

# Identifying Better Features Relevant to Classification

Frances Crouch

A DISSERTATION

Submitted to

The University of Liverpool

in partial fulfilment of the requirements  
for the degree of

MASTER OF SCIENCE

# Abstract

Decision trees are widely used across multiple disciplines to perform feature selection. Traditional decision tree algorithms take a greedy approach to growing trees, quickly producing reasonable but suboptimal results. Growing trees according to a Bayesian approach could outperform a greedy approach, however conventional methods used to explore the solution space are too computationally expensive to be practical.

The recent development of SMC samplers provides an effective way to parallelise a Bayesian decision tree. This project has developed a prototype parallel implementation of such a model, which has not been attempted before. The performance of the algorithm has been assessed with two contrasting datasets: genetic data relating to pancreatic disease and tweets indicating adverse drug side-effects.

The initial conclusions from this study suggest that a speedup of around 70 can be achieved by just changing the software used to implement the algorithm and further speedup can be obtained through parallelisation. The Bayesian model implemented here searched for a single tree. The results suggest that this can outperform a conventional decision tree, but better results were obtained using Random Forest, indicating an opportunity to develop a parallelisable Bayesian sum-of-trees model.

# Student Declaration

I confirm that I have read and understood the University's Academic Integrity Policy.

I confirm that I have acted honestly, ethically and professionally in conduct leading to assessment for the programme of study.

I confirm that I have not copied material from another source nor committed plagiarism nor fabricated data when completing the attached piece of work. I confirm that I have not previously presented the work or part thereof for assessment for another University of Liverpool module. I confirm that I have not copied material from another source, nor colluded with any other student in the preparation and production of this work.

I confirm that I have not incorporated into this assignment material that has been submitted by me or any other person in support of a successful application for a degree of this or any other university or degree-awarding body.

SIGNATURE \_\_\_\_\_

DATE            September 21, 2016

# Contents

<b>1</b>	<b>Introduction</b>	<b>1</b>
<b>2</b>	<b>Background</b>	<b>3</b>
2.1	The Bayesian CART algorithm . . . . .	3
2.1.1	The Prior Distribution . . . . .	4
2.1.2	The Prior Likelihood . . . . .	5
2.1.3	Metropolis-Hastings . . . . .	5
2.2	Existing Parallelisation Techniques . . . . .	6
2.3	SMC Samplers . . . . .	7
2.3.1	Importance Sampling . . . . .	7
2.3.2	Resampling . . . . .	8
2.3.3	The General Approach . . . . .	8
2.3.4	BCART with SMC Samplers . . . . .	8
2.3.5	Parallelisation of SMC Samplers . . . . .	9
2.4	Research Methodology . . . . .	9
2.5	The Data . . . . .	9
2.5.1	Twitter Data . . . . .	10
2.5.2	Pancreatic Data . . . . .	10
2.5.3	Fish Data . . . . .	10
2.5.4	Cyber Security Data . . . . .	11
2.5.5	Commercial Data . . . . .	11
<b>3</b>	<b>Design</b>	<b>12</b>
3.1	Decision Tree and Random Forest . . . . .	12
3.2	Serial BCART . . . . .	12
3.2.1	Code Structure . . . . .	12
3.3	Parallel BCART . . . . .	14
3.3.1	Brief Introduction to Spark . . . . .	14
3.3.2	Code Structure . . . . .	14
3.4	Evaluation . . . . .	16
3.4.1	Predictive Performance . . . . .	16
3.4.2	Runtime . . . . .	16
3.5	Hardware . . . . .	17
3.6	Changes from the Original Design . . . . .	17
<b>4</b>	<b>Results</b>	<b>19</b>
4.1	Pancreatic Data . . . . .	19
4.1.1	Runtime Results . . . . .	19

4.1.2	Predictive Performance . . . . .	23
4.2	Twitter Data . . . . .	27
4.2.1	Runtime Results . . . . .	27
4.2.2	Predictive Performance . . . . .	28
4.2.3	Computing Error . . . . .	31
4.3	Fish Data . . . . .	32
<b>5</b>	<b>Learning Points</b>	<b>33</b>
5.1	Code Reuse and Recursive Functions . . . . .	33
5.2	Random Numbers . . . . .	34
5.3	Caching . . . . .	35
5.4	Normalising weights . . . . .	35
<b>6</b>	<b>Professional Issues</b>	<b>37</b>
<b>7</b>	<b>Evaluation</b>	<b>38</b>
7.1	Strengths . . . . .	38
7.2	Weaknesses . . . . .	38
<b>8</b>	<b>Conclusions</b>	<b>40</b>
8.1	Runtime . . . . .	40
8.2	Predictive Performance . . . . .	40
8.3	Further Research . . . . .	41
<b>A</b>	<b>Algorithms</b>	<b>44</b>
A.1	Serial BCART . . . . .	44
A.2	Parallel BCART . . . . .	45
A.3	Parameter Tuning . . . . .	46
<b>B</b>	<b>Code</b>	<b>47</b>
B.1	Code Listing . . . . .	47
B.2	Code Structure . . . . .	49
B.2.1	Serial BCART . . . . .	49
B.2.2	Parallel BCART . . . . .	51
<b>C</b>	<b>Results</b>	<b>54</b>
C.1	Pancreatic Dataset . . . . .	54
C.2	Twitter Dataset . . . . .	57
C.3	Fish Dataset . . . . .	61
<b>D</b>	<b>Learning Points</b>	<b>62</b>

# List of Figures

2.1	A Generic CART model . . . . .	4
3.1	Class Diagram for Serial BCART . . . . .	13
3.2	Spark Implementation of BCART . . . . .	15
3.3	Class Diagram for Parallel BCART . . . . .	15
4.1	Pancreatic: Comparative Runtimes . . . . .	19
4.2	Example Runtime of the Parallel BCART Algorithm (local mode, 512 samples, 1,000 iterations) . . . . .	20
4.3	Pancreatic: Speedup . . . . .	21
4.4	Runtime Comparison for Cache and Non-Cache Scripts . . . . .	22
4.5	Pancreatic: AUC Comparison . . . . .	23
4.6	Serial BCART Parameter Tuning . . . . .	24
4.7	Pancreatic: Serial and Parallel Implementation Comparison . . . . .	25
4.8	Pancreatic: Parallel BCART Parameter Tuning . . . . .	26
4.9	Twitter: Comparative Runtimes . . . . .	27
4.10	Twitter: Speedup . . . . .	27
4.11	AUC of all Samples at One Iteration . . . . .	28
4.12	Twitter: AUC Comparison . . . . .	29
4.13	Twitter: Serial and Parallel Implementation Comparison . . . . .	30
4.14	Parallel BCART Parameter Tuning . . . . .	30
4.15	AUC Result of running a DT model with the Fish Dataset . . . . .	32
5.1	Demonstration of Random Numbers in Spark . . . . .	34

# Chapter 1

## Introduction

Feature selection is the process of choosing a subset of features from a feature space, and can provide various benefits, including: reducing the time to generate a prediction, decreasing storage demands, reducing training time, improving the performance of a model and ease of visualisation and data understanding [1].

This project explores binary classification problems with a very large feature space, where it is important to quickly generate predictions or using less features has a monetary advantage, and hence feature selection is necessary.

A real life problem where feature selection is advantageous is analysing genetic data for disease diagnose. The number of variables in the dataset can exceed the tens of thousands, the accuracy of a predictive model is critical to correctly diagnosing the patient, and for some diseases it can be crucial to treat a patient quickly, so the speed of generating a prediction is important.

Similarly, in analysing social media, classification problems have a large feature space due to the number of words in a language. A user can access the Twitter API, while filtering on up to 100 words, for free. If the predictive model uses more than 100 features, the user will need to pay for the data.

Decision tree algorithms, such as Classification and Regression Trees (CART) or Random Forest (RF), inherently use feature selection. However, they adopt a greedy approach which can generate suboptimal results [2].

Bayesian Additive Regression Trees (BART) are the Bayesian extension of the RF algorithm, building a Bayesian “sum-of-trees” model, which has been shown to outperform other classification algorithms [3]. The approach starts by defining a prior and likelihood, and uses methods, such as Markov chain Monte Carlo (MCMC), to navigate towards “good” trees. However, these traditional search methods can be too computationally expensive to be practical, inhibiting the ability to adequately tune the model’s parameters and to produce better predictions.

The aim of this project was to explore how the runtime of a Bayesian tree model could be reduced by parallelisation, using the recent development of Sequential Monte Carlo (SMC) samplers [4].



## Chapter 2

# Background

### 2.1 The Bayesian CART algorithm

The literature mentions several Bayesian tree models [3, 5, 6, 7]. Time did not allow for thorough analysis of all papers, so this project considered the algorithms proposed by Chipman et al., as they have comparatively more citations and open source implementations.

Chipman et al. have offered both a single tree model (BCART) [5] and a sum-of-trees model (BART) [3]. While ensemble methods generally perform better, the focus of this project is parallelisation and speedup, rather than accuracy, so for simplicity, this project was based BCART.

A decision tree is a predictive model which maps a set of features,  $x = (x_1, x_2, \dots, x_n)$ , to a target value,  $y$ . This project considers binary classification problems, and hence  $y \in \{0, 1\}$ . A CART model can be defined as a tree,  $T$ , with  $b$  terminal nodes and  $b - 1$  internal nodes. Instances of the data filter down through the internal nodes according to the split features and split values. For example in Figure 2.1, the first node splits the data with feature  $x_1$  and a split value of 3. Thus, the internal nodes create non-overlapping subdivisions in the solution space.

Each instance settles into one of the terminal nodes. The class of a terminal node is determined as the modal value of the label,  $y$ , corresponding to its instances. To generate a prediction, a test instance takes the class label of the terminal node it resides in.

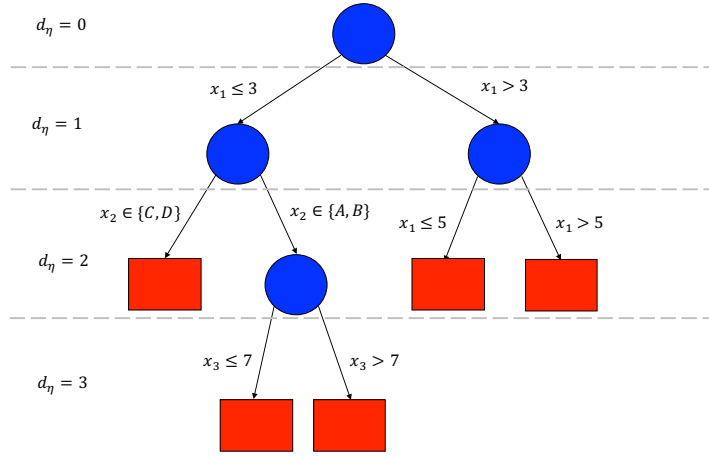


Figure 2.1: A Generic CART model

Training a decision tree model involves determining the split features and values at each node. A greedy decision tree chooses the features which best split the data (according to various metrics). The Bayesian approach involves defining a prior distribution,  $p(T)$ , and finds “good” trees by searching for areas of high posterior probability, where the posterior is:

$$P(T|X, Y) \propto P(Y|X, T)P(T)$$

As with many Bayesian approaches, even when using conjugate priors to reduce the amount of calculation required, exhaustive search of the posterior is impractical. Instead, methods such as MCMC are used to explore the posterior as they migrate towards areas of high posterior probability.

