \\
x_{2,c}: &\quad 1 \quad 1 \quad 0 \\
x_{3,c}: &\quad 1 \quad 1 \quad 0 \\
x_{4,c}: &\quad 1 \quad 1 \quad 1 \\
x_{5,c}: &\quad 1 \quad 1 \quad 1
\end{aligned}
$$

Figure 6.5: Decoding clustering solutions

In this solved solution, we can see that there are 3 clusters because there are 3 unique sequences of 1's and 0's. Here, the first column represents the potential inclusion in the first cluster, the second column in the second cluster, and so on. The first data point represented by the first row, with the sequence '100', forms a unique cluster. Data points 2 and 3, with the sequence '110', are grouped together in a second cluster. Similarly, data points 4 and 5 share the sequence '111', placing them in a third cluster. We employ this method to allow us to precisely determine cluster assignments directly from the SAT solver's output.

To further improve the simplicity of our library, if a user has clustered a dataset with a feature dimensionality of 1 or 2, we provide simplistic visualizations, that is automatically produced and saved upon finding a solution to the *images* directory of the repository.

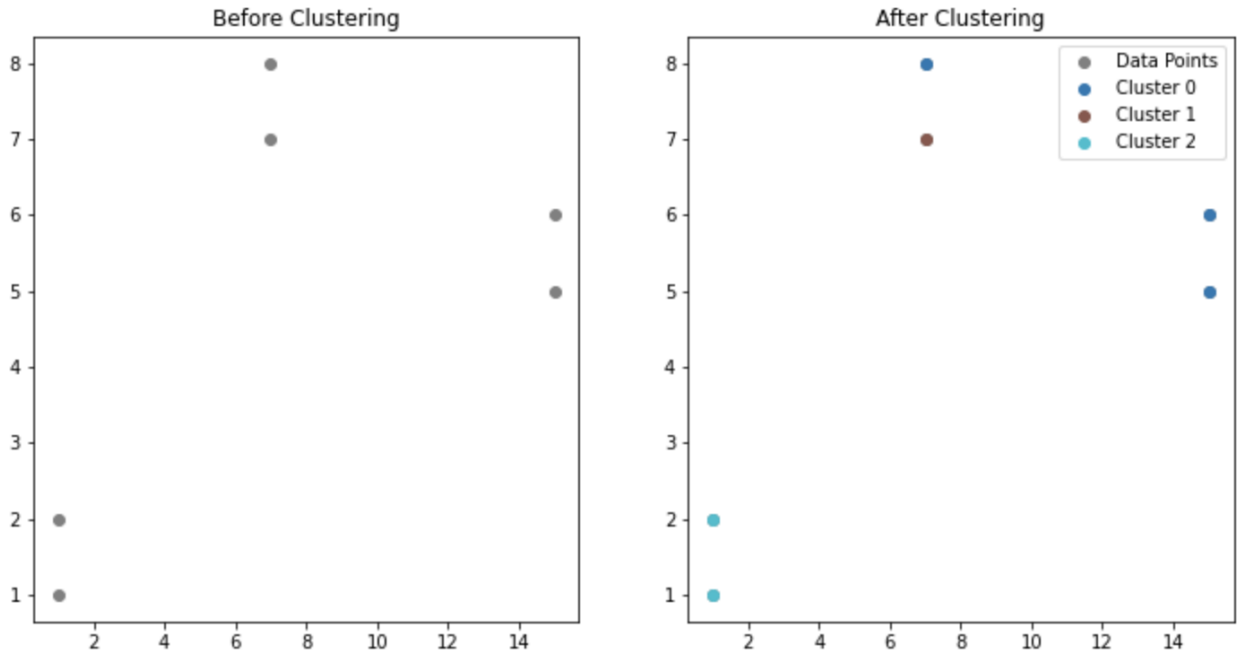

Figure 6.6: Clustering visualization for decoded SAT solution for Feature size = 2

6.3 Scikit-learn Integration

To enhance the usability of our SAT models we have integrated the popular scikit-learn library as an interface for tree model evaluation.

6.3.1 SATreeClassifier()

The SATreeClassifer sub-module we have built is used to transform the model object produced by the main module into a tree estimator to be used on out of sample data. Simply the user should input the model into this wrapper, to which after the model can be used to make predictions on out of sample data. We have included many methods from scikit-learn into this interface such as confusion matrices, F1-score, precision and more that the user can call[2]. The code excerpt below highlights how to initialize the object as well how to use some of its methods. We continue to use the maximum accuracy problem we defined earlier as illustration.

[2]For more information on methods visit sckit-learn `https://scikit-learn.org/stable/modules/generated/sklearn.metrics.classification_report.html`Footnote

```
# Classifier - sklearn Integration
# Build model
from SATreeCraft import SATreeClassifier
model = SATreeClassifier(max_accuracy_numerical_problem.model)

# EXAMPLE TEST DATA
X_test = dataset[5:160]
y_test = true_labels_for_points[5:160]

# To get the accuracy score - use the SATreeClassifier.score() method
accuracy = model.score(X_test, y_test)
print(f"Accuracy: {accuracy}")

# To get the classification report
# prints out recall, f1, support
report = model.get_classification_report(X_test, y_test)
print("Classification Report:")
print(report)

# To get the confusion matrix
conf_matrix = model.get_confusion_matrix(X_test, y_test)
print("Confusion Matrix:")
print(conf_matrix)
############################## Output ###################################
Accuracy: 0.967741935483871
Classification Report:
              precision    recall  f1-score   support

           0       0.95      0.98      0.96        54
           1       0.97      0.96      0.96        71
           2       1.00      0.97      0.98        30

    accuracy                           0.97       155
   macro avg       0.97      0.97      0.97       155
weighted avg       0.97      0.97      0.97       155

Confusion Matrix:
[[53  1  0]
 [ 3 68  0]
 [ 0  1 29]]
```

