

Final Report

|  |
| --- |
| School of Computing  Faculty of Engineering AND PHYSICAL SCIENCES |

<Full Title of Project>

<Full Name of Author>

Submitted in accordance with the requirements for the degree of

<Name of Degree> (*e.g.* BSc Computer Science)

**<Session>** (*e.g.* 2021/22)

**<Module code and name>** (*e.g.* *COMP3931 Individual Project*)

The candidate confirms that the following have been submitted*:*

*<As an example>*

|  |  |  |
| --- | --- | --- |
| **Items** | **Format** | **Recipient(s) and Date** |
| *Final Report* | *PDF file* | *Uploaded to Minerva (DD/MM/YY)* |
| *Scanned participant consent forms* | *PDF file/file archive* | *Uploaded to Minerva (DD/MM/YY)* |
| *Link to online code repository* | *URL* | *Sent to supervisor and assessor (DD/MM/YY)* |
| *User manuals* | *PDF* | *Sent to client and supervisor (DD/MM/YY)* |

The candidate confirms that the work submitted is their own and the appropriate credit has been given where reference has been made to the work of others.

I understand that failure to attribute material which is obtained from another source may be considered as plagiarism.

(Signature of student)

© <Year of Submission> The University of Leeds and <full name of candidate>

# Summary

*<Concise statement of the problem you intended to solve and main achievements (no more than one A4 page)>*

# Acknowledgements

*<This page should contain any acknowledgements to those who have assisted with your work. Where you have worked as part of a team, you should, where appropriate, reference to any contribution made by others to the project.>*

*Note that it is not acceptable to solicit assistance on ‘proof reading’ which is defined as “the systematic checking and identification of errors in spelling, punctuation, grammar and sentence construction, formatting and layout in the text”; see*

https:://www.leeds.ac.uk/secretariat/documents/proof\_reading\_policy.pdf

# Table of Contents (example of how to format)

Summary iii

Acknowledgements iv

Table of Contents (example of how to format) v

Chapter 1 - Introduction 1

Chapter 2 – Background Research 2

2.1 Binary and Multiclass Classification 2

2.1.1 Definition and Overview 2

2.1.2 Classification Instances 2

2.1.2.1 Binary Classification Instance 2

2.1.2.2 Multiclass Classification Instance 2

2.1.3 Applications in Industry 3

2.2 Decision trees 3

2.2.1 What are Decision trees? 3

2.2.2 Formal Definition 4

2.2.3 Diagram Example 4

2.3 Constructing Decision Trees 5

2.3.1 The Process of Constructing Decision Trees 5

2.3.2 Overview of Existing Algorithms 5

2.3.2.1 Heuristic Algorithms 5

2.3.2.2 Exact Algorithms 6

2.3.3 Importance of Smallest Decision Trees 7

2.3.1 Computational Problem of Constructing Smallest Decision Trees 7

2.4 The Selected Algorithm for Implementation 7

2.4.1 Description of the Algorithm 7

2.4.1.1 Choice of Algorithm 7

2.4.1.2 Pseudo Code 8

2.4.1.3 Explanation of Pseudo Code 9

2.4.2 Correctness of Algorithm 10

2.4.3 Runtime Analysis of Algorithm 11

Chapter 2 12

Methods 13

2.1 Table example 13

2.2 Figure example 13

Method 13

Chapter 3 15

Results 16

Chapter 4 17

Discussion 18

4.1 Conclusions 18

4.2 Ideas for future work 18

References 19

Appendix A 20

Self-appraisal 21

A.1 Critical self-evaluation 21

A.2 Personal reﬂection and lessons learned 21

A.3 Legal, social, ethical and professional issues 21

A.3.1 Legal issues 21

A.3.2 Social issues 21

<Discussion of social issues> 21

A.3.3 Ethical issues 21

<Discussion of ethical issues> 21

A.3.4 Professional issues 21

Appendix B 23

External Materials 24

# Chapter 1 - Introduction

The field of machine learning, which includes everything from data mining to advanced AI technologies, often relies on decision trees. Central to many of these applications are decision trees, favoured for their simplicity and interpretability. These models are particularly useful in situations where it is crucial to trace how decisions are made, providing a clear path from input to outcome.

However, creating the smallest possible decision tree that can correctly handle all examples in a dataset is a formidable challenge. This problem becomes increasingly difficult as datasets grow and complexity, leading to increasing computational demands.

Traditionally, heuristic methods are used to build trees, these methods prioritise computational efficiency and are designed to quickly produce “good enough” trees that handle large datasets effectively but do guarantee the smallest or most accurate tree. On the other hand, there are exact methods that aim to find the smallest possible tree by exhaustively exploring possible solutions. These methods output an optimal decision tree, but they are computationally intensive, making them impractical for large datasets.

The challenge of finding the smallest possible decision tree is an NP-hard problem, meaning that unless P=NP no polytime algorithm exists to solve the problem for all cases. Despite these challenges, “A General Theoretical Framework for Learning Smallest Interpretable Models” by Ordyniak et al. (2024) proposes a fixed-parameter tractable algorithm (an algorithm that is exponential only in the size of a fixed parameter while polynomial in the size of the input) for computing smallest decision trees that represents given data.

The goal of this project is to develop the proposed algorithm that initially handles simple datasets with boolean (true or false) features and categories. We will then enhance the algorithm to work with any type of feature or category. This expansion makes the method more versatile and applicable to more scenarios.

Furthermore, in this project we will explore possible optimisations to the algorithm and analyse performance improvements through metrics such as runtime and number of trees constructed. To evaluate the effectiveness of the implementation, it will undergo a comparative analysis against other established exact methods. This evaluation will involve constructing decision trees across a diverse range of datasets and comparing performance.

# Chapter 2 – Background Research

## 2.1 Binary and Multiclass Classification

### 2.1.1 Definition and Overview

Binary classification is a supervised learning algorithm where the goal is to predict, given input features, one of two potential classes or categories to which an instance belongs. Each instance in binary classification is labelled with one of two classes (Lorena et al., 2008), therefore it is an obvious choice for problems with opposite outcomes such as “yes” or “no”, “positive” or “negative”, “healthy” or “diseased”.

