Optimisation of Classification Tree Models

***By***

***Andrew Rippington – Reg no. 1701077***

***School of Computer Science and Electronic Engineering***

***BSc Data Science and Analytics***

***Supervised by: Dr Michael Fairbank***

***Second Assessor: Dr Serafeim Perdikis***

***Word count: 10,968***

**Acknowledgements**

With thanks to Dr Michael Fairbank and Dr Daniel Karapetyan for both supervising and supporting me throughout this project.

**Abstract**

We attempt to improve upon current state-of-the-art classification methods with a novel approach to optimising classification trees. Classification tree algorithms are widely used machine learning algorithms capable of classifying input data into subsets and thus predicting an output. We provide the algorithm with a dataset (arbitrary) and thus produce a classification model.

Next, we optimise that model. Note that we are not optimising the classification algorithm itself (through parameter tuning etc.) as is often done, but instead we are optimising the data structure (tree) that the algorithm initially produces.

The problem of creating an optimal classification tree is known to be NP-Complete, meaning that it cannot currently be solved in polynomial time. Thus, classification algorithms use heuristics to make good decisions at a local level, despite the fact that these are often sub-optimal on a global level. Once these decision are made, they are final (they do not backtrack).

Here, instead, we build a classification tree using a standard algorithm and then optimise the tree structure itself. Effectively, we are attempting to reduce the optimal classification tree problem to an optimisation task.

Attempts at optimising classification trees have been made before, however, our approach here is novel.

Using this approach, we attempt to do the following:

1) Improve upon the predictive accuracy of state-of-the-art classifiers

2) Reduce the size (number of nodes) of classification trees while maintaining performance, making them more interpretable to humans

**Table of Contents**

1. Statistical Classification

1.01 Introduction & Definition *p.7*

1.02 Project Scope *p.7*

1.1 Classification Algorithms (Classifiers)

1.1.1 Definition & Types of Classifier *p.8*

1.1.2 Methodology and Limitations *p.8*

1.2 Classification Trees

1.2.1 Structure of Decision/Regression Trees *p.9*

1.2.2 Interpretability *p.10*

1.3 C4.5 Algorithm

1.3.1 Origin p.10

1.3.2 Base Cases *p.10*

1.3.3 Information Gain & Entropy *p.10-11*

1.3.4 Information Gain & Entropy Example Calculation *p.12-13*

1.3.5 Algorithm Logic *p.14*

1.3.6 Variable Types *p.14*

1.3.7 Heuristics *p.14-15*

1.4 Implementation of the C4.5 Algorithm

1.4.1 Data Parsing *p.16*

1.4.2 Software Architecture *p.16-17*

1.4.3 Pseudocode *p.17-18*

1.4.4 Evaluation Metrics *p.18*

**2.0** Visualisation

2.0.1 Technologies *p.19*

2.0.2 Implementation *p.19*

2.0.3 Graphs *p.19*

**3.0** Optimisation

3.0.1 Definition *p.20*

3.0.2 Optimisation V Machine Learning *p.20*

3.1` Optimisation Algorithms

3.1.1 Background & Definitions *p.21*

3.1.2 Local Search *p.21*

3.1.3 Problems in Optimisation *p.21*

3.1.4 No-Free-Lunch *p.21*

3.1.5 Pareto Front & Project Scope *p.22*

3.2 Implementing the Optimisation Framework

3.2.1 Previous Research *p.22*

3.2.2 Parameters *p.22*

3.2.3 Methodology *p.23-24*

3.2.4 Deep Copying & Functional Implementation *p.25*

3.2.5 Pruning *p.26-27*

**4.0** Experiments & Results

4.0.1 Hardware *p.28*

4.0.2 Parameters & Design *p.28*

4.0.3 Datasets *p.28*

4.0.4 Independent Variable Selection *p.29*

4.0.5 Analysis *p.29*

4.0.5.1 Heart Disease Dataset *p.29-34*

4.0.5.2 Abalone Dataset *p.35*

**5.0** Performance Comparison with Published C4.5 Libraries *p.36-37*

**6.0** Project Planning *p.38-39*

**7.0** Conclusions & Future Work *p.40-41*

**8.0** References *p.42-44*

**Terminology**

* Classification Tree 🡪 An umbrella term referring to either a decision tree or a regression tree.
* Instance 🡪 A single observation. A single row in a dataset.
* Class 🡪 A sub-population/category.
* Attribute 🡪 A single variable. Used as an input.
* Model 🡪 A single classification tree structure.
* Local 🡪 With respect to a defined range.
* Global 🡪 Over an entire function domain.
* Extrema 🡪 Collective name for both maxima and minima (The largest and smallest values of a function, either locally or globally)

**List of Figures**

Fig. 1 - A basic (non-binary) decision tree *(p.9)*

Fig. 2 - C# function to calculate the entropy of the target column in a DataTable *(p.15)*

Fig. 3 - Class diagram of the C4.5 algorithm and optimisation framework *(p.16)*

Fig. 4 - An example decision tree model produced by our code *(p.17)*

Fig. 5 - A .DOT file opened with Visual Studio Code and GraphViz extensions *(p.19)*

Fig. 6 - A visualisation of tree structure optimisation *(p.24)*

Fig. 7 – C# Constructor to perform deep copying of a Node *(p.25)*

Fig. 8 - C# Method for cloning a classification tree *(p.25)*

Fig. 9 – C# Tree-pruning function to remove redundant nodes *(p.26)*

Fig. 10 - Unpruned decision tree *(p.27)*

Fig. 11 - Pruned decision tree *(p.27)*

Fig. 12 - 'Heart Disease' Results Table *(p.29)*

Fig. 13 – Test accuracies by parameter configuration *(p.30)*

Fig. 14 - Standard deviation of test accuracies *(p.31)*

Fig. 15 - Average time taken to train a single tree model for each value of **i** *(p.31)*

Fig. 16 - Test accuracy distributions of parameter configurations *(p.32)*

Fig. 17 - Average accuracy scores for i=1000 configurations with varying X values *(p.33)*

Fig. 18 - Average tree sizes  *(p.34)*

Fig. 19 - 'Abalone' Results Table *(p.35)*

Fig. 20 - weka.J48 results *(p.36)*

Fig. 21 - Test accuracy comparison table *(p.36)*

1. Statistical Classification

1.0.1 Introduction & Definition

In the context of statistics and machine learning, classification is a supervised learning problem with the task of identifying which sub-population a new observation belongs to. This is achieved using previous knowledge in the form of training data observations whose labels (category memberships) are already known. In this respect, classification differs from clustering (unsupervised learning) which does not have a set of pre-defined categories and instead uses some measure of ‘distance’ to group observations.

We can consider classification as a problem of the following form:

* Given an input X and an output Y, find the function that maps X🡪Y. [1]

1.0.2 Project Scope

In this project we will be looking at the following types of classification::

* Binary classification: There exist two classes, i.e. “yes/no”, “pass/fail”, “0/1”
* Multiclass classification: There are > 2 target classes, an instance belongs to one and only one class

We must be careful not to confuse multiclass with multi-label classification (in which an instance can belong to more than one class). We will be dealing strictly with binary and multiclass tasks whereby each class is mutually exclusive and independent.

Regression can be viewed as a classification task where the target variable is continuous, i.e. predicting height, weight. In the scope of this project, we also look at discretising regression problems and thus converting them into classification tasks. To do this, we define thresholds on the target variable and consider each partition as a distinct class. [2]

*For example, we could turn height prediction into a 4-class classification problem:*

*Target Variable: Height*

*Classes: 0-100cm, 101-150cm, 151,200cm, >201cm*

In these discretisations, the more classes we define the more precise our classification model can be. However, the accuracy is also likely to decrease. Note also that some other statistical models, such as logistic regression, are already classification methods.

* 1. Classification Algorithms (Classifiers)

1.1.1 Definition & Types of Classifier

A classifier is any algorithm which sorts data into labelled classes. There exist many types of classifier, though determining which is best often depends on the nature of the specific dataset. [3]

Classifiers can be broadly divided into the following:

***Classification Trees:*** (***See section 1.2)***

***Discriminant Analysis:*** A fitting function estimates the parameters of a Gaussian distribution for each class. Prediction is achieved by finding the class with the smallest misclassification cost for each new instance. [4]

***Support Vector Machines:*** Attempts to find a hyperplane in N-dimensional space that distinctly classifies the data points. [5]

***Naïve Bayes:*** A family of algorithms based on conditional probability models [6]

***K-Nearest Neighbour:*** Classifies data points based on some similarity measure. [7]

***Logistic Regression:*** Using an equation representation, linearly combines input values with coefficients to predict an output class. [8]

1.1.2 Methodology and Limitations

Classifiers utilise training data to ‘understand’ how input variables relate to each target class. A classification model is built using each training instance, with a single hypothesis being created which covers the entire instance space. This hypothesis can then be used to predict the class of a new instance, given the values of variables other than that of the target variable.

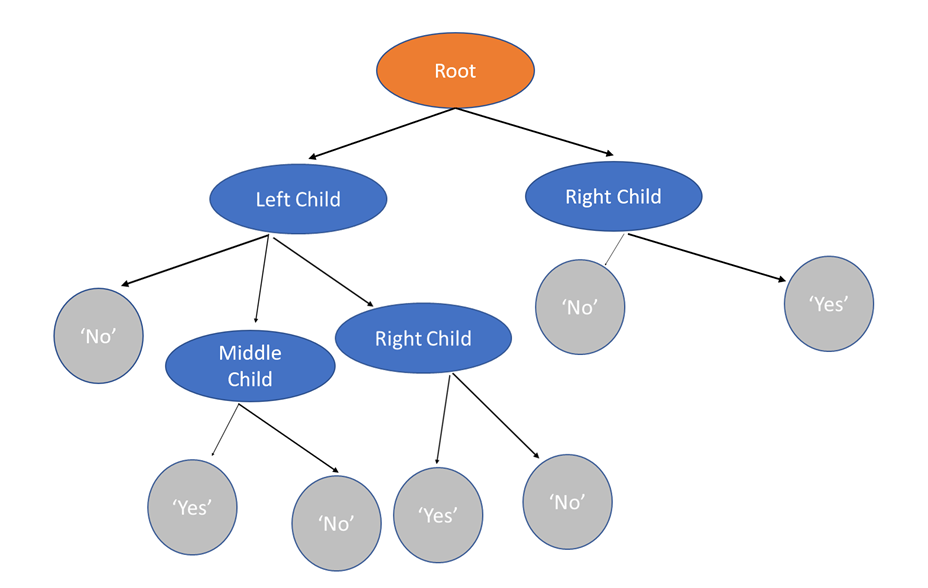
Overfitting is a large problem with not only classification, but machine learning problems in general. [9] Overfitting can be defined as creating a model than fits the training data too well and takes into account idiosyncrasies of the specific data sample used. This leads to a lack of generalisation power, as the model is too complex/specific to describe the entire population of data rather than just the sample used for training. We can reduce overfitting by employing methods such as the ‘Row Limiting’ parameter *(described in section 1.4, base case 2).* Similarly, we can underfit our model. This refers to producing a model that is too general and does not capture the structure of the data. Both overfitting and underfitting reduce the performance of our models, thus should be avoided.

* 1. Classification Trees

1.2.1 Structure of Decision/Regression Trees

In this project, we consider decision/regression tree classification algorithms. Decision trees and regression trees are predictive models consisting of data structures representing nodes, branches and leaves.

* Node - Represents a threshold/condition on a certain attribute, i.e. if N < 5, follow path to left child, else follow path to right child.
* Branch – A sub-tree
* Leaf – A class label



*Fig. 1 - A basic (non-binary) decision tree.*

The initial node in Fig. 1 (coloured orange) is known as the root, which is the starting point of classification. It contains the first condition in the tree structure, and thus every observation in the training set will ‘enter’ this node.

Trees limited to a binary split at each node are referred to as binary trees. In this project, we do not limit ourselves to binary trees and thus there may be more than two paths leading from a node.

Trees which have a categorical target variable are referred to as decision trees and those which have a continuous target are referred to as regression trees. As aforementioned, a regression tree problem can be converted into that of a decision tree by discretising the target variable. In this report we will use the term ‘classification tree’ to refer to both decision and regression trees.

The learning process of classification trees is accomplished by relevant feature selection. This is a key step in creating an efficient tree structure, though it also causes problems. By reducing the geometric quantity of data as we approach leaf nodes, we risk creating an overly complex model and thus classification trees are particularly prone to overfitting. [10]

1.2.2 Interpretability

Classification trees are an extremely popular machine learning tool due to their interpretability. They are often easier to understand/more intuitive than other classification methods and can be visualised naturally. As the size of a tree grows, however, its interpretability may be significantly reduced. One of the aims of this project is to research simplifying decision trees while maintaining performance. That is, reduce the number of nodes in a decision tree without negatively affecting predictive accuracy. By doing this we attempt to keep a classification tree model accessible to a wide range of professionals without compromising and losing information. Interpretability is currently a hot topic within the data science community [11]- If we can create understandable explanations for our models, they can lead to more intuitive decision making at a human level. This also follows the principles of Occam’s Razor, insinuating that the simplest explanation is often the best (and often has the most generalisation power).