Code Excerpt 6.6: Building predictive models from SAT solutions

6.3.2 Other Utilities

Another useful tool we have built out is a k-fold accuracy tester function, where a user is able to specify the number train test splits they would like and it will compute a model on each split. The function returns the test accuracy for each split in an array as well as a mean performance. We have built two variants of this to support both solving problems via LOANDRA and the default PySAT solvers. These are single functions where a user can provide all dataset information and constraints without having to initialize the main module.

```
from SATreeCraft.utils import k_fold_tester, k_fold_tester_loandra
# k fold methods
k = 5
depth = 2
minimum_support = 0

# default solver variant
accuracies, mean_score = k_fold_tester(k=k, depth=depth, dataset=dataset,
    ↪ true_labels_for_points=true_labels_for_points,  labels=labels,
    ↪ features=features, min_support_level=minimum_support)

# LOANDRA solver variant
accuracies, mean_score = k_fold_tester_loandra(k=k, depth=depth, dataset=
    ↪ dataset, true_labels_for_points=true_labels_for_points,  labels=
    ↪ labels, features=features, min_support_level=minimum_support)

print("K-fold: ", accuracies)
print("Mean Accuracy: ", mean_score)

################################## Output ###################################
K-fold : [0.94444444 0.94444444 0.88888889 0.88571429 0.94285714]
Mean Accuracy: 0.9212698412698412
```

Code Excerpt 6.7: Building predictive models from SAT solutions

7 Experiments for Software

7.1 Experiment Set-Up

This section outlines the testing methodology applied to our software, where we construct decision trees across a wide range of datasets, differing in size and types of features to solve all types of problems described. We validate our software by comparing our time and cost of solutions to the results seen in [5], and present new results for new classification constraints. We also present optimization results for our clustering objective. One key measure of interest is the *Total Time to Solve* which we define in Eq. 26:

$$Total\ Time(s) = t_{Build\ Clauses} + t_{Solve\ SAT\ Formula} + t_{Decode\ SAT\ Solution} \tag{26}$$

Please note that there can be discrepancies in time between our library and the reference papers [5,6] due to factors in hardware. The hardware specifications are slightly different as [5,6] performs its tests on two 12-core Intel E5-2697v2 CPUs and 128G of ram, whereas our tests were performed on an Intel Core i9 CPU and 16GB of ram. For a few tests, we record the time it takes for both the default PySAT solvers and LOANDRA. Please note that the reference papers use LOANDRA for all MaxSAT problems, and uses the MiniSAT solver for standard CNF problems such as minimum height.

7.2 Minimum Height Tree Problem Tests

The software's performance is verified by its ability to identify the most optimal minimum-height decision tree for a given training dataset paralleling the approach taken in [5]. We compare our solutions to [5] to validate our designs, as well as report the total time it took to find the solutions in order to compare efficiency and performance. We record both time it takes to solve using external solvers and the default.

7.3 Maximum Accuracy Tree Problem Tests

We find the optimal solution for the maximum accuracy objective at three distinct tree depths: 2, 3, and 4 for a given dataset and compare it to the test results in [5]. We record the minimum cost found as well as the time it takes to solve the problem.

7.4 Classification User Constraints Tests

We execute three distinct sets of experiments on the maximum accuracy problem with variations. In the first series, for the predetermined depth levels used in the original problem,

we introduce a minimum support constraint with five varying settings, where the parameter S is set to $\{0, 2, 5, 10\}$. A 5-fold cross-validation is performed on different training-test splits to assess accuracy. We document the total elapsed time for completing these the 5-fold experiment as well as the mean accuracy for each configuration and compare it to the results see in [5].

The second set of tests is similar to the minimum support constraint test, however, we now apply the minimum margin constraint where M varies from $\{1, 4, 10, 20\}$ as done in [5], and record the mean accuracy and total time to completion.

Lastly for classification, we test the accuracy and time to build oblivious tree solutions that has never been presented before.

7.5 Clustering Tests

For clustering tests, we evaluate datasets by setting the number of clusters to three and experimenting with various values of the approximation parameter ε, specifically {0, 0.1, 0.5, 0.75, and 1.0}. We record the maximum diameter for each cluster and the total time for the decoding and printing out the solution.

7.6 Datasets

To test the software we have built is correct, we use the same datasets for computing minimal heights and fix depth cost solutions that are used in [5] that is retrieved from [25]. These datasets' features and sizes are reported in Table 7.1. Please note that we use the same data manipulation techniques presented in the original paper for certain datasets in order to verify our solutions. $|X|$ reports total size of the dataset, $|F_N|$ reports the number of numerical features, $|F_B|$ reports the number of binary features, and $|F_C|$ reports the number of categorical features. $|C|$ represents the total number of distinct classes present in the dataset.