In contrast, multiclass classification entails classifying instances into three or more classes (Lorena et al., 2008). This scenario is common in situations where outcomes are not limited to two possibilities, such as classifying types of crops, recognising various languages or diagnosing multiple types of diseases. Due to the increased number of possible outcomes, the decision-making process is inherently more complex.

Binary and Multiclass Classification has widespread applications across various industries. For instance, in healthcare, binary classifiers are utilised to classify patient diagnoses as either 'positive' or 'negative' for specific conditions, which could be crucial for early intervention (Κούρου et al., 2015). These algorithms are also employed in the finance industry to differentiate between fraudulent and legitimate transactions, improving security measures (De Sá et al., 2018).

### 2.1.2 Classification Instances

A classification instance in machine learning refers to a dataset made up of multiple examples that are processed by a classification algorithm (Ordinyak et al., 2024). Each of these examples, which are part of a larger dataset, consists of a classification and a set of features. The features are the measurable properties or characteristics of the instance, whereas the classification designates the category to which the instance is assigned based on the features.

#### 2.1.2.1 Binary Classification Instance

In binary classification, each example is labelled with one or two possible classes. Here the classification instance can be defined as a tuple (Ordinyak et al., 2024), where:

* is a set of examples, with each example consisting of a set of features values derived from F. In binary classification the domain of these values is {0,1}.
* is a set of features, which are the attributes or variables considered in the classification process.
* is a classification function , which maps each example in E to one of the two possible classes: 0 or 1.

*Table 1: representing an example of a binary classification instance.*

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | Feature: Sunny? | Feature: Humid? | Feature: Rain? | Classification: Go for a run? |
| Example 1 | 1 | 0 | 0 | 1 |
| Example 2 | 1 | 1 | 0 | 0 |
| Example 3 | 0 | 1 | 1 | 0 |
| Example 4 | 0 | 0 | 1 | 0 |

Table 1 illustrates a binary classification instance example where different weather conditions influence the decision to go for a run. Each row in the table represents a unique set of conditions on a specific day, captured by three binary features: “Sunny?”, “Humid?” and “Rain?”. These features are encoded as “1” for yes and “0” for no. The classification outcome, “Go for a run?”, is determined based on these conditions, where “1” suggests it is advisable to go for a run, and “0” indicates otherwise.

#### 2.1.2.2 Multiclass Classification Instance

Multiclass classification extends the concept of binary classification by involving three or more classes into which the examples can be classified. The only difference in the formal definition is is now a classification function . Where represents the range of possible classes. For example, could represent {“Football”, “Cricket”, “Golf”} when determining which sport to play based on weather conditions.

## 2.2 Decision trees

### 2.2.1 What are Decision trees?

Decision trees represent a hierarchical structure for decision-making, where each node signifies a decision based on certain attributes, and the branches denote the outcome of these decisions (Kingsford and Salzberg, 2008) The process initiates from the root node and progresses through internal nodes (representing attributes tests) to the leaf nodes, which hold the decision outcomes – class labels - for classification tasks or continuous values for regression tasks. This process of systematically dividing the entire dataset into smaller subsets mirrors the divide-and-conquer approach often used in problem-solving. By applying this method decision trees aim to simplify complex datasets into subsets that are easier to analyse and make predictions from. (Kingsford and Salzberg, 2008)

The simplicity and interpretability of decision trees make them suitable for a wide array of applications:

Classification and prediction: Their primary use in categorising instances into distinct classes based on attribute values supports applications in a variety of fields such as medical diagnosis, weather forecasting and sales predictions (Rahmatillah et al., 2023) among others.

Regression: Beyond classification, decision trees can be predictors of continuous outcomes, making them invaluable tools in forecasting sales, evaluating real estate prices, and other quantitative analyses (Choy et al., 2012)

The representation of decision trees is both straightforward and visually engaging, enhancing their appeal:

Nodes: The nodes in the decision tree are the entities that make decisions. The root node represents the entire dataset, internal nodes correspond to attribute tests which split the dataset, and leaf nodes represent the outcome of these decision paths.

Branches: These represent the decision outcome at each node, guiding the path to later nodes or leaf outcomes

This structured approach allows decision trees to transparently communicate the logic behind decision-making processes, which makes it easier to grasp and apply them in a variety of fields (Kotsiantis, 2011).

### 2.2.2 Formal Definition

In the context of this report, we provide a formal definition for a decision tree. A decision tree M is a tuple which is a model for a classification instance , such that is a rooted binary tree and , where A is a set of all possible classifications, the function assigns to every inner node of a feature in , and to every leaf node of , a class from . The function, , assigns to every inner node of a real number threshold value, and to every leaf node of , .

The classification function of a decision tree is defined as follows for an example . We define as a function such that , where is a feature in and is a real number value for the feature . Starting at its root of we do the following at every inner node of : If we continue with the left child of , otherwise, we continue with the right child of . The process is repeated until a leaf node is reached, at which point the example is classified as .

A decision tree provides classifications for an entire set of a classification instance by applying the classification function to each example, mapping them to classes in .

### 2.2.3 Diagram Example

A diagram of a diagram

Description automatically generated

*Figure 1: Decision Tree Diagram Illustrating Conditional Branching for Classification*

In Figure 1, we observe a decision tree diagram. The root of the tree poses the initial condition involving the feature ‘Xi’, whether the value of this feature for the current example is less than a certain threshold ‘A’. Depending on the outcome of this evaluation, The process splits into two distinct paths. The left path queries if the example’s value for the feature ‘Xj’ is less than the threshold ‘B’. Whereas the right path queries if the example’s value for the feature ‘Xk’ is less than threshold ‘C’. These paths lead to terminal nodes (leaves) that denote the final classifications of the examples: ‘X’ or ‘Y’.

## 2.3 Constructing Decision Trees

Constructing decision trees involves building models that predict the value of a target variable by learning simple decision rules deduced from data features. Example algorithms, the computational challenges involved, and the significance of minimising tree size, will all be covered in this section.

### 2.3.1 Overview of Existing Algorithms

#### 2.3.1.1 Heuristic Algorithms

Heuristic algorithms solve decision tree problems using rules that find good-enough solutions efficiently without guaranteeing the optimal solution (Hartmann et al., 1982). This approach helps in managing computational costs and complexity, making heuristic algorithms useful in large datasets where an exhaustive search is impractical.