* 1. C4.5 Algorithm

1.3.1 Origin

The C4.5 algorithm is a particular classification tree induction algorithm, developed by Ross Quinlan in 1994. [12] Despite its age, C4.5 is still widely used as a benchmark algorithm and retains cutting edge performance in 2020. [13] The algorithm initially gained popularity after ranking 1st in the ‘Top 10 Algorithms in Data Mining’ paper published by Springer LNCS in 2008. [14]

1.3.2 Base Cases

Consider the training set **S = s1`, s2, …, si** with each sample representing an already classified n-dimensional vector, where **n** = number of input variables. At each node, C4.5 selects the attribute for splitting based upon which has the highest normalised information gain with respect to the target variable. The algorithm performs this heuristic recursively until one of the following base cases is reached:

Base Cases

1. **All instances reaching a node have identical class labels. (Thus, convert the node into a leaf node)**
2. **Row Limiting parameter reached: Fewer than the minimum permitted number of instances enter a node. (Thus, convert to a leaf node and label with the most common class from the remaining instances)**

*N.B. This is a crucial parameter of the algorithm. Changing this value can change performance drastically. If this value is too low, we overfit our model. If this value is too high, we underfit our model.*

1. **All attributes have already been used for splitting in a particular branch (Thus, once again convert to leaf node and label with the most common class.)**

*N.B. Theoretically, if an input attribute is continuous it could be used for splitting multiple times in the same branch. However, this is not usually done in C4.5 implementations.*

1.3.3 Information Gain & Entropy

C4.5 induces trees based upon the principle of information gain.

Within the context of information theory and machine learning in general, information gain is a synonym for Kullback–Leibler divergence, which is the amount of information gained regarding one random variable from observing another random variable. [12]

However, an important distinction must be made. In the context of classification trees specifically, information gain is synonymous with mutual information. This is the conditional expected value of the Kullback–Leibler divergence of the univariate probability distribution of X taken from the conditional distribution of X,Y, where X and Y are random variables. This is also equal to the reduction of Shannon’s Entropy in a system.

*Where is the Kullback–Leibler divergence.* [15]

*Note that normalised information gain will equal 0 if either of the two variables are homogenous, thus there is no information gain from observing one with respect to the other.*

Entropy can be defined as the level of disorder in a system, or the measure of impurity. [16]

Within the context of this project, we consider two measure of entropy:

**Entropy(T) 🡪 using the frequency table of one attribute, T:**

***c*** *= number of distinct values for this variable*

***pi*** *= the probability of value i being selected at random (i.e. pi = 0.5 if there is equal representation of two distinct values)*

Note that log base 2 is arbitrary and is used only to normalize the entropy value. If we wish to keep entropy within the range 0-1, then log2 must be replaced by log*k*, where k = c = the number of distinct values contained within this variable.

**Entropy(T,X) 🡪 using the frequency table of two attributes, T, X, (i.e. where T is the target variable and X is an input variable):**

***c*** *= a distinct value of X*

***P(c)*** *= the probability of c occurring*

***E(c)*** *= the entropy of c*

Information gain is then simply calculated by:

That is, the reduction of entropy in variable T given the variable X .

1.3.4 Information Gain & Entropy Example Calculation

For example, consider the following data, with ‘Breed’ as the target variable:

|  |  |  |  |
| --- | --- | --- | --- |
| **Height** | **Colour** | **Length** | **Breed** |
| Medium | Brindle | Long | Dachshund |
| Short | Cream | Long | Dachshund |
| Tall | Brindle | Normal | Labrador |
| Medium | Brindle | Normal | Labrador |

Firstly, we need to calculate the Entropy of the target variable itself:

Thus,

Our target variable’s initial Entropy is therefore 1.

Note, expressions in the form refer to calculating the initial entropy on that class, with the input values i and j equal to the frequency each class occurs. For example;

Next, we need to calculate the joint entropy from splitting on each variable. This can be done using the joint frequency tables of each variable alongside the target variable.

**Height:**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | | Breed | | Total |
| Dachshund | Labrador |
| Height | Short | 1 | 0 | 1 |
| Medium | 1 | 1 | 2 |
| Tall | 0 | 1 | 1 |

**Colour:**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | | Breed | | Total |
| Dachshund | Labrador |
| Colour | Brindle | 1 | 2 | 3 |
| Cream | 1 | 0 | 1 |

**Length:**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | | Breed | | Total |
| Dachshund | Labrador |
| Length | Long | 2 | 0 | 2 |
| Normal | 0 | 2 | 2 |

Finally, we can calculate the Information Gain from splitting on each variable (simply by subtracting the entropy after splitting from the initial entropy):

In this case, we can see that splitting on the ‘Length’ attribute gives us the highest information gain and thus would be selected by C4.5’s heuristic.

1.3.5 Algorithm Logic

The C4.5 algorithm itself consists of the following steps:

***Pseudocode***

1. Check for base cases *(as seen in section 1.3.2)*
2. Find the information gain from splitting on each attribute
3. Split on the attribute with highest information gain
4. Repeat until a base case is found, at which point a label is assigned to the remaining instances

Once this process has been recursively applied to each sub-set of data, we have created a tree model which can then be used for prediction. To predict the class of a new observation, we iteratively traverse the tree’s nodes (starting at the root) until we reach a leaf. The label of the leaf node reached is the predicted class for our observation.

1.3.6 Variable Types

C4.5 contains functionality to handle both continuous and categorical input data, as well as implementing tree pruning to remove redundant nodes.

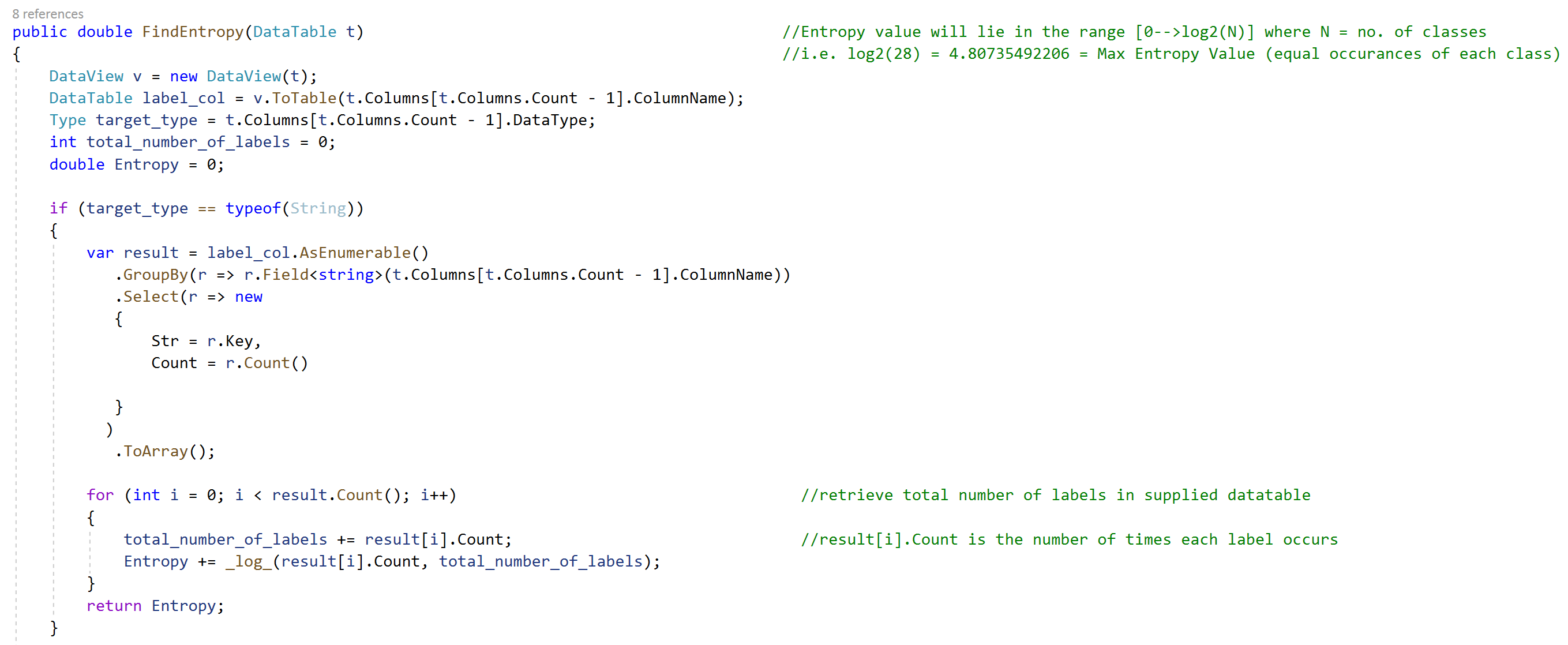
For categorical input variables, C4.5 immediately partitions the dataset into N branches, N= number of distinct values for that variable. In many cases, this will be a binary split, i.e. if the attribute is “Yes/No?” and the possible values are “Yes” and “No”. However, the same concept is applied to categorical input variables with a greater number of possible values. That is, if a categorical attribute has 3 distinct possible values, the node will split the data into three child nodes – one for each value.

For continuous input variables, C4.5 uses a similar concept to determine the ‘best’ splitting threshold. Given a continuous attribute, the algorithm calculates the information gain from splitting on each existing value. The threshold with the highest information gain is selected, and the continuous data is split at that threshold (less than or equal to the threshold 🡪 left child, greater than the threshold 🡪 right child). To perform a split, C4.5 calculates the best threshold from each attribute and selects the single highest information gain found (and thus splits on the associated attribute and threshold).

1.3.7 Heuristics

Machine learning algorithms utilise heuristics to make good decisions on a local level, although they are unlikely to be optimal globally. For instance, our heuristic here is the measure of Information Gain to determine how we split the data. A split may be optimal in the initial instance (in terms of our heuristic), but instead if we made a sub-optimal split at the same location, the overall tree could perform better by producing superior splits further down the tree.

The exact implementation of entropy in this project can be seen in Fig. 2.



…

*Fig. 2 - C# function to calculate the entropy of the target column in a DataTable.*

The function shown in Fig.2 returns the entropy of target column (of type: string). The method also proceeds to define functionality for dealing with a continuous target variable (of type: int, double), using the same heuristic to detect the best splitting threshold.

* 1. Implementation of the C4.5 Algorithm

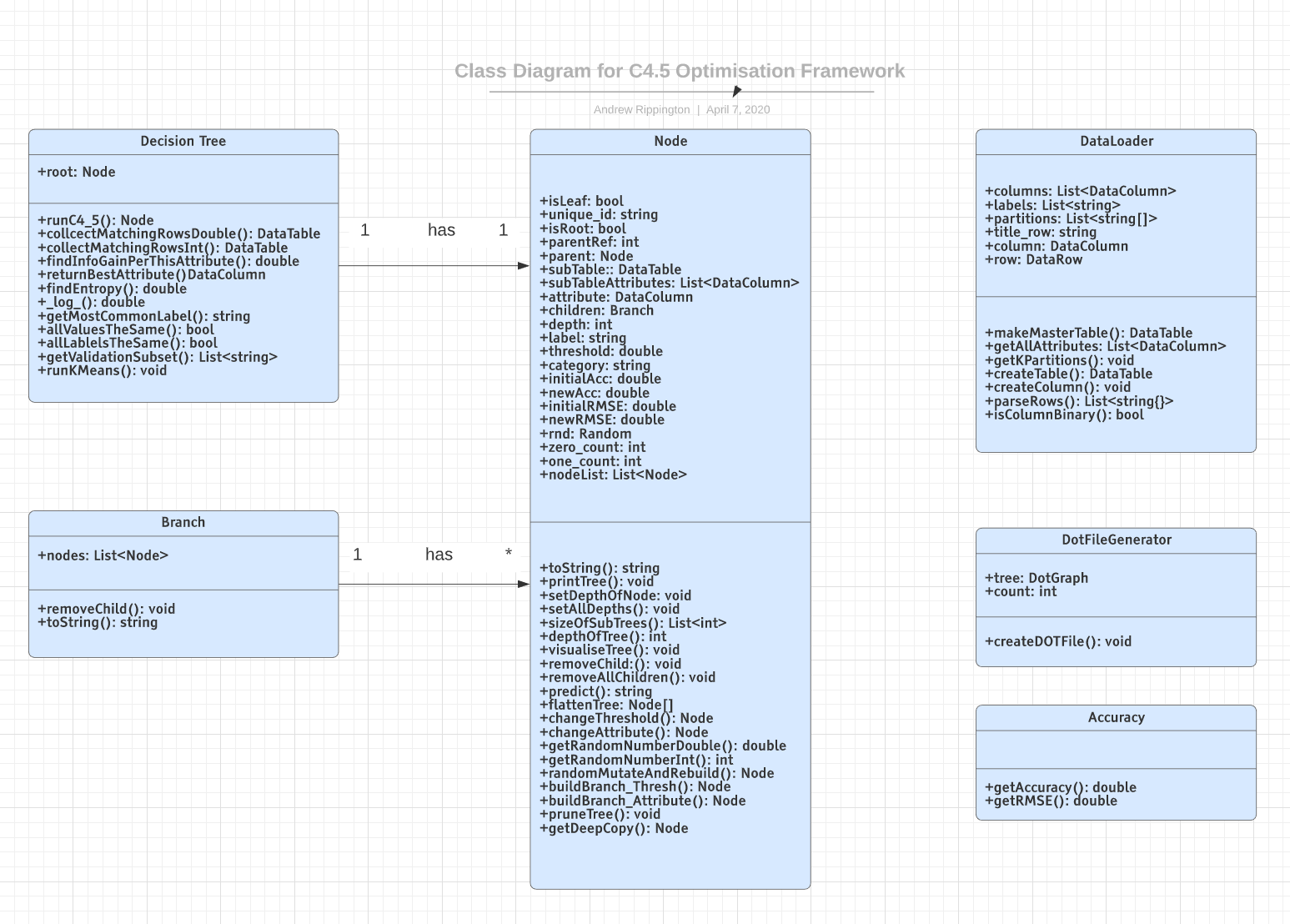
.

1.4.1 Data Parsing

Microsoft’s C# programming language was used to build a C4.5 decision tree library from scratch. No third-party libraries were used.