Table 7.1: Dataset characteristics

Type	Name	$\|X\|$	$\|F_N\|$	$\|F_B\|$	$\|F_C\|$	$\|C\|$
N	Banknote	1372	4	0	0	2
	Breast Cancer	116	9	0	0	2
	Cryotherapy	90	5	1	0	2
	Immunotherapy	91	6	1	0	2
	Ionosphere	351	32	2	0	2
	Iris	150	4	0	0	3
	User Knowledge	258	5	0	0	4
	Vertebral Column	310	6	0	0	2
	Wine	178	13	0	0	3
C	Credit Approval[†]	653	6	4	5	2
	Promoter	106	0	0	57	2
	Soybean Large[†]	266	5	16	14	15
	Protease Cleavage	746	0	0	8	2
	Protease Cleavage(/4)[*]	186	0	0	8	2
B	Car[‡]	1728	6	0	0	2
	Monk2	169	4	2	0	2

[†] Records with missing values were removed.

[‡] Classes were merged as in [5].

[*] 25% of the original dataset is used [5].

8 Results

8.1 Minimum Height Problem Results

For this objective we follow the same rules as in [5], that we give the system 30 minutes to find and decode a solution and record the minimal depth. If the system cannot find a solution in the time limit, we use the notation $T/O[d]$ to indicate the tree depth level the iterative solver was attempting to solve by the end of the process. If a solution was not found within or even beyond the time limit due to extended durations, we denote the height as "?". As seen in Table 8.1 and Table 8.2 we see that our Python library has been verified as our solutions are the same depths as our reference. However, we have noticed that there are slight differences in speed potentially due to the differences in hardware. It can be seen that LOANDRA takes more time to solve, which can be explained by the fact that LOANDRA was designed to solve for MaxSAT problems with weighted CNF formulas, whereas this problem is a standard CNF.

Table 8.1: Experimental results of min-height problem for numerical feature datasets

Data Set	**Type**	**Python Library**			**[5]**	
		Min Depth	**Time(s)**	**LOANDRA Time(s)**	**Min Depth**	**Time(s)**
Banknote	N	4	9	180	4	5.82
Breast Cancer	N	4	2.9	42	4	6.59
Cryotherapy	N,B	4	0.3	1.8	4	0.08
Immunotherapy	N,B	4	0.5	3.6	4	0.18
Ionosphere	N,B	?	T/O[4]	T/O	?	T/O[4]
Iris	N	4	0.5	1.4	4	0.04
User Knowledge	N	5	3	120	5	1.31
Vertebral Column	N	5	100	240	5	87.35
Wine	N	3	0.45	2.2	3	0.11
Monk2	B	6	3	25	6	2.73
Car	B	8	T/O[7]	T/O	8	T/O[8]

Table 8.2: Experimental results of min-height problem for categorical feature datasets

Data Set	**Type**	**Python Library**			**[5]**	
		Min Depth	**Time(s)**	**LOANDRA Time(s)**	**Min Depth**	**Time(s)**
Promoter	C	4	180	320	4	224
Protease	C	?	T/O[5]	T/O[5]	?	T/O[5]
Protease(/4)	C	4	1	3	4	0.69
Soybean Large	C	?	T/O[6]	T/O[6]	?	T/O[6]
Credit Approval	C,N,B	?	T/O[5]	T/O[6]	?	T/O[6]

8.2 Max Accuracy Problem Results

For the max accuracy problem we use a 15-minute time limit as in [5]. We report the cost that was found as well as the time it took for each specified depth. If a solution is not found in time, we report the time as T/O, and if a solution is never found due to extended durations we report the cost as "?". Tables 8.3 and 8.4 prove that our software produces the same solutions as [5], but with longer solving times for larger datasets when using the default PySAT solvers. However, utilizing LOANDRA, our library finds solutions in times that are the same, or even surpass, the efficiency of the original implementation seen in[5].

Table 8.3: Results of fixed depth problem of categorical feature datasets

Data Set	**Type**	**Depth**	**Python Library**			**[5]**	
			Time(s)	**Cost**	**LOANDRA Time(s)**	**Time(s)**	**Cost**
Promoter	C	2	325	12	325	316.71	12
		3	T/O	3	T/O	T/O	3
		4	12	0	9.5	7.83	0
Protease Cleavage	C	2	T/O	98	T/O	T/O	98
		3	T/O	101	T/O	T/O	101
		4	T/O	39	T/O	T/O	39
Protease Cleavage(/4)	C	2	6.2	13	6.2	18.9	13
		3	2.6	1	2.6	7.37	1
		4	1.1	0	1.1	1.27	0
Soybean Large	C, N, B	2	180	152	20	16.22	152
		3	T/O	104	T/O	T/O	104
		4	T/O	35	T/O	T/O	35
Credit Approval	C, N, B	2	T/O	84	65	106	84
		3	T/O	76	T/O	T/O	76
		4	T/O	60	T/O	T/O	60