An example of a heuristic algorithm is the Classification and Regression Tree (CART) algorithm (Denison et al., 1998). The CART algorithm is a widely used decision tree learning technique that is used for both classification and regression tasks. It is a method that produces a binary tree by recursively partitioning the data space.

For classification, CART uses the Gini impurity index as a measure to choose the best split at each node. This metric helps determine which feature and which value of that feature will best separate the data into two groups. Initially, CART grows a full tree by continuing to split the data specified by a stopping parameter. After the full tree is built, CART employs a pruning method called cost-complexity to avoid overfitting. This involves trimming the less significant branches from the tree.

A disadvantage of CART and other heuristic algorithms to construct decision trees is these methods are greedy as they make the locally optimal choice at each step of construction without considering the global optimality. This can lead to suboptimal tree structures that are not the best possible representation of the data (Dsemirović et al., 2020).

While heuristic algorithms are favoured for their computational efficiency and effectiveness, they fall short of providing exact solutions. This limitation means that for tasks of building the smallest possible model that correctly classifies all examples in a classification instance, heuristic algorithms will not be the best choice to find a solution (Demirović et al., 2020).

#### 2.3.1.2 Exact Algorithms

Exact algorithms aim to find the optimal solution to the decision tree construction problem by exploring multiple combinations of features and splits. This search ensures that the resultant tree is the smallest possible tree that correctly classifies all examples from the classification instance.

An example of an exact algorithm is the SAT-based method as explored by Narodytska et al. (Narodytska et al., 2018). SAT-based algorithms for decision tree construction are grounded in the principles of satisfiability problems from logical theory. They reformulate the task of constructing a decision tree as an SAT problem. The SAT (Boolean satisfiability) problem is the task of determining whether there exists an assignment of truth values that satisfies a given Boolean formula.

This process of constructing decision trees, with SAT-based methods, starts with problem encoding, where each potential split at every node is converted into logical variables. Furthermore, the relationships and constraints that define a valid decision tree, including the condition that each example in the classification instance must be correctly classified, are encoded as logical clauses. The next phase is constraint formulation, where each logical clause is crafted to represent a specific rule that the tree must satisfy to classify the data accurately. For example, a rule could state that if an example has a particular feature value, it must be directed to a specific leaf of the tree. Additional constraints, such as limiting the depth of the tree – to prevent overfitting, may be formulated.

Once the logical framework has been built, the SAT solver is used. The solver is provided with the complete set of constraints in the form of a logical formula. It begins a search for an assignment of true or false values to the potential splits of the tree at each node, which correspond to the decision paths in the tree, that satisfy the entire formula. The solution provided by the SAT solver is then used to construct the decision tree. The true or false variables indicate which splits should be used while constructing the tree. Splits labelled as ‘true’ are included in the final tree structure, whereas those marked as ‘false’ splits are removed.

The primary benefit of exact algorithms is their ability to produce the most accurate and simplest decision tree that correctly classifies all examples in the classification instance. However, the exhaustive search for the optimal decision tree is computationally intensive and often impractical for large datasets.

### 2.3.2 Importance of Smallest Decision Trees

Occam’s Razor is a principle from philosophy that suggests “entities should not be multiplied beyond necessity”. In the context of machine learning, this translates to favouring simpler models when possible. A simpler model that solves a problem is preferred over a more complex one. Therefore, smaller decision trees are highly valued for their interpretability and reduced likelihood of overfitting (Leiva, 2019). Symbolic models become increasingly opaque if their size increases (Ordinyak et al., 2024). Meaning a smaller model is easier to understand and explain, making it more transparent for decision-makers who rely on the model.

### 2.3.3 Computational Problem of Constructing Smallest Decision Trees

Constructing the smallest possible decision tree is recognized as an NP-hard problem (Ordinyak et al., 2024), which means that unless P=NP no known polynomial-time algorithm can solve this problem for all cases.

Parameterised complexity is a framework used in computer science to analyse the complexity of problems beyond the conventional big O notation, which only considers the overall input size. This approach is particularly useful for addressing NP-hard problems, where traditional computational methods may not yield efficient solutions. In parametrised complexity the goal, a problem is considered in terms of a chosen parameter, typically a characteristic of the input, and the goal is to analyse how the complexity of solving the problem scales when this parameter is small, regardless of the total size of the input.

A problem is said to be fixed parameter tractable if it can be solved in time , where is a computationally manageable function of the parameter , and is the input size. This means the problem’s harder computational aspects are confined to , which ideally grows slowly with , making the problem more manageable when is small.

From a theoretical perspective, there has been intensive research on the parameterised complexity of finding the smallest decision trees (Ordinyak and Szeider, 2021) This research has revealed that the problem is fixed-parameter tractable when parameterised by the solution size plus a bound on the number of features any two examples differ. This tractability when certain parameters are small provides a pathway to develop algorithms that can effectively tackle what would otherwise be an intractable problem.

## 2.4 The Selected Algorithm for Implementation

This project adopts an algorithm inspired by the principles outlined in the paper “A General Theoretical Framework for Learning Smallest Interpretable Models” (Ordinyak et al., 2024). The paper introduces several algorithms that apply to decision trees, decision sets, decision lists, binary decision diagrams and ensembles of these models. In this section, we will be providing a description of the algorithm, the complexity of the algorithm and the correctness of the algorithm.

### 2.4.1 Description of the Algorithm

#### 2.4.1.1 Choice of Algorithm

In this project, we implement algorithm 1, which is a generic algorithm for finding a minimum model of size at most for any strongly extendable model-type T, and algorithm 6, which is an algorithm for finding a full set of strict extensions for decision trees. Together these algorithms form an exact algorithm capable of finding the smallest decision tree which correctly classifies all examples in a provided binary classification instance.

This algorithm is initially tailored to manage datasets with binary features and classes. From this starting point, the project extends the algorithm’s capability, adapting it to handle a broader domain of classes and numerical feature values.

The algorithm is fixed-parameter tractable with respect to the maximum size of the decision tree. This parameter effectively limits the number of nodes in the tree, which directly influences the computational effort required to build the model. This approach allows the algorithm to ignore larger trees early in the process.