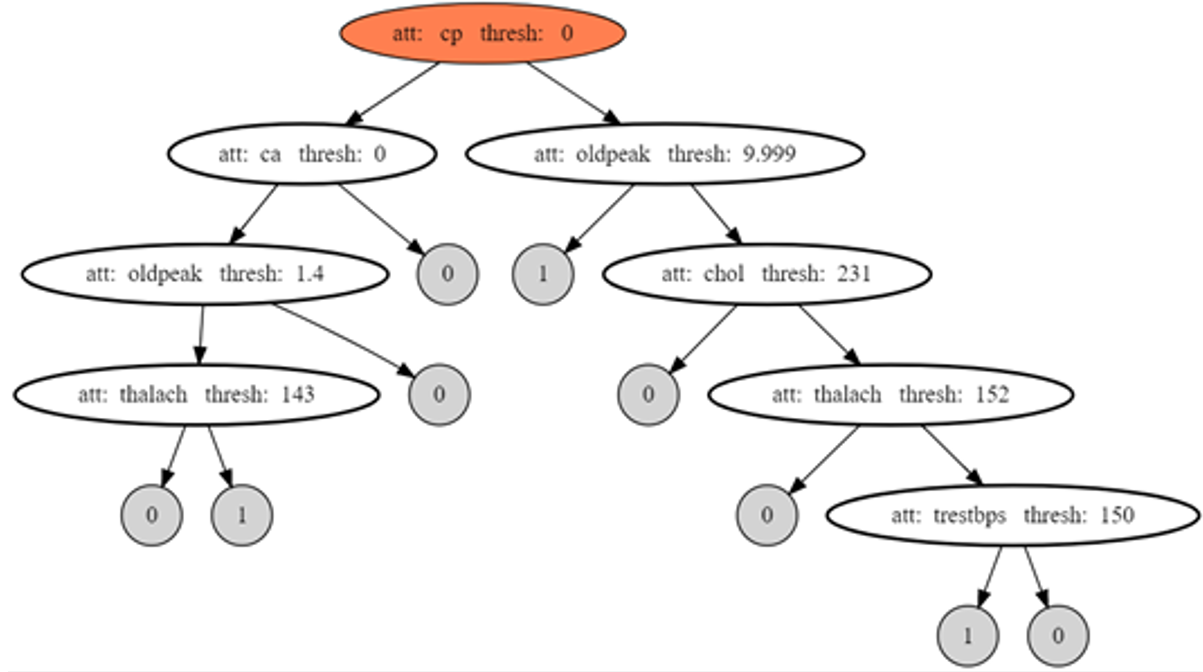
The framework is generalised and can accept any CSV file including binary, categorical and continuous data. The type of each variable is first detected, before being converted into a DataTable format. Any input variables with only 2 distinct values are converted into a binary format to slightly improve our algorithm’s computational complexity.

1.4.2 Software Architecture



*Fig. 3 - Class diagram of the C4.5 algorithm and optimisation framework.* [17]

Fig.3 shows our code structure. Each tree is stored as a recursive data structure of type Node. A node has references to each of its children, thus an entire decision tree can be expressed via its root node as recursively traversing from the root allows us to reach any node in the tree. Due to this recursive nature of our data structure, all methods within the Node class are recursive and thus able to encapsulate performing a method on the entire tree.



*Fig. 4 - An example decision tree model produced by our code, induced on the ‘Heart Disease’ dataset* [18] *Each node shows which attribute is used for splitting and the threshold at which the split is made.*

Each Node stores the following information:

* ‘subTable ‘– the rows of the DataTable which ‘enter’ this node
* ‘attributes’ – the variables that have not already been used for splitting on the path to this node
* ‘attribute’ – the variable used for splitting at this node
* ‘children’ – the immediate children of this node
* ‘parent’ – this node’s parent
* ‘parentRef’ – an index used to identify which position this child lies in the tree (i.e. a node who is it’s parent’s left child will have a parentRef of 0, while a right child a parentRef of 1.)
* ‘unique\_id’ – a unique identifier specific to this node
* ‘isLeaf’ & ‘isRoot’ – Boolean identifiers showing whether this node is the root of a tree or a leaf of a tree (Both can be false in the case that this node is neither)
* ‘depth’ – the level of the tree that this node is located at
* ‘label’ – the class this node belongs to (leaf nodes only)
* ‘threshold’ – the threshold used to split at this node

1.4.3 Pseudocode

In this project, we utilise K-fold cross validation and automatically partition our data *(see section 3.2.3)* However, in the simple case that the training and test sets are pre-split and stored in separate files, our code proceeds as follows:

**Data Preparation**

1. Read data from training set CSV file
2. Extract each variable from the data (by index)
3. Identify data types of each variable and create a DataColumn of the corresponding type.
4. Create a DataTable with all DataColumns (including the target variable)
5. Add all data to the DataTable as DataRows
6. Add copies of all DataColumns apart from that of the target variable to a new list, referred to as ‘attributes’
7. Initiate C4.5 using the DataTable and attributes list:



*N.B. The ‘forceBestAttribute’ argument is used for optimisation and will be discussed in section 3.2.3*

**Run algorithm (as previously described)**

1. Check for base cases.
2. Find best attribute and threshold for splitting (measure information gain for each attribute and potential threshold)
3. Split data on the combination with the highest information gain. (The DataTable is split into 2+ sub-tables, one for each child of the splitting node.)
4. Repeat recursively for all nodes produced.

**Prediction Algorithm**

1. Read data from testing set CSV file.
2. For each observation, starting at the root, iteratively traverse the decision tree model using the ‘attribute’ and ‘threshold’ fields of each node to guide which path we should take.
3. When a leaf node is reached, store the leaf node’s ‘label’ value as our prediction for this test observation.
4. Compare our predicted values with the actual target variable values to obtain an accuracy/RMSE score.

1.4.4 Evaluation Metrics

We have three evaluation metrics that we use to determine performance of our model.

*For a discrete target variable:*

1. Accuracy: Calculated as a percentage of how many predicted values were correct/incorrect.

*For a continuous target variable:*

1. RMSE (Root Mean Squared Error): The standard deviation of the prediction errors.

Where: f = predicted values, o = actual values

*Usage Example: One of the datasets used in this project was the ‘Abalone’ dataset* [19] *which has a target variable with 28 distinct classes. As these classes are the integers 1-28 and the target variable is numeric, it would make sense to use RMSE as our evaluation metric, though Accuracy could also be used.*

1. Size: The number of nodes comprising a tree model (including root, excluding leaves).
2. Visualisation

2.0.1 Technologies

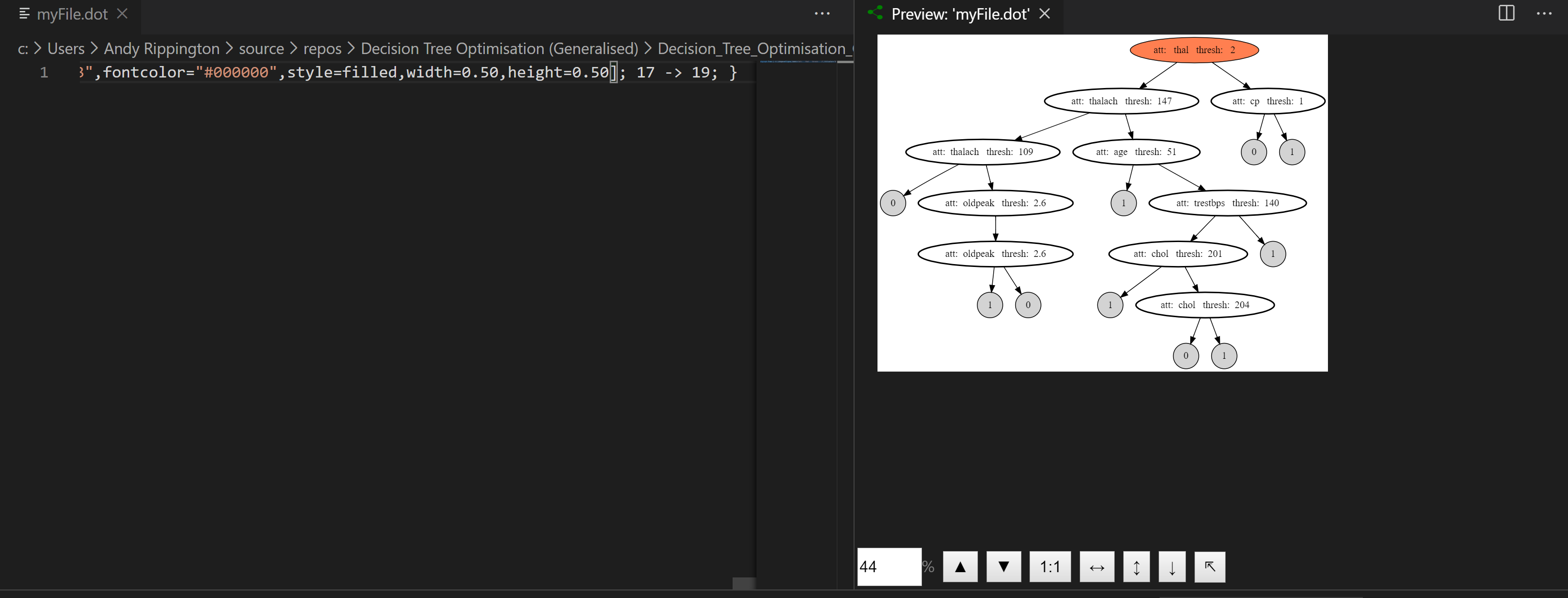
To visualise our decision tree models, we utilise the following tools:

* GraphViz [20]
* GraphViz (dot) language support for VS Code [21]
* GraphViz web previewer [22]
* GraphViz VS Code Visualisation package [23]
* DotNetGraph [24]

2.0.2 Implementation

Firstly we convert our decision tree data structure into a .DOT file by implementing the DotNetGraph library via a method specific to our tree architecture and saving the resulting file locally.

We can then open our .DOT file with the GraphViz VS Code Preview package to obtain a visualisation of the tree in our right-hand side pane, as seen in Fig. 5.



*Fig. 5 - A .DOT file opened with Visual Studio Code and GraphViz extensions.*

By implementing these steps within our main algorithmic framework, we can automatically generate a .DOT file for each tree and view the trees produced in real time.

2.0.3 Graphs

The graphs within this report were generated using RStudio, utilising the ggplot2 and dplyr libraries. The R code used is also stored within the Git repository alongside all source code for this project. Some simple graphs were also created with Microsoft Excel.

1. Optimisation

3.0.1 Definition

Mathematical optimisation refers to the maximisation or minimisation of some real function. This is achieved by choosing inputs from a permitted set and measuring the output of the function. [25]

We can represent any optimisation task with the following simple framework:

from some set **A** to real numbers **R**

where we aim to minimise/maximise **R**.

The permitted set is often a subset of Euclidian space , limited by a set of constraints/inequalities that the elements of A have to satisfy. The function, f, is often referred to as the objective function and the set, A, is referred to as the search space.

It’s also worth noting that minimisation and maximisation problems are interchangeable, as minimising the negative of an objective function is equal to maximising it.

3.0.2 Optimisation V Machine Learning

Optimisation and machine learning have close ties, though are not without difference. Both attempt to minimise some loss function (i.e. in classification, we attempt to minimise the discrepancy between our predictions and the actual unseen labels.). The difference is that optimisation is concerned with minimising some loss function for observed data, while machine learning attempts to minimise some loss function with relation to unseen data – this is synonymous with predicting. [26]

In this project, we utilise both machine learning and optimisation techniques. We use machine learning by creating a classification model with C4.5, before optimising the model that is produced in an attempt to better the performance of C4.5 alone.

* 1. Optimisation Algorithms

3.1.1 Background & Definitions

An optimisation algorithm is any procedure which iteratively compares solutions and alters parameters/inputs until an optimal/satisfactory solution is found. The input/parameter altering step is usually implemented with some form of heuristic – some method designed to find solutions to problems more quickly than classical methods. Heuristics are used as in most cases the search space of an optimisation problems is too large for the optimal solution to be found via brute force. The heuristic, instead, utilises either some function or weighting to steer the algorithm towards an optimal solution (or approximate solution) within the search space. [27]

The following distinctions should be made regarding our terminology:

* Algorithm – Finitely terminating (can be guaranteed to find an optimal solution, but not necessarily within a feasible timeframe) [28]
* Iterative Method – Convergent (guaranteed to continuously get closer to an optimal solution from the initial starting point, but not necessarily reach it.) [29]
* Heuristic – An algorithm which is not mathematically guaranteed to find an optimal solution, but is useful within our framework. (often traverses the search space faster or more effectively and is used for that purpose) [27]

3.1.2 Local Search

A local search algorithm is an optimisation technique which begins at a candidate solution, before moving onto a neighbouring solution. The decision of which neighbour to move to is usually decided by a metaheuristic which measures and selects the neighbour maximising some criterion. In this project, we use C4.5 both as our initial tree induction method, but also as a hill-climbing heuristic in our optimisation framework to select which neighbouring state is ’better’ (We are forcing our search to move to a neighbouring solution with less entropy.) [30]

3.1.3 Problems in Optimisation

A problem with optimisation in general is that of local extrema. Local extrema are solutions which dominate the neighbouring search space, but are not the single best solution in the entire search space. In this case, our hill-climber can no longer be utilised as our current solution is better than all of its immediate neighbours. To escape, we must implement a form of mutation which changes our position in the global search space. Often, a mutation will make the solution worse, but with the goal of improving it beyond the previous local maximum/minimum in the long term.

It can be difficult to distinguish local extrema from global extrema, as to do so often requires knowledge of the global maximum/minimum itself. One technique to deal with this problem is that of saving the best solution ever found. By doing so, we can ensure that we will have a saved copy of the global maximum/minimum if we do encounter it during running time.

3.1.4 No-Free-Lunch

A relevant topic regarding optimisation is the ‘no free lunch’ theorem. It is a result which states that for certain types of problems, the computational cost of finding a solution within a problem class is identical for all solutions methods, when averaged across all problems in that class. [31] From this information we can infer that we should select an appropriate optimisation method for our specific problem domain, but that there is no single best solution method that should be applied in every case.