Table 8.4: Results of fixed depth problem of numerical feature datasets

Data Set	**Type**	**Depth**	**Python Library**			**[5]**	
			Time(s)	**Cost**	**LOANDRA Time(s)**	**Time(s)**	**Cost**
Banknote	N	2	298	100	15.3	16.83	100
		3	363	23	74	105.79	23
		4	14.5	0	19.7	18.98	0
Breast Cancer	N	2	10	19	4.7	5.07	19
		3	500	9	420	242	9
		4	30	0	18.2	20	0
Cryotherapy	N,B	2	0.2	5	0.30	0.57	5
		3	0.2	1	0.40	0.73	1
		4	0.3	0	0.70	0.75	0
Immunotherapy	N,B	2	0.3	8	0.30	0.99	8
		3	2	4	2.0	3.81	4
		4	0.5	0	1.0	1.27	0
Ionosphere	N,B	2	400	29	115	155.06	29
		3	T/O	21	T/O	T/O	21
		4	T/O	10	T/O	T/O	10
Iris	N	2	0.1	6	0.40	0.6	6
		3	0.1	1	0.40	0.77	1
		4	0.4	0	0.70	0.82	0
User Knowledge	N	2	2	35	0.70	1.94	35
		3	4	10	3.2	3.29	10
		4	4	1	3.3	3.86	1
Vertebral Column	N	2	120	45	10	15.79	45
		3	T/O	32	T/O	T/O	32
		4	T/O	15	T/O	T/O	15
Wine*	N	2	0.5	6	0.70	1.25	6
		3	0.5	0	1.1	1.62	0
		4	-	-	-	-	-
Monk2	B	2	4.1	57	0.90	2.74	57
		3	T/O	42	T/O	T/O	42
		4	T/O	32	T/O	T/O	32
Car	B	2	T/O	250	14.34	12.67	250
		3	T/O	180	T/O	T/O	180
		4	T/O	122	T/O	T/O	122

* A depth of 3 for the Wine dataset perfectly classifies all training examples

8.3 Classification User Constraints Results

We report both the impact on accuracy results and time to solve when adding the minimum margin and minimum support constraints to SAT formulae for the fixed height problem. Table 8.5 presents the 5-fold cross-validation mean accuracy results for various settings of the minimum margin constraint, and Table 8.6 shows the mean validation accuracy for the minimum support constraint. The results align with the original implementations, though there are some discrepancies due to potential variations in the train-test splits used in our tests. We utilized the LOANDRA solver to record results for these problems, as we determined that LOANDRA was more effective for MaxSAT problems in the previous section. Please note that we report results for only a subset of the datasets, as many from the original reference paper require several hours to complete single tests, which is beyond the capabilities of our available hardware. However, the datasets we were able to test, show a quite strong correspondence with the expected outcomes. We do not set a time limit for these tests.

Table 8.5: Minimum margin constraint impact on model accuracy

Dataset	**Depth**	**Avg Test Accuracy**				**Total Time** (s)				[5] **Avg Test Accuracy**				[5] **Total Time** (s)			
		M =1	M = 4	M = 10	M = 20	M =1	M = 4	M = 10	M = 20	M =1	M = 4	M = 10	M = 20	M =1	M = 4	M = 10	M = 20
Banknote	2	0.91	0.91	0.91	0.91	44.37	45.75	43.89	42.35	0.91	0.91	0.91	0.91	71.8	62.9	64.09	62.68
	3	0.98	0.97	0.97	0.97	196.94	148.25	121.41	167.95	0.97	0.97	0.97	0.97	274.42	264.88	191.61	209.57
	4	0.99	0.99	0.99	0.99	53.29	54.35	46.07	47.18	0.98	0.98	0.98	0.98	45.15	44.69	40.65	44.04
Iris	2	0.93	0.94	0.94	0.92	1.38	1.23	1.13	1.12	0.93	0.92	0.9	0.93	1.09	1.06	1.03	1.16
	3	0.93	0.95	0.93	0.95	1.9	2.17	1.79	2.13	0.93	0.95	0.94	0.95	1.46	1.48	1.38	1.4
	4	0.95	0.95	0.91	0.92	3.8	4.2	6.28	6.14	0.93	0.93	0.93	0.93	1.93	1.95	1.83	1.88
Breast Cancer	2	0.76	0.69	0.74	0.76	13.24	9.15	7.47	5.23	0.74	0.75	0.75	0.72	15.27	14.58	8.8	7.91
	3	0.7	0.74	0.75	0.72	220.52	287.29	351.01	182.41	0.77	0.73	0.73	0.7	372.61	317.35	276.37	319.79
	4	0.64	0.65	0.63	0.63	17.59	14.6	9.7	7.9	0.62	0.66	0.65	0.6	19.34	16.87	10.4	7.34