#### 2.4.1.2 Overview of Algorithm

This algorithm works with annotated decision trees , (Ordyniak et al., 2024) where is a model as previously defined, and is an annotation of the model with examples that will help guide the search for possible simple extensions. Therefore, every leaf in will have an example assigned to it.

The algorithm starts by constructing two rudimentary decision trees, each consisting of one node. One with the classification of 0 and the other with a classification of 1. Following the initial set up, the algorithm checks if these trees are a model for (meaning every example in is correctly classified by the tree). If any example is misclassified by these initial trees, the algorithm proceeds to generate a set of “strict extensions”. A strict extension involves generating a new tree by adding an inner node and a corresponding leaf to the tree, thereby creating a branch specifically designed to correctly classify the misclassified example. These extensions are a complete set of all possible ways to extend the current tree to correctly classify the example.

For each of these generated extensions, the algorithm recursively evaluates whether this modified tree is a model for . If not, it uses another misclassified example to generate further extensions until it creates a tree that is a model for , or exceeds the predetermined size limit.

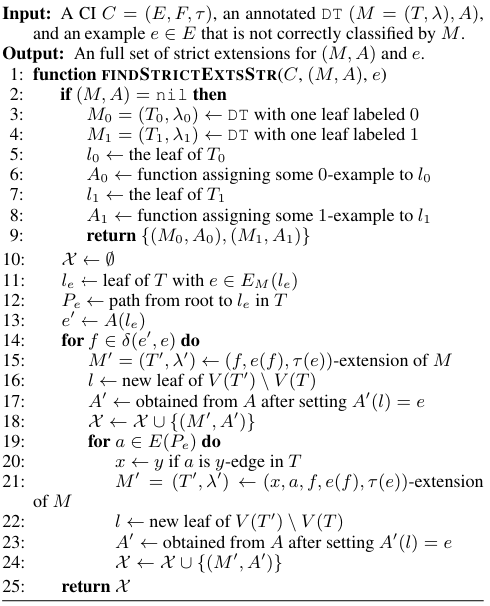
Among all trees generated through the recursive process, the algorithm selects the smallest tree that successfully classifies all examples and return the optimal tree. If no tree can be constructed within the size limit, the algorithm returns .

#### 2.4.1.2 Pseudo Code

A math equations on a white background

Description automatically generatedFigure 2 is a snippet from “A General Theoretical Framework for Learning Smallest Interpretable Models” (Ordinyak et al., 2024). Which shows the pseudocode for Algorithm 1. This is a bounded depth branching algorithm, an algorithm which works by successively branching on various decisions or possibilities at each node of a search tree. However, the algorithm will only explore branches down to a certain depth.

*Figure 2 - Algorithm 1: Generic algorithm for finding a minimum extendable model of size at most s for any strongly extendable model type T (Ordinyak et al., 2024).*

Figure 3 is a snippet from “A General Theoretical Framework for Learning Smallest Interpretable Models” (Ordinyak et al., 2024). Which shows the pseudocode for Algorithm 6. This algorithm’s objective is to take the decision tree passed in as a parameter and create extensions of the tree to make it a better model for .

*Figure 3 - Algorithm 6: Algorithm for finding a full set of extensions for decision trees (Ordinyak et al., 2024).*

#### 2.4.1.3 Explanation of Pseudo Code

This algorithm work with annotated decision trees , where is a model as previously defined, and is an annotation of the model with examples that will help guide the search for possible simple extensions. Therefore, every leaf in will have an example assigned to it, this example would be correctly classified in and reach this leaf when traversing the tree.

The function FindOptModelStr() is the initial function which will be called. The parameter is the binary classification instance we are building a decision tree to represent, and is the maximum size of the decision tree, in the context of the report this is defined as the number of nodes in the tree. When this function is called, it calls the function FindOptExtStr( with an empty annotated tree ().

The function FindOptExtStr( outputs a minimum model for C of size at most that extends (if this doesn’t exist it will output ) given a binary classification instance , an integer , and an annotated tree . The function works as follows: first, it checks if is a model for , meaning the model correctly classifies all examples in . If it does the algorithm returns , else the function proceeds to check if the size of the tree is equal to or larger than and returns , if this is true. Else the algorithm proceeds. Next, the algorithm defines as any example from which is not correctly classified by M and calls FindStrictExtStr() which returns a set of strict extensions for for – which is defined as an example that is incorrectly classified by the current model. In terms of this project, a strict extension is defined as a new tree which adds two nodes (internal and leaf) and a single branch to the old tree so that it correctly classifies . The algorithm then proceeds to call itself for every extension in , and if these extensions are not a model for C it will continue to find extensions for another . It does so for every extension of the model and every example in until it is a model for , or the size is above or equal to , in which case it returns. This would be the base case of the recursive function. Then on line 14, the algorithm checks if the model returned is not , and if the size of the model returned is less than the previous model stored in , if so, it supersedes the previous best model by updating . This iterative refinement continues across all initial extensions in , progressively extending into a model for . Ultimately, ensuring the algorithm exhaustively surveys all possible models bounded by the size of .

Before explaining the final function, it is necessary to explain what is meant by -extensions of and -extensions of . The first of which works as follows: first we create a new root with the feature , then if , which refers to the value from the example for the feature , is 0 then the 0-child of the new root is a new leaf which is labelled with the classification of e and the 1-child is the original root, else the 1-child of the new root is that new leaf and the 0-child is the old root. . -extensions of are applied as follows: for every edge in the path let where is the parent node of in the tree , and a is an -edge between and . We create a new node with the feature and replace the edge with an -edge to the node . If is 0 then we add a 0-edge from the node to a new leaf labelled with and a 1-edge to node , otherwise, we add a 1-edge to the new leaf and a 0-edge to .

The function FindStrictExtStr() works as follows: if the parameter is then the function returns the annotated decision trees and , the former consists of only one leaf labelled with a classification of 0 and annotated with , the latter consists of one leaf labelled with a classification of 1 and annotated with . Otherwise, the function continues. is declared as an empty set. is declared as a leaf of the tree which would currently reach when traversed from the root to a leaf node. is declared as the path from the root to the leaf and is declared as the example which is the current annotation of . Then for every feature which has a different value between the two examples and , the function adds to all the -extensions of and all the -extensions of for each edge in path , where the new leaf is annotated by .