### 2.1.1 The Prior Distribution

The prior is defined as the probability of generating the tree,  $T$ , given the rules  $p_{SPLIT}$  and  $p_{RULE}$ . Firstly,  $p_{SPLIT}$  establishes whether a node splits:

$$p_{SPLIT}(d_\eta, T) = \alpha(1 + d_\eta)^{-\beta}$$

The parameter  $d_\eta$  is the depth of the node and the parameters  $\alpha$  and  $\beta$  determine the size and shape of the trees. For higher values of  $\alpha$  the trees are deeper and for higher values of  $\beta$  the trees are more bushy.

If a node is split,  $p_{RULE}$  determines the split feature,  $x_i$ , and split value,  $s_i$ . The feature is selected uniformly from all the features, and similarly, the value selected uniformly from the range of that feature:

$$p_{RULE}(\rho|d_\eta, T) = \frac{1}{(|X|range(x_i))}$$

The prior can then be calculated as the product of probabilities (across all nodes in the tree) of:

- An internal node splitting;
- Selecting an internal node's split feature and value;
- A terminal node not splitting.

The set of terminal nodes is given the notation  $T_n$ , such that the prior is:

$$P(T) = \prod_{\eta \notin T_n} p_{SPLIT}(d_\eta, T) p_{RULE}(\rho | d_\eta, T) \prod_{\eta \in T_n} (1 - p_{SPLIT}(d_\eta, T))$$

### 2.1.2 The Prior Likelihood

As the label,  $y$ , is binary, the probability an instance belongs to class 0 or class 1 can be represented by a Bernoulli distribution, with probabilities  $p$  and  $1 - p$  respectively. So it is natural to define the likelihood,  $p(Y|X, T)$ , using a Beta-Bernoulli conjugate prior:

$$P(Y|X, T) = \frac{1}{Be(\gamma, \delta)^b} \prod_{i=1}^b Be(\gamma + n_{i,0}, \delta + n_{i,1})$$

where:

- $n_{i,k} = \sum_j I(y_{i,j} = k)$  for  $k = 0, 1$  (e.g.  $n_{i,0}$  is the number of instances with a label of 0 at terminal node  $i$ );
- $y_{i,j}$  is the label corresponding to the  $j^{th}$  instance in the  $i^{th}$  terminal node;
- $n_i = n_{i,1} + n_{i,0}$ .

Note: typically the parameters of the Beta distribution are denoted as  $\alpha$  and  $\beta$ , however to avoid confusion with the  $p_{SPLIT}$ ,  $\gamma$  and  $\delta$  have been used.

This results in a high likelihood for trees with terminal nodes that contain homogeneous labels.

### 2.1.3 Metropolis-Hastings

MCMC is used to explore the posterior, generating random samples from a probability distribution by constructing a Markov chain, such that the chain will spend more time in areas of high posterior probability. The BCART algorithm specifically uses the Metropolis-Hastings (MH) algorithm. Broadly speaking, this works by generating a candidate sample from a current sample, and choosing to accept it if it is better. A proposal distribution,  $q$ , is used to generate the candidate sample.

In BCART, the candidate tree,  $T^*$ , is generated from the current tree,  $T$  and the proposal distribution,  $q(T^*|T)$ , which randomly selects between four options with equal probability:

- GROW - split a randomly selected terminal node using  $p_{RULE}$ ;
- PRUNE - collapse a randomly selected parent of two terminal nodes;
- CHANGE - randomly select an internal node and reassign the split value,  $s_i$ ;
- SWAP - swap the splitting rule of a randomly selected parent-child pair.

Hence, the candidate tree,  $T^*$ , is generated by either growing, pruning, changing or swapping a node in the tree,  $T$ . So, the proposal distribution is ‘blind’ to the data.  $T^*$  is accepted or rejected based on the acceptance function:

$$a(T^*|T) = \min \left\{ \frac{q(T^i|T^*)p(Y|X, T^*)p(T^*)}{q(T^*|T^i)p(Y|X, T^i)p(T^i)}, 1 \right\}$$

This process, of generating a candidate tree and deciding whether to accept, is iterated 5,000 times, and then this chain is repeated 500 times. Finally, “good” trees are selected as those with the highest the area under the ROC curve (AUC), calculated using the training data. The full algorithm is included in Appendix A.1.

## 2.2 Existing Parallelisation Techniques

Parallelisation techniques considered include:

- **Data parallelism:** Pratola et al. introduced a parallel BART algorithm (the sum of trees version) by dividing the data across the available processors, applying the model concurrently to each subset of the data and aggregating the results [8]. Given the number of instances in the datasets being used for this project, see Section 2.5, this approach would not yield a high speedup.
- **Multiple MCMC chains:** The BCART algorithm runs many repeats of the MCMC step. These could be run concurrently. However, typically, the first  $n$  samples of an MCMC chain are discarded to allow the chain time to reach the equilibrium distribution, referred to as ‘burn-in’. Hence, the speed up of such an approach is limited when there is a long burn-in time and this approach does not generally scale well over large clusters [9].
- **Parallelising a single MCMC chain:** This approach aims to speed up the computation time by parallelising the calculations in each iteration of the MCMC step [9]. Here, this could be implemented, for example, by different processors calculating:  $q(T^i|T^*)$ ,  $p(Y|X, T^*)$ ,

$p(T^*)$  and so on, for the acceptance distribution. However, this would require a lot of communication between processors (which is expensive), would not scale well over a large cluster and would be difficult to balance the work load between processors, resulting in idle time.

## 2.3 SMC Samplers

An alternative parallelisation approach is to use SMC samplers. First introduced by Del Moral et al. [4], they come under the umbrella of SMC methods. Compared to other techniques within this group, there are relatively few published papers. For example, searching “SMC samplers” in Google Scholar returns around 4,060 results, whereas “particle filter” returns around 2.6m results. Furthermore, of those papers available, none could be found that apply SMC samplers to a Bayesian CART model and many, particularly the papers published in statistics journals, are theoretical and provide little insight into how the technique can be applied. Fortunately, I had access to a paper (currently under review) which provides a more accessible explanation [10].

This section starts with an overview of the key concepts of SMC samplers.

### 2.3.1 Importance Sampling

Importance sampling is used when it is inefficient or impossible to draw samples from the distribution of interest. Assume we want to find:

$$\mathbf{E}[f(x)] = \int f(x)p(x)dx \quad (2.1)$$

where  $p$  is the probability density function. If we could sample from  $p$ , we could estimate  $\mathbf{E}[f(x)]$  as:

$$\mathbf{E}[f(x)] \approx \frac{1}{n} \sum_{i=1}^N f(X_i)$$

where  $X_i \sim p$ . Now consider a positive probability density function  $q$ . Adapting equation 2.1 by multiplying and dividing by  $q$  gives:

$$\mathbf{E}[f(x)] = \int f(x)p(x)dx = \int f(x)\frac{p(x)}{q(x)}q(x)dx \approx \frac{1}{n} \sum_{i=1}^N f(X_i)\frac{p(X_i)}{q(X_i)}$$

where, now, the samples  $X_i$  are drawn from the proposal distribution  $q$ .

The ratio  $p(x)/q(x)$  is referred to as the ‘importance weights’ and denoted as  $w(x)$ . Essentially, this is re-weighting the integral to correct for the fact we not sampling from  $p$  but from  $q$ . As the weight corresponds to the

probability distribution  $p$ , the higher the weight of a sample  $X_i$ , the higher the probability.

The distribution  $p$  is referred to as the target probability distribution. Often with Bayesian inference, the target distribution is difficult to calculate in full, and so an ‘unnormalised’ target distribution is used instead.

### 2.3.2 Resampling

A common issue with importance sampling, referred to as ‘degeneracy’, is when all but a few of the weights have negligible value. This results in the majority of the computational effort focusing on samples which have very low probabilities. If the ‘effective sample size’,  $N_{eff}$ , falls below a threshold, then resampling will be performed.  $N_{eff}$  is defined as:

$$N_{eff} = \frac{1}{\sum_i \tilde{w}^i{}^2}$$

where  $w^i$  are the weights. Resampling can be thought of ‘survival of the fittest’. We generate a new population of samples by sampling with replacement from the existing population, according to the weights. This helps to remove insignificant samples, but note that resampling does not aid thorough exploration of the solution space.

### 2.3.3 The General Approach

The general SMC samplers approach starts by generating an initial sample, of size  $N$ , from the proposal distribution and calculating the weights for the sample. Similarly to the MH algorithm, a new sample is generated by applying the proposal distribution  $q$ . The weights are then updated such that:

$$\hat{w}^i = w^i \frac{p(\hat{x}^i)}{p(x^i)} \frac{L(x^i|\hat{x}^i)}{q(\hat{x}^i|x^i)},$$

for  $i = 1, \dots, N$  and  $L(x^i|\hat{x}^i)$  is the  $L$ -kernel. As  $p$  is an unnormalised target distribution, the weights must also be normalised. If the effective sample size drops below  $N/2$ , then resampling is performed. This process of generating new samples and updating the weights is repeated iteratively.

### 2.3.4 BCART with SMC Samplers

This section outlines the parallel BCART algorithm used in this project. SMC samplers have not previously been used in the context of Bayesian tree models. The algorithm developed for this project has been based on those presented by Green et al. [10], see Appendix A.2.

First, an initial sample of trees is generated using *PSPLIT* and *PRULE*, and the weights set equal to likelihood.

The proposal distribution  $q(T^*|T)$  is defined as in Section 2.1.3. The  $L$ -kernel is defined as  $q(T|T^*)$ , and hence the weight update becomes:

$$\hat{w}^i = w^i \frac{q(T^i|T^*)p(Y|X, T^*)p(T^*)}{q(T^*|T^i)p(Y|X, T^i)p(T^i)}$$

Notice that this is the same expression as that in the acceptance formula in the serial BCART algorithm.

### 2.3.5 Parallelisation of SMC Samplers

Many of the steps in this algorithm are trivially parallelisable, as they involve applying the same function to the population of trees. These steps have been highlighted in green text in Appendix A.2.

The normalisation step requires summing the weights. While this could be parallelised, it was not considered necessary as the number of samples is sufficiently small that it takes minimal time.

The resampling step is the least straightforward to parallelise and the technique would vary given different architectures. The approach taken in this project is explained in Section 3.3.

## 2.4 Research Methodology

It was not considered appropriate to use synthetic data for this project, as this would not necessarily demonstrate that the performance of the algorithms for real data with noise or errors etc.