Table 8.6: Minimum support constraint impact on model accuracy

Dataset	**Depth**	**Avg Test Accuracy**				**Total Time** (s)				[5] **Avg Test Accuracy**				[5] **Total Time** (s)			
		S = 0	S = 2	S = 5	S = 10	S = 0	S = 2	S = 5	S = 10	S = 0	S = 2	S = 5	S = 10	S = 0	S = 2	S = 5	S = 10
Banknote	2	0.91	0.92	0.92	0.91	51.59	150.38	229.71	242.28	0.91	0.91	0.91	0.91	71.00	68.23	87.46	118.32
	3	0.97	0.97	0.97	0.97	242.30	807.70	498.02	773.00	0.97	0.97	0.98	0.97	267.00	305.00	467.74	406.70
	4	0.98	0.98	0.98	0.98	86.07	165.65	257.95	303.12	0.98	0.98	0.98	0.99	44.61	167.80	354.03	702.00
Iris	2	0.93	0.93	0.93	0.93	1.29	5.65	5.90	4.55	0.93	0.92	0.91	0.95	1.13	2.88	3.83	5.24
	3	0.95	0.93	0.93	0.93	2.61	11.68	9.60	11.22	0.93	0.95	0.92	0.96	1.45	4.94	7.65	12.88
	4	0.94	0.91	0.95	0.93	3.97	20.55	15.91	12.89	0.93	0.93	0.92	INF	1.76	9.39	26.93	4.07
Breast Cancer	2	0.73	0.78	0.72	0.77	18.32	23.50	19.50	27.40	0.74	0.75	0.75	0.75	14.53	19.36	15.91	24.17
	3	0.77	0.74	0.81	73.00	964.55	876.73	793.20	892.00	0.77	0.71	0.71	0.71	372.00	319.00	824.00	3039.00
	4	0.74	0.69	0.72	0.63	16.05	21.61	35.00	49.53	0.62	0.56	0.65	INF	18.99	53.97	4511.00	203.00

8.4 Oblivious Tree Results

We present the model performance on all dataset for the new oblivious tree constraint encodings that is unique to SATreeCraft. We set a time limit for 5 hours on this problem, and apply this test only for the max accuracy problem for a given depth. We utilize the same depths as seen in previous tests. If the 5-fold accuracy is not reported in the time limit we report it as "??" and represent the timeout as T/O.

Table 8.7: Oblivious tree accuracy results on categorical feature datasets

Dataset	**Depth**	**Avg Test Accuracy**	**Total Time (s)**
Credit Approval	2	0.85	82
	3	??	T/O
	4	??	T/O
Protease Cleavage	2	0.83	638.6
	3	??	T/O
	4	??	T/O
Protease Cleavage (/4)	2	0.75	12.72
	3	0.72	8.86
	4	0.63	5.66
Promoter	2	0.77	77
	3	??	T/O
	4	??	T/O
Soybean Large	2	0.41	68.5
	3	??	T/O
	4	??	T/O

Table 8.8: Oblivious tree accuracy results on numerical feature datasets

Dataset	**Depth**	**Avg Test Accuracy**	**Total Time** (s)
Banknote	2	0.92	38.46
	3	0.97	125.89
	4	0.99	193.79
Breast Cancer	2	0.64	14.33
	3	0.68	2824
	4	0.66	15071
Cryo Therapy	2	0.90	1.76
	3	0.86	3.22
	4	0.87	2.95
Immuno Therapy	2	0.82	1.43
	3	0.78	9.30
	4	0.73	3.68
Ionosphere	2	0.90	44.00
	3	??	T/O
	4	??	T/O
Iris	2	0.93	2.37
	3	0.95	2.08
	4	0.94	3.92
User Knowledge	2	0.86	4.02
	3	0.92	10.71
	4	0.93	32.35
Vertebral Column	2	0.80	12.61
	3	0.80	3392
	4	??	T/O
Wine	2	0.88	4.31
	3	0.93	6.05
	4	0.89	6.75
Car	2	0.86	35.00
	3	??	T/O
	4	??	T/O
Monk	2	0.55	2.53
	3	0.59	2.53
	4	0.62	1544.88

8.5 Clustering Solution Results

In this section we examine the trade off between time to solve and optimality of the maximum diameter problem upon adjusting our ε parameter. We use a subset of the datasets that were used in [6] as proof of functionality of our clustering mechanisms. We give 15 minutes for the each dataset to complete all depths. If the test could not complete within time we report '??' for accuracy and "T/O" for time. We report the euclidean distance as the measure of maximum diameter between points in each cluster $k_i \in K$.

It can be seen as we increase the threshold parameter in most cases, the time to solve the problem is significantly reduced in most cases of the datasets. However, as seen in each table for the given values we have tested, the value of ε can have negligible impact on producing optimal solutions (as seen with the wine dataset) or can significantly reduced the optimality of the maximum diameters found (such as the ionosphere dataset). However, it is important to note that it can take severely long times to find solutions to much larger datasets with varying features if no ε parameter is initialized, as seen with the Ionosphere dataset.

Table 8.9: $K = 3$ clusters, $\varepsilon = 0$

Dataset	**Depth**	k_1	k_2	k_3	**Time** (s)
Wine	2	445.67	458.13	455.00	42.00
	3	445.67	458.13	455.00	50.00
	4	445.67	458.13	455.00	107.00
Iris	2	2.70	2.45	2.60	15.00
	3	2.58	2.55	2.42	15.30
	4	2.57	2.58	2.42	16.00
Ionosphere	2	?	?	?	T/O
	3	?	?	?	T/O
	4	?	?	?	T/O

Table 8.10: $K = 3$ clusters, $\varepsilon = 0.1$

Dataset	**Depth**	k_1	k_2	k_3	**Time** (s)
Wine	2	445.67	458.13	455.00	16.75
	3	445.67	458.13	455.00	15.95
	4	445.67	458.13	455.00	17.70
Iris	2	2.70	2.64	2.60	0.87
	3	2.57	2.58	2.42	1.16
	4	2.62	2.58	2.42	2.01
Ionosphere	2	7.98	8.01	8.03	5.45
	3	8.01	8.01	8.02	7.30
	4	8.01	8.03	8.01	10.93