### 2.4.2 Correctness of Algorithm

To show the correctness of the algorithm, we will need to prove the following: the algorithm returns a model M’ which is a model for C and of minimum size, and the algorithm considers all models of size less than or equal to s.

The algorithm begins by checking if the current model already satisfies the classification instance . If is a model for the algorithm terminates and returns , as shown in line 4 of Figure 2, thus affirming the correctness of this trivial case. If is not a model for it checks if its size is at least , then it returns . As strict extensions increase the size of the tree by two (creating two new nodes), the model does not consider models larger than .

For now, we can assume that FindStrictExtStr( returns a full set of strict extensions for M – meaning that it returns all trees which are an extension of M to correctly classify . For each strict extension ’ produced, the algorithm recursively calls itself, checking if is not a model for , the recursive nature of the algorithm ensures that the further extensions for each tree in are explored. The recursive calls effectively perform a depth-first search through the space of all possible models of size less than or equal to .

It remains to show that the set of strict extensions returned by FindStrictExtStr( is a full set of strict extensions of . The way to minimally extend a tree so it classifies correctly is to create 2 new nodes, an internal node, and a leaf. The internal node could be anywhere in the path . So, for the number of branches in the path, there are as many possible places to add the new node (-extension ) plus creating a new root (). This new node would branch out to the new leaf which would classify correctly. To change the direction of the example traversing the tree, there must be a difference in at least one of the features in compared to the annotation of , therefore there are extensions created for all features that have different values for and . So, applying both types of extensions for each differing feature the functions create a full set of strict extensions.

### 2.4.3 Runtime Analysis of Algorithm

Towards showing the run time of the algorithm as a whole, we will start with algorithm 6: we first note that almost all single-line operations in the algorithm can be achieved in constant time (Ordinyak et al., 2024). The only exceptions are lines 11 and 12 to obtain and . Both exceptions can be achieved in time (height of tree)(in the worst case the size of the path is the height of the tree, we obtain that the run time of the function is dominated by the two for loops in line 14 and 19. Together this can be achieved in time (), let be defined as the number of features any two examples differ in and be defined as the height of the tree (T being the tree of . This is because for the first loop, in the worst case, the algorithm may loop for every feature in (such as in or , and within this loop, we have another loop which loops for every edge in which in the worst case is equal to the height of the tree.

For algorithm 1, the initial call to FindOptModelStr merely calls to FindOptExtStr with the same parameters plus an initialised , so its cost would be constant. FindOptExtStr checks if the current model is a model for , This would involve traversing the decision tree for each example in . As previously defined, is a set of examples in C, therefore this step would take time. Next FindOptExtStr checks if is greater than or equal to , this check would take constant time. Next, the function finds an example incorrectly classified by the tree, in the worst case this would involve traversing the tree for every example and, therefore would take time. The call of FindStrictExtsStr, as previously explained, would take time. The algorithm then iterates over generated extensions and calls itself; this recursive step is the most significant contributor to run time. Each call to FindStrictExtStr could potentially generate up to extensions and then the function calls itself on each of these extensions. Let’s denote the maximum depth of the recursive call as , which would be the maximum depth of the recursion tree. This depth would not exceed , since the size of the tree is bounded by . The number of recursive calls at each level can potentially be so the total number of recursive calls can be approximated as .

Each recursive call includes checking the model for examples and the generation of up to strict extensions, thus the complexity of each recursive call remains . Therefore, the total runtime for the algorithm as a whole could be approximated as . Since is bounded by , and assuming the height of a tree does not exceed , we can simplify the approximation of the runtime to .

Correct this !! s is not height of tree

# Chapter 3 – Methods and Implementation

In this chapter, we detail the methodologies and strategies used to develop and improve the algorithm introduce in chapter 2.4. We begin by outlining the project management methodology, emphasising the adoption of Agile practices and the importance utilising version control and coding standards, which are crucial for maintaining code integrity. Following this we delve into the specifics of each development sprint. The first sprint focuses on creating a minimum viable product that implements the decision tree algorithm capable of handling boolean features and classes. The subsequent sprint expands the algorithm capabilities to allow for a wider range of feature types and optimises the algorithm to enhance performance.

The decision tree algorithms are implemented entirely in python, using its native functionalities using only one external library – copy.

## 2.1 Project Management Methodology

2.1.1 Agile Development with Sprints

Agile project management was chosen for its iterative nature, which supports structured development through clearly defined, short passes known as sprints. This approach is particularly effective for a project like ours, where the goal is to initially develop a minimum viable product and incrementally enhance its capabilities. Agile allows for systematic testing and refinement at the end of each sprint, facilitating continuous improvement. Splitting the project into sprints, ensure that each phase of the project contributes to a progressively more efficient and capable software, with opportunities for refinement.

### ****2.1.2 Version Control and Coding Practices****:

Version control was managed using Git, which facilitated efficient handling of code changes. Regular commits were made to track progress and ensure that each alteration was documented and reversible. For coding standards, functions and variables were named clearly and descriptively to improve readability and maintainability. Comments within the code were made concise and informative. Additionally, documentation comments were used to explain each function’s purpose, its parameters, and the expected return values, further ensuring that the code could be easily understood when reviewing, for example in subsequent sprints or when debugging. Commit history can be seen on the following GitHub link – INSERT GITHUB LINK.

## 2.2 Sprint 1: Initial Implementation

### 2.2.1 Goals of Sprint 1

The objectives of Sprint 1 of the project focussed on establishing a solid foundation by creating the minimum viable project (MVP). The key goals were:

* Development of the algorithm: Implementing the simplest iteration of the algorithm outline in section 2.4, capable of classifying datasets with boolean features and classes.
* Ensuring accurate Classification: Verification of the algorithm’s capability to precisely classify all examples in the dataset.
* Validation of the algorithm’s integrity: Utilising classification instances with known minimum decision trees to validate the algorithm’s functionality.
* Implementing terminal output: Ensuring the algorithm can print a representation of decision trees generated to the terminal.

