Efficient and effective machine learning on clinical data with missing information

Adeyinka Lipede

August 25, 2017

MSc in High Performance Computing

The University of Edinburgh

Year of Presentation: 2017

Abstract

The automatic detection and localisation of anatomical landmarks shall prove invaluable for a number of medical imaging applications. In 2014, Toshiba Medical Visualization Services Europe (TMVSE) proposed a technique for automatic detection and localisation[1] based on a previous study by Criminisi et al.,[2] implementing a highly efficient solution utilising random forests comprised of probabilistic boosted trees. The method suggested by TMVSE attempted to deal with effects occurring at image edges due to missing data, due to cropping or otherwise, using trees built with a relaxed version of Quinlan's C4.5 algorithm. Some performance issues were encountered using this new technique. This project attempts to build an efficient version of C4.5 random forests, in order to serve the needs of TMVSE.

Contents

1 Introduction **1**

1.1 Background . . . 1

1.2 Project Aims . . . 2

1.3 For the reader . . . 2

2 Current Environment **4**

2.1 C4.5 Algorithm . . . 4

2.1.1 C4.5 Decision Tree . . . 4

2.1.2 Application of Random Forest to Data . . . 8

2.2 TMVSE Method . . . 9

2.3 TMVSE Requirements . . . 10

3 Working Methodology **12**

3.1 Environment . . . 12

3.2 Hardware . . . 12

3.3 Software . . . 12

3.3.1 Working with Scikit-Learn . . . 12

3.3.2 Testing and Validation . . . 14

3.3.3 Profiling and Debugging . . . 14

3.4 Experimental Methodology . . . 16

3.5 Data Usage and Methodologies . . . 16

4 Implementing C4.5 Random Forests **17**

4.0.1 Binary Tree . . . 17

4.0.2 Missing Values and Incomplete Samples . . . 18

4.0.3 Initial Accuracy Tests . . . 20

4.1 Optimised Implementation . . . 20

4.1.1 Categorical Data . . . 20

4.1.2 Modifying missing values 22
4.1.3 Sample Counting 22
4.1.4 Additional Split Criteria 23
4.1.5 Tree Builder 25
4.1.6 Early stopping criteria 26
4.2 External Interface 28
4.2.1 Rejected Considerations 29

5 Conclusions **32**
5.1 Reflections 32
5.2 Extensions to this work 32
5.2.1 Streaming Data 32
5.2.2 Tree Level Parallelism 33

List of Tables

3.1 Hardware used for tests of the code used for this project. 12

4.1 A collection of terms used in discussions regarding missing data. 18
4.2 Comparison of times taken to fit a C4.5 Decision tree to the MNIST dataset, for various degrees of missing data, using different encodings of missing values. 22
4.3 Comparison of times taken with C4.5 trees using the information gain ratio, and entropy. 25
4.4 Comparison of time and accuracy when training a random forest of 100 trees (max leaf nodes = 1000) for various values of minimum impurity decrease. 50% of data is missing. 27
4.5 Comparison of time and accuracy when training a random forest of 100 trees (max leaf nodes = 1000) for various values of minimum impurity decrease. 50% of data is missing. 28
4.6 Comparison of time and accuracy when training a random forest of 100 trees (max leaf nodes = 1000) using a Taylor's series approximation to the logarithmic function. 50% of data is missing. 31

5.1 Comparison of time taken to train a random forest of 100 trees (max leaf nodes = 1000) using varying numbers of threads. 50% of data is missing. 33

List of Figures

2.1 A basic decision tree, being used to predict a person's level of highest education. Each row in the larger table represents a sample, typically denoted as a vector x_i. The table to the right holds the actual target values, or *class* for each sample, typically denoted by y_i. 5

2.2 A random forest. In the same vein as fig.2.1, trees are trained on vectorsχ_i, with classes y_i. In the case of random forests, the vector χ_i is a subset of the vector presented to the decision tree in fig.2.1, x_i. 8

2.3 Figure highlighting how TMVSE generate samples using bounding boxes. Sample 4 has missing voxels, though clearly these voxels will not represent air. 10

3.1 Flow of data through the SKLearn code for decision trees. **a.** User data is passed to the treebuilder. **b.** The treebuilder tells the splitter which samples should be looked at, for the node in question. **c./d.** For each split position, for each feature, the splitter requests an impurity improvement. **e.** The feature and split position with the best impurity improvement are passed back to the treebuilder. **f.** The treebuilder uses this information to build the next node in the tree. 15

4.1 Representation of sample distribution to child nodes. **a.** All samples are complete. An array is used to hold all samples, and the child nodes are simply passed the start and end positions for their partitions. The samples can then be shuffled at will by each child node.**b.** The fourth sample is missing. C4.5 dictates that both child nodes should receive this sample. If one samples array is used, as in *a.*, sorting/shuffling of samples cannot occur. **c.** The fourth sample is missing. It is pushed to the end of the samples array. The child nodes are passed the start and end positions for their own partitions, and are free to shuffle these samples. The incomplete sample is stored separately for each child. . . 19

4.2 Illustration of sample counting by the criterion. The total number of samples is known to the criterion already. The blue and purple arrows represent the number number of samples that *could* be counted by the criterion. Clearly it is most efficient to measure the number of samples for child 1 and 2 (the blue arrow), and then calculate the number of samples in child 3 by subtracting from the total number. 23

4.3 Comparison of accuracy on increasing missing data, using the C4.5 algorithm with information gain criterion, and the CART algorithm with information gain. The 5-fold cross validity score is used here as a measure for accuracy. 24

4.4 Plot of fraction of missing data vs. 5-fold cross validity score, for a C4.5 and CART random forest, both using the entropy criterion. For the CART tree, missing values are simply set to zero. 26

4.5 Illustration showing how a tree of arbitrary number of child nodes can be represented as a binary tree. 31

5.1 Illustration showing how a tree of arbitrary number of child nodes can be represented as a binary tree. 34

Acknowledgements

It is no exaggeration to say that this project, and this course could not have been made possible without the help of a number of people. Firstly, I would like to give thanks to Ian Poole, Alison O'Neal, and Malcolm Illingworth over the course of this project, for their constant encouragement and patience. I would also like to give thanks to Mario Antoniolietti, Ben Morse, and Terry Sloan, who have been incredibly helpful when things have been more difficult throughout the year. Finally, this course would not have been possible without my family and girlfriend, without whose constant support, I would not have been able to do this course.

Abbreviations

TMVSE	Toshiba Medical Visualization Systems Europe
DAL	Detection and Localization
3D	Three dimensional
CT	Computer Tomography
MRI	Magnetic Resonance Imaging
DICOM	Digital Imaging and Communications in Medicine
NaN	Not a Number

Chapter 1

Introduction

1.1 Background

The detection and localization (DAL) of anatomical landmarks[*] is an important requirement for many medical imaging tasks. When working with DICOM data, a first step for many machine learning algorithms might be to detect the region or regions of interest to be worked on.

This can be done manually by a medical expert, but can often be a large undertaking to curate a database of appropriately labelled images in the volumes required. Also, where pre-labelled data has been procured, labelling may not be consistent across sources. For this reason amongst others, automatic detection has been a topic of interest for many studies.

Several approaches have been proposed to segment regions of interest, using a wide variety of techniques. These vary both in effectiveness and computational complexity. Indeed, while deep convolutional neural networks are often seen as the 'state of the art' for this type of problem, training these can take many hours, to days, dependent on the number of layers of the network[†]. Classifying a single image can also take a very long time, for the same reason, especially if said image is large. A study carried out by

[*]'Landmark' here refers to a site of interest on the body. A simple example here might be the centre of an eye socket, or the base of a gland.

[†]Convolutional neural networks are an algorithm often used for imaging problems. As a gross simplification, a number of transformations are applied to various representations of the original image, in an iterative manner, until some arbitrary minimum on error is achieved. We may take depth as a proxy for algorithmic complexity.

Criminisi et al.[2] in 2009 outlined an efficient method for automatic DAL using random decision forests comprised of probabilistic boosting trees. Using this technique, they were able to develop a model capable of determining landmarks within an image in approximately 2 seconds.

The above method did not address (or specify how it addressed) effects encountered at the edge of an image. A great degree of diversity between imaging subjects is possible, and images may have been cropped differently if they were provided from different sources. The treatment of this missing data can then have bearing on the resultant effectiveness of the algorithm used.