3.1.5 Pareto Front & Project Scope

In some frameworks, we may wish to simultaneously optimise two objective functions – intuitively known as multi-objective optimisation. In this case, a Pareto Optimal solution would be a sequence of elements in A which is not dominated by any other design. That is, if we have two objective functions, a pareto optimal solution would be a solution where neither of its two objective function values can be beaten by another design. The set of solutions which improve upon each other’s objective functions at the expense of the second objective function is known as the Pareto set. The Pareto set lies on a curve, and it is the prerogative of the designer to determine which solution in the Pareto set is optimal and where the trade-off is preferred. [32]

In this project, we consider a simple iterative local search approach to optimisation, and will be looking at multi-objective optimisation for the accuracy and size of classification tree models. Instead of forcing a mutation to escape local extrema, we intermittently (every **X** iterations) randomise the validation subset that we are optimising with respect to, in order to avoid overfitting. Based on this, we prefer to investigate one form of optimisation on a larger variety of parameter configurations – though in future, the scope of this project could be expanded to include alternative optimisation techniques.

3.2 Implementing the Optimisation Framework

3.2.1 Previous Research

The optimisation of classification trees is considered a very complex problem. [33] Research has been conducted in this domain by producing a graph of all reduced trees for a specific dataset. However, that research differed from this as its purpose was to reduce geometric complexity measures such as weighted depth of trees. We do not consider weighting of trees in this project – we remain impartial to tree geometry with the single evaluation metric in terms of size merely being the total number of nodes.

There has also been research into the minimisation of classification tree size via combinatorial optimisation [34]. However, in our case, we are optimising decision trees with respect to size but simultaneously attempting to improve accuracy. In this respect, our approach here is novel.

3.2.2 Parameters

Optimisation of decision tree models provides a unique problem, in that with each **X** iterations of our optimisation cycle we must select a new random subset of our validation data to evaluate our model’s accuracy. This is a necessary task as by doing so we can prevent overfitting of our model to the validation set.

Noting that we use C4.5 as our hill-climbing operator to rebuild branches after value mutations, our model parameters are as follows:

• ***Row limiting parameter (C4.5)*** - minimum number of observations permitted to enter a node without converting to a leaf)

• ***K*** – number of partitions to split our data into for k-fold cross validation, set to 10 by default

• ***O*** – accuracy/RMSE or size (or both)

• ***V*** (Validation set size) – the proportion of our training data used as a validation set

• ***Vi*** (Validation subset size) – the proportion of our validation set used to check accuracy of a model

• ***i*** – the number of times to perform our optimisation cycle

* ***X*** – number of optimisation iterations implemented before re-selecting a random subset of our validation set

3.2.3 Methodology

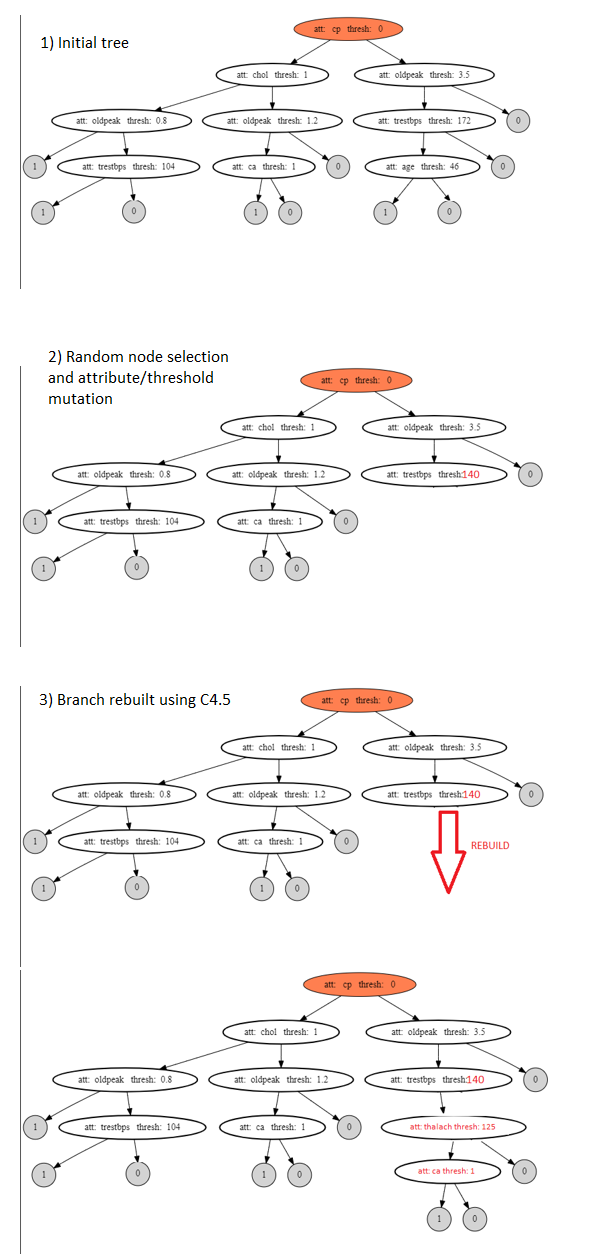
Our optimisation algorithm is thus as follows:

1. **Randomly shuffle the dataset**
2. Split data into K partitions (each with 45% training, 45% validation, 10% testing) *(default proportions)*
3. Produce an initial tree with C4.5 on partition **K*i***’s training data
4. ***Optimisation*** - Optimise tree for O (accuracy and/or size) with respect to a random subset of the validation set (**Vi**). *(see below)*
5. Repeat step (3) ***i*** times
6. Test final classification tree model against the reserved test set
7. Repeat steps (1-6) **K** times
8. Retrieve average results

***Step(4) - Optimisation***

1. Randomly select a node from the tree – we produce an array of nodes(Node[]) by traversing the tree structure and storing all nodes, before using a random number generator to select a node by index.
2. Randomly mutate the attribute/threshold – we randomly select either the attribute or threshold for mutation. If attribute is selected; we randomly pick an attribute from the list of remaining attributes stored in that node (the attributes not yet used higher up in the same branch). If threshold is selected, we randomly choose a new threshold by storing all remaining values for this attribute in an array and selecting one. By ‘remaining values’, we refer to the values of this attribute held by each of the observations that reach this node.
3. Rebuild the branch using C4.5 – If ‘attribute’ is selected, this is implemented by running the C4.5 algorithm with the optional input parameter ‘forceBestAttribute’, which forces C4.5 to skip the step of calculating best split in relation to Information Gain, and instead split on the supplied attribute (if the supplied attribute is numeric, C4.5 will still determine the best threshold to split upon from within that attribute). If ‘threshold’ is selected, we use a manually defined function to split the remaining data by our new threshold and run C4.5 on the two newly created sub-tables. This function also adds a reference from each of the new nodes to its’ parent node and vice versa to ensure that we maintain one single tree structure in storage.
4. Calculate Accuracy/Size - we calculate the size of our new tree and the accuracy with respect to the current validation subset being used.
5. Revert/Proceed – if the accuracy and/or size has worsened, we revert the change. This is implemented by creating a deep copy of each tree structure before a mutation occurs so that we can revert control back to this copy if necessary. If the accuracy and/or size has improved, we keep the change. If we are using both objective functions, accuracy/RMSE and size, we consider a new tree ‘improved’ if one of these values is superior to the last and the other value has not worsened. Depending on which optimisation technique we use, we may occasionally decide not to reverse a change even if it worsens the accuracy and/or size (to escape local extrema).

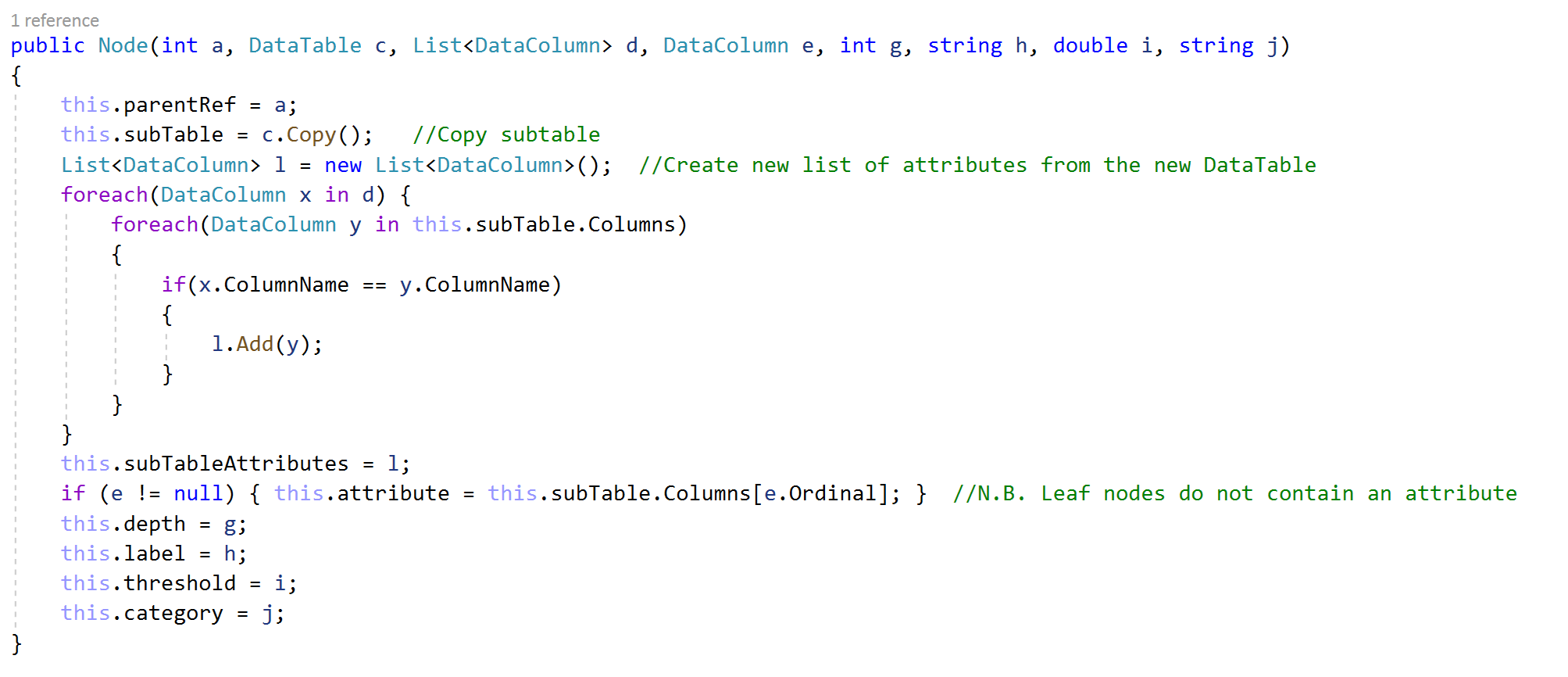
If our target variable is numeric/continuous, our framework will automatically calculate both the Accuracy and RMSE of our model so that both are available for analysis. However, a distinction must be made as to whether we wish to optimise Accuracy or RMSE, as their relationship is not necessarily linear.



*Fig. 6 - A simple example of tree structure optimisation. The accuracy and size of the tree is checked before and after this cycle is applied. If the new tree is superior in terms of our given metric, it is kept. If it is worse, we revert back to our initial tree.*

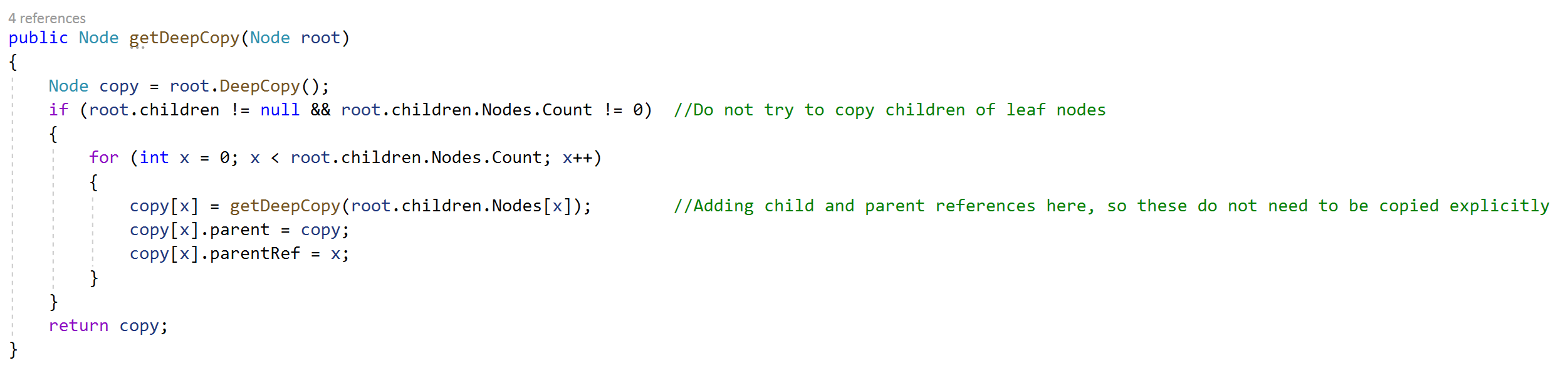
3.2.4 Deep Copying & Functional Implementation

To implement deep copying of a tree, a custom constructor (Fig.7) was defined to copy a single node.