### 2.2.2 Implementation Details

2.2.2.1 Software Architecture Overview:

The software architecture of the project is designed to be modular and clear, involving three Python files: ‘main.py’, ‘dt.py’, and ‘config.py’. Each file has a specific role, allowing easier debugging and better scalability.

‘main.py’: Acts as the entry point of the application. Housing key functions ‘FindOptModelStr’ and ‘FindOptExtStr’, as outlined in section 2.4. The integration between ‘main.py’ and ‘dt.py’ is established through the invocation of ‘FindStrictExtStr’ within ‘FindOptExtStr’.

‘dt.py’: Hosts the core functionalities and definitions related to the decision tree structure. This includes the ‘TreeNode’ class, responsible for defining the nodes of the decision tree, and the ‘DecisionTree’ class, managing the overall tree structure and providing essential methods.

‘config.py’: Declares a two-dimensional array ‘C’, representing the dataset, and a one-dimensional array ‘CFeatures’, listing the names of the features in C, for universal accessibility across modules. This compartmentalisation fosters ease of modification and scalability, permitting alterations to specific components without disrupting unrelated segments.

2.2.2.2 Data Structures and Objects

In the implementation, various data structures are strategically employed to enhance system efficiency and functionality. Arrays are used for storing and accessing classification instances and feature lists. The linear structure of arrays facilitates efficient indexing and iteration (such as accessing feature values for traversal of the tree), aligning with the representation of the dataset ‘C’, where each row represents an example, and the final index is the classification of that example. The indexing of ‘CFeatures’ aligns seamlessly with the indexes of examples within ‘C’, ensuring that the value for each example corresponds to the feature of the same index. This alignment simplifies data retrieval within the algorithm.

Moreover, hash tables (dictionaries in python) are utilised within the ‘DecisionTree’ class to enable efficient mapping and lookup operations. This choice ensures constant-time complexity for add and lookup operations, optimising performance during tree operations.

The decision tree implementation revolves around two essential classes: ‘TreeNode’ and ‘DecisionTree’. The ‘TreeNode’ class encapsulates the properties of each node in the tree structure. ‘TreeNode’ stores the following attributes:

* ‘feature’: Represents the feature associated with node. For leaf nodes, this attribute is set to ‘None’.
* ‘value’: Indicates the classification outcome for leaf nodes, denoted as either 0 or 1. For other nodes, this attribute remains ‘None’.
* ‘child0’: Points to the left child node.
* ‘child1’:Points to the right child node.
* ‘id’: Uniquely identifies each node, ensuring consistency when making copies of the tree structure.

The ‘DecisionTree’ class manages the overall tree structure by holding the root node. Leveraging the ‘child0’ and ‘child1’ attributes of each node, the class facilitates the construction of the decision tree. Additionally, it maintains a dictionary named ‘leafExampleMap’, which associates an example with each leaf node, contributing to the representation of an annotated decision tree.

2.2.2.3 Implementation of Key Functions

‘FindOptExtStr’ in ‘main.py’:

A computer screen shot of a computer code

Description automatically generatedThe ‘FindOptExtStr’ function plays a pivotal role in constructing the minimal decision tree for dataset ‘C’, as per Algorithm 6 outlined in section 2.4. It initiates by validating whether the decision tree ‘M’ correctly models ‘C’, iteratively assessing each example’s classification through the ‘predict’ function from the ‘DecisionTree’ class.

*Figure 3: the code for ‘predict’ function.*

The ‘predict Function involves traversing the decision tree based on the feature of the node and the corresponding value in the example, as shown in Figure 3. This approach is used in different function where traversing through the decision tree is needed. By abstracting the classification process into a separate function, it becomes easier to extend the algorithm to handle more complex classification scenarios, such as handling an increased domain for feature values, without modifying ‘FindOptExtStr’.

If an incorrect classification is detected , the function breaks out the loop, to ensure that it doesn’t perform unnecessary checks, storing the current example in ‘incorrectExample’. If ‘M’ fails to model ‘C’ correctly, ‘FindOptExtStr’ compares the size of ‘M’ to the parameter ‘s’, the maximum allowed size of the tree, if the size is equal to or more than ‘s’ the function returns ‘None’. Otherwise, the function invokes ‘FindStrictExt’ from ‘dt.py’ to generate strict extensions using ‘M’ and ‘incorrectExample’. It then recursively explores these extensions using a recursive depth first search strategy to identify a minimal tree that correctly models ‘C’. This approach is particularly effective for this use case because it explores all possible trees that classify ‘C’ and returns the smallest one.

‘FindStrictExtStr’ in ‘dt.py’:

The function begins by checking if ‘M’ is ‘None’, in which case it creates two basic trees consisting of a node each – one for each classification (0 and 1). Otherwise, ‘FindStrictExtStr’ proceeds by identifying the leaf (‘eLeaf’) and path (‘ePath’) the misclassified example reaches in the current tree. This is accomplished using ‘FindLeafAndPathForExample’, which traces the path and example takes through the tree (traversal occurs in similar way to ‘predict’ function shown above), adding each node it passes to an array ‘path’ and return both this path and the last node in the path as a tuple.

Subsequently, ‘getExampleForLeaf’ retrieves an example (‘e\_’) at the leaf ‘eLeaf’. Next the array ‘disagreeFeatures’ is populated by invoking ‘DisagreeFeatures’, which identifies the specific features where ‘e’ and e\_’ differ by comparing their values. For each feature in disagreeFeatures we perform a -extension to create a tree ‘M\_’. This is done by first creating a new leaf (‘l’) which has a ‘value’ equal to the classification of incorrect example (‘e’), and a new node ‘n’ which extends the tree by acting as the new root and setting its children to a copy of the existing tree and ‘l’. When extending the tree we make a copy of ‘M’ and ‘LeafExampleMap’, so we don’t alter the structure of the old tree. We then call the function ‘AddExampleToLeaf’ on ‘M\_’ to assign ‘e’ to the newly created leaf ‘l’. ‘M\_’ is appended to the array of strict extension ‘X’.