In 2014, Toshiba Medical Visualization Systems Europe (TMVSE) proposed a modification to the decision forest method. Their algorithm used a random forest comprised with a simplified version of the C4.5 decision tree, devised by Quinlan,[3] which provides a method for dealing with missing data. They would now like to implement the C4.5 algorithm fully within the Scikit-Learn framework (referred to interchangeably as SKLearn), to test its effectiveness, and to integrate the code better with their coding practices.

1.2 Project Aims

This project aims to implement an efficient version of the C4.5 random forest method suggested by TMVSE. The code should follow the style of the Scikit-Learn project, in the hope that it might be accepted into the main codebase. It should also provide sufficient options that a user might be able to experiment, trading a small amount of accuracy for increases in speed.

1.3 For the reader

There are five chapters in total in this document. The user may choose to peruse these in a different order, so a summary is provided of each.

Chapter 2 gives the reader an in-depth explanation of the background theory for the project, setting the context for the experiments carried out later. The algorithm is compared with another related algorithm, and some extra detail is given regarding the requirements of TMVSE, to provide an explanation for various design choices.

Chapter 3 details the environment that was used for all code developed during this project, and will be useful for anyone trying to extend it. A summary of which datasets were chosen for investigation, as well as the techniques that were used to analyse the accuracy of the code is also provided.

Chapter 4 is an implementation walkthrough, discussing design choices and incremental changes, with appropriate timings and justifications. Rejected design choices are also discussed. Finally, the conclusion is given in Chapter 5.

Chapter 2

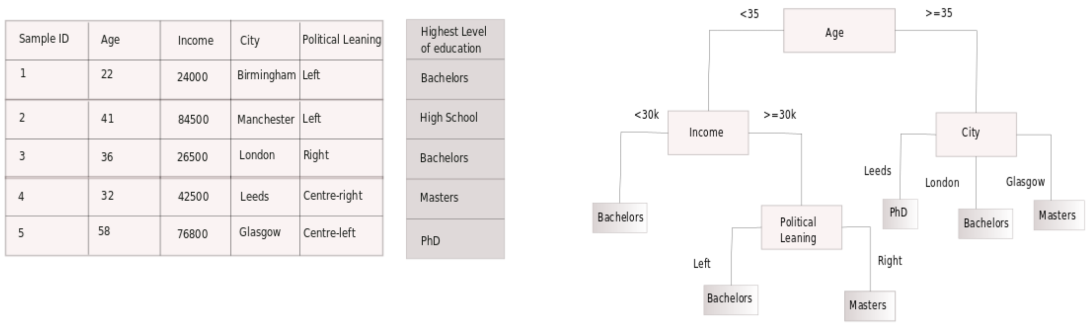
Current Environment

2.1 C4.5 Algorithm

As stated previously, TMVSE utilises a random forest comprised of decision trees built using the C4.5 algorithm. Some explanation for these concepts is provided below.

2.1.1 C4.5 Decision Tree

Decision tree algorithms are a set of supervised machine learning algorithms for classification. These types of algorithms produce a model which can accept a vector of input values (hereby referred to as a sample), and produce a predicted target value, or class. Perhaps the simplest way to conceptualise them is as a simple collection of 'if-then-else' rules. This is illustrated in fig.2.1. For each node in the tree, the sample is subjected to a simple test. Dependent on the result of the test, it is sent down a particular tree branch. Eventually the sample reaches a terminal, or leaf node, for which a target value, or class is defined.



Sample ID	Age	Income	City	Political Leaning
1	22	24000	Birmingham	Left
2	41	84500	Manchester	Left
3	36	26500	London	Right
4	32	42500	Leeds	Centre-right
5	58	76800	Glasgow	Centre-left

Highest Level of education
Bachelors
High School
Bachelors
Masters
PhD

Figure 2.1: A basic decision tree, being used to predict a person's level of highest education. Each row in the larger table represents a sample, typically denoted as a vector x_i. The table to the right holds the actual target values, or *class* for each sample, typically denoted by y_i.

To build, or train the model, an original set of training samples are used, for which along with their input feature[*] values, a known class is associated. The training process is explained as follows.

At each node in the tree the 'best' test is searched for, hereby referred to as an optimal split. This split should be the one that partitions the training samples amongst their respective classes in the 'best' possible manner. 'Best' here means the split that separates samples results in as little variation in classes as possible in any one partition. A quantitative measure of the degree of mixture of classes amongst partitions can be calculated, and is known as the *impurity*. The features are iterated over, and the one that reduces, or improves the overall impurity by the greatest amount is chosen as a test at the node. An *impurity criterion* is a heuristic that can be used to calculate the quality of a split. For

[*]Using the analogy of a vector, a feature refers to a particular index of the sample. From fig.2.1, this could also be seen as a column in the table.

the C4.5 algorithm, the *entropy*, or *information gain* is suggested to measure impurity, and the criterion suggested to measure the impurity improvement is information gain ratio.[3] If we have K classes present at a node with S samples, the probability of any sample having a class k is:

$$P(C_k) = \frac{freq(C_k, S)}{|S|} \tag{2.1}$$

Taking C as a discrete random variable over classes on the node, its *information*, or *entropy* is defined as[†]

$$H(C) = -\sum_{k=1}^{K} P(C_k) \times log_2(P(C_k)) \tag{2.2}$$

If a parent node is partitioned according to some test T into n child partitions, or child nodes, the new entropy is defined as the sum of entropies of all child nodes, weighted according to the fraction of samples W_i/W_S that are in each node:

$$H_T(C) = \sum_{i=1}^{n} \frac{|W_i|}{|W_S|} \times H(C_i) \tag{2.3}$$

The gain in entropy from the test is simply $gain(T) = H(C) - H_T(C)$. The maximum possible value for this occurs when $\forall k \; P(C_k) = 1$, or every child node contains only one class. Often decision tree algorithms use this criterion to decide optimal splits, but the C4.5 algorithm goes further to define a quantity known as the split info:

$$H_{split(T)} = -\sum_{i=1}^{n} \frac{|W_i|}{|W_S|} \times log_2(\frac{|W_i|}{|W_S|}) \tag{2.4}$$

The gain ratio is then $gain(T)/H_{split(T)}$. This places a penalty on splits with a large number of child nodes. An extreme example of why this is useful would be if the feature tested was discrete, and was a unique sample ID number, for example. Ideally such a feature would have been removed from the dataset prior to training by the user. However, if the criterion used was entropy gain, each child node would contain only one class, and so this feature would be selected as optimal for splitting - even though the test would be useless during testing.

When a test involves a discrete feature, samples are partitioned according to every value seen. For a continuous feature at a node with S samples, the samples are partitioned

[†]This can be understood as the average number of bits needed to determine which class a sample belongs to at the node.

into two child nodes. The algorithm tests $S - 1$ possible thresholds for this feature, to determine the maximum gain ratio. Once a feature is set as a test at a parent node, it is never used again for any child nodes. This process is continued moving down the tree until every node contains only one class, or there are no features left to test.

From the Scikit-Learn decision tree implementation page, for a balanced binary decision tree (i.e. one containing only continuous features, or discrete features with only two values), training complexity is $O(n_{features}n_{samples}^2 log_2(n_{samples}))$.[4]

C4.5 with Missing Data

An important benefit when using the C4.5 algorithm is its ability to handle feature data that has been omitted, both for training and testing. Simple approaches for dealing with this missing data might be to remove any samples affected, or simply set the missing feature to some default value.

The C4.5 algorithm handles this as follows. During training, the gain is weighted according to the probability that a sample has a known value:

$$gain(T) = P_{known} \times (H(C) - H_T(C)) \tag{2.5}$$

The split info is also modified, by treating the unknown samples as an extra group, such that the sum runs over $n + 1$ partitions. The *incomplete* samples are then passed to all child nodes, with their weights modified according to the fraction of known samples that went to each node. This may in practice increase training time dramatically. If a very large percentage of features are missing, then most splits will be suboptimal, and the tree will continue to grow, until it produces very small leaf nodes.