Similarly, the algorithm could have been evaluated using a single set of data, however, it was considered more appropriate to analyse how the algorithm performs in different problem domains.

The serial and parallel implementations of the BCART algorithm have been compared against a DT and a RF model, in terms of both the predictive performance and the computation time.

## 2.5 The Data

The following datasets were considered, see Appendix B.1 for listing of all scripts used to preprocess the data.

### 2.5.1 Twitter Data

This dataset is being used as part of a project to identify adverse side-effects of various medicinal drugs. The dataset consists of a tweet and the MedDRA medical term for the side effect being described. The feature space is large as it comprises all the words used in all the tweets.

Some tweets are labeled with more than one MedDRA symptom. Only tweets labeled with a single symptom have been considered and the control instances selected as tweets which do not mention an adverse side-effect.

Given the ambiguity of some tweets (some may relate to adverts instead of an individual's complaint) each instance has a relevance score. The data was filtered to only include tweets with the highest scores of 7 or 8.

Each tweet was transformed into a binary representation, with 1 indicating a word occurs in a tweet. This was implemented using the `CountVectorizer` function available in the `scikit-learn` library<sup>1</sup>. The `ngram_range` and `min_df` parameters were set such that the feature set included uni and bi-grams appearing in at least 2 documents.

This generated a dataset with 3,354 instances and 7,888 features. As discussed in Section 4.2, the BCART algorithm did not perform well with such a sparse dataset, so the preprocessing was adjusted to only include words appearing in at least 10 documents. This generated a dataset of with just 935 features.

### 2.5.2 Pancreatic Data

Royal Liverpool University Hospital is investigating biomarkers relevant to pancreatic disease with RNA data from 92 humans, 55 of whom had severe pancreatic disease. The data includes the gene expression of 54,675 genes.

Little preprocessing was required, however the labels and features needed to be matched up as they were stored on separate spreadsheets.

### 2.5.3 Fish Data

University of Liverpool experimented with fish exposed to different chemicals, to analyse the effect on their gene expression. The aim being to predict the pollutants in water.

The data includes 14,182 genes from 482 experiments (including a single chemical and a combination of several chemicals). The control used was samples of fish exposed to just a solvent in the water. This project has only analysed the results of single chemical experiments.

---

<sup>1</sup>[http://scikit-learn.org/0.15/modules/generated/sklearn.feature\\_extraction.text.CountVectorizer.html](http://scikit-learn.org/0.15/modules/generated/sklearn.feature_extraction.text.CountVectorizer.html)



### 2.5.4 Cyber Security Data

Microsoft published data for a Kaggle competition to classify different malware files<sup>2</sup>. The data includes information on the content of various malware files. Initially it was considered that this dataset could be use as part of this project, however investigating the data further highlighted the following issues:

- The dataset only includes malware files, and not any non-malware files, meaning there is no “control” data for a binary classification task;
- The data is in hexadecimal representation and so would require significant preprocessing which was not practical given the time constraints of the project.

Therefore, this dataset was not used for the project.

### 2.5.5 Commercial Data

The initial description of the project specified using data from Shop Direct (provided by the Consumer Data Research Centre). However, discussions with the Data Lead at Shop Direct highlighted difficulties in gaining access to this data due to both the commercial sensitivity of the data and Data Protection issues. Hence, this was not considered a viable source.

---

<sup>2</sup><https://www.kaggle.com/c/malware-classification/data>

# Chapter 3

## Design

### 3.1 Decision Tree and Random Forest

The DT and RF models were written with just a single script each, using the scikit-learn library<sup>1</sup> (see Appendix B.1). Ideally, each of the 10-15 parameters would be tuned. Only 4 were tuned given the time available, see Appendix A.3 for details.

Given the short runtime and the results' variability (especially for the pancreatic dataset), each model has been repeated 500 times.

### 3.2 Serial BCART

#### 3.2.1 Code Structure

The serial BCART algorithm has been written from scratch in Python, with inspiration taken from the open source code available on GitHub<sup>2</sup>.

There are no suitable Python libraries and only a few open source implementations, with the majority written in R. Using R was not viable due to my lack of experience of R and the limited parallelisation options.

The algorithm has been implemented with 2 scripts (see Appendix B.1):

1. **BayesianTree.py** - defines the two classes, Node and Tree, and a function to calculate the acceptance distribution for the Metropolis-Hastings algorithm, see Figure 3.1 for the attributes and methods of each class.

The decision tree has been implemented using a tree data structure, and so is composed of nodes with three pointers to: a parent node, a left child and a right child.

---

<sup>1</sup>[http://scikit-learn.org/0.15/supervised\\_learning.html#supervised-learning](http://scikit-learn.org/0.15/supervised_learning.html#supervised-learning)

<sup>2</sup><https://github.com/acbecker/BART/blob/master/tree.py>

The Node class stores these pointers, the split feature and value, and flags indicating what type of node it is i.e. left or right child.

The Tree class contains methods to generate a proposal tree, calculate the likelihood, calculate the performance metrics and also two debugging functions (creating a diagram of the tree and checking the attribute values are correct).

Appendix B.2.1 provides a description of each attribute and method.

2. **SerialBCART.py** - an implementation of the serial algorithm using the Node and Tree class methods (see Appendix A.1 for the algorithm).

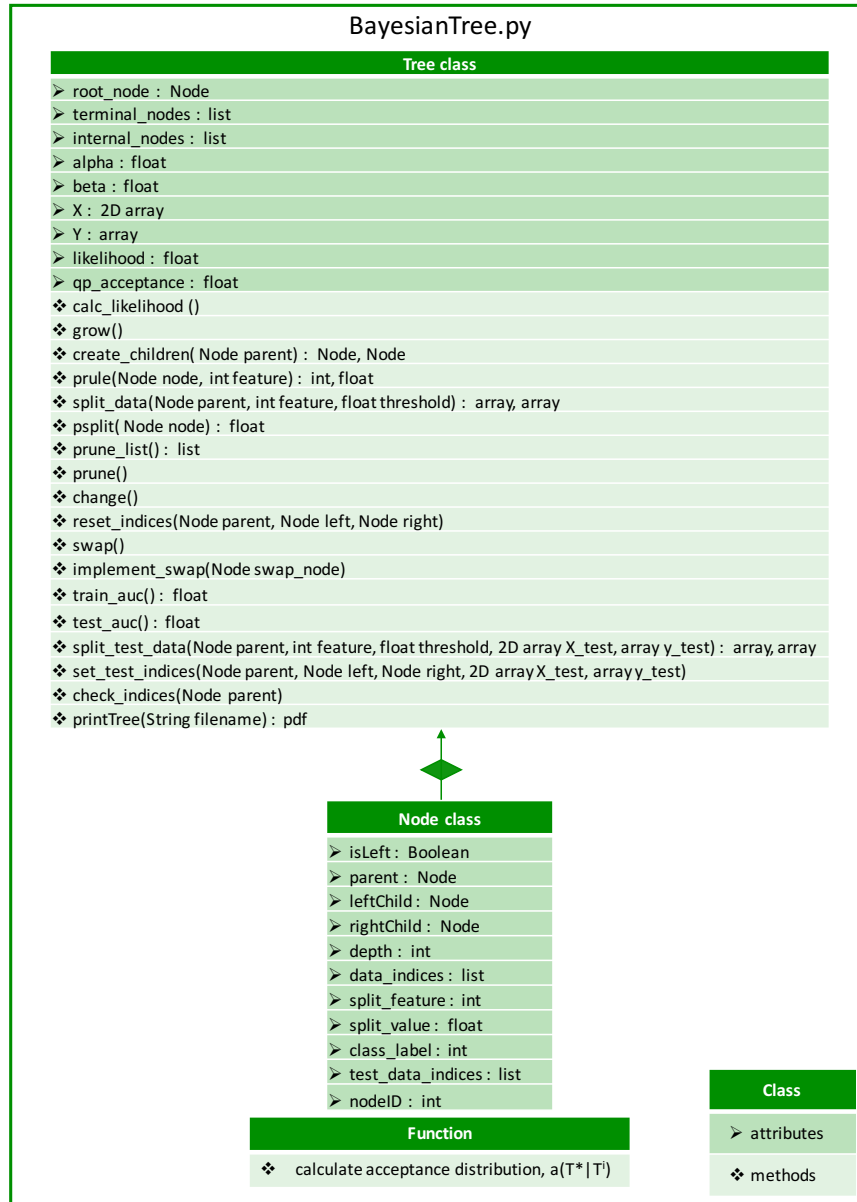


Figure 3.1: Class Diagram for Serial BCART

Appendix A.3 details the serial BCART parameters and their values.

### 3.3 Parallel BCART

The parallel scripts were written in Python using Apache Spark<sup>3</sup>. Spark is a higher-level parallelisation tool compared with MPI or OpenMP, so is easier to program, makes the code more accessible to a data science audience and is specifically built to handle large datasets. There are no open source implementations of SMC samplers currently available.

#### 3.3.1 Brief Introduction to Spark

Spark uses resilient distributed datasets (RDDs) for parallelisation, where either a ‘transformation’ or an ‘action’ can be applied to generate a new RDD or calculate a value from an existing RDD. The following transformations and actions have been used:

1. **map(func)** - returns a new RDD by applying a function to an existing RDD. This is used for all steps that are trivially parallelisable, as the same function is applied to each row in the RDD.
2. **mapPartitions(func)** - returns a new RDD by applying a function to each partition of an existing RDD. This is used for the resampling step.

The code calculates how many times each tree should be copied, as per a stratified sampling approach. This is stored as a list and broadcast. For example, given a sample of four trees, the list may be (0, 2, 1, 1), indicating that 2 copies of the second tree and one copy of the third and fourth tree should be created.

Each partition generates the new samples relating to its partition. So, assume the four trees are divided into two partitions. One would generate the 2 copies of tree 2 and the other would generate the single copies of tree three and four.

3. **collect()** - returns all the elements of the dataset as an array in the driver program, e.g. collecting the weights from the RDD. This is also used to ensure the runtime measurements are accurate as the Spark transformations are ‘lazy’, only executing when an action is called.
4. **broadcast()** - creates a read-only variable cached on each machine. The training and testing data is broadcast at the beginning of the program.

#### 3.3.2 Code Structure

For the BCART algorithm, each row of the RDD consists of a tuple (tree\_id, tree, weight). Figure 3.2 describes, step-by-step, how the algorithm has been implemented in Spark.