*Fig. 7 – C# Constructor to perform deep copying of a single Node.*

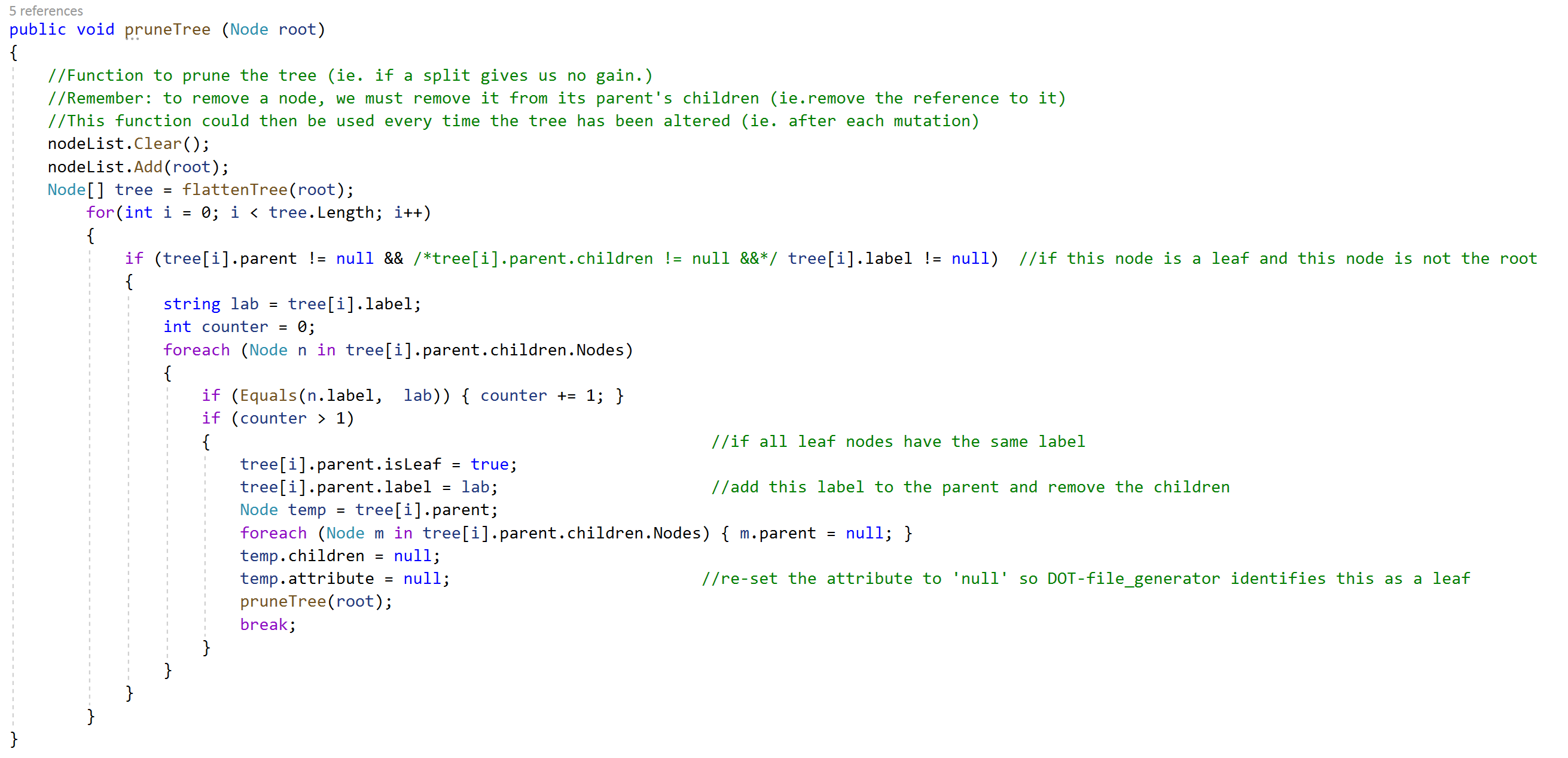
This is then wrapped in a function (Fig. 8) to iterate over the tree, thus copying the entire tree structure.



*Fig. 8 - C# method for cloning a classification tree.*

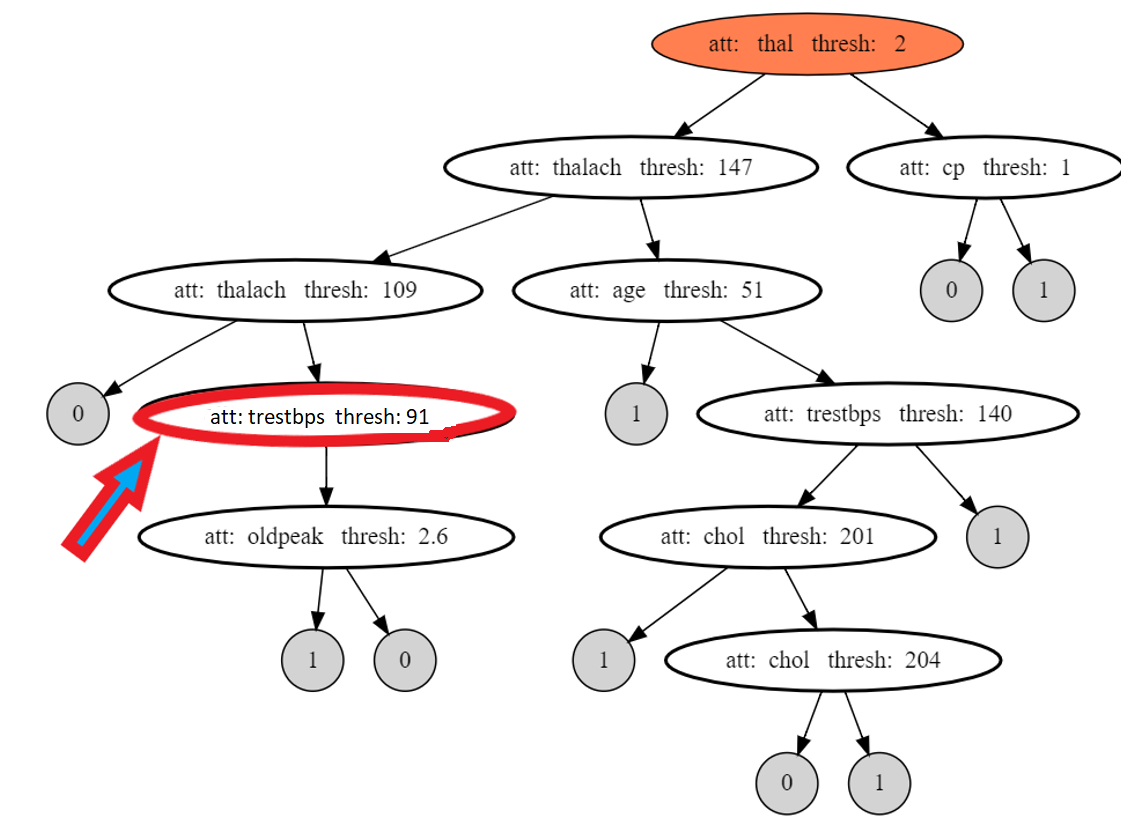
An important technical note is that in the .NET C# language, we must then explicitly define a single no-argument constructor for the class Node. This is because once a constructor of a class has been defined (with any number of arguments), the implicit parameterless constructor is no longer called by default. It should also be noted that any method defined in the class Node, which is applied to a single node, then has to be wrapped in another function (like that seen in Fig 8) to operate on an entire tree structure.

3.2.5 Pruning

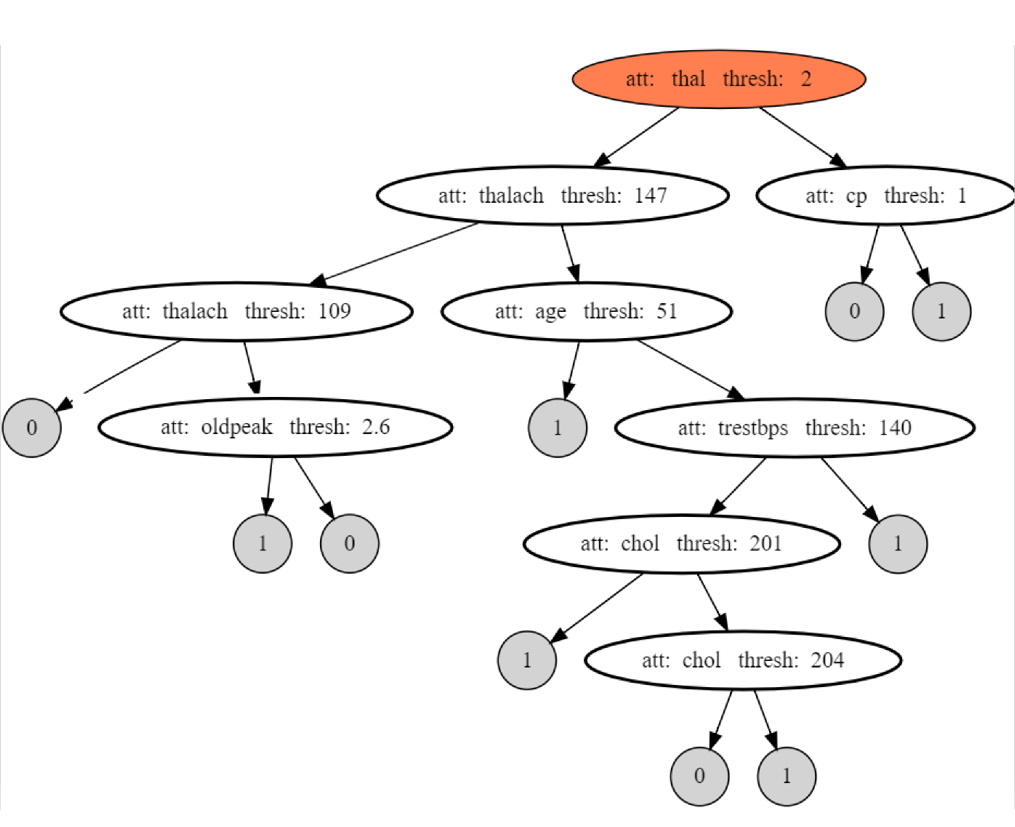


*Fig. 9 – C# Tree-pruning function to remove redundant nodes*

Our optimisation framework implicitly prunes our tree after each optimisation iteration to avoid redundant splits. Redundant nodes can lead to trees containing more nodes than necessary and thus distort our calculations when minimising the size of trees. Within the base C4.5 algorithm, it is impossible for redundant nodes to exist within our tree (because we utilise the Information Gain heuristic to select favourable splits). However, once we begin to optimise a tree, we could encounter a situation whereby our randomly selected threshold/attribute leads to zero Information Gain. In cases such as these, our pruning algorithm traverses the tree to remove redundant nodes. It performs this by allowing the redundant node’s parent to directly reference the redundant node’s child, thus ‘skipping’ the obsolete node within our tree reference structure. Redundant nodes can be easily identified as such because they by nature only have one child (As their information gain is equal to zero, all observations that enter will lie within the same side of the ‘splitting criteria’, thus one of the redundant node’s branches leads to all of the data that entered it initially and the other branch leads to an empty set. As an empty set does not initiate the C4.5 algorithm and thus construction of a new node, the redundant node only has one child.) This can be visualised intuitively, as seen in Fig. 10.



*Fig. 10 - An example of an unpruned decision tree. Note that the highlighted node is redundant as all data travels ‘through’ it but no splitting is performed.*



*Fig. 11 - The same tree as Fig.10, after pruning. We can see that the redundant node has been erased, with it’s parent and child nodes now being directly connected.*

1. Experiments & Results

4.0.1 Hardware

All experiments were run on an ASUS Zenbook UX510UWK, running Windows 10 Home with the following components:

* Intel® Core™ i7-7500U CPU @ 2.70GHz, 2904 MHz, 2 Cores, 4 Logical processors
* 16.0GB Installed Physical RAM

4.0.2 Parameters & Design

Results are retrieved using the C# ClosedXML library to write output data to a .xlsx file. Each value stored in the file represents the average result from a 10-fold cross validation run.

Parameters:

* ***R*** - row limiting parameter
* ***K*** – number of partitions to split our data into for k-fold cross validation (default: 10)
* ***O*** – accuracy/RMSE or size (or both)
* ***Tr*** – Training set size (as a proportion of the full dataset)
* **T** - Test set size (as a proportion of the full dataset)
* ***V*** - Validation set size set (as a proportion of the full dataset)
* ***Vi*** - Validation subset size (as a proportion of the validation set)
* ***i*** – the number of times to perform our optimisation cycle
* ***P*** – probability that a change is kept if it worsens the solution
* ***X*** – number of optimisation iterations implemented before re-selecting a random subset of our validation set
* ***D*** – Dataset

The scope for experimentation is extremely large with this number of parameters, therefore there is much left outstanding that cannot be included in this project. Note that both the training and test set sizes, Tr and T, are implicitly determined by the value of K. (i.e if K=10, Tr=90% and T = 10% **or** if k=5, Tr=80% and T =20%.)

The data points recorded are the result of traversing our final trees with the training, validation and test sets respectively. The score from ‘training accuracy’ is a measure of how closely our final model was fitted to the training data, likewise with ‘validation accuracy’. The true indicator of predictive power is captured by the ‘test accuracy’ measurement.

4.0.3 Datasets

We utilise two public datasets to train our models:

Cleveland Heart Disease Data (binarized): *(*[*http://archive.ics.uci.edu/ml/datasets/Heart+Disease*](http://archive.ics.uci.edu/ml/datasets/Heart+Disease)*)* [18]

* 303 observations
* 14 attributes (10 continuous, 4 nominal)
* Target Variable: 2 classes

Abalone Data: *(*[*http://archive.ics.uci.edu/ml/datasets/Abalone*](http://archive.ics.uci.edu/ml/datasets/Abalone)*)* [19]

* 4177 observations
* 8 attributes (7 continuous, 1 nominal)
* Target Variable: 28 classes

4.0.4 Independent Variable Selection

We consider the effects of varying ***i***, the number of optimisation cycles performed and ***X***, the number of optimisation cycles performed before we randomise the validation subset. For this, we utilise both the Heart Disease and Abalone datasets (***D***). Selecting any of the other defined parameters to investigate would be equally valid, though ***i, X and D***have been selected as they relate most directly to optimisation performance, rather than the performance of C4.5 itself.