During testing, if a missing value is encountered at a node, the testing sample is sent down all of the child paths of that node. Each path is weighted according to the percentage of training samples that took each path. This changes the classification of a sample to a probability statement over all classes. Generally the class with the highest probability is chosen as the classification value for the sample. As the maximum size of a binary decision tree is $log_2(n_{samples})$, the complexity for training becomes $O(log_2(n_{samples}))$. This will occur if the training sample is comprised solely of missing data.

2.1.2 Application of Random Forest to Data

A well-known issue with decision trees is their tendency to overfit. This is simply to say that the tree fits ’too well’ to the training dataset, such that it lacks robustness for unseen datasets. The random forest is a technique that deals with this issue, and is able to increase accuracy, creating a more complex classifier without overfitting.[5] It is one of a number of ensemble techniques; in particular a bootstrapping[‡] method. The general behaviour is shown in fig.2.2. Random subsets of the original dataset are taken (with replacement), and used to build multiple decision trees. When predicting using these trees, their results are then averaged.

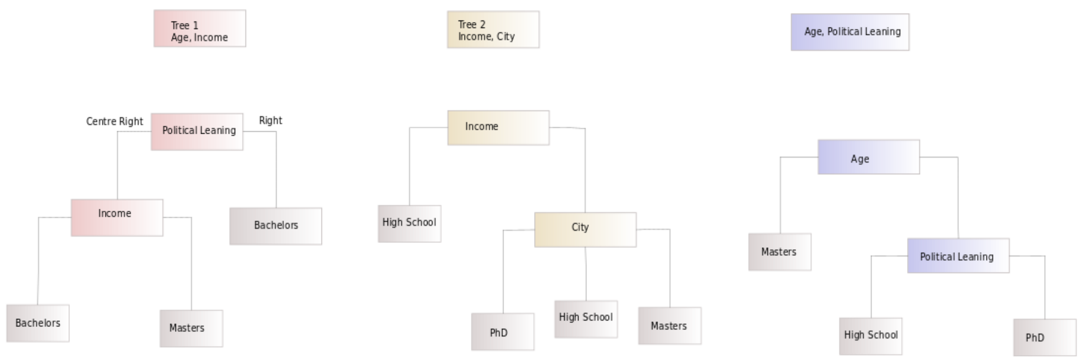


Figure 2.2: A random forest. In the same vein as fig.2.1, trees are trained on vectorsχ_i, with classes y_i. In the case of random forests, the vector χ_i is a subset of the vector presented to the decision tree in fig.2.1, x_i.

[‡]Bootstrapping refers to splitting a dataset into subsets with replacement.

2.2 TMVSE Method

A brief schematic of TMVSE's method for DAL is provided in fig.4.1. A sample is defined in the following way. For each voxel[GLOSSARY] in each two-dimensional image slice in the 3D dataset, its intensity as well as that of its M nearest neighbours are recorded. This is used as a single sample, where the intensities are feature values. In order to maximise the use of the information present, the neighbours of these edge voxels[§] are marked as having unknown values. A C4.5 random forest is then trained on these samples.

[§]A voxel is a **Vo**lumetric pixel, and is analogous to a pixel in a 2D image.

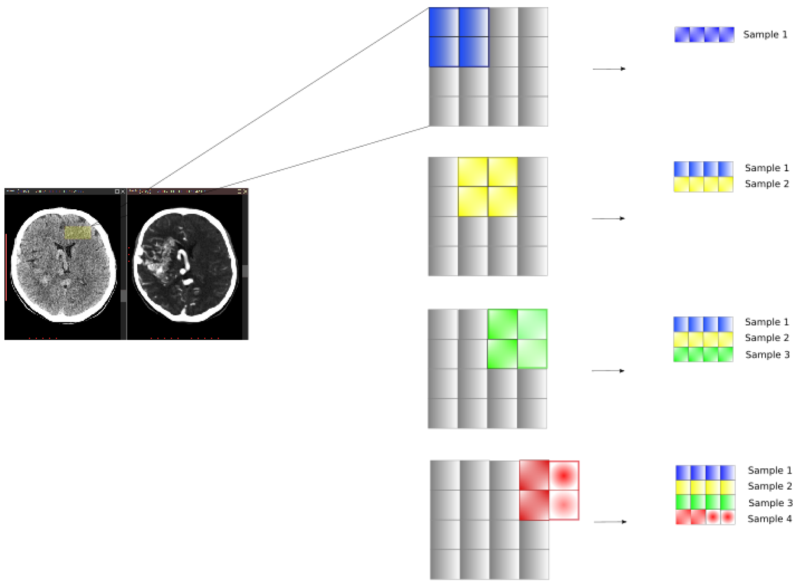


Figure 2.3: Figure highlighting how TMVSE generate samples using bounding boxes. Sample 4 has missing voxels, though clearly these voxels will not represent air.

2.3 TMVSE Requirements

Much of the work done by the data scientists at TMVSE is currently is written in Python. The language is fairly high-level, and easy to use, so allows for rapid prototyping. There are also a number of libraries which can be used for many common machine learning tasks. Of these, the Scikit-Learn framework is used extensively. TMVSE would therefore like for C4.5 decision trees and random forests to be implemented within Scikit-Learn.

Ideally, the code would be merged into the main Scikit branch, ensuring that the C4.5 code can be used alongside any updates to Scikit. This code is intended to run on a typical desktop or laptop, with only the typical dependencies needed for the normal Scikit code to run.

Chapter 3

Working Methodology

3.1 Environment

3.2 Hardware

All code for this project was run on a Clevo W650RB laptop. Specifications are shown in 3.1. This was done to keep in line with the requirement from TMVSE that code run on commodity (non-high performance) machines.

CPU	RAM	GPU
Intel(R) Core(TM) i7-6700HQ CPU @ 2.60GHz	Kingston 16GB (2x8192MB) DDR3	Nvidia GT940M

Table 3.1: Hardware used for tests of the code used for this project.

3.3 Software

3.3.1 Working with Scikit-Learn

Scikit-Learn is an open-source package is written using a mixture of Python and Cython code.[6] Cython is a library that allows a user to dynamically write C extensions, using an easier Python-style syntax. This is particularly useful for performance critical regions of code, and is needed for many calculations in Scikit. Some of the benefits of using this include the use of contiguous C arrays and access to pointers, rather than the overhead that comes with the use of Python's memory structures. Each *.pyx* Cython file is compiled to a *.c* file, through the use of the user's C compiler. Optimisation flags can be passed to the compiler in much the same way as with standard C code.

Scikit-Learn has a large open-source community and has existed for some time now, and as such enables many optimisations for a wide variety of algorithms. However, the code is not particularly well optimised for large datasets. This is because most of its algorithms are designed for use with datasets held in memory, and the possibility to stream data from disk is not currently implemented. A few Scikit-Learn algorithms are designed to fit in increments, but this is not the case for decision trees. Unfortunately this placed some limitations on the size of dataset that could be used for testing, due to a lack of sufficient RAM.

Scikit-Learn for Decision Trees

In order to provide some context to the changes made in Chapter 4, a brief overview is provided detailing how the code ties together.

The decision tree algorithm implemented in Scikit-Learn is the CART[8] tree. This is very similar to the C4.5 tree algorithm described earlier, but (1) only binary splits on continuous features are supported, (2) the gain described in Chapter 2 is used as a split criterion in place of gain ratio, and (3) missing data is not supported.

There are four main components to the Scikit-Learn decision tree implementation, each implemented as a class.

Tree

The CART tree is a binary tree structure, whose nodes are represented as an array of structs. Each node in the tree is a struct containing information such as which feature to split on, the ID of the child nodes, or the threshold with which to split at. This is used during the testing phase - a sample moves down the array, starting at the root node, until a leaf node is reached. The tree also maintains an one-dimensional array of the weights of samples with each class, for each node. Once a sample reaches a leaf node, the node's class weights are retrieved. Using these, a prediction can be made, taking the class with highest weight to be the predicted value. Alternatively, class probabilities can be returned.

TreeBuilder

The tree is built using one of two graph traversal algorithms - a depth-first algorithm, and a best-first algorithm*. If left to grow fully, these produce identical trees. However, if limitations are placed on the extent of the tree, the depth-first algorithm will use less memory for building, but the best-first algorithm will produce trees better fitted on the training data.

Splitter

For each node to be expanded, the tree builder calls a splitter class. In short, this determines the optimal split for the node, and calculates various meaningful quantities to be used in calculations for child nodes.