---

<sup>3</sup><http://spark.apache.org>

No	Step	Serial/ Parallel	Spark function	Time measurement
Initialise				
1	Read in the data and broadcast the testing and training datasets.	Serial	Broadcast variable	
2	Create a tree which has just a root node. Use this to generate an RDD where each row is a tuple of (tree_id, root_node, tree).	Serial	Create RDD	
3	Generate and broadcast random numbers for step 4.	Serial	Broadcast variable	
4	Generate the initial sample by using a map transformation to grow each tree in the RDD and then update the weight. This returns an RDD where each row	Parallel	Map transformation and collect action	
5	Get the AUC from the training data and the testing data.	Parallel	Map transformation and collect action	
For loop:				
6	Calculate the normalised weights and N_eff.	Serial	n/a	
If $N_{eff} < N/2$ :				
7	Calculate and broadcast which trees should be resampled as per a stratified sampling approach.	Serial	Broadcast variable	
8	Generate the new sample by applying a mapPartitions transformation.	Parallel	MapPartitions transformation	
Else:				
9	Generate a new RDD with the normalised weights.	Serial	Create RDD	
10	Generate and broadcast random numbers for step 11.	Serial	Broadcast variable	
11	Use a map transformation to generate a new sample of trees from the proposal distribution. Collect the new RDD (the weights will be used in step 6).	Parallel	Map transformation and collect action	
12	Get the AUC from the training data and the testing data.	Parallel	Map transformation and collect action	



Key  
 Runtime of whole program  
 Runtime of transformations

Figure 3.2: Spark Implementation of BCART

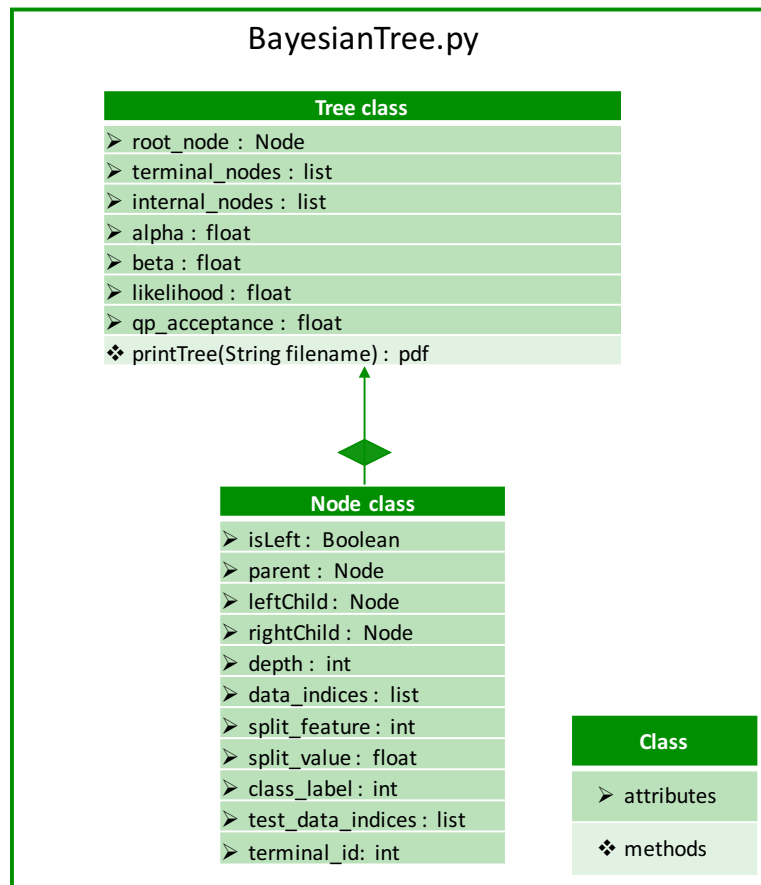


Figure 3.3: Class Diagram for Parallel BCART

The program consists of five scripts:

- **ParallelBCART.py** - implements the parallel Algorithm (see Appendix 2);
- **ParallelBCART\_BayesianTree.py** - defines the Tree and Node classes, see Figure 3.3 for class diagram. The Tree and Node attributes are similar to the serial implementation, except that the training data is no longer held as a Tree attribute, but broadcast out. The only method the Tree class has is the `printTree` method;
- **ParallelBCART\_initialise.py** - generates the initial starting sample and calculates the starting weight;
- **ParallelBCART\_proposal.py** - applies the proposal distribution to the current sample and updates the weight value;
- **ParallelBCART\_evaluate.py** - calculates the AUC of the training and test data.

The algorithm has similar parameters to serial BCART, see Appendix A.3.

## Alternative Code Structure

Originally, the random numbers were generated within the map transformations. It was discovered that the RDD must be cached for this to work, see full discussion in Sections 5.2 and 5.3. This project analysed the runtime of cache and non-cache versions of the algorithm. See Appendix B.1 for a discussion of the key differences between the two scripts.

## 3.4 Evaluation

### 3.4.1 Predictive Performance

$K$ -folds cross-validation was used for perform parameter tuning. This was considered appropriate due to the small number of instances in some of the datasets. Typically  $k = 10$ , however  $k = 5$  is more reasonable here.

The area under the ROC curve (AUC) has been used as the performance metric. It gives a more balanced result as it considers all four sections of the confusion matrix whereas precision, recall or f-score can be easily biased.

### 3.4.2 Runtime

The overall runtime of each algorithm was recorded. In addition, for the parallel algorithm, the runtime of each step of the program (as set out in Figure 3.2) was recorded. In particular, the analysis focuses on the runtime and speedup of the whole program (yellow bar) and the transformations (pink bars).

### 3.5 Hardware

All scripts have been run on a MacBook Pro with Intel Core i7, 16GB RAM, with 2 cores each with 2 threads.

Ideally the parallel script would be executed on a computer with at least 4 cores to provide better evidence of speedup and it was the intention to run the parallel scripts on the Dawson cluster at the Hartree Centre which has 6 nodes, each with 24 cores. However, while the program would run in local mode on Dawson, it failed to run in cluster mode, see Appendix B.2.2. Researching this error highlighted that this is an unresolved bug <sup>4</sup> and contacts at Hartree could not provide any assistance in trying to resolve the issue.

Alternatives to using the Hartree computers included:

- Using the University cluster, however this does not have Spark installed on it;
- Sourcing a 4 core computer or building a small cluster from multiple computers, which was not practical with the time remaining on the project.

Therefore, the parallel scripts have only been run on the aforementioned computer and hence it is difficult to draw strong conclusions from the experiments performed, but they provide some insight nevertheless.

Note: testing the code on the Hartree computers highlighted differences in software versions (Python 2.6 and not 2.7). Similarly, the computers do not have the Graphviz or the scikit-learn libraries, so the code had to be adapted for this.

### 3.6 Changes from the Original Design

The changes from the original project design are:

- The parallel script could not be executed on the Hartree clusters, see Section 3.5;
- The parallel and serial scripts could not use the same Tree and Node classes, see Section 5.1;
- The small number of instances in the fish dataset meant the results were not meaningful, see Section 4.3;
- A gradient-boosted tree model was initially considered, resulting in the single tree BCART model being compared with two ensemble methods. The DT model provided a more direct comparison;
- The RF model was tuned on the `max_features` parameter. Originally the values for tuning were ‘auto’, ‘log2’ and ‘None’. The latter con-

---

<sup>4</sup><https://issues.apache.org/jira/browse/SPARK-10795>

siders all of the features at every split, which took a disproportionate amount of time to run, around 100 times longer than the other values. Hence, this value was dropped from the parameter tuning;

- The original design document stated that several different symptoms would be analysed from the Twitter data, however, time did not allow for this, and ‘pain’ was selected as the most commonly occurring symptom.



## Chapter 4

# Results

### 4.1 Pancreatic Data

#### 4.1.1 Runtime Results

##### Comparison of the Tree Algorithms

Figure 4.1 provides the runtimes of the four algorithms. RF and DT take just a matter of minutes to run 500 iterations, even when the parameter settings require them to generate 100 trees or perform calculations on every feature in the data set.

Algorithm	Time in minutes			Iterations	Parameter settings
	Average	Min	Max		
Random Forest	2.4	1.3	4.1	500	All
Decision Tree	1.8	0.1	5.1	500	
Serial BCART	1,011.2	981.2	1,030.3	5	$\beta = 1.5$
Parallel BCART	13.9	11.1	15.4	40	Min split value = 0

Figure 4.1: Pancreatic: Comparative Runtimes

As expected, the serial BCART algorithm takes a long time to run, around 17 hours. The surprising result is that the parallel implementation of BCART, run in local mode with Spark, takes around 14 mins. So by just changing the software used to implement BCART, it is possible to obtain a speedup in the region of 70. The reasons for this could be due to:

- The parallel algorithm has just 1,000 iterations compared with 5,000 in the serial algorithm;
- Spark may be better configured to optimise data handling than the Python code;
- The serial algorithm is continually switching between different operations, whereas the parallel algorithm repeatedly applies the same operation, which could make better use of the hardware;

- The Tree class in the serial algorithm includes the training data as an attribute, whereas the parallel algorithm broadcasts the data, and so, the serial Tree objects will be much larger.

Note: the BCART times in Figure 4.1 are for just a single parameter setting, as the serial runtime precluded numerous trials. The runtime did appear to vary given different parameter settings.

### Parallel BCART Analysis

Figure 4.2 gives an example breakdown of the parallel BCART runtime when executed using one thread. We see that around 97% of the total runtime relates to Spark transformations, and hence can be run in parallel. At first this appears to be promising in terms of the speedup. However, the average runtime of each transformation, at each iteration, is less than half a second (see the right most column) and given the expense of coordinating threads, little speedup is achieved on these transformations, see Figure 4.3.

Step	Spark function	Serial/ Parallel	Total time (secs)	Average time (secs)
Spark set up	Create SparkContext object	Serial	2.14	
Reading data	n/a	Serial	5.70	
Broadcasting data	Broadcast variable	Serial	0.38	
Build forest of root node trees	Create RDD	Serial	0.23	
Generating random numbers for initialisation step	Broadcast variable	Serial	0.04	
Generate initial sample from proposal distribution	Map transformation	Parallel	1.63	
Calculate AUC for initial sample	Map transformation	Parallel	0.45	
Calculate sum of weights and N_eff	n/a	Serial	2.20	0.00
Calculate trees to resample	Broadcast variable	Serial	3.80	0.00
MapPartition to get resample	MapPartition transformation	Parallel	232.55	0.23
Normalise weights	Create RDD	Serial	0.00	0.00
Generating random numbers for proposal step	Broadcast variable	Serial	4.79	0.00
Perform proposal step	Map transformation	Parallel	122.74	0.12
Calculate AUC for new sample	Map transformation	Parallel	383.51	0.38
<b>TOTAL SERIAL</b>			19.26	0.01
<b>TOTAL PARALLEL</b>			740.88	0.74
<b>TOTAL</b>			760.14	0.75

Figure 4.2: Example Runtime of the Parallel BCART Algorithm  
(local mode, 512 samples, 1,000 iterations)

Figure 4.3 shows the average results of running the parallel script on a single laptop with 2 cores, each with 2 threads, a total of ten times. The highest speedup achieved is just 1.24 with 2 threads, and only the AUC calculation appears to really benefit from the additional core. These results are not surprising given the small grain size and the limitation of just 2 cores.