4.0.5 Analysis

We set R = 15 for Heart Disease data and R = 250 for Abalone data as these values performed optimally in brief preliminary testing for their respective datasets, though these values are still arbitrary and should be treated as such.

Note that increasing the value of ***X*** causes us to randomise the validation subset more infrequently, hence ***X***=0 for each value of **i** is placed **after** the largest X value in Fig. 12 as it refers to randomising the validation subset as infrequently as possible (not at all).

4.0.5.1 Heart Disease Dataset

Constants: *R = 15, K = 10, O = accuracy & size, T = 10%, Tr = 45%, V = 45%, Vi = 50%, P = 0 (no forced mutation), D = Heart Disease dataset,* *Optimisation Method: Iterative Local Search*

|  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  | | | | Test Accuracy | | | | Size | | | |  |
| ***i*** | ***X*** | Tr. Acc. | Va.  Acc. | Min | Avg. | Max | Std. | Min | Avg. | Max | Std. | Avg. Training Time (seconds) |
| **0** | **n/a** | 66.51 | 61.34 | 43.91 | 61.35 | 75.37 | 6.12 | 14.1 | 24.5 | 38.3 | 4.3 | 0.41 |
| **10** | **2** | 68.03 | 63.50 | 42.22 | 62.86 | 71.58 | 5.75 | 15.9 | 23.7 | 31.0 | 3.3 | 0.74 |
| **10** | **5** | 67.78 | 63.85 | 46.79 | 63.24 | 73.62 | 5.72 | 13.2 | 23.4 | 32.1 | 3.7 | 0.73 |
| **10** | **0** | 68.50 | 64.65 | 42.79 | 63.63 | 74.34 | 5.66 | 14.2 | 24.0 | 36.0 | 4.6 | 0.77 |
| **100** | **2** | 69.28 | 68.25 | 48.88 | 65.33 | 77.20 | 5.47 | 10.9 | 17.6 | 29.5 | 3.0 | 3.15 |
| **100** | **5** | 69.56 | 68.31 | 51.24 | 66.02 | 74.92 | 4.78 | 9.8 | 17.9 | 26.3 | 3.1 | 3.33 |
| **100** | **10** | 69.87 | 68.37 | 51.82 | 66.37 | 78.50 | 4.91 | 12.3 | 17.9 | 26.9 | 2.9 | 3.45 |
| **100** | **0** | 70.57 | 69.45 | 54.89 | 67.34 | 76.28 | 4.22 | 12.1 | 17.9 | 26.8 | 2.6 | 3.54 |
| **1000** | **2** | 69.43 | 69.49 | 50.36 | 67.33 | 74.92 | 4.41 | 7.0 | 10.4 | 14.7 | 1.8 | 32.29 |
| **1000** | **5** | 69.90 | 70.06 | 54.40 | 68.06 | 75.90 | 4.37 | 5.9 | 10.3 | `15.1 | 1.5 | 33.04 |
| **1000** | **10** | 69.48 | 70.10 | 56.24 | 67.23 | 76.59 | 4.78 | 7.3 | 10.8 | 14.7 | 1.7 | 31.22 |
| **1000** | **50** | 70.20 | 71.28 | 51.65 | 68.61 | 75.93 | 4.42 | 7.5 | 11.0 | 16.4 | 1.5 | 31.86 |
| **1000** | **100** | 71.85 | 72.22 | 59.02 | 69.16 | 76.56 | 3.62 | 7.6 | 11.4 | 15.9 | 1.8 | 33.63 |
| **1000** | **500** | 71.29 | 72.11 | 55.66 | 69.07 | 76.24 | 3.62 | 8.7 | 12.4 | 18.7 | 1.9 | 32.19 |
| **1000** | **0** | 71.74 | 72.54 | 54.04 | 68.99 | 77.22 | 4.06 | 8.3 | 13.0 | 18.6 | 2.1 | 30.80 |

*Fig.12 - Summary of the Test Accuracies and Sizes of 100 runs in each of the 12 configurations, where each of those runs consists of 10-fold cross validation. Tr. Acc. = Average training accuracy, Va. Acc. = Average validation accuracy. Avg. Training Time refers to the average time taken to produce a single model on a single partition of data.*

*Fig. 13 - Graph showing test accuracies by parameter configuration, Heart Disease dataset.*

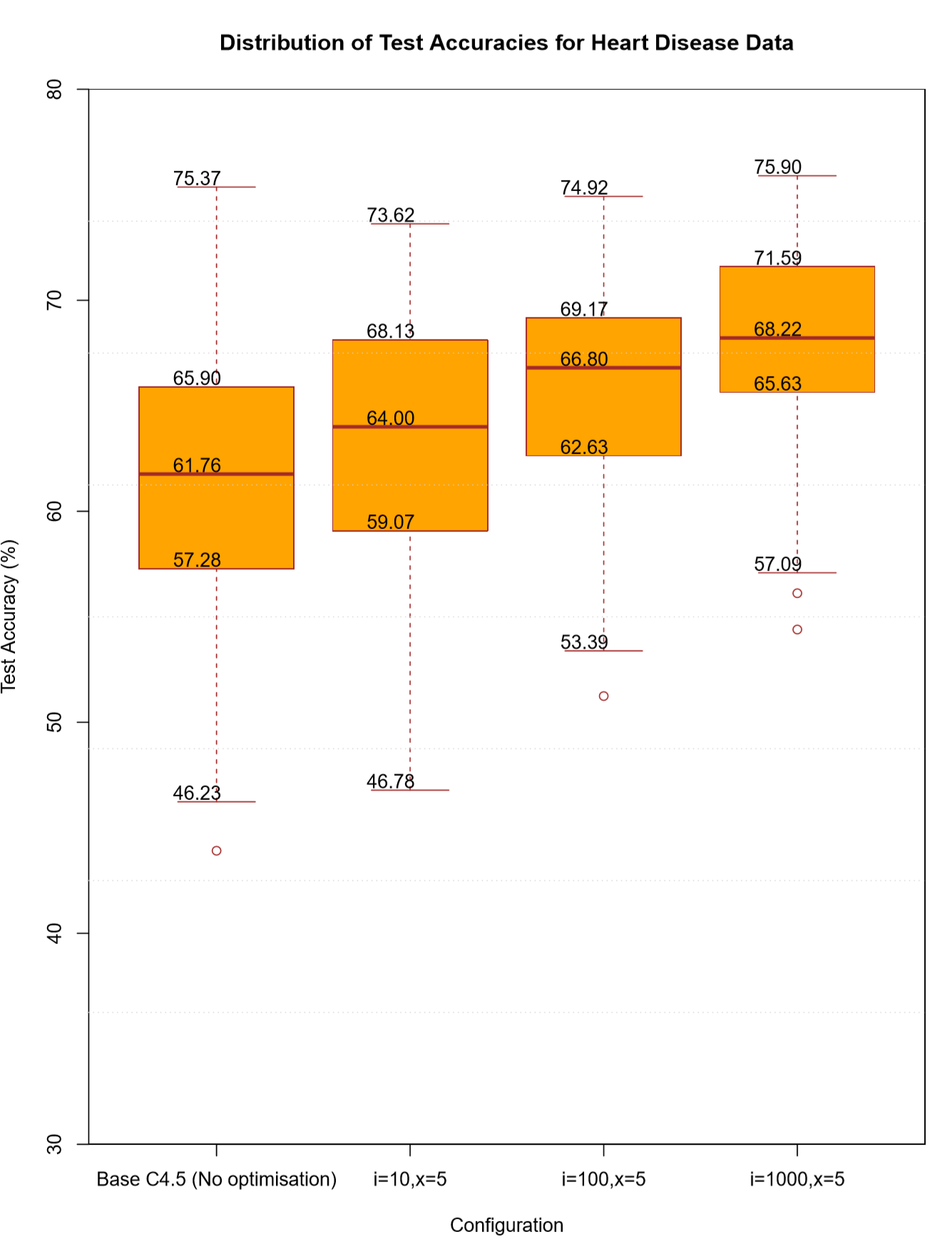
Fig.13 demonstrates that the minimum, average and maximum test accuracies increase as we increase the number of optimisation iterations performed, **i**. We can see that average test accuracy increases at a more stable rate, while there are more fluctuations in terms of the minimum and maximum accuracy values, which is to be expected as these are outliers. The x axis variables have been ordered lexicographically, first in terms of increasing ***i*** and then in terms of increasing ***X.***

We can see from Fig.14 that all optimised models have decreased variance of test accuracies compared to C4.5 alone. Based on this, we tentatively suggest that by performing optimisation on a tree, our model can become more robust and thus be expected to perform more consistently than a tree produced by C4.5 alone. This result could be particularly beneficial when we are to build a classification model on a small sample of data, as we may be able to increase the reliability of our model without requiring additional training data.

*Fig. 14 - Standard deviation of test accuracies for all configurations, Heart Disease dataset.*

Increasing ***i*** beyond 1000 was out of scope for this project due to the computational complexity seen in Fig.15. We see an exponential training time increase as we increase ***i*** by a factor of 10. Consider that to perform 100 runs, each of 10-fold cross validation – we produce 1000 individual models. As the average training time of a single model for ***i***=1000 is 32.15 seconds, the total run time is 8 hours, 50 minutes and 52 seconds. Methods to reduce computational complexity of our framework are discussed in section 8.0.

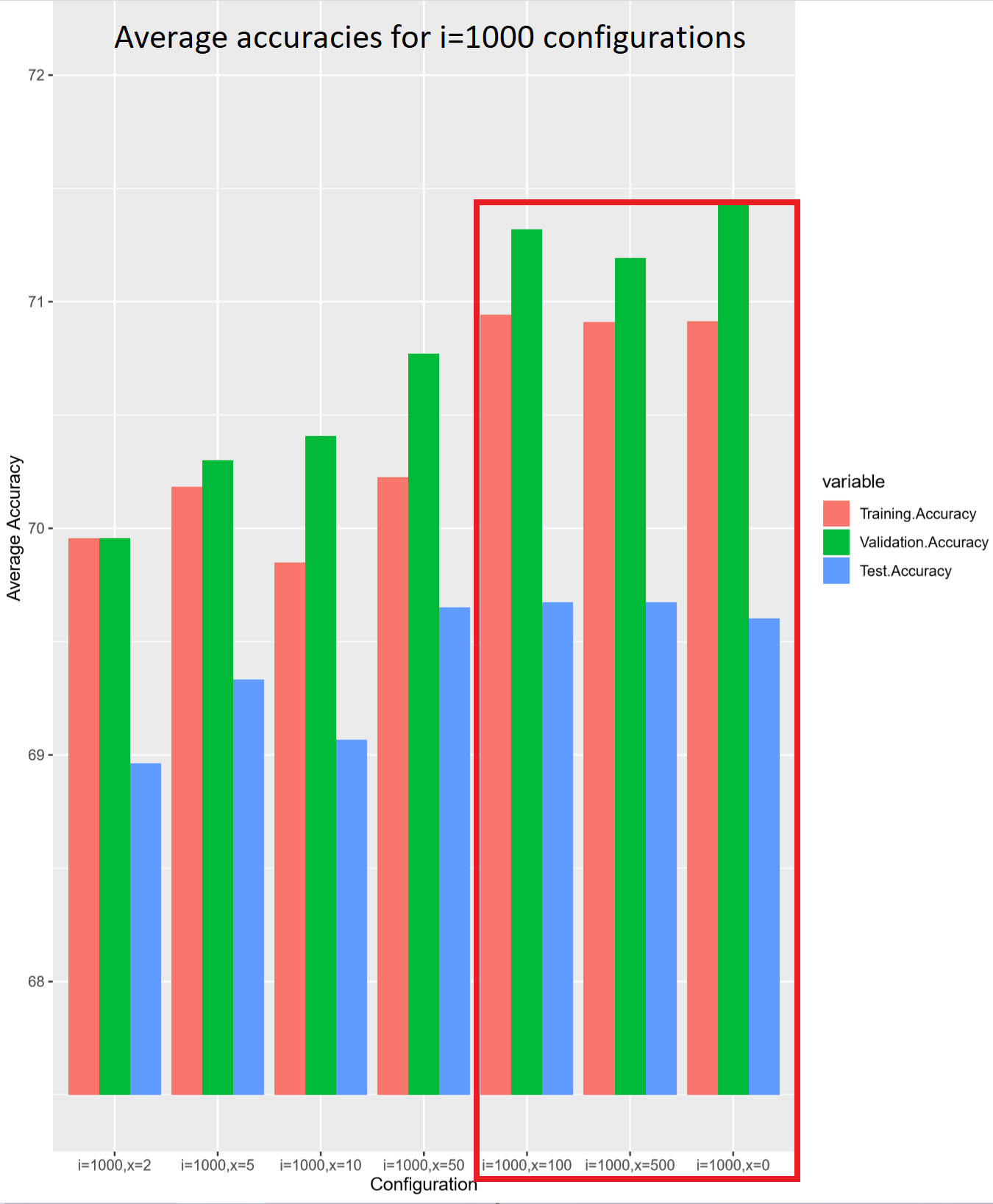
*Fig. 15 - Average time taken to train a single tree model for each value of* ***i****, Heart Disease dataset.*



*Fig. 16 - Test accuracy distribution of different parameter configurations, with* ***X****=5, Heart Disease dataset.*

Fig.16 demonstrates that the more we optimise a model, the higher the 1st quartile, median and 3rd quartile of test results. This can also be seen by looking at the decreasing standard deviation values in Fig.14 as we increase the value of ***i.*** This project leaves scope for increasing ***i*** further to investigate whether accuracy of our models continues to increase, perhaps where greater computing power is available. The results show that we are yet to become stuck at a local extrema at 1000 optimisation iterations, but this is likely to be the case at some unknown larger value of ***i.*** It is unclear how and when a local extrema will become prevalent, however, due to us randomising the validation subsets we optimise with respect to.