Criterion

Finally, for each potential split, a criterion of choice calculates the change in entropy for the split in question.

A diagram highlighting the above is shown in fig.3.1

Things to note

The code for this project was developed with the intention of eventually contributing to the Scikit-Learn framework, and as such was written in a similar style to the CART decision tree implementation. This was done to allow interoperability between with other sections of the framework.

3.3.2 Testing and Validation

Where possible, the decision tree tests within Scikit-Learn were modified to run on C4.5 trees.

3.3.3 Profiling and Debugging

To enable debugging symbols within Python, a version was compiled in debugging mode.[9] As GDB 7.2+ supports Cython,[10] it was installed for debugging. Code profiling

*These are explained in full in Appendix TREEBUILDERSSS

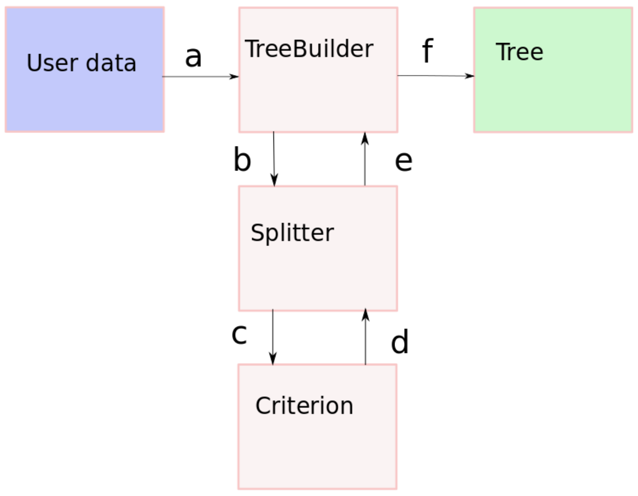


Figure 3.1: Flow of data through the SKLearn code for decision trees. **a.** User data is passed to the treebuilder. **b.** The treebuilder tells the splitter which samples should be looked at, for the node in question. **c./d.** For each split position, for each feature, the splitter requests an impurity improvement. **e.** The feature and split position with the best impurity improvement are passed back to the treebuilder. **f.** The treebuilder uses this information to build the next node in the tree.

was done using Python's *yep* module. Cython compilation flags were modified during profiling (no optimisations, debugging flag).

3.4 Experimental Methodology

All experimental timings were tested for a single decision tree. In all cases, the code was run five times. A single tree was chosen for timings in opposition to a random forest, as the two are directly linked, and it was simpler to establish an initial state for the tree (e.g. number of features in use) for a single tree.

Where significant changes were made to the code, timings were gleaned prior to the change being made for later comparison. Otherwise, analyses on parameter modification can be checked using the code provided.

3.5 Data Usage and Methodologies

As TMVSE was interested primarily in the classification of images, some priority was given to testing this code with continuous data. However, later in the project it became apparent that discrete or mixed valued data may be useful in a handful of cases, and so a discrete dataset was also used.

Continuous data was tested on a preprocessed, reduced size MNIST dataset.[11] This dataset was recommended by TMVSE, and is a collection of handwritten digits. Conveniently, Scikit-Learn provides this dataset. The learning task is to determine which digit has written.

Discrete data was tested on the UCI Chess dataset.[12] This dataset contains the positions of a number of chess pieces, and the learning task is to determine whether a draw will occur, or the optimal number of moves for a win.

Effectiveness of the algorithm was measured using the random forest 'out of bag' error from the random forest.

Chapter 4

Implementing C4.5 Random Forests

The following chapter details the motivation behind the decisions made when implementing C4.5 decision trees. Some results are also provided, comparing the accuracy of a small dataset with missing data, against the CART algorithm with various imputation methods.

Motivations and Constraints

The code for this project was developed with the intention of eventual submission to the Scikit-Learn project. Because of this, there were a few constraints that it was developed to adhere to. Where possible, the Scikit-Learn contributor guidelines[13] were followed. By staying as close to the original CART decision tree code as possible, it was possible to take advantage of functions that built on top of this code. As an example highlighting this - once the base decision tree was implemented, minimal modifications were necessary to enable C4.5 random forests.

4.0.1 Binary Tree

As discussed earlier, the data that TMVSE intends to use this code primarily on image data. As the sample features are continuous voxel intensities, an initial focus was given to a binary implementation of the C4.5 tree. This led to fast initial development, as much of the base structure was already in place. Eventually, support for data with

Missing Value	A feature value found to be missing
Missing Sample	A sample with a missing value for the currently investigated feature, at the current node
Complete Sample	A sample for which no missing value have ever been found for an optimal split (including parent nodes)
Incomplete Sample	A sample for which a missing value was *previously* found for an optimal split at a parent node

Table 4.1: A collection of terms used in discussions regarding missing data.

discrete features would be implemented through the implementation of a non-binary tree.

4.0.2 Missing Values and Incomplete Samples

Perhaps the largest difference between the C4.5 and CART algorithms is their handling of data with missing values. This was responsible for most of the changes to the original code. For clarity in the following discussion, a few phrases are provided with their meanings in table 4.1.

Training

Fig.[partition] shows how training samples are partitioned amongst nodes by the splitter, during the training process in Scikit-Learn. This is viable because it is impossible for sibling nodes to share samples. For each feature, the samples in a node partition are sorted according to the feature value of each sample. Once the optimal feature to split on is discovered, the start and end positions for subsets of the partition are passed down to child nodes. This is not possible when training with missing data using C4.5. According to the C4.5 algorithm, if a sample has a missing value for the optimal feature, it should be marked as an incomplete sample, and passed to all child nodes. The sample's weight should also have different weights for each child node, representing the proportional size of the known samples of the child node, compared with the parent node. Clearly this mixture of complete and incomplete samples cannot be represented in the same manner. Fig.[c4.5partition] further highlights this. Instead, a second, *mixed_samples* array of size $n_{samples}$ was defined for the splitter, containing both complete and incomplete samples. A *completeness_weight* array was also defined, denoting the adjusted sample weight for all samples.

By doing this, it was posible to avoid the need for redefinition of many *Criterion* class