Table 8.11: $K = 3$ clusters, $\varepsilon = 0.5$

Dataset	**Depth**	k_1	k_2	k_3	**Time** (s)
Wine	2	445.67	458.13	455.00	5.52
	3	445.67	458.13	455.00	6.46
	4	445.67	458.13	455.00	7.67
Iris	2	2.95	2.65	2.93	1.09
	3	2.97	2.65	2.93	1.55
	4	2.97	2.26	2.60	2.12
Ionosphere	2	8.50	8.50	8.30	5.38
	3	8.50	8.10	8.40	6.65
	4	8.50	8.50	8.40	11.04

Table 8.12: $K = 3$ clusters, $\varepsilon = 0.75$

Dataset	**Depth**	k_1	k_2	k_3	**Time** (s)
Wine	2	445.67	458.13	455.00	4.73
	3	445.67	458.13	455.00	5.11
	4	445.67	458.13	455.00	5.92
Iris	2	2.95	2.70	2.93	0.84
	3	2.95	2.80	2.92	1.25
	4	2.87	2.80	2.43	2.05
Ionosphere	2	8.00	8.00	8.10	4.96
	3	8.20	8.10	7.70	7.07
	4	8.10	8.10	8.20	11.30

Table 8.13: $K = 3$ clusters, $\varepsilon = 1$

Dataset	**Depth**	k_1	k_2	k_3	**Time** (s)
Wine	2	445.67	458.13	455.00	5.33
	3	445.67	458.13	455.00	4.76
	4	445.67	458.13	455.00	6.23
Iris	2	2.97	2.25	2.61	0.78
	3	2.90	2.97	2.42	1.21
	4	2.65	2.97	2.93	2.00
Ionosphere	2	8.86	8.38	8.25	4.85
	3	8.67	8.90	8.90	6.35
	4	8.60	9.00	8.30	9.95

9 Conclusion

In this design study, we successfully implemented a powerful toolkit that enhances both the usability and accessibility of SAT, aiming to engage a broader range of data science and machine learning practitioners. We have enabled users to solve a wide range of problems within both classification and clustering. We have constructed a library that includes robust encodings of those decision tree problems to ensure exact optimal solutions, created a modular design that is simple to use , and we have enabled user customization for adding additional constraints onto user problems. The library's implementation has also been verified, as we have been able to match the results for both time efficiency and model performances in the original source papers that we base our design off of. Exclusive to our library, we have added new features that extends from source material such as adding new tree structure constraints.

There are many avenues for future work that can be taken to enhance the current state of this library. The primary addition that should be implemented is the bi-criteria optimization objective in clustering that finds the pareto optimal solution when optimizing for both maximum diameter and minimum split. Another addition that could be focused upon is the implementation of oblivious trees that have both the same feature and threshold selected at each depth level of a tree. Finally, there can be more improvements to the current code base in terms of cleanliness.

References

[1] A. Navada, A. N. Ansari, S. Patil, and B. A. Sonkamble, "Overview of use of decision tree algorithms in machine learning," 2011 IEEE Control and System Graduate Research Colloquium, pp. 37–42, 2011. doi:10.1109/icsgrc.2011.5991826

[2] L. Hyafil and R. L. Rivest, "Constructing optimal binary decision trees is NP-complete," Information Processing Letters, vol. 5, no. 1, pp. 15–17, May 1976. doi:10.1016/0020-0190(76)90095-8

[3] G. Nanfack, P. Temple, and B. Frénay, "Constraint enforcement on decision trees: A survey," ACM Computing Surveys, vol. 54, no. 10s, pp. 1–36, Sep. 2022. doi:10.1145/3506734

[4] N. Narodytska, A. Ignatiev, F. Pereira, and J. Marques-Silva, "Learning optimal decision trees with sat," Proceedings of the Twenty-Seventh International Joint Conference on Artificial Intelligence, 2018. doi:10.24963/ijcai.2018/189

[5] P. Shati, E. Cohen, and S. A. McIlraith, "SAT-based optimal classification trees for non-binary data," Constraints, vol. 28, pp. 166-202, 2023. [Online]. Available: https://doi.org/10.1007/s10601-023-09348-1

[6] P. Shati, E. Cohen, and S. McIlraith, "Optimal decision trees for interpretable clustering with constraints," Proceedings of the Thirty-Second International Joint Conference on Artificial Intelligence, 2023. doi:10.24963/ijcai.2023/225

[7] Leo Breiman, Jerome H Friedman, Richard A Olshen, and Charles J Stone. Classification and regression trees. Wadsworth & Brooks/Cole Advanced Books & Software, 1984. (accessed Jan. 7, 2024).

[8] J. R. Quinlan, "Induction of decision trees - machine learning," SpringerLink, https://link.springer.com/article/10.1007/BF00116251 (accessed Jan. 7, 2024).

[9] C. Bessiere, B. O'Sullivan, and E. Hebrard, "Minimising Decision Tree size as Combinatorial Optimisation," ResearchGate, `https://www.researchgate.net/publication/221632866_Minimising_Decision_Tree_Size_as_Combinatorial_Optimisation` (accessed Jan. 8, 2024).