		Runtime with 512 samples				Runtime with 4096 samples			
		No of threads							
		1	2	3	4	1	2	3	4
Step no.	Step Description	1	2	3	4	1	2	3	4
4	Generating initial sample	1.61	1.63	1.76	1.90	2.50	2.23	2.56	2.54
8	Resampling	0.26	0.25	0.27	0.28	2.03	1.78	1.82	1.71
11	Generate new sample from proposal distribution	0.13	0.12	0.13	0.15	0.96	0.68	0.73	0.65
12	Calculate the AUC	0.40	0.25	0.25	0.26	2.92	1.54	1.82	1.30
	Total	85.82	69.15	73.31	77.13	594.56	409.49	445.78	376.69
	Speedup		1.24	1.17	1.11		1.45	1.33	1.58

Average runtime (seconds) of the Spark transformations in the parallel BCART algorithm as the number of threads increases from 1 to 4. The number of samples was set at 512 and 4096 and the number of iterations set at 100. The step no. relates to Figure 3.1.

Figure 4.3: Pancreatic: Speedup

To investigate increased grain size, the experiment was run using 4,096 samples (see the second half of the table in Figure 4.3). Again, the speedup is limited for threads 3 and 4 as there are just 2 cores, but the total speedup for 2 threads does increase to 1.45.

Note: for these experiments the number of iterations was reduced to 100 to allow time for multiple runs to understand the variability of the results. The full results are included in Appendix C.1.

## Analysis of Caching

As mentioned in Section 3.3.2, this project explored a second implementation of the parallel BCART algorithm, where the RDDs are cached instead of being recalculated. Each script was run with the same parameter settings, for 1 and 2 cores, a total of ten times, see Figure 4.4.

These experiments show that the non-cache script actually runs quicker than the cache script. While the resampling step and calculating the AUC take less time when the RDDs are cached, applying the proposal distribution takes significantly longer with caching. This is because the RDD must be checkpointed which involves saving the RDD to disk (see Section 5.3). Interestingly, the cache code does not gain much speedup, just 1.02, over two cores, compared with a speedup of 1.25 for the non-cache code.

Given these results, the cache script was not used for any further experiments.

		Cache script		Non-cache script		Difference	
		No of cores					
Time measurement		1	2	1	2	1	2
Total runtime (secs)		999.72	980.51	661.61	528.12	338.11	452.40
Total runtime (mins)		16.66	16.34	11.03	8.80	5.64	7.54
Speedup			1.02		1.25		
Average runtime across 1,000 iterations	Resample	0.24	0.24	0.27	0.26	-0.03	-0.02
	Applying proposal distribution	0.46	0.43	0.14	0.12	0.32	0.31
	Calculating AUC	0.28	0.28	0.30	0.19	-0.03	0.09
Total runtime of all 1,000 iterations (secs)	Resample	236.89	243.86	269.82	258.94	-32.93	-15.08
	Applying proposal distribution	459.52	427.14	135.68	121.15	323.85	305.99
	Calculating AUC	276.41	284.39	302.54	189.79	-26.14	94.60
Total runtime of all 1,000 iterations (mins)	Resample	3.95	4.06	4.50	4.32	-0.55	-0.25
	Applying proposal distribution	7.66	7.12	2.26	2.02	5.40	5.10
	Calculating AUC	4.61	4.74	5.04	3.16	-0.44	1.58

Figure 4.4: Runtime Comparison for Cache and Non-Cache Scripts

## 4.1.2 Predictive Performance

### Comparison of the Tree Algorithms

Figure 4.5 gives the final AUC results for the RF, DT and parallel BCART (note: serial BCART has been excluded as the lengthy runtime prevented full parameter tuning, see Section 4.1.2).

An initial observation is that the results across the different data splits vary quite substantially. Similarly, for all of the algorithms, the AUC varies from one run to another. The DT has a standard deviation of around 11% across all 500 iterations (see Appendix C.2). This could be due to:

- The small number of instances available for training;
- The dataset being unbalanced. There are more severe instances than mild, 55 and 37 respectively (see Appendix C.1 for the composition of each of the 5 splits).

In particular the results from split 3 are poor, where the testing data has more mild instances. This could indicate that the small number of instances precludes strong conclusions.

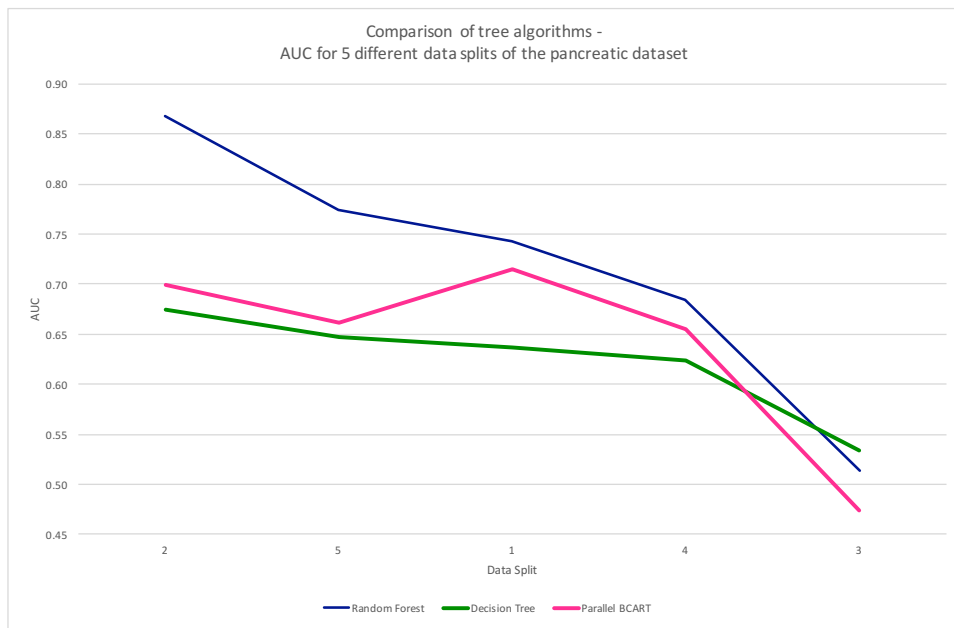


Figure 4.5: Pancreatic: AUC Comparison

## Serial BCART

Figure 4.6 shows how the serial BCART performed across the 5 data splits with varying values of  $\beta$ . Given the runtime of the serial BCART algorithm it was not possible to perform the full parameter tuning or repeatedly run the code to ensure reliable results, so no further parameter tuning was done with the serial script. However, these results did provide some insight. The algorithm appears to perform better, across all the data splits, with  $\beta$  equal to 1.5. In these experiments, the `min_split_value` was set to 0. As a higher value of  $\beta$  reduces the size of the trees, it may help to prevent the trees from overfitting, analogous to pruning a decision tree.

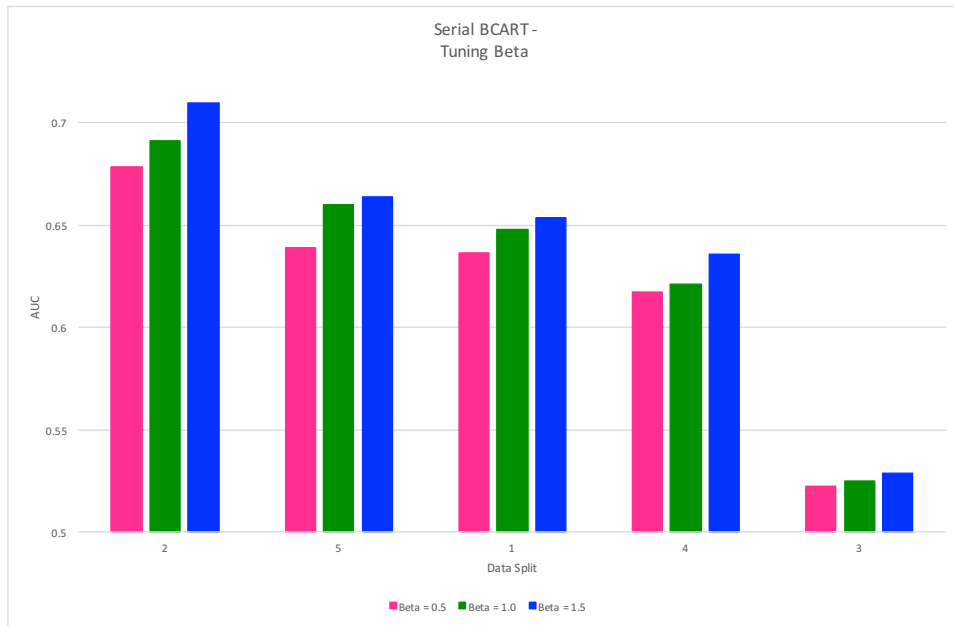


Figure 4.6: Serial BCART Parameter Tuning

## Comparing the Serial and Parallel Implementations

To test whether the two implementations are equivalent, both scripts were run multiple times and a comparison of the average results performed. The same parameter settings and data split have been used, with the serial code run a total of 5 times and the parallel code 40 times, see Figure 4.7.

Algorithm	AUC			
	Average	Min	Max	$\sigma$
Parallel BCART	0.657	0.492	0.770	0.06
Serial BCART	0.658	0.647	0.664	0.01
Difference	-0.001	-0.155	0.105	0.06

Results of running each script with data split 1  
(beta = 1.5 and minimum split value = 0)

Figure 4.7: Pancreatic: Serial and Parallel Implementation Comparison

This highlights a couple of interesting points:

- The average results of the parallel and serial algorithms have a difference of just 0.1%, suggesting that the two implementations are equivalent;
- The parallel algorithm appears to generate much more variable results. This could be due to the algorithm “getting stuck” in the solution space. The parallel algorithm resampled in 99% of the iterations, on average. This is an indication that the algorithm is not exploring the solution space fully but getting stuck in a local minimum. It may even be the case that, after a few iterations, all the trees derive from the same “best” tree found in the initial sample, and hence the sample trees centre around a single point in the solution space.

## Pancreatic: Parallel BCART Parameter Tuning

The variation in results from one run to another makes it difficult to confidently tune parameters. Ideally, the scripts would be repeatedly run for all parameter combinations many times, but time did not allow for this. This section explains the parameter tuning performed on the BCART algorithm, using the parallel implementation.

The first stage was to run three times for the minimum split values of 0, 10 and 20,  $\beta$  equal to 1.5 and 512 samples (see Figure 4.8). A low AUC average ruled out a minimum split value of 0 (overfits trees), but it was not clear which of 10 and 20 was better. The second stage involves running the script for a further 7 times. While these results were still not clear cut, it seems to suggest that a minimum split value of 20 gives a better, less varying AUC.