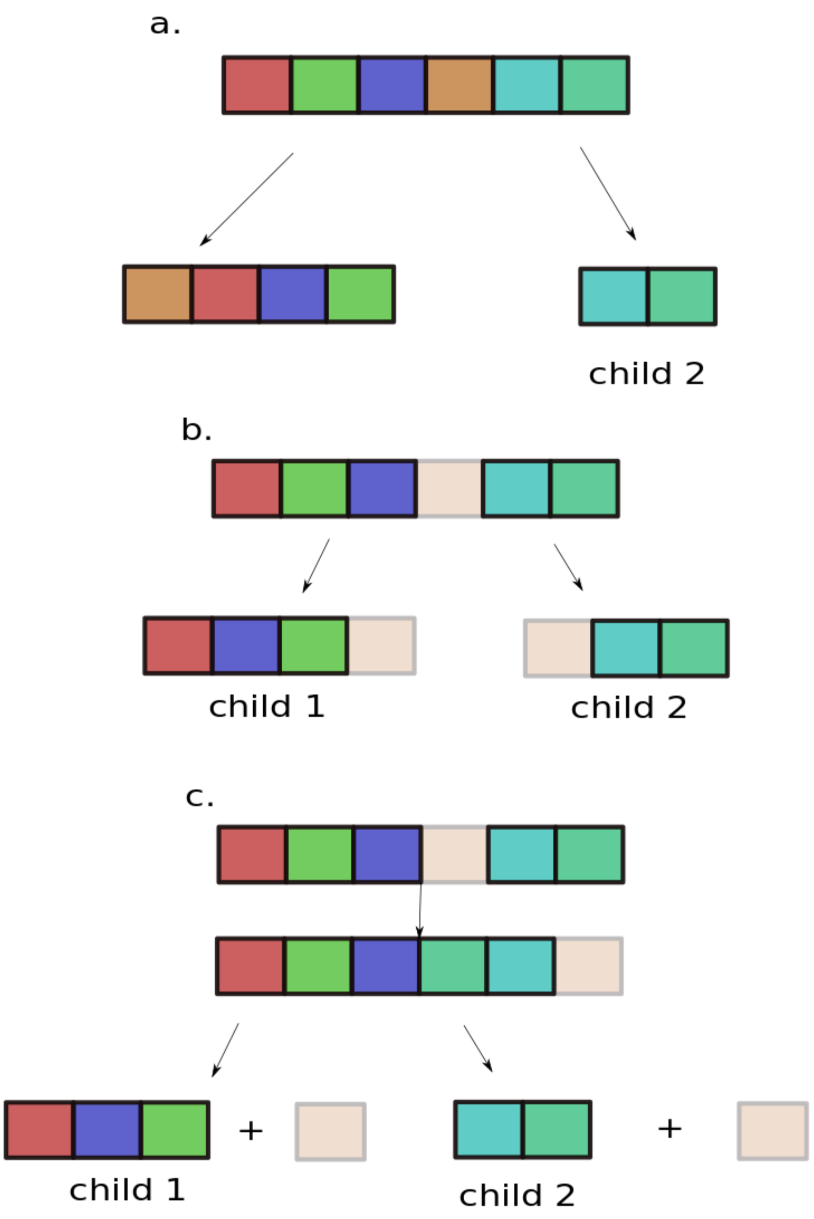


Figure 4.1: Representation of sample distribution to child nodes. **a.** All samples are complete. An array is used to hold all samples, and the child nodes are simply passed the start and end positions for their partitions. The samples can then be shuffled at will by each child node.**b.** The fourth sample is missing. C4.5 dictates that both child nodes should receive this sample. If one samples array is used, as in *a.*, sorting/shuffling of samples cannot occur. **c.** The fourth sample is missing. It is pushed to the end of the samples array. The child nodes are passed the start and end positions for their own partitions, and are free to shuffle these samples. The incomplete sample is stored separately for each child.

functions. The class reference to the array of complete samples could simply be replaced, along the partition boundary positions. It was also necessary to change the *TreeBuilder* classes slightly. The *DepthFirstTreeBuilder* and *BestFirstTreeBuilder* work in a similar manner. Data relevant to next node expansion is stored using a stack, for the *DepthFirstTreeBuilder*, and a priority queue, for the *BestFirstTreeBuilder*. In simple terms, these are both implemented as arrays of structs, each struct relating to a specific node. Pointers to memory locations for partial weights, and incomplete samples were added to these structs, as well as an integer value for the number of incomplete samples at the node.

4.1 Optimised Implementation

4.1.1 Categorical Data

Perhaps the biggest modification to the code after the initial implementation was the inclusion of the capability to deal with categorical data. As explained in section *2.1.1*, if a discrete feature is chosen as optimal for a split, a child node is created for each feature value. Trees need not be binary, and nodes can have an arbitrary number of children. However, the code base is designed specifically for binary trees. To accommodate for this, special *NonBinary* classes were created. These replaced the **Tree**, **TreeBuilder**, and **Splitter** classes. In order to support the addition of other tree building algorithms, 'C45-like' code was segregated from *NonBinary* code. A C4.5 tree builder and splitter were implemented, inheriting from the relevant *NonBinary* code. Additional 'multi-child' functions were also created within the **Criterion** classes.

Within the constraints imposed, only the *best-first* algorithm for tree building was implemented for C4.5. The rationale for this was the expectation that using this algorithm should allow for the creation of 'better', more accurate trees, when using factors limiting tree growth. Initially, only the 'gain ratio' was enabled as a possible split criterion.

Motivation

The capability for dealing with a larger assortment of data was the main driving force for making these changes. Trees could be built specifically on discrete data, or a discrete feature could be added to a continuous datasets, to provide a tree with some extra relevant information. Using the context of the DICOM data TMVSE works with, connected component labelling[16][17][*] could be used before training to provide an initial approximate guess as to the region that each voxel occupies. Aside from the benefit of adding the capability to deal with different types of data, the following shows how training for categorical features is actually computationally cheaper than for continuous ones.

To reiterate, when analysing the split quality for a discrete feature with C4.5, samples are assigned to different nodes, according to their value. In the context of Scikit-Learn, the distribution of samples amongst nodes is represented by an array of sample numbers. The act of 'assigning' samples is simply to shuffle and sort these numbers over the array, and then to define a start and end position on the array for each node. Scikit uses the introsort[18] algorithm for sorting, which has a time cost of $O(n_{samples}log(n_{samples})$. For each continuous feature, samples are sorted once, and then $n_{samples} - 1$ split qualities are tested. For discrete features, only one split quality need be tested. Two choices were considered for discrete value representation.

1. Pass the splitter references to the original (string) values supplied by the user.
2. Note all possible values for each discrete feature, and map each to a number. Pass these numbers to the splitter.

The first choice allows for user-supplied values to be used as-is, reducing some initial time taken for modifying data. By assigning each value to a number, the introsort algorithm could be used. This would require a one-pass scan through the samples (to check for which values are present), and possibly the use of a hashmap to store value-to-number encodings. Access time for the use of this structure is $O(n_{samples})$.[19] Scanning across all samples is $O(n^2_{samples})$ for each feature. Therefore the total cost at a node is $O(n^2_{samples}n_{features})$. The latter option requires an initial modification to the data provided by the user, of $O(n^2_{samples}n_{features})$, but following this, complexity is only $(O(n_{samples}log(n_{samples}))$.

4.1.2 Modifying missing values

As discussed previously, consideration was taken over how missing values should be presented by the user. For numerical data, Scikit only prevents the use of the values

[*]Connected component labelling describes a series of simple techniques to perform very basic segmentation, of a kind. Essentially, neighbouring pixels that look the same (where 'the same' is defined by equal pixel intensities) are labelled as a distinct region.

Fraction missing	**Time (NaN encoding)/(ms)**	**Time (∞ encoding)/(ms)**
0	987	678
0.1	2576	1941
0.2	7894	5169
0.3	10127	7814
0.4	13550	8126
0.5	18865	10264

Table 4.2: Comparison of times taken to fit a C4.5 Decision tree to the MNIST dataset, for various degrees of missing data, using different encodings of missing values.

$\pm\infty$ and NaN. The choice to use the value of NaN for missing data was made, as it seemed to make the most intuitive sense. However, dealing with these values in the splitter caused issues. As outlined in the IEEE 754 standard for floating point arithmetic,[20] all comparisons involving NaN return false. In practical terms, this meant that missing samples had to be separated from other node samples, before these samples could be sorted for a feature. This operation was of $O(n_{samples})$ for each feature. This was countered by copying the initial user dataset, and substituting all NaN values for $+\infty$. This provided a natural way to separate missing samples, for calculation of split criterion. Table 4.1.2 provides some timing results for various amounts of missing data, when encoding values with NaN and ∞.

4.1.3 Sample Counting

After implementing categorical data functionality, it was found that building trees was significantly slower. Profiling showed that a large proportion of training time was spent in the *multi_update* **criterion** function. It is a relatively simple function, but is called many times. The basic purpose of this function is to sum and note the total weight of samples within each child node, and the weight for each class. Initially, this was done in a naiive manner - samples were iterated over and weights were simply calculated for each node in turn (fig. naiive-count). However, the sample and class weights for the parent node were already known, meaning that these values for the final child node could be simply be calculated as a remainder.

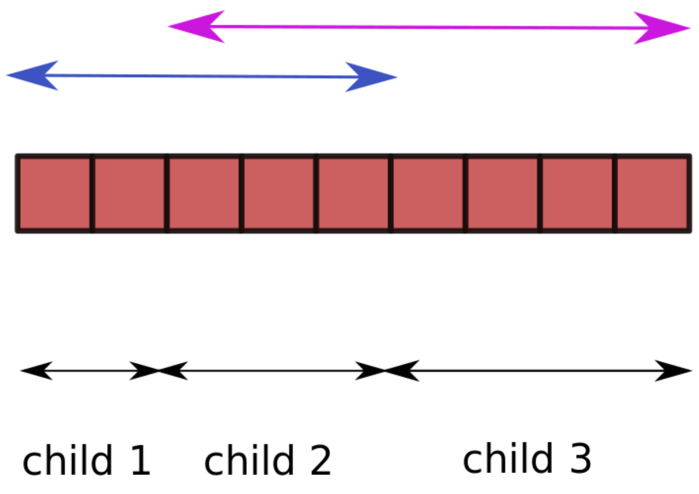


Figure 4.2: Illustration of sample counting by the criterion. The total number of samples is known to the criterion already. The blue and purple arrows represent the number number of samples that *could* be counted by the criterion. Clearly it is most efficient to measure the number of samples for child 1 and 2 (the blue arrow), and then calculate the number of samples in child 3 by subtracting from the total number.