The function then continues by looping over the nodes in ‘ePath’ within the previous loop. For each iteration, we perform a -extension creating a tree ‘M\_copy’ . First the function creates a deep copy of ‘M’ (‘M\_copy’) using ‘deepcopy’ from the copy library. Then we use two functions (‘computePath’ and ‘findEquivalentNode’) to find the equivalent node (node in ‘M\_copy’ that is in the same position in the tree structure as the node in ‘M’) of the current node in ePath (‘copyEPathNode’) and the equivalent of the next node (‘copyEPathChild’). Then in a similar way as previously we create a new leaf, ‘l’ and new node ‘n’ (setting the children of ‘n’ to ‘l’ and ‘copyEPathNodeChild’). We set one of the children of ‘copyEPathNode’ to ‘n’ to link it to the tree. Finally, we add the example to the leaf ‘l’ and add the tree ‘M\_copy’ to ‘X’. Once the nested loop has completed the function return ‘X’

2.2.2.4 Testing Completion of Goals of Sprint 1

During Sprint 1, the primary objectives centred on establishing a minimum viable product, verifying correct classification, and ensuring comprehensible visual output of decision tree structures.

* ‘TestDecisionTree’ function was developed to validate correct classification of all examples within the dataset.
* Acceptance testing involved manual execution of the algorithm using datasets with established minimum decision trees, confirming output alignment with expected results.
* ‘PrintTree’ function was manually tested to ensure clear depiction of decision tree structures in the terminal, aligning output with expected tree structures.

A screenshot of a computer program

Description automatically generatedThe output of the implementation is shown in Figure 4, showcasing the structure of the decision tree generated by the ‘PrintTree’ function. Additionally, ‘TestDecisionTree’ verifies all examples’ correct classification according to the constructed decision tree.

*Figure 4: Final output of the implementation, as displayed in terminal.*

### 2.2.3 Challenges Encountered

During Sprint 1, a significant challenge arose involving the creation of leaves within the decision tree with incorrectly assigned examples, leading to erroneous tree structures.

The root of the problem was identified through a meticulous debugging process. It was discovered that while extending the tree, the algorithm sometimes generated nodes that were logically unreachable based on previous decisions made higher up in the tree. For instance, if a node determined that feature ‘A’ must be 0 for a specific path the algorithm could attempt to create a branch where ‘A’ is 1 further down the same path – a contradiction because the decision at the earlier node already excluded this possibility.

A screenshot of a computer program

Description automatically generatedTo address this flaw, a new function named ‘validatePath’ was implemented. This function plays a crucial role in ensuring that any new extensions or nodes added to the tree are logically consistent with the decisions already made along the path from the root to that node.

*Figure 5: code for ‘validatePath’*

*Figure 5: the code for ‘validatePath’*

The ‘validatePath’ function works as follows: If the node is not a leaf, the function recursively checks both children (left and right). It updates the ‘decisions’ dictionary to reflect the decisions made at the nodes (0 for left child and 1 for right child) along the path to the leaf. Once it reaches a leaf the function retrieves the examples assigned to this leaf. It then verifies that each example adheres to the decisions (features and their values) made along the path to this leaf. If any example does not conform to these decisions, the function returns false.

The ‘validatePath’ function is recursive, to simplify the process of navigating through the tree – each recursive call handles the validation for one node, automatically traversing down to the leaves.

## 2.3 Sprint 2: Feature Expansion

### 2.3.1 Goals of Sprint 2

For Sprint 2 of the project, the objectives are focused on expanding the functionality of the decision tree developed in Sprint 1. The key goals for this sprint include:

* Extend the algorithm to efficiently handle datasets with more than two class labels.
* Modify the algorithm to accommodate numerical features in addition to boolean values.
* Modify the ‘PrintTree’ function for a more intuitive display of decision nodes that involve numerical thresholds.

### 2.3.2 Implementation Details

2.3.2.1 Implementation

To extend the algorithm from Sprint 1, ‘main.py’ remain unchanged. However, a new file , ‘dt\_integer.py’ was created to replace ‘dt.py’. In this file, only a few changes were made to accommodate for an increased domain for feature values.

Extending the algorithm to manage datasets with more than two class labels involves a straightforward enhancement where the class of each leaf node is set to the class of the example assigned to it. This natural extension allows the algorithm to handle multiple classes without needing significant changes to the underlying logic. Therefore, when creating the two basic decision trees at the start of ‘FindStrictExtsStr’ we created one tree with the classification of ‘incorrectExample’ and the other with the classification of the next example in ‘C’ with a different classification.

To extend the domain of feature values, first we need to store all features, along with possible values. This is done in the ‘config.py’ file by creating a dictionary named ‘featureValuesMap’ which lists all features in ‘CFeatures’ as keys in the dictionary and assigns to these features all the possible values for that feature from the dataset ‘C’.

The implementation of this is relatively simple, as show in in figure 6.

*A screen shot of a computer program

Description automatically generated*

*Figure 6: code to initialise ‘featureValuesMap’.*

The main difference in this extension was the function ‘DisagreeFeatures’, where instead of searching for features where two examples differ, the function searches for all (‘feature’, ‘value’) pairs from ‘featureValuesMap’ where the two examples had values on either side of that threshold (represented by ‘value’ in the tuple). This approach ensures that every potential threshold to split at is considered, allowing the algorithm to accept any numerical value for features.

Finally, every function which traversed the tree was altered to include comparisons between feature values and a node’s ‘threshold’ attribute. Examples’ which were less than or equal to the threshold followed the branch to the left child, whereas other examples followed the branch to the right child.

2.3.2.2 Testing and Validation

The method for testing the goals for sprint 2, was relatively similar to the testing for Sprint 1. In Sprint 2 we altered the ‘PrintTree’ function to include the threshold for each node. For a clearer depiction of the decision tree structures.

A computer screen shot of white text

Description automatically generatedAcceptance testing was performed, which involved manually executing the algorithm using datasets with known minimum decision trees, these datasets included multiple classes and numerical values for the features.

*Figure 7: Output for ‘PrintTree’*

### 2.3.3 Challenges Encountered`

Integrating categorical features into the decision tree algorithm, which inherently functions on binary decisions typically suited for numerical data, presented a significant challenge. The core issue was adapting the algorithm, which uses binary comparisons (less than and more than), to handle non-ordinal categorical data.

To address this, categorical data were represented as integers, allowing the decision tree to continue making binary splits. However this integer representation introduced a concern regarding arbitrary order of categories, which could influence the decision tree’s effectiveness and accuracy.