[10] A. Schidler and S. Szeider, "Sat-based decision tree learning for large data sets," Proceedings of the AAAI Conference on Artificial Intelligence, vol. 35, no. 5, pp. 3904–3912, 2021. doi:10.1609/aaai.v35i5.16509

[11] M. Bilenko, S. Basu, and R. J. Mooney, "Integrating constraints and metric learning in semi-supervised clustering," Twenty-first international conference on Machine learning - ICML '04, Jul. 2004. doi:10.1145/1015330.1015360

[12] I. Davidson, S. S. Ravi, and L. Shamis, "A Sat-based Framework for Efficient Constrained Clustering," Proceedings of the 2010 SIAM International Conference on Data Mining, 2010. doi:10.1137/1.9781611972801.9

[13]] P. Vossler, S. Aghaei, N. Justin, and N. Jo, "ODTlearn: A package for learning optimal decision trees for ... - arxiv.org," arxiv, https://arxiv.org/pdf/2307.15691.pdf (accessed Jan. 8, 2024).

[14] G. Aglin, S. Nijssen, and P. Schaus, "Pydl8.5: A library for Learning Optimal Decision Trees," Proceedings of the Twenty-Ninth International Joint Conference on Artificial Intelligence, 2020. doi:10.24963/ijcai.2020/750 (accessed Jan. 8, 2024).

[15] A. Biere, M. Heule, and H. Maaren, "Handbook of Satisfiability," Frontiers in Artificial Intelligence and Applications, vol. 185, 2009. doi:10.3233/faia336

[16] Zhaohui Fu and Sharad Malik. "On solving the partial MAX-SAT problem," International Conference on Theory and Applications of Satisfiability Testing (SAT), pages 252–265. Springer, 2006.

[17] F. Avellaneda, "Efficient inference of Optimal Decision Trees," Proceedings of the AAAI Conference on Artificial Intelligence, vol. 34, no. 04, pp. 3195–3202, 2020. doi:10.1609/aaai.v34i04.5717

[18] H. Hu, M. Siala, E. Hebrard, and M.-J. Huguet, "Learning optimal decision trees with maxsat and its integration in AdaBoost," Proceedings of the Twenty-Ninth International Joint Conference on Artificial Intelligence, 2020. doi:10.24963/ijcai.2020/163

[19] H. Verhaeghe, S. Nijssen, G. Pesant, C.-G. Quimper, and P. Schaus, "Learning optimal decision trees using constraint programming," Proceedings of the Twenty-Ninth International Joint Conference on Artificial Intelligence (IJCAI-20) Sister Conferences Best Papers Track, vol. 25, no. 3–4, pp. 226–250, 2020. doi:10.1007/s10601-020-09312-3

[20] Gaël Aglin, Siegfried Nijssen, and Pierre Schaus. Learning optimal decision trees using caching branch-and-bound search. In AAAI Conference on Artificial Intelligence (AAAI), pages 3146–3153, 2020.

[21] L. Castin and B. Frenay, "Clustering with decision trees: Divisive and agglomerative approach," esann, https://www.semanticscholar.org/paper/Clustering-with-Decision-Trees%3A-Divisive-and-Castin-Fr%C3%A9nay/93475c4ce36dece5711627cc8c48520a95477eb3 (accessed Jan. 8, 2024).

[22] A. Ignatiev, J. Marques-Silva, and A. Morgado, "SAT technology in python," PySAT, https://pysathq.github.io/ (accessed Jan. 8, 2024).

[23] C. Sinz, "Towards an optimal CNF encoding of boolean cardinality constraints," Principles and Practice of Constraint Programming - CP 2005, pp. 827–831, 2005. doi:10.1007/11564751_73

[24] P. Langley and S. Sage, "Oblivious Decision Trees and Abstract Cases," cdn.aaai.org, https://cdn.aaai.org/Workshops/1994/WS-94-01/WS94-01-020.pdf (accessed Jan. 8, 2024).

[25] Dheeru Dua and Casey Graff. UCI machine learning repository, 2017. URL: http://archive. ics.uci.edu/ml. (accessed Jan. 8, 2024).

[26] J. Berg, E. Demirović, and P. J. Stuckey, "Core-boosted linear search for incomplete maxsat," Integration of Constraint Programming, Artificial Intelligence, and Operations Research, pp. 39–56, Jun. 2019. doi:10.1007/978-3-030-19212-9_3

A Appendix: Standard Complete Trees Vs. Oblivious Trees

The following figures highlight the difference between the complete tree and oblivious tree implementations within this library. We use the wine dataset as an example with depth of $d = 2$, and find the most optimal solutions with and without the oblivious tree structure constraint enforced.