4.1.4 Additional Split Criteria

Tests were carried out to compare accuracy between a random forest comprised of C4.5 trees, and one comprised of CART trees. The C4.5 forest used the gain ratio criterion, while the CART forest used gain. Results are shown in fig.4.3. The outcome was unexpected, but it was surmised that the reason for this may have been due to the nature of the missing data. Firstly, the assumption made when dropping values were dropped from the dataset was that these values were MCAR†. This was simulated as follows. A

†MCAR stands for **M**issing **C**ompletely **At** **R**andom, and describes the nature by which data might be missing from a dataset. With MCAR, it is assumed that there is no way to predict which values might be missing. This is in contrast to **M**issing **At** **R**andom, in which case the probability of a value being missing for a sample could be predicted from the other feature values, or **M**issing **N**ot **At** **R**andom, whereby a

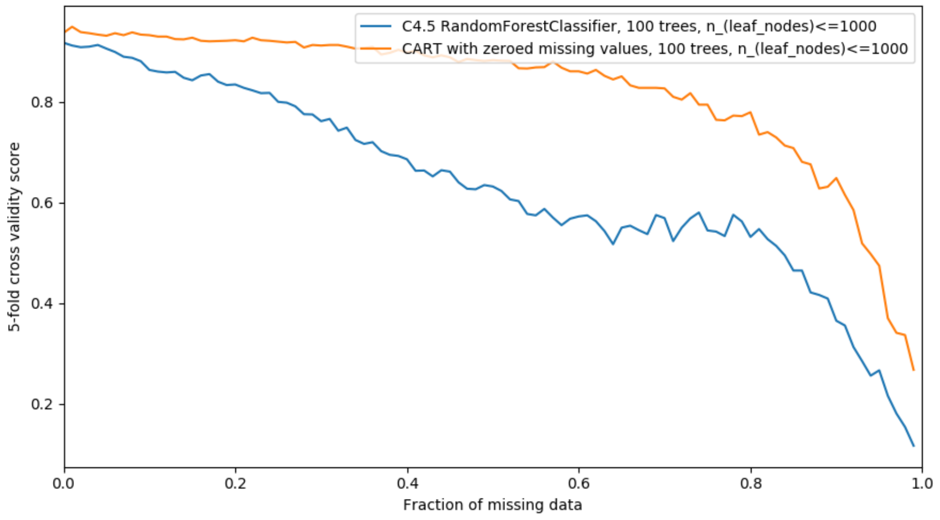


Figure 4.3: Comparison of accuracy on increasing missing data, using the C4.5 algorithm with information gain criterion, and the CART algorithm with information gain. The 5-fold cross validity score is used here as a measure for accuracy.

function was defined to iterate over all feature values within the dataset, and a random number was chosen from a uniform random distribution $U(0,1)$[‡]. If the random number chosen was less than some specified fraction, the feature value was dropped (or set to zero). In this way, fifty percent of values within the dataset could be removed or zeroed, simply by providing a value of 0.5 to the function. Choosing to drop values in this way may not have been the best choice for the C4.5 algorithm. With this assumption, missing data could reasonably be treated simply as noise, and set to some arbitrary number. When building the random forest, their contribution to error would presumably be 'averaged out', especially given these missing values are evenly distributed. C4.5 instead tries to place a penalty splits using the split information term from information gain, dependent on the number of missing values. In contrast, as shown earlier, TMVSE data will not be MCAR - samples closer to the edge of an image will be more likely to have missing data. As such, they may find that C4.5 with gain ratio performs better.

To deal with this however, and to provide additional flexibility to the user, the gain cri-

feature's value is directly related to whether it might be missing.

[‡]A random variable modelled by a uniform random distribution $U(a,b)$ takes a value in the range $(x_0, x_1; a \leq x_0 < x_1 < b)$ with probability $\frac{x_1 - x0}{b-a}$. If $a = 0, b = 1$ the variable has an equal chance of being any value in the range $[0,1)$.

Fraction missing	**Time (information gain ratio)/(ms)**	**Time (entropy)/(ms)**
0	514	222
0.1	1764	1542
0.2	4768	2951
0.3	6616	4393
0.4	7400	6077
0.5	9887	8122

Table 4.3: Comparison of times taken with C4.5 trees using the information gain ratio, and entropy.

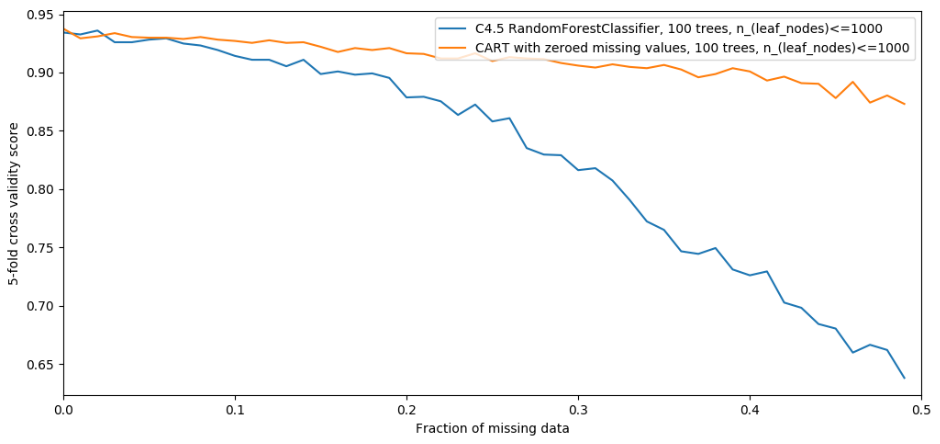


Figure 4.4: Plot of fraction of missing data vs. 5-fold cross validity score, for a C4.5 and CART random forest, both using the entropy criterion. For the CART tree, missing values are simply set to zero.

terion (**Entropy** in Scikit-Learn) was updated to deal with missing data and an arbitrary number of child nodes. Fig.4.4 shows how this improved accuracy slightly for this test. The accuracy still drops with increasing missing data, but An additional benefit was that the calculation of the gain was less computationally intensive than that of the gain ratio. The split information term is the only term differing between these calculations. As it involves the use of a logarithmic function, it can confer a large time cost. Table 4.1.4 demonstrates how time compared between using the information gain and entropy for various amounts of missing data, on a tree with a maximum of 1000 leaf nodes.

4.1.5 Tree Builder

The *BestFirstTreeBuilder* was modified to allow for training with categorical data. As stated earlier, this should produce better trees of limited growth than the *DepthFirst-TreeBuilder*. However, the memory to requirement to grow these trees is greater using the *BestFirstTreeBuilder*.[21] The best-first graph traversal algorithm utilises a priority queue, which in Scikit is implemented using a heap, implemented as an array of structs. Each of these structs essentially holds information regarding the node to be expanded. In the original SKLearn implementation, this array of structs is freed at the end of all computation. However, in dealing with incomplete samples, partial sample weights, and an arbitrary number of child nodes, it was not possible to do this. The program would gradually consume more and more memory, until limits of RAM were reached. This memory thus had to be freed early.

4.1.6 Early stopping criteria

Table 4.1.4 shows how timing differed for different proportions of missing data on the MNIST dataset. These times posed a clear issue for training. The increased time taken to complete training was thought to stem especially from the 'greedy' nature of the C4.5 algorithm. By assessing the how the conditions needed to determine whether a node is a terminal one are changed with incomplete samples, we can see why this is the case.

1. If a node has an entropy of zero, then its samples are all of the same class. When calculating the improvement that a particular split makes to overall entropy, the C4.5 algorithm ignores missing samples. When considering only the known samples, each node may contain one class each, marking the split as optimal. However, when expanding the child nodes, the node entropy of each is recalculated using all samples. As the unknown samples may not be of the same class as the complete node samples, the node is not declared to be a leaf node, and is further expanded. For large proportions of missing values, nodes are rarely marked as leaves, and trees will tend to grow to much greater depths. This will also tend to lead to rather overfitted trees.