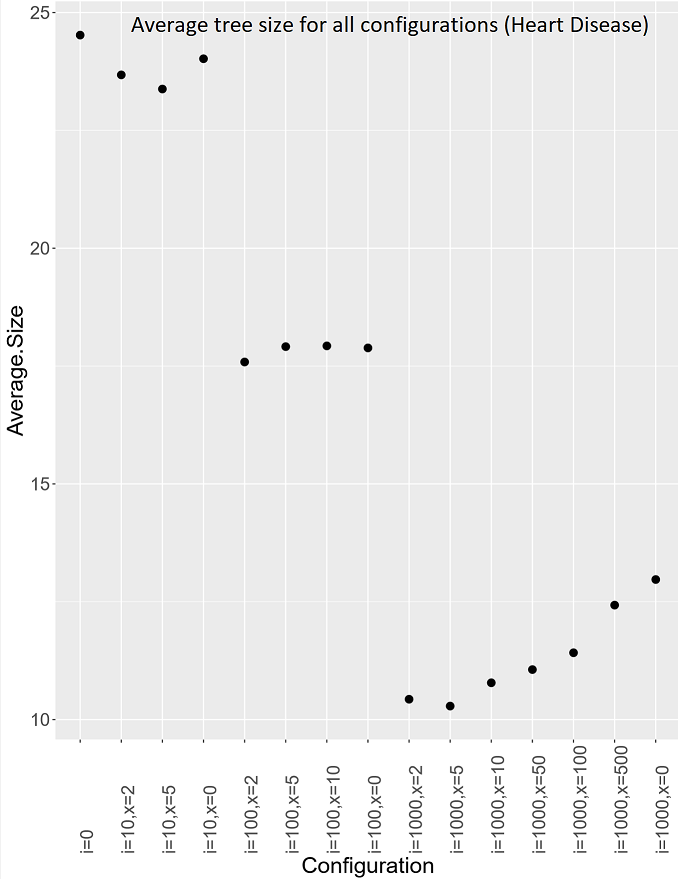
For lower values of ***i***, we see that not randomising the validation subset (***X***=0) is preferable – perhaps as optimisation has not been running long enough to cause overfitting. However, once ***i*** is increased to 1000, we see that randomising the subset infrequently (***X***=100) performs slightly better than running 1000 optimisation iterations on a single partition of the validation set. We observe that while the validation accuracy increases almost linearly the more infrequently we randomise the validation subset, the test accuracies plateau. This can be seen by the similar test accuracy scores but (generally) increasing validation accuracy scores of ***X***=100, ***X***=500, X=0 in Fig.17.



*Fig. 17 - Average accuracy scores for* ***i****=1000 configurations with varying* ***X*** *values on the Heart Disease dataset. The red box highlights the configurations for which test and training accuracy plateau/decrease, though validation accuracy continues to increase.*

We observe that tree size decreases with an increase in ***i***, as seen in Fig.18. This result suggests that by increasing the number of optimisation iterations, we are able to reduce the size of our trees significantly. On the Heart Disease dataset, C4.5 without optimisation produced trees with an average size of 24.5 nodes. Our most effective configuration for minimising size (***i***=1000,***X***=5), produced trees with an average size of 10.3. This shows an approximate 58% decrease in tree size, while simultaneously improving predictive performance.

Interestingly, for **i=1000,** the average tree size remains larger the less frequently we randomise our validation subset (synonymous with increasing ***X***). This suggests that by spending more time optimising with respect to a single subset of the validation data, we are less able to reduce tree size (however, accuracy continues to increase).



*Fig. 18 - Scatter plot showing the average tree size for each configuration on Heart Disease data, with increasing* ***i*** *(left to right)*

4.0.5.2 Abalone Dataset

Constants: *R = 250, K = 10, O = accuracy & size, T = 10%, Tr = 45%, V = 45%, Vi = 50%, P = 0 (no forced mutation), D = Abalone dataset, Optimisation Method: Iterative Local Search*

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  | | | | Test Accuracy | | | | | Size | | | |  |
| ***i*** | ***X*** | Tr. Acc. | Va.  Acc. | Min | Avg. | Max | Std. | RMSE | Min | Avg. | Max | Std. | Total Training Time (seconds) |
| **0** | **n/a** | 25.95 | 23.28 | 20.87 | 23.56 | 24.87 | 1.25 | 2.79 | 14.0 | 17.5 | 21.3 | 2.3 | 48.61 |
| **10** | **2** | 26.26 | 23.89 | 22.55 | 23.83 | 25.45 | 0.94 | 2.81 | 15.7 | 18.0 | 21.7 | 1.9 | 74.63 |
| **10** | **5** | 26.50 | 24.09 | 22.17 | 24.15 | 25.45 | 0.94 | 2.82 | 16.0 | 18.9 | 23.0 | 2.4 | 69.01 |
| **10** | **0** | 26.03 | 23.77 | 21.43 | 23.89 | 25.52 | 1.12 | 2.82 | 14.1 | 17.3 | 20.1 | 1.9 | 77.10 |
| **100** | **0** | 25.60 | 24.69 | 22.31 | 24.10 | 25.59 | 1.03 | 2.86 | 14.7 | 17.9 | 21.0 | 2.4 | 238.62 |

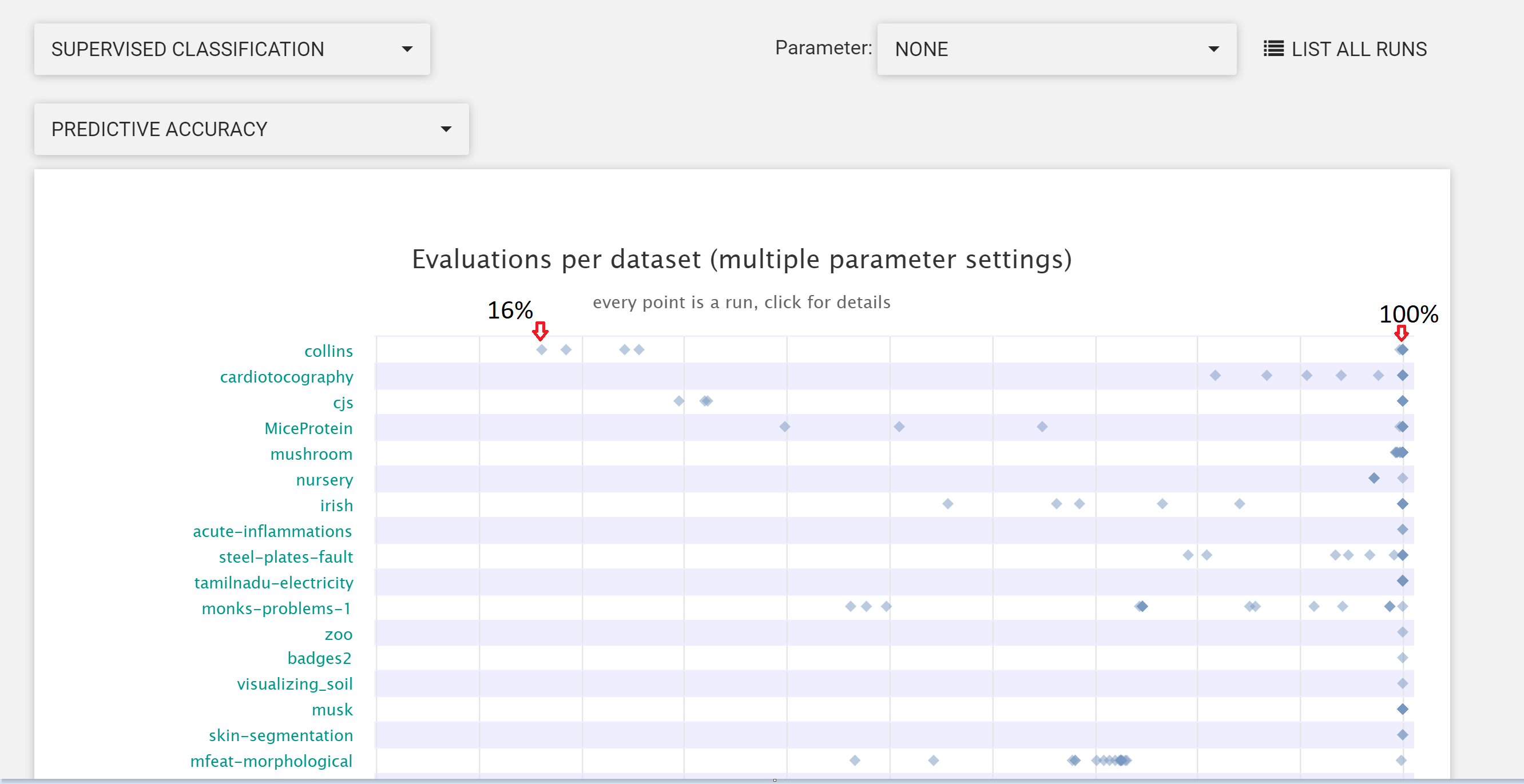
*Fig. 19 - Summary of the Test Accuracies and Sizes of 10 runs in each of the 12 configurations, where each of those runs consists of 10-fold cross validation. Tr. Avg. = Average training accuracy, Va. Avg. = Average validation accuracy. Note that we ran fewer experiments on the Abalone dataset due to it’s size and therefore the computational complexity to train/optimise.*

Note that as we are optimising with respect to prediction accuracy, RMSE is not an objective function and is recorded above in Fig. 19 as an observation rather than a target. Our RMSE metric suggests that we are treating this model as a regression tree, while measuring Accuracy suggests we are treating this as a decision tree. Both performance metrics can be considered within the same set of results and the distinction between regression/decision is trivial in this case as the model can be considered either, though the appropriate metric should be analysed depending on which is chosen. In this test we selected accuracy to be our objective function, rather than RMSE, and this explains why RMSE is seen to increase as ***i*** increases.

We observe that results are less clear for the Abalone dataset, possibly due to the fact that is it much larger than the Heart Disease data and thus many more iterations of optimisation are required to achieve clear results (which requires computing power we do not have access to). Only a very slight performance increase on base C4.5 (***i=0***) was achieved in terms of test accuracy, with all optimised models having an average test accuracy *less than or equal to 0.59 higher* than that of our base algorithm.

5.0 Performance Comparison with Published C4.5 Libraries

Here, we show how our results compare to benchmark instances on both the Heart Disease and Abalone datasets. A ‘run’ refers to the average results of running an algorithm once on a dataset using 10-fold cross-validation.



*Fig. 20 - Showing predictive accuracy of the weka.J48 implementation of C4.5 (Java library, produced by Weka) on selected datasets.* [12][35]

Fig.20 shows us that an identical algorithm can have vastly different predictive accuracy on an identical dataset (I.e. 16% V 100% prediction accuracy for the ‘collins’ dataset). These discrepancies in performance can be explained by different algorithmic parameters, which have differing optimal values unique to each dataset/distribution. Based on this, it can be difficult to directly compare our models with those runs where parameter tuning has been performed as we are likely optimising a tree which has initially been produced by C4.5 with sub-optimal parameters (therefore our initial tree is a sub-optimal model with respect to what C4.5 is capable of producing).

|  |  |  |
| --- | --- | --- |
| Classifier | Dataset | |
| Heart Disease  (Best 10-fold run) | Abalone  (Best 10-fold run) |
| Optimised C4.5 (C#) | 78.50 | 25.45 |
| Sklearn.DecisionTreeClassifier (Python) | 78.22 | 26.27 |
| Weka.J48 (Java) | 74.59 | 26.74 |

*Fig. 21 - Test accuracy comparison to published classifiers [36]*

We compare our results to those of the two most popular C4.5 algorithm implementations, as demonstrated in Fig.21.

The structure of OpenML [36] encourages users to upload verifiable results on particular datasets, for instance, by uploading an algorithm with specified settable parameters, the dataset, and the results. Due to this nature, it is unlikely that users would submit runs with poor performance – therefore we cannot extract an ‘average’ result from the data available as it is likely skewed towards the top end of the algorithm’s performance. Due to this, we compare ‘best’ runs rather than ‘average’ runs for an algorithm. Further data collection is crucial in future to compare the **average** performance of our optimised C4.5 against the average performance of published libraries. To make a proper comparison to published classifier libraries, we should run parallel experiments ourselves so that data collection and parameter configuration is consistent across all frameworks/languages – thus we are making a direct comparison of the classification algorithms themselves. As we are comparing only the best runs in Fig. 21 rather than comparing true averages, we may have to consider our framework as a meta-algorithm which performs 100 x 10-fold cross validation runs and selects the best performing model found. If we consider this to be the case, we see that our meta-algorithm outperforms both standard benchmark algorithms (J48 and scikit) on the Heart Disease dataset (However, to retrieve more significant evidence for this result, the meta-algorithm itself should be run 100 times itself and average results taken – this has not been done as it is considered out of project scope).

Based on our chosen metric, we observe that our optimised C4.5 outperforms both the scikit (Python) and J48 (Java) versions of C4.5 on the Heart Disease dataset but has poorer test accuracy on the Abalone dataset. This can be explained by us running too few iterations of optimisation on the Abalone data as it is much larger, thus we likely do not give our algorithm ample time to improve performance. We are applying our optimisation procedure to a tree produced by C4.5 with sub-optimal parameters, and then comparing this to the results of algorithms with (likely) near-optimal parameters.

By starting with a sub-optimal tree, we are able to improve said tree to the extent that it outperforms trees produced by algorithms with near-optimal parameters. However, we also start with a sub-optimal tree for the Abalone dataset, but do not have time to optimise it enough to match or outperform models produced by C4.5 runs with near-optimal parameters. This logic helps us to understand why our framework does not perform as well on the Abalone dataset, but exceeds benchmark performance for the Heart Disease dataset.

Moving forward, we should implement an additional step within our project and thus do the following:

* 1) Optimise C4.5 parameters
* 2) Optimise the model produced

We have produced evidence that optimisation of trees does improve the accuracy and minimise the size of our trees from the non-optimised version. A worthwhile experiment would be to apply the same optimisation methods after initial C4.5 parameter optimisation has been performed to see if we can improve results further.