## 2.4 Sprint 3: Optimisations

### 2.4.1 Goals of Sprint 3

Prevent the construction of identical trees: This goal involves developing mechanisms within the decision tree algorithm to ensure that no two trees with the same structure and classification outcomes are generated. Identical trees can occur when different extensions of a tree can ultimately lead to two or more of the same trees. Preventing this not only makes the overall algorithm more efficient by reducing computational redundancy but also simplifies the analysis of the model’s decision path.

Prevent splitting of the same feature in a path:

### 2.4.2 Challenges

*Explain why the method didn’t work for preventing construction of identical trees, and how we didn’t find a way to achieve this goal*

### 2.4.2 Implementation

*How I implemented the second goal or preventing split of the same feature*

### 2.4.3 Testing

*Tested by creating trees with and without optimisation and showing that we still get the same final tree, however we construct less trees*

# Chapter 3

# Results

<Results, evaluation (including user evaluation) *etc*. should be described in one or more chapters. See the `Results and Discussion' criterion in the mark scheme for the sorts of material that may be included here.>

Results

validation

* Binary – use known decision trees, multiple
* Numerical

Evaluations

* Compare to SAT
* Compute upper and low bounds – binary search to find s, compare to SAT, return first one we find

# Chapter 4

# Discussion

<Everything that comes under the `Results and Discussion' criterion in the mark scheme that has not been addressed in an earlier chapter should be included in this final chapter. The following section headings are suggestions only.>

## 4.1 Conclusions

<Text in 11-point size and 1.5 line spacing.>

## 4.2 Ideas for future work

<Text in 11-point size and 1.5 line spacing.>

# References

1. Lorena, A.C., De Leon Ferreira De Ponce Carvalho, A.C. and Gama, J. 2008. A review on the combination of binary classifiers in multiclass problems. *Artificial Intelligence Review*. **30**(1–4), pp.19–37.
2. Ordyniak, S., Paesani, G., Rychlicki, M. and Szeider, S. 2024. A general theoretical framework for learning smallest interpretable models. *Proceedings of the ... AAAI Conference on Artificial Intelligence*. **38**(9), pp.10662–10669.
3. Κούρου, Κ., Exarchos, T.P., Exarchos, K., Karamouzis, M.V. and Fotiadis, D.I. 2015. Machine learning applications in cancer prognosis and prediction. *Computational and Structural Biotechnology Journal*. **13**, pp.8–17.
4. De Sá, A.G.C., Pereira, A.C.M. and Pappa, G.L. 2018. A customized classification algorithm for credit card fraud detection. *Engineering Applications of Artificial Intelligence*. **72**, pp.21–29.
5. Poo, Z.Y., Ting, C.Y., Loh, Y.P. and Ghauth, K.I. 2023. Multi-Label Classification with Deep Learning for Retail Recommendation. *Journal of Informatics and Web Engineering (Online)*. **2**(2), pp.218–232.
6. Kingsford, C. and Salzberg, S.L. 2008. What are decision trees? *Nature Biotechnology*. **26**(9), pp.1011–1013.
7. Rahmatillah, I., Astuty, E. and Sudirman, I.D. 2023. An Improved Decision Tree Model for Forecasting Consumer Decision in a Medium Groceries Store. *2023 IEEE 17th International Conference on Industrial and Information Systems (ICIIS*.
8. Choy, L.H.T., Ho, W.K.O. and Mak, S. 2012. Housing attributes and Hong Kong real estate prices: a quantile regression analysis. Construction Management & Economics. 30(5), pp.359–366.
9. Kotsiantis, S. 2011. Decision trees: a recent overview. Artificial Intelligence Review. 39(4), pp.261–283.
10. Hartmann, C., Varshney, P.K., Mehrotra, K. and Gerberich, C. 1982. Application of information theory to the construction of efficient decision trees. IEEE Transactions on Information Theory. 28(4), pp.565–577.
11. Denison, D.G.T., Mallick, B.K. and Smith, A.F.M. 1998. A Bayesian CART algorithm. *Biometrika*. **85**(2), pp.363–377.
12. Demirović, E., Lukina, A., Hébrard, E., Chan, J., Bailey, J., Leckie, C., Ramamohanarao, K. and Stuckey, P.J. 2020. MurTree: Optimal classification trees via dynamic programming and search. arXiv (Cornell University).
13. Narodytska, N., Ignatiev, A., Pereira, F.S. and Marques-Silva, J. 2018. *Learning optimal decision trees with SAT* [Online]. Available from: https://doi.org/10.24963/ijcai.2018/189.
14. Leiva, R.G., Anta, A.F., Mancuso, V. and Casari, P. 2019. A Novel Hyperparameter-Free Approach to Decision Tree Construction That Avoids Overfitting by Design. *IEEE Access*. **7**, pp.99978–99987.
15. Ordyniak, S.; and Szeider, S. 2021. Parameterized Complexity of Small Decision Tree Learning. Proc. IJCAI 2021, 6454–6462.

# Appendix A

# Self-appraisal

<This appendix must contain everything covered under the ’self-appraisal’ criterion in the mark scheme. Although there is no length limit for this section, 2-4 pages will normally be suﬃcient. The format of this section is not prescribed, but you may like to consider the following sections and subsections.>

## A.1 Critical self-evaluation

## A.2 Personal reﬂection and lessons learned

## A.3 Legal, social, ethical and professional issues

<Refer to each of these issues in turn. If one or more is not relevant to your project, you should still explain *why* you think it was not relevant.>

### A.3.1 Legal issues

<Discussion of legal issues>

### A.3.2 Social issues

### <Discussion of social issues>

### A.3.3 Ethical issues

### <Discussion of ethical issues>

### A.3.4 Professional issues

<Discussion of professional Issues>

# Appendix B

# External Materials

<This appendix should provide a brief record of materials used in the solution that are not the student's own work. Such materials might be pieces of codes made available from a research group/company or from the internet, datasets prepared by external users or any preliminary materials/drafts/notes provided by a supervisor. It should be clear what was used as ready-made components and what was developed as part of the project. This appendix should be included even if no external materials were used, in which case a statement to that effect is all that is required.>