2. If it was not possible to find a split that improved upon the entropy of the parent node, then the parent is marked as a leaf node. With the C4.5 algorithm, this tends to happen only occasionally, when a node has a large proportion of incomplete samples. In this case, the modified weights of incomplete samples can become

Minimum Impurity Decrease	**Time/(ms)**	**Accuracy**
0	19414	0.64
0.002	8226	0.72
0.004	3901	0.75
0.006	2589	0.76
0.008	2049	0.76
0.01	1673	0.78
0.012	1456	0.75

Table 4.4: Comparison of time and accuracy when training a random forest of 100 trees (max leaf nodes = 1000) for various values of minimum impurity decrease. 50% of data is missing.

sufficiently small that the limit of floating point precision is reached, and entropy is calculated to be zero. It will generally take many splits to reach this point.

To improve upon these times, the user can choose to manually relax the conditions necessary to mark a node as a leaf. To deal with the first point discussed above, the parameter *min_impurity_decrease* could be used. Table 4.1.6 shows how the time and accuracy changed when training random forests for very small modification to this parameter. Indeed a value of 0.01 allowed for a 91% reduction in time, with improved accuracy.

As is seen, small values of this parameter actually increased both training speed and accuracy. It would be wise to experiment with this parameter when training on large datasets. Unfortunately, the *min_impurity_decrease* can also affect tree building when dealing with complete samples. In order to specifically target slowdown due to incomplete samples, the *min_fraction_complete* parameter was introduced. If any node contains a smaller fraction of complete samples than the value specified, the node is marked as a leaf. Some results using this parameter are shown in table 4.1.6. Similarly to the *min_impurity_decrease*, a 94% reduction in time taken was achieved, with improved accuracy. When using these values for the parameters together, the accuracy of 0.78 was maintained, but time taken further reduced to 890ms.

Minimum Fraction Complete	**Time/(ms)**	**Accuracy**
0	19414	0.64
0.002	2511	0.75
0.004	2313	0.75
0.006	1796	0.75
0.008	1414	0.77
0.01	1193	0.78
0.012	1073	0.78

Table 4.5: Comparison of time and accuracy when training a random forest of 100 trees (max leaf nodes = 1000) for various values of minimum impurity decrease. 50% of data is missing.

4.2 External Interface

It would be useful to eventually provide access to all decision trees through the same interface. A user might use an extra parameter, *algorithm="CART"/"C4.5"* to distinguish the type of tree. However for this project, a new tree class was created, called a *C45DecisionTree*. By doing this, (1) tests for each tree type could be isolated, and (2) it removed the need to implement error messages for a few features not implemented as of the end of the project. Intuitively, it seemed to make sense to reserve the value of NaN to mark values as missing in a test dataset. This may be changed in a future update however, to prevent the accidental marking of values as missing due to some incorrect mathematical calculation during data preparation. For discrete features, the parameter *na_string=**string*** was provided, with a default of *NA*, to denote missing values.

Some consideration was given to how data should be presented by the user, if data with a mixture of discrete and continuous valued features was provided. In general, Scikit-Learn accepts NumPy[§] arrays to train its estimators. Depending on how the data was produced, it may be presented to Scikit-Learn as contiguous data, or as an array of Python objects. The latter is especially true if the data was imported as a *.csv* file, using the *Pandas* module, which is a common way to load stored data. Creating a NumPy array from a Python list also has this effect.

[§]NumPy arrays are generally faster than Python lists. This is because 'true' C-style contiguous arrays can be built. A convenient interface is also provided for a wide number of memory modification operations.

Internally, Scikit copies this data to a contiguous array of double values. This is useful for cache locality. If the data is already in this format, only a reference to the data is stored. Clearly, this is the best-case. However, if discrete feature data is used with strings or characters (whether this data is comprised entirely of discrete data, or a mixture of discrete and continuous data), the situation is more complicated. Strings *can* be stored contiguously with NumPy via the use of a padded one-dimensional char array, but this is a complicated requirement to impose. Doing this with NumPy would require that the length of the longest string is known. Another approach, shown in fig[separate-disc_cont] would require the user to explicitly define a struct within NumPy. A final decision was made to allow data to be stored simply as arrays of Python objects. Internally, this was then converted to a contiguous format. Unfortunately, this meant that if using discrete data, the entire dataset would need to be copied. This will be a limiting factor for large datasets.

The parameters used to initialise a *C45DecisionTree* are the same as for a *DecisionTree*, save for the addition of the *na_string=**string*** parameter discussed above, and the *min_complete_samples_fraction*. The *fit* function was modified to require a feature metadata file when using discrete data, describing the possible values that might occur for each discrete feature. The interface for using random forests is exactly the same, with two additional optional parameters. The options *algorithm=**string***, *min_complete_samples_fraction=**float*** and *na_string=**string*** are now provided.

4.2.1 Rejected Considerations

Logarithmic Approximation

Running speed tests on complete data showed that using the C4.5 decision tree with the 'gain ratio' split criterion was significantly slower than using a normal decision tree with the 'entropy' (gain) criterion.

Profiling the slower code showed that a large proportion of time was spent within the *multi_update* criterion function. The logarithmic function also used a large amount of code time. As discussed earlier, the gain ratio differs from the gain by a division over split information. As this involves an expensive logarithmic calculation for each child node, this was likely where a large amount of code time was spent. An attempt was made to improve this situation by noting that absolute values for the gain ratio need not be known. These values are only used to compare different potential node splits,

Number of Taylor's series terms	**Time/(ms)**	**Accuracy**
Actual function	19845	0.64
1	780	0.23
2	876	0.25
3	1230	0.33
4	1580	0.38
5	2043	0.45

Table 4.6: Comparison of time and accuracy when training a random forest of 100 trees (max leaf nodes = 1000) using a Taylor's series approximation to the logarithmic function. 50% of data is missing.

so only their relative values were necessary. Recalling equation 2.4, the contribution to split information for each child node is of the form $xlog(x)$, where $0 < x \leq 1$. The Taylor's series[¶] for this function is

$$\sum_{n=2}^{\infty} \frac{(-1)^n (x-1)^n}{n(n-1)} + (x-1) \tag{4.1}$$

Unfortunately, this function is not monotonic[‖] for x between 0 and 1. Were this the case, an approximation to the function of $x - 1$ would give the same relative gain ratio values[**]. This meant that using an approximation may contribute to choosing different splits (and hence incorrect trees). However, choosing an appropriately large number of approximation terms to use should prove sufficient. Some experimentation was done to test this, checking tree accuracy and speed against Taylor series terms, and is shown in table.4.2.1. It was decided eventually that speed improvements were too small, in comparison to the drop in accuracy caused by this approximation.

Non Binary as Binary Tree

A binary representation of a non binary tree was briefly considered when implementing discrete feature capabilities. A basic illustration of this is shown in fig.4.5. This was

[¶]A Taylor's series is a sum representation of a function, in terms of its derivatives at a specified point in the function's domain.

[‖]A function is monotonic over a specified range if it only increases or decreases in that range.

[**]This is also because terms with increasing powers of x decrease in size.

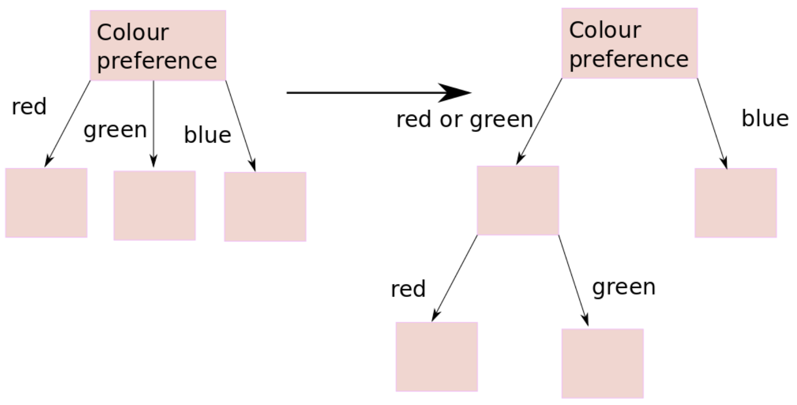


Figure 4.5: Illustration showing how a tree of arbitrary number of child nodes can be represented as a binary tree.

eventually rejected as it introduced complications when trying to limit tree growth, and make tree visualisation confusing for the end-user.