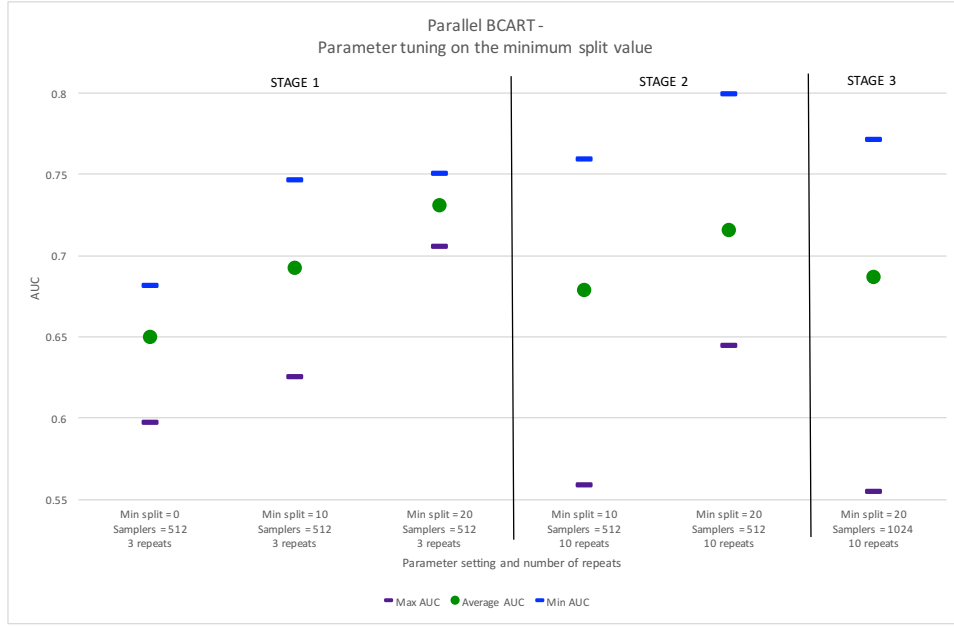


Figure 4.8: Pancreatic: Parallel BCART Parameter Tuning

Finally, for the third stage, the number of samples was increased to 1,024. It was expected that this would improve the predictive performance of the algorithm, however, this was not this case. This suggests that the proposal distribution is not heavy-tailed enough, resulting in a single sample dominating the estimate, and hence taking more samples does not improve the result.

The solution to this is to change the prior and proposal distributions to be more heavy-tailed. One way to do this is to build up trees according to the number of terminal nodes. For example, determine the number of terminal nodes from a Poisson distribution and then generate the tree, instead of building the trees using  $p_{SPLIT}$  and  $p_{RULE}$ .

The results from serial BCART suggest that  $\beta = 1.5$  gives better results. A higher value of  $\beta$  makes more bushy trees more likely. Having a minimum split value of 20 will already lead to shorter trees given there are around 70 training instances, and hence it is assumed the value of  $\beta$  will not have a significant impact on the result and so no further parameter tuning was performed due to time restraints.



## 4.2 Twitter Data

### 4.2.1 Runtime Results

Firstly, the total runtimes of all the tree algorithms are compared in Figure 4.9. The RF and DT algorithms take just minutes to complete 500 repeats of the algorithm, serial BCART takes around 14 hours and parallel BCART around 15 minutes. Broadly speaking, this is a similar pattern to the pancreatic runtimes, as expected, and an initial speedup of around 60.

Algorithm	Time in minutes			Iterations	Parameter settings
	Average	Min	Max		
Random Forest	5.1	2.7	8.3	500	All
Decision Tree	1.8	1.0	2.7	500	
Serial BCART	848.6	731.8	888.7	5	$\beta = 1.5$
Parallel BCART	14.9	14.4	15.5	10	Min split value = 10

Figure 4.9: Twitter: Comparative Runtimes

Figure 4.10 shows the runtimes of the parallel BCART script with 1, 2, 3 and 4 threads. Note that the resampling step is not included as the algorithm never resamples, see Section 4.2.2.

With 512 samples, there is a speedup of 1.31 from using 2 threads, and when the number of samples increases to 1,024, the speedup increases to 1.40. There is no significant difference between the runtime when 2, 3 or 4 threads are used, a full results table is included in Appendix C.2. This is a very similar result to the pancreatic runtimes.

Note: initially the number of samples was increased to 4,096, to allow direct comparison with the pancreatic results. However, this gave a “java.lang.OutOfMemoryError: Java heap space” error, so the number of samples was reduced down to just 1,024.

		Runtime with 512 samples				Runtime with 4096 samples			
		No of threads							
Step no.	Step Description	1	2	3	4	1	2	3	4
4	Generating initial sample	2.19	2.06	2.27	2.29	2.87	2.53	2.58	2.74
11	Generate new sample from proposal distribution	0.45	0.38	0.39	0.39	1.00	0.79	0.75	0.76
12	Calculate the AUC	0.64	0.41	0.40	0.41	1.29	0.76	0.71	0.71
	Total	130	99	97	100	262	188	177	179
	Speedup		1.31	1.33	1.30		1.40	1.48	1.47

Average runtime (seconds) of the Spark transformations in the parallel BCART algorithm as the number of threads increases from 1 to 4. The number of samples was set at 512 and the number of iterations set at 100. The step no. relates to Figure 3.1.

Figure 4.10: Twitter: Speedup

## 4.2.2 Predictive Performance

### Initial Experiments

As mentioned in Section 2.5, originally the Twitter data was preprocessed to have a total of 7,888 features. However, the BCART algorithm only achieved on AUC of around 0.55, while with limited parameter tuning the DT algorithm achieved around 0.82. This could be due to the following:

1. The data is very sparse. Of the 7,888 features around 5,000 appear in just 2 or 3 instances. The BCART algorithm will only grow a node if there is a minimum threshold of instances in each of the terminal nodes (the minimum split value). It is thought that it takes a long time to find the features which more evenly split the data by randomly selecting among all the features, hence why a greedy approach is more effective in this situation;
2. The trees tend to be very small and in some cases just the root node. This is because the proposal distribution is just as likely to prune a tree as it is to grow it. If it does decided to grow, it is likely to select a feature that will not split the data evenly, and hence the node will not actually be grown;
3. In contrast to the results of the Twitter data, the algorithm never resamples. Figure 4.11 shows the AUC of all the samples at one iteration. Clearly, there are 4 trees that perform significantly better than the other trees, and yet it doesn't resample, see Section 4.2.3.

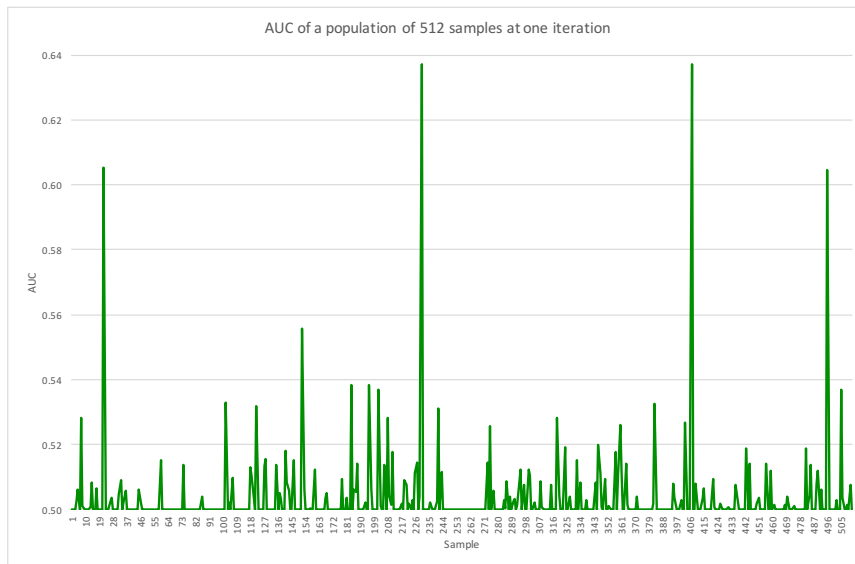


Figure 4.11: AUC of all Samples at One Iteration

For the remainder of this section, the analysis is based on preprocessed data where words appearing in 10 or more tweets were selected, resulting in 785 features.

## Comparison of Tree Algorithms

Figure 4.12 shows that BCART still struggles to compete against both the DT and RF algorithms, only achieving an AUC of around 0.62, compared with around 0.95. Even with just 785 features, the data is still too sparse and BCART takes too long to find the good features.

Interestingly the DT is the best performing algorithm when its `max_features` parameter, which sets how many features are considered when looking for the best split, was set as “None”, and hence considers every feature at every split. So it appears that when the data is very sparse, a greedy approach is very effective.

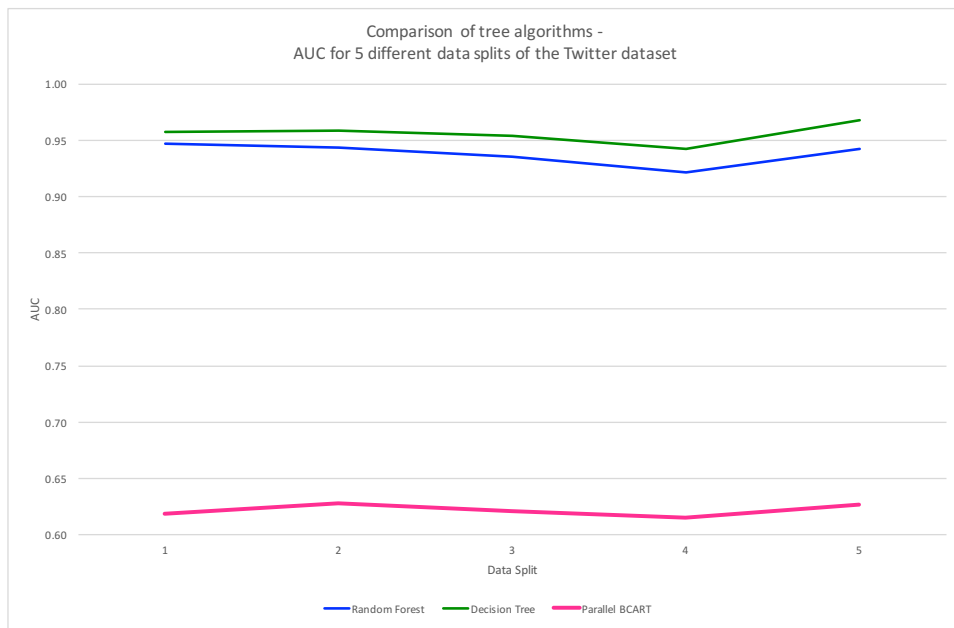


Figure 4.12: Twitter: AUC Comparison

## Comparing the Serial and Parallel Implementations

The serial and parallel implementations of BCART appear to give different results with the serial implementation achieving around a 4% higher AUC, see Figure 4.13. As the data is very sparse, it could be that the parallel implementation, with just 1,000 iterations, is unable to explore the solution space as much as the serial implementation, which has 5,000 iterations.