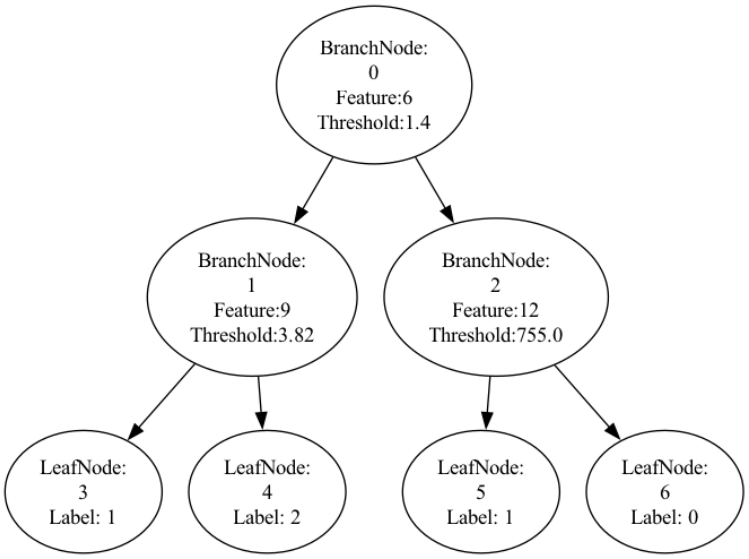

Figure A.1: Optimal solution on wine dataset without oblivious tree constraint

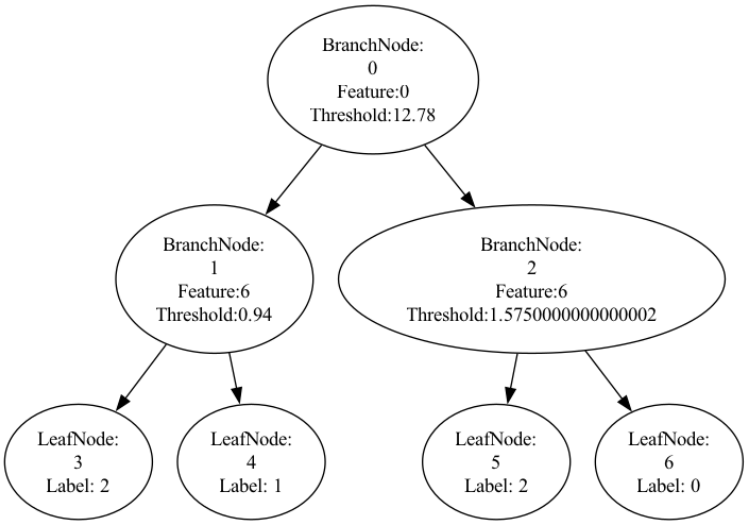

Figure A.2: Optimal solution on wine dataset with oblivious tree constraint

B Appendix: ε–Approximation Leading to Sub-Optimal Clustering Solutions

In this section we observe the impact of using the ε allowance on finding clustering solutions for the maximum diameter problem.

Consider the following dataset $X = \{x_0; x_1; x_2; x_3; x_4; x_5\} = \{(1,1)\ ;\ (1,2)\ ;\ (7,7)\ ;\ (7,8)\ ;\ (15,5)\ ;\ (15,6)\}$, and $(x_2, x_3) \in CL$. We would like to solve this problem testing two values of ε and compare the differences in optimality for $k = 3$ clusters. We test with $\varepsilon = 0$ (global optimal solution) and $\varepsilon = 1$. The solutions produced by our library below highlight how ε can impact optimality.

```
-----Define SATreeCraft Module with  epsilon = 0
clustering_problem.solve()
>>>Output:
{Cluster 0: [2, 4, 5], Cluster 1: [3], Cluster 2: [0, 1]}
{Cluster 0 Max Diameter: 8.246211251235321, Cluster 1 Max Diameter: 0,
    ↪ Cluster 2 Max Diameter: 1.0}

-----Define SATreeCraft Module with  epsilon = 1
clustering_problem.solve()
>>>Output:
{Cluster 0: [3, 4, 5], Cluster 1: [2], Cluster 2: [0, 1]}
{Cluster 0 Max Diameter: 8.54400374531753, Cluster 1 Max Diameter: 0,
    ↪ Cluster 2 Max Diameter: 1.0}
```

Code Excerpt B.1: Cluster Assignmnets and solutions

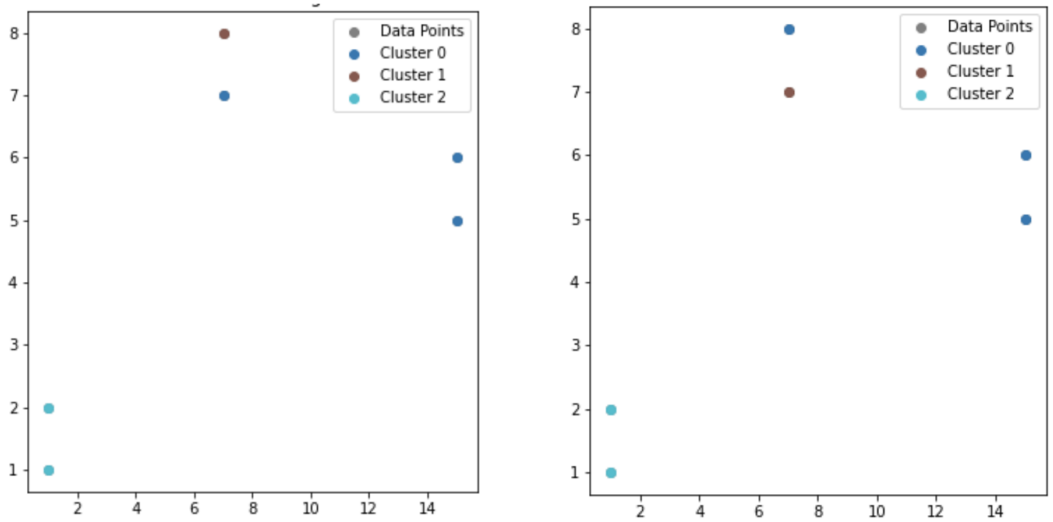

Figure B.1: Optimal clustering solutions found with $\varepsilon = 0$ (left) and $\varepsilon = 1$ (right)