Chapter 5

Conclusions

5.1 Reflections

Overall, this project seeemed to have mixed success - C4.5 random forests were implemented within Scikit-Learn, but some more work is yet to be done for this to be usable to TMVSE on the industrial scales planned. This unfortunately meant that testing the code with datasets fully representative of the problem to be solved was not possible. The code was also not completely in a form in which it could be passed on to the Scikit-Learn team, as of the end of this project, though that may be addressed in the future.

5.2 Extensions to this work

The following section describes a number of interesting avenues for investigation, following from the work completed during the course of this project. Some of these were intended to be examined, but owing to the constraints of time, were left.

5.2.1 Streaming Data

Although it is not currently supported in Scikit-Learn, the capability to stream a serialized version of a dataset would be useful for large datasets. Currently, all data is loaded entirely into memory, which could be prohibitive for a user intending to run a large training exercise on commodity hardware.

Number of threads	Time/(ms)
1	1945
2	2143
3	2351
4	2334

Table 5.1: Comparison of time taken to train a random forest of 100 trees (max leaf nodes = 1000) using varying numbers of threads. 50% of data is missing.

5.2.2 Tree Level Parallelism

When training a random forest, the building of each individual tree is completely independent, and is hence embarrassingly parallel. Indeed, Scikit-Learn allows a user to build these trees on a single machine using multiple threads with the **Joblib**[22] module. However, for all tests using the C4.5 algorithm, using more than one thread was seen to slow down overall training time (table5.2.2. It was surmised that this was likely due to the complexity involved in building a single C4.5 decision tree. Given the amount of data moved around to deal with incomplete samples and their respective weights, cache thrashing would be very likely. Parallellisation would perhaps be more useful on smaller sections of code, while building a single tree. Based on the section of code with the largest contribution to overall run time, the *multi_update* function would be an appropriate starting point to attempt this.

Application to Cloud Computing

An *export* function is provided by Scikit-Learn. This allows a user to serialise decision trees. Using the example of TMVSE, training could be completed on a large DICOM dataset on a high compute, high memory AWS instance in a short time, and then the classifier copied back to a local machine. TMVSE provided some statistics on data training time using their (relaxed) implementation of C4.5 random forests. Approximately three hundred trees were each trained on a dataset of 180000 samples, and 2500 features, taking roughly five minutes to train per tree. This would correspond to a total training time of 25 hours. The following shows how this could potentially be completed in a fraction of the time, at low cost on a number of on-demand AWS EC2 instances.

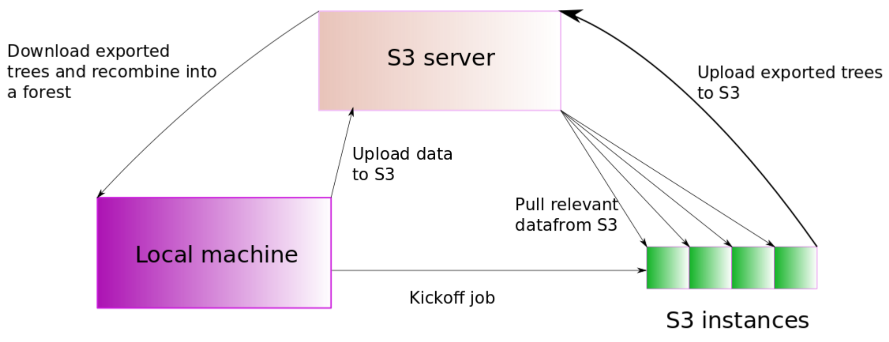

Download exported trees and recombine into a forest
S3 server
Upload exported trees to S3
Upload data to S3
Pull relevant datafrom S3
Local machine
Kickoff job
S3 instances

Figure 5.1: Illustration showing how a tree of arbitrary number of child nodes can be represented as a binary tree.

As of today, the on-demand hourly cost for one r3.large[*] AWS instance is \$0.17. The cost for 25 hours worth of training time on one instance would be \$4.00. Similarly, the cost for 5 r3.large instances for 5 hours would be \$4.00. As the calculation of each decision tree is independent, the data used to train these trees could simply be distributed amongst instances. Assuming fairly evenly balanced features (and given that instances are not competing for resources, as in the case of the multithreaded approach), the speedup from doing this could be reasonably assumed to be fairly linear with the number of instances used, excluding the time to copy data. However, even this time could be minimised through a choice to use S3 storage located in the same region, which offers very fast upload and download speeds[†]. If tree level parallelism offered even a 2x speedup for each tree, this would reduce the total time for training from 25 hours, to 2.5, for half the price ($2.5 \times 5 \times \$0.16 = \$2.00$).

A schematic for such a system is shown in fig.5.1.

[*]This particular instance would be chosen for having sufficient RAM to hold an entire DICOM dataset in memory.

[†]Using a server located in the same region should also prevent some of the issues related to the transfer of sensitive patient data.

Bibliography

[1] MA.Dabbah; S.Murphy; H.Pello; R.Courbon; E.Beveridge; S.Wiseman; D.Wyeth; I.Poole *2014 Detection and location of 127 anatomical landmarks in diverse CT datasets*

[2] A.Criminisi; J.Shotton; S.Bucciarelli *Decision forests with long-range spatial context for organ localization in CT volumes* Medical Image Computing and Computer-Assisted Interventation (MICCAI), 69-80 (2009)

[3] J.R.Quinlan *C4.5: programs for machine learning* C4.5: programs for machine learning, pp. 22-35

[4] Scikit-Learn *Scikit-Learn Decision tree information page* http://scikit-learn.org/stable/modules/tree.html

[5] T.K.Ho *Random Forests* Proceedings of the 3rd International Conference on Document Analysis and Recognition, Montreal, QC, 14âĂŞ16 August 1995. pp. 278âĂŞ282.

[6] *Cython homepage* http://cython.readthedocs.io/en/latest/src/quickstart/overview.html

[7] *Scikit-Learn scaling overview* http://scikit-learn.org/stable/modules/scaling_strategies.html

[8] L.Breiman; J.H.Friedman, R.A.Olshen, C.J.Stone. *Classification and Regression Trees* CRC Press; 1984

[9] *Python compilation overview* http://pythonextensionpatterns.readthedocs.io/en/latest/debugging/debug_

[10] *Cython debugging overview* http://cython.readthedocs.io/en/latest/src/userguide/debugging.html

[11] *MNIST data* http://yann.lecun.com/exdb/mnist/

[12] *UCI Chess dataset* https://archive.ics.uci.edu/ml/ datasets/Chess+(King-Rook+vs.+King)

[13] *Scikit-Learn Contribution Guidelines* http://scikit-learn.org/stable/developers/contributing.html

[14] SciPy numpy.array Documentation *Scipy Documentation* https://docs.scipy.org/doc/numpy/reference/generated/numpy.array.html

[15] J.Pearl *Heuristics: Intelligent Search Strategies for Computer Problem Solving* Addison-Wesley, 1984. p. 48.

[16] H.Samet; M.Tamminen *Efficient Component Labeling of Images of Arbitrary Dimension Represented by Linear Bintrees* IEEE Transactions on Pattern Analysis and Machine Intelligence. IEEE. 10 (4): 579

[17] M.B.Dillencourt; H.Samet; M.Tamminen *A general approach to connected-component labeling for arbitrary image representations* Journal of the ACM. J. ACM. 39 (2): 253

[18] D.Musser *Generic Algorithms* http://www.cs.rpi.edu/ musser/gp/algorithms.html

[19] *CPP Reference* http://en.cppreference.com/w/cpp/container/unordered_map/operator_at

[20] IEEE Computer Society (August 29, 2008) *IEEE Standard for Floating-Point Arithmetic* IEEE Std 754-2008

[21] W.Zhang; R.E.Korf *Depth-first vs. best-first search: new results*

[22] Joblib *Joblib documentation page* https://pypi.python.org/pypi/joblib

[23] AWS *AWS EC2 on-demand pricing page* https://aws.amazon.com/ec2/pricing/on-demand/

[24] AWS AWS multipart upload page http://docs.aws.amazon.com/AmazonS3/latest/dev/uploadobjusingmp

[25] AWS AWS multipart download page https://aws.amazon.com/blogs/developer/parallelizing-large-downloads-for-optimal-speed/