If the last 4,000 iterations of the serial implementation are ignored, the results are much more comparable, with just a 0.1% difference, supporting the hypothesis that the two implementations are equivalent.

Algorithm	AUC			
	Average	Min	Max	$\sigma$
Parallel BCART	0.618	0.608	0.624	0.006
Serial BCART	0.659	0.657	0.660	0.001
Difference	-0.041	-0.049	-0.036	0.005
Serial BCART - only the first 1,000 iterations	0.619	0.610	0.621	0.005
Difference	-0.001	-0.002	0.002	0.001

Results of running each scripts with data split 1, beta = 1.5 and minimum split value = 10.

Figure 4.13: Twitter: Serial and Parallel Implementation Comparison

## Parallel BCART Parameter Tuning

The parallel BCART was run 3 times for each parameter setting using the first data split. The standard deviation on these results was minimal, around 0.1%, and regardless of the parameter setting the results were very similar (a range of just 1.15%). Plotting these results shows that using a minimum split value of 20 gives slightly better results, see Figure 4.14.

The script was then run on each data split with  $\beta$  at 0.5, 1.0 and 1.5 and the minimum split value at 20. Again these results showed little variance and it seems the value of  $\beta$  makes little difference to the results. Similarly, when the number of samples was increased to 1,024, there was not a significant difference in the AUC. See Appendix C.2 for the full table of results.

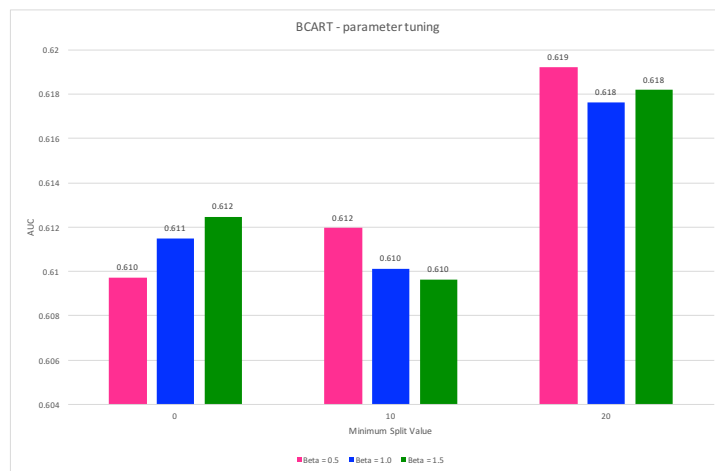


Figure 4.14: Parallel BCART Parameter Tuning

### 4.2.3 Computing Error

After this analysis was performed, it became apparent that there was an error in the BCART computations, due to an error in calculating the weights. Calculating the likelihood for the weights uses a Beta function where the parameters values are the count of each of the instances. If a tree has just a root node, then it contains 1,329 pain instances and 1,351 control instances (for data split 1). Then,  $Be(1 + 1329, 1 + 1351) = 4.6 \times 10^{-809}$  - recorded as zero in the computation.

This explains the differences between the pancreatic and Twitter results, namely, that the algorithm nearly always resampled with pancreatic data but never did with the Twitter data, and the differences in the AUC results compared to the other tree algorithms.

Curiously, neither the parallel or serial scripts threw an error or warning message indicating the calculations were dividing by zero, which is why the error was not picked up until after the above analysis had been performed. A resolution to this issue would be to calculate the weights on a logarithmic scale, however, time ran out to implement this before the project deadline.

### 4.3 Fish Data

As mentioned in Section 2.5, the fish dataset has very few instances, the smallest just 5 and the largest 13 for a particular chemical. The DT script was run using the chemical with the largest number of instances, ‘Levo’, so the dataset has a total of 26 including the controls, which was split into a training and testing splits of 20 and 6, respectively.

Over the 500 iterations, there is a standard deviation of around 21%, see Appendix C.3 for table of the results. Figure 4.15 shows the AUC from each repeat for one of the parameter settings. With just 6 testing instances, the AUC varies from 1.0 to 0.33.

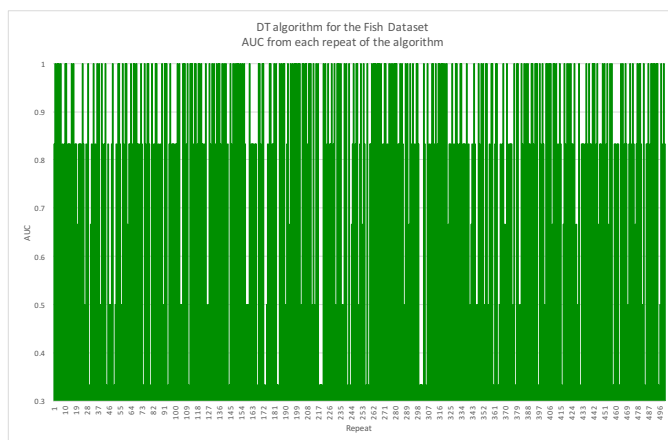


Figure 4.15: AUC Result of running a DT model with the Fish Dataset

This suggests that there just aren’t enough instances to both effectively train and accurately test the model. Therefore, no further experiments were performed with this dataset.

## Chapter 5

# Learning Points

The main challenges faced in this project were a lack of:

- General programming experience;
- Standard libraries for Bayesian tree models;
- Experience of Apache Spark;
- Familiarity with the Hartree environment;
- Prior knowledge of Bayesian tree models, MCMC, the HM algorithm and SMC samplers.

Even just writing the serial code was a challenge and my knowledge of Spark has been completely self-taught. The remainder of this section discusses some of the aspects of Spark which were important for this project.

### 5.1 Code Reuse and Recursive Functions

Originally, it was anticipated that the `BayesianTree.py` script, containing the `Tree` and `Node` classes, could be used for both the serial and parallel implementations, to maximise code reuse. As the `Tree` class has a tree data structure, recursive functions are the natural choice to iterate over all the nodes.

However, when trying to execute `Tree` class methods within Spark, the results were not as expected. For example, it seems that as soon as a `return` command is used, the executor terminates, see Appendix D for a simple example. Attempts to use recursive function within the map transformations were unsuccessful. There were similar issues with calling functions from within a map transformation.

My solution to these issues was to completely rewrite the parallel scripts such that the recursive functions were converted into non-recursive functions, and each map transformation was stand-alone without calling any other functions. Given more time or guidance, it may be possible to find more elegant solutions to these issues.

## 5.2 Random Numbers

Random numbers are required in several steps of the BCART algorithm. At first it appeared that repeatedly calling a function that generates a random number, would overwrite the previous random number in the RDD. This is illustrated in Figure 5.1. Line 5 shows the output of calling `random_test`. This is a simple map transformation which appends a random number to each row in the RDD. In line 6, the function is called for a second time and the output shows that the random numbers from the previous call have been reset.

```
In [1]: import numpy as np
        from random import randint

In [2]: def random_test(row):
        seed = int(os.urandom(4).encode('hex'), 16)
        rs = np.random.RandomState(seed)
        rand = rs.randint(100)

        return row, rand

In [3]: rdd_rand = sc.parallelize(np.arange(0,10))

In [4]: rdd_rand = rdd_rand.map(random_test)

In [5]: for row in rdd_rand.collect():
        print row

(0, 5)
(1, 14)
(2, 16)
(3, 84)
(4, 37)
(5, 16)
(6, 89)
(7, 79)
(8, 83)
(9, 51)

In [6]: rdd_rand = rdd_rand.map(random_test)

In [7]: for row in rdd_rand.collect():
        print row

((0, 89), 65)
((1, 84), 78)
((2, 40), 33)
((3, 42), 69)
((4, 55), 47)
((5, 67), 15)
((6, 88), 19)
((7, 5), 69)
((8, 80), 7)
((9, 42), 25)
```

Figure 5.1: Demonstration of Random Numbers in Spark

It took a while to realise the issue here. Once an action has been called on an RDD, the RDD is discarded. Only if the RDD is explicitly cached (within the code) will it be stored and reused. As the BCART algorithm is iterative, the RDD must either be cached or the random numbers generated by the master node in serial and broadcast.

I investigated a number of other ways to resolve this issue:

- The pyspark mllib library can be used to generate an RDD of random



numbers<sup>1</sup>, but this would require several RDDs to be merged together which would probably be expensive;

- The Spark SQL module can append a column of random numbers to a DataFrame<sup>2</sup>, but unfortunately it is not possible to populate a DataFrame with a tree object as it only supports standard data types.

## 5.3 Caching

To cache an RDD, at first, appears simple. All that is required is to add `.cache()` to the end of a RDD transformation. But it actually took a while to debug the script, due to the following issues:

- Spark builds up a RDD lineage, a graph of all the parent RDDs of a particular RDD. With 1,000 iterations, the lineage becomes very long, and Spark throws a ‘StackOverflow’ error. To truncate the lineage, an RDD can be checkpointed: the RDD is saved to disk and all references to the its parent RDDs are removed.
- The cache script also threw a ‘SocketTimeoutException’. The solution to this error is to increase heart beat interval between the executor and the driver.

The Spark documentation suggests that caching the RDDs is appropriate for iterative algorithms<sup>3</sup>. However, it actually took longer to run and took significantly more time to debug.

## 5.4 Normalising weights

It was originally intended that the weight would be normalised in parallel. However, while writing and testing the non-cache script some strange results occurred. At some point the weights were not normalised by the sum of the weights, but by some random and very large number. This resulted in weights approaching a zero value, and then when normalised in the next iteration they end up as “inf”. Attempts to fix the problem have included using a broadcast variable instead of passing the sum to the function, but this did not prove successful.

Curiously, the exact same function did work in the cache script, which suggest that the issue may relate to the RDDs not being cached, although it is not clear why this would be the case.

To avoid delaying the progress of the project, a temporary fix to the problem was to serialise the normalisation. This involved creating a new RDD with

---

<sup>1</sup><http://spark.apache.org/docs/latest/mllib-statistics.html#random-data-generation>

<sup>2</sup><https://spark.apache.org/docs/1.6.1/api/python/pyspark.sql.html#pyspark.sql.functions.rand>

<sup>3</sup><http://spark.apache.org/docs/latest/quick-start.html#caching>

the normalised weights. The runtime results highlight that this step takes, on average, just 0.05 seconds for the pancreatic data and 0.13 seconds for the Twitter data, see Appendix C.1 and C.2.

The parallel normalisation, in the cache script, has a comparable runtime, and hence, it seems that there is little runtime difference between creating a new RDD and applying this simple map transformation.

Similarly, it was found that using the `collect()` action to return the whole RDD was quicker than applying a map function to just return the weight value. This highlights the point that it is not always quicker to use parallelisation. Although it is worth noting that this may not be the case for much larger RDDs, and for very large RDDs it may not be possible to call the `collect()` action.