Our data suggests that our optimised C4.5 algorithm is not currently suitable for large datasets as it is too computationally complex to perform meaningful optimisation, but is capable of outperforming state-of-the-art decision tree algorithms for small-medium sized datasets. More extensive analysis and further data is required, however, on the performance of published decision tree algorithms to draw any decisive conclusions (Thus we must be tentative with our suggestions.)

To perform an in-depth comparison of our framework with standard published libraries (including an ‘average’ performance comparison), we would likely need to implement them ourselves to collect adequate performance data. Additional scope to this project could include such a comparison.

6.0 Project Planning

This project was managed with a Kanban methodology, utilising a Kanban board as the primary interactive tool. The project was divided into three stories, representing the following time periods and consisting of the key points outlined:

1. October 2019-January 2020

* Initial project scope chosen
* Dataset-specific algorithm completed

1. January 2020 – March 2020

* Generalised algorithm and optimisation completed

1. March 2020 – April 2020

* Experiment design and execution
* Report write-up and technical documentation produced

Momentum was fairly consistent throughout the time period of the project, though dipped slightly out of term-time (particularly December/January). After returning to work in January, it was noticeably more difficult to begin working on code that was written 6 weeks prior and time had to be taken re-familiarising with the code and the project itself. This break was time inefficient, as perhaps if the project was looked at more consistently over the Christmas period then the re-familiarising time needed wouldn’t have been as extensive.

Mid-way through this project the scope was changed slightly, with the decision being made not to pursue the initial optimisation method of CMCS. Instead, we decided to focus on iterative local search with more emphasis on understanding the optimisation of classification tree models in general. Adapting to this change was relatively straight-forward as the initial code for our algorithm was generalised enough to make applying any form of optimisation achievable.

Within this project, we identified the following risks and resolved them as follows:

* **Real-time performance too slow**: Ensured that we could perform a ‘reasonable’ amount of optimisation within a given time frame. In our case, we achieved this by switching the main initial dataset used in our project (Abalone) to a slightly smaller benchmark dataset (Heart Disease) to reduce computational complexity.
* **Inefficient architecture:** Identified as a risk at the beginning of the project when deciding to utilise the DataTable class instead of arrays. The decision was made to continue using DataTable as a more generalised framework could be achieved, allowing us to apply our framework to a wider variety of datasets.
* **Unrealistic schedule:** The decision to switch from CMCS to iterative local search as or optimisation technique was based on this. It was decided that deploying CMCS would not be feasible given the project timeframe considering the size and complexity of CMCS code and the lack of available resources/support materials.

As aforementioned, the scope for experimentation within our framework is extremely large and investigating the effects of varying any of the algorithmic parameters would be a worthwhile experiment. The risk of not achieving our initial goal in this project was rectified by our results, which show that optimisation of classification tree models has been achieved in terms of both size and accuracy.

The biggest challenges faced were in relation to debugging code. There were a few occasions when finding the source of a bug took longer than expected (up to 8 full days), possibly because I previously had little experience of writing recursive code and initially found it counter-intuitive. Throughout the time working on this project, however, my ability to identify flaws in my code and intuition of where to look for bugs increased drastically. An important learning experience from this project was observing how ideas, which initially seemed complex to me, can be implemented within only a few lines of code. Seeing this in practice and implementing these ideas myself has increased my confidence in my programming ability. Where before I would not have felt capable of achieving certain tasks, I now feel much more able to research ideas and implement them independently.

I feel as though the planning of this project was mostly successful. There was a natural split as this project was effectively in two parts – firstly implementing the C4.5 library, then producing an optimisation framework. I was able to initially focus on implementing C4.5 before selecting an optimisation technique at a later date. I feel as though this allowed me to focus on implementing C4.5 correctly without being distracted by how I was going to optimise it without fully understanding it first. Concentrating initially on C4.5 gave me a deep understanding of the algorithm, thus selecting and applying an optimisation technique afterwards became considerably easier.

A defensive style to programming was used at times throughout this project, with less focus on the speed of certain bits of code and more focus on getting it to work as intended. I feel as though this helped me to complete the project on time and am confident that it was the correct decision. Going forward, however, there is scope to improve the speed of our code so that we are able to run more optimisation iterations on a similar level of hardware.

1. Conclusions & Future Work

An unintentional discovery from this project was that optimising a tree model appears to improve robustness and increases the reliability of the model – this is shown by the reduction in standard deviation of the accuracy scores with increasing optimisation. This was not an anticipated achievement when initially devising the project, however this is an encouraging result that can be tested more extensively in future.

As mentioned in our analysis, we are able to better the base C4.5 algorithm for predictive power on smaller datasets but the framework is not currently suitable for larger datasets. We have also produced a framework to consistently minimise the size of trees while maintaining performance. Access to the actual tree data structures produced by published classifiers is difficult to find, as most classifier analysis consists of statistics describing the accuracy of the model, rather than the model geometry itself. However, as we have been able to successfully minimise the size of classification trees without reducing accuracy, the trade-off for increased interpretability is therefore an increase in training time. This compromise may be beneficial in terms of science communication (perhaps in a corporate setting to make decision making more intuitive or to visualise why decisions have been made, or in general where the target audience are not scientists).

The most immediate task which would have been completed with more time is to streamline the code’s usability. Currently, altering parameters requires visiting different areas of the code. A neater way to present this project would be to have a single location where we could alter parameter values, which are then passed to the relevant methods where they are required (For instance, in a single ‘runProgram’ method with parameters as supplied arguments).

The most critical alteration that needs to be made is to improve the speed of our code so that more optimisation iterations can be performed within a reasonable time budget. This process could also be sped up by ensuring that we optimise the parameters of C4.5 before applying our optimisation framework to the tree itself – this would ensure that we are starting with a near-optimal tree (in terms of C4.5’s capabilities), rather than starting optimisation on a sub-optimal model.

There is much scope remaining to investigate altering different parameters of our optimisation model, as well as expanding the optimisation framework itself. For instance, we could add further hill-climbing operators so that we are not always utilising C4.5 to build branches. A different method of optimisation could also be used, such as CMCS [37].

There is also scope on a lower level to perform more targeted optimisation. For example, mutating and rebuilding a node close to a leaf is likely to have less of an impact on the overall structure and performance of the tree than mutating and rebuilding a node close to the root. Node selection could be refined to the level where selection from different areas of the tree is dependent on an additional parameter, such as the initial accuracy of the tree before mutation. If the accuracy is low, we could add a higher weighting to selecting a node near the root. If the accuracy is high, we could add a higher weighting to selecting a node near a leaf.

Alternatively, we could add weightings to our nodes based on how many observations enter them. For example, the root node is entered by all of our data, whereas nodes further down the tree have some fraction of the initial data entering them. By applying a higher probability of selecting nodes for mutation based upon the quantity of data they are entered by, we would simultaneously prevent the creation of disproportionately long branches. As we have not considered tree dimensions in this project, we may be prone to creating unbalanced trees. By adding a probabilistic selection metric, we would be able to prevent such a problem.

This project was never initially intended to be used as a .NET library for C4.5, however the code has enough generalisation power that it could, with modifications, be used as such. If we were to fully deploy our framework to create a single best tree model, we would of course not utilise a test set. The full dataset would be partitioned into an initial training set and a validation set, of some size. As we would only require a single tree to model the full dataset, we would be able to utilise all available training time to optimise one tree, rather than creating many trees as we have done in this project for testing purposes. We would therefore be able to exploit many more iterations of optimisation in the training process, while keeping training time reasonable. Considering this, our code could be used as a C4.5 and/or optimisation library for classification tree induction on small datasets – capable of classifying datasets, producing trees, optimising trees and making predictions in real time.

8.0 References

|  |  |
| --- | --- |
| [1] | E. Alpaydin, Introduction to Machine Learning, Cambridge, MA: MIT Press, 2014. |
| [2] | R. T. &. R. T. Clemen, Making Hard Decision with Decision Tools Suite, Boston, MA: Duxbury Press, 2004. |
| [3] | E. C. S. B. D. A. Manuel Fern´andez-Delgado, “Do we Need Hundreds of Classifiers to Solve Real World Classification Problems?,” *Journal of Machine Learning Research,* vol. 15, 2014. |
| [4] | R. A. Fisher, “The Use of Multiple Measurements in Taxonomic Problems,” *Annals of Eugenics,* vol. 7, p. 179–188, 1936. |
| [5] | V. N. V. C Cortes, “Support Vector Networks,” *Machine Leaning,* vol. 20, no. 3, p. 274, 1995. |
| [6] | I. Rish, “An empirical study of the naive Bayes classifier.,” in *IJCAI 2001 workshop on empirical methods in artificial intelligence*, 2001. |
| [7] | S. D. P Cunningham, “K-Nearest neighbour classifiers.,” University College Dublin, 2007. |
| [8] | D. G. e. a. Kleinbaum, “Logistic regression,” Springer-Verlag, New York, 2002. |
| [9] | T. Dietterich, “Overfitting and Undercomputing in Machine Learning,” ACM Computing Surveys, Department of Computer Science, Oregon State University , 1995. |
| [10] | A. I. D Ignatov, “Decision Stream: Cultivating Deep Decision Trees,” in *IEEE 29th International Conference on Tools with Artificial Intelligence (ICTAI)*, Boston, MA, 2017. |
| [11] | L. &. B. e. a. Gilpin, “Explaining Explanations: An Overview of Interpretability of Machine Learning,” in *DSAA*, 2018. |
| [12] | R. Quinlan, C4.5: Programs for Machine Learning, Morgan Kaufmann Publishers, 1994. |
| [13] | T. B. A. a. N. A. S. H. W. Nugroho, “Performance Improvement of C4.5 Algorithm using Difference Values Nodes in Decision Tree,” in *6th International Conference on Cyber and IT Service Management (CITSM)*, Parapat, Indonesia, 2018. |
| [14] | K. Q. e. a. Wu, “Top 10 algorithms in data mining,” Springer-Vertlag, London, 2008. |
| [15] | S. Guiasu, Information Theory with Applications, New York: McGraw-Hill, 1977. |
| [16] | Y. S. e. a. O Popova, “Entropy and Algorithm of the Decision Tree for Approximated Natural Intelligence,” 2019. |
| [17] | “Lucid Chart,” [Online]. Available: https://www.lucidchart.com/. |
| [18] | W. S. M. P. R. D. A Janosi, “Heart Disease Data Set,” UCI Machine Learning Repository, 1 July 1988. [Online]. Available: http://archive.ics.uci.edu/ml/datasets/Heart+Disease. [Accessed December 2019]. |
| [19] | T. 7. Marine Resources Division, “Abalone Data Set,” UCI Machine Learning Repository, 1994. [Online]. Available: http://archive.ics.uci.edu/ml/datasets/Abalone. [Accessed October 2019]. |
| [20] | E. R. G. e. a. J Ellson, “Graphviz and dynagraph – static and dynamic graph drawing tools,” Springer-Verlag, 2003. |
| [21] | S. v. S. (. Stephanvs), “This extension provides GraphViz (dot) language support for VS Code.,” Stephanvs, 12 December 2015. [Online]. Available: https://github.com/Stephanvs/vscode-graphviz. [Accessed February 2020]. |
| [22] | M. D. (. mdaines), “A hack to put Graphviz on the web.,” mdaines, 8 December 2018. [Online]. Available: https://github.com/mdaines/viz.js/. [Accessed February 2020]. |
| [23] | J. P. (. joaompinto), “Graphviz (dot) language support for Visual Studio Code,” joaompinto, 26 February 2020. [Online]. Available: https://github.com/joaompinto/vscode-graphviz. [Accessed February 2020]. |
| [24] | V. F. (. VRFZ), “DotNetGraph C#/.NET,” [Online]. Available: https://github.com/vfrz/DotNetGraph. [Accessed 2020]. |
| [25] | D. N. W. J A Synman, Practical Mathematical Optimization - Basic Optimization Theory and Gradient-Based Algorithms, Springer International Publishing, 2018. |
| [26] | Y. B. A. F. N Le Roux, Optimization for Machine Learning, Cambridge, MA: MIT Press, 2012. |
| [27] | J. Pearl, Heuristics: Intelligent search strategies for computer problem solving, United States, 1984. |
| [28] | Y. Gurevich, “Algorithms: A Quest for Absolute Definitions,” *Bulletin of the European Association for Theoretical Computer Science,* vol. 81, 2003. |
| [29] | E. S. e. a. A Amritkar, “Recycling Krylov subspaces for CFD applications and a new hybrid recycling solver,” *Journal of Computational Physics,* vol. 303, pp. 222-237, 2015. |
| [30] | H. A. S. A Newell, “Completer Science as Empirical Inquiry: Symbols and Search,” in *Communications of the ACM*, 1976. |
| [31] | W. G. M. D H Wolpert, “No Free Lunch Theorems for Search,” The Santa Fe Institute, Santa Fe, NM, 1996. |
| [32] | Y. Censor, Pareto optimality in multiobjective problems., 1977. |
| [33] | M. J. M. M. S. Z. I G Chikalov, On Optimization of Decision Trees, Berlin: Springer, 2005. |
| [34] | E. H. B. O. C Bessiere, “Minimising Decision Tree Size as Combinatorial Optimisation,” in *LNCS*, Berlin, 2009. |
| [35] | E. Frank, “Weka J48,” [Online]. Available: https://weka.sourceforge.io/doc.dev/weka/classifiers/trees/J48.html. [Accessed 16 April 2020]. |
| [36] | J. N. v. R. B. B. L. T. J Vanschoren, “OpenML: networked science in machine learning.,” SIGKDD Explorations, 2013. |
| [37] | B. G. D Karapetyan, “Conditional Markov Chain Search for the Simple Plant Location Problem improves upper bounds on twelve Körkel-Ghosh instances (Optimization Problems in Graph Theory),” Springer , 2018. |