## Chapter 6

# Professional Issues

This section outlines how the project was conducted in accordance with the British Computer Society (BCS) Code of Conduct [11] and Code of Good Practice [12].

Regarding public interest, the datasets used in this project were sourced from existing research projects within the University, and hence, the appropriate ethical considerations had already been made. The data has not been shared with anyone outside of the University of Liverpool or the Hartree Centre.

This project has provided the opportunity to develop my programming, research, planning and machine learning skills. I have regularly sought feedback on my work and have provided an honest evaluation of the project in this report.

The relevant authority for this project was the University of Liverpool, and to a lesser extent, the Hartree Centre. The data has not been shared outside the University of Liverpool and the Hartree Centre, and the results presented in this report are undoctored.

Regarding my duty to the profession, the scripts produced for this project are well documented and existing libraries used where possible.

## Chapter 7

# Evaluation

### 7.1 Strengths

This was an ambitious project, both in terms of the challenge in digesting so many new concepts and the volume of work in coding an algorithm without an existing library, both in serial and parallel, and analysing three different datasets, alongside the preprocessing need to do this.

SMC samplers had not been applied in the context of Bayesian CART models before or been implemented in Spark. Additionally, there are relatively few papers on SMC samplers, the majority of which are very theoretical.

The project produced some interesting results, chiefly reducing the runtime of the BCART algorithm to 10 minutes just by changing the software, generating a speedup of around 70. This enabled insight into how the algorithm works and how it can be improved, and also allowed for more comprehensive parameter tuning.

In addition to this, the project explored two different parallel implementations, demonstrating how different approaches can affect the runtime.

### 7.2 Weaknesses

The most disappointing aspect of the project was not being able to run the BCART algorithm on the Hartree cluster, as this prevented making any strong conclusions on the speedup.

It was also disappointing that the results on the Twitter dataset were invalid. Ideally, there would have been time to rectify this.

It would have been desirable to perform more thorough parameter tuning for the BCART algorithm on the pancreatic dataset, but the variation in the results, runtime of the script and time restriction precluded this.

Similarly, there was no time to fully investigate the effect of different pre-processing techniques on the Twitter data. For example, experimenting with removing stop words, using a different range of n-grams or using a non-binary representation.

Finally, if time have allowed, it would have been interesting to analyses several different symptoms from the Twitter data.

## Chapter 8

# Conclusions

### 8.1 Runtime

As expected, the DT and RF algorithms have a very short runtime, while the serial BCART algorithm took many hours. The surprising result from this project was how much speedup can be gained by just changing the software, as much as 70, with the parallel BCART code run in local mode. Reducing the BCART runtime down to around 10 - 15 minutes makes it possible to gain greater insight into the algorithm, gain more reliable and accurate results by running many more repeats and enable more thorough parameter tuning.

While it is not possible to draw definite conclusions on the speedup of parallel BCART using a single computer with just 2 cores, the results showed the potential to obtain a speedup of around 1.25 with 512 samples and an even greater speedup as the number of samples increases.

Caching the RDDs was expected to reduce the runtime of the parallel scripts, but this was not the case as the RDDs have to be checkpointed which involves saving to disk.

### 8.2 Predictive Performance

The results from the pancreatic data show that the BCART algorithm has potential as it outperformed the DT. While BCART did not perform better than RF, ensemble methods do generally perform better. Although the variance in the results perhaps indicates that more data is required to make any solid conclusions.

The frequency with which the algorithm resamples suggests that the prior and proposal distributions are suboptimal, and hence there is room to improve these.

While it appeared that the BCART struggled with very sparse data, no conclusions could be drawn from the Twitter data results due to the error

in calculating the weights.

### 8.3 Further Research

Firstly, further work would focus on addressing the weakness described in Section 7.2 and to source and analyse further real life datasets, to provide more definitive conclusions.

An obvious extension to this project is to implement a sum-of-trees model, instead of just the single tree.

Given the number of features in these datasets, BCART may well miss some of the “good” features because it just takes too long to search through all of them. It could be more efficient to add a greedy element to the algorithm. For example, the features could be ranked, just as in the DT algorithm, but instead of always choosing the best feature, pick a feature with probability proportional to its weighting.

It should be noted that ranking the features is very expensive, as seen with the RF where it could increase the runtime by as much as 100 times. Therefore, it could be beneficial to parallelise this step.

It would be interesting to experiment with different prior and proposal distributions, that are more heavy-tailed. Currently, they do not create a sufficiently diverse set of trees.

Alongside running the scripts on a cluster, the runtime of the algorithm could potentially be improved by tuning the Spark configurations, such as, optimising memory usage and the level of parallelisation.

# Bibliography

- [1] I. Guyon and A. Elisseeff, “An Introduction to Variable and Feature Selection,” *J. Mach. Learn. Res.*, vol. 3, pp. 1157–1182, Mar. 2003.
- [2] S. K. Murthy and S. Salzberg, “Decision tree induction: How effective is the greedy heuristic?,” in *Proc. of the First Int. Conf. on Knowledge Discovery in Databases*, pp. 222–227, 1995.
- [3] H. A. Chipman, E. I. George, and R. E. McCulloch, “BART: Bayesian additive regression trees,” *The Annals of Applied Statistics*, pp. 266–298, 2010.
- [4] P. Del Moral, A. Doucet, and A. Jasra, “Sequential monte carlo samplers,” *Journal of the Royal Statistical Society: Series B (Statistical Methodology)*, vol. 68, no. 3, pp. 411–436, 2006.
- [5] H. A. Chipman, E. I. George, and R. E. McCulloch, “Bayesian CART Model Search,” *Journal of the American Statistical Association*, vol. 93, pp. 935–948, Sept. 1998.
- [6] Y. Wu, H. Tjelmeland, and M. West, “Bayesian CART: Prior Specification and Posterior Simulation,” *Journal of Computational and Graphical Statistics*, vol. 16, pp. 44–66, Mar. 2007.
- [7] D. G. T. Denison, B. K. Mallick, and A. F. M. Smith, “A Bayesian CART algorithm,” *Biometrika*, vol. 85, pp. 363–377, June 1998.
- [8] M. T. Pratola, H. A. Chipman, J. R. Gattiker, D. M. Higdon, R. McCulloch, and W. N. Rust, “Parallel Bayesian Additive Regression Trees,” *arXiv:1309.1906 [stat]*, Sept. 2013. arXiv: 1309.1906.
- [9] D. J. Wilkinson, “Parallel bayesian computation,” *Statistics Textbooks and Monographs*, vol. 184, p. 477, 2006.
- [10] P. Green and S. Maskell, “Estimating the parameters of dynamical systems from Big Data using Sequential Monte Carlo samplers,” *In review*.
- [11] “The british computer society code of conduct.” <http://www.bcs.org/upload/pdf/conduct.pdf>. Accessed: 16/09/2016.
- [12] “The british computer society code of good practice.” <http://www.bcs.org/upload/pdf/cop.pdf>. Accessed: 16/09/2016.



- [13] T. K. Ho, “The random subspace method for constructing decision forests,” *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 20, pp. 832–844, Aug. 1998.
- [14] L. Breiman, “Random Forests,” *Machine Learning*, vol. 45, no. 1, pp. 5–32.

# Appendix A

## Algorithms

### A.1 Serial BCART

---

**Algorithm 1** Bayesian CART Algorithm using MCMC

---

```
1: for  $r = 1 : 500$  do
2:    $T^0 = \text{root node}$ 
3:   for  $i = 1 : 5000$  do
4:     Generate a candidate tree  $T^*$  from  $q(T^*|T)$ 
5:     Sample  $u \sim U(0, 1)$ 
6:      $a(T^*|T) = \min \left\{ \frac{q(T^i|T^*)p(Y|X, T^*)p(T^*)}{q(T^*|T^i)p(Y|X, T^i)p(T^i)}, 1 \right\}$ 
7:     if  $u < a(T^*|T)$  then
8:        $T^{i+1} = T^*$ 
9:     else
10:       $T^{i+1} = T^i$ 
11:      Record the AUC of the tree based of the training data and the
        testing data
12:    end if
13:  end for
14: end for
15: Calculate result:
    1. For each chain, find the best tree (the highest AUC from the training
        data).
    2. For each chain, get the AUC from the testing data for the best tree.
    3. Return the average AUC across all chains.
```

---

## A.2 Parallel BCART

---

**Algorithm 2** Bayesian CART Algorithm using SMC samplers

---

```

1: Initialise:
   Generate initial sample  $\{T^1, T^2, \dots, T^N\}$  from  $q(T)$ 
   Set initial weights:  $w^i = \frac{p(Y|X, T^i)p(T^i)}{q(T^i)}$ , for  $i = 1, \dots, N$ 
   Calculate AUC of each tree on both the testing and training data
2: for  $i = 1 : \text{iterations}$  do
3:   Normalise weights:  $\tilde{w}^i = \frac{w^i}{\sum_j w^j}$ , for  $i = 1, \dots, N$ 
4:    $N_{eff} = \frac{1}{\sum_i \tilde{w}^i{}^2}$ 
5:   if  $N_{eff} < \frac{N}{2}$  then
6:     Resample to get  $\{\hat{T}^1, \hat{T}^2, \dots, \hat{T}^N\}$ 
7:     Reset weights:  $w^i = \frac{1}{N}$ , for  $i = 1, \dots, N$ 
8:   end if
9:   Sample  $\{\hat{T}^1, \hat{T}^2, \dots, \hat{T}^N\}$  from  $q(\hat{T}|T^i)$ 
10:  Set new weights:  $\hat{w}^i = w^i \frac{L(T^i|\hat{T}^i)}{q(\hat{T}^i|T^i)} \frac{p(Y|X, \hat{T}^i)p(\hat{T}^i)}{p(Y|X, T^i)p(T^i)}$ , for  $i = 1, \dots, N$ 
11:   $w^i = \hat{w}^i$ , for  $i = 1, \dots, N$ 
12:  Calculate AUC of each tree on both the testing and training data
13: end for
14: Calculate result:
    1. For each sample, find the best tree (the highest AUC from the training
       data), across all iterations.
    2. For each sample, get the AUC from the testing data for the best tree.
    3. Return the average AUC across all samples.

```

---

Green text = apply map transformation

Orange text = apply mapPartitions transformation

## A.3 Parameter Tuning

The tables below detail the parameter tuning performed on each of the models.

Decision Tree			
Parameter	Type	Description (from scikit-learn documentation)	Parameter values for cross-validation
criterion	string, optional (default="gini")	The function to measure the quality of a split. Supported criteria are "gini" for the Gini impurity and "entropy" for the information gain.	gini, entropy
max_features	int, float, string or None, optional (default="auto")	The number of features to consider when looking for the best split. If "auto", then $\text{max\_features} = \sqrt{n\_features}$ . If "log2", then $\text{max\_features} = \log_2(n\_features)$ . If None, then $\text{max\_features} = n\_features$ .	auto, log2, None
max_depth	integer or None, optional (default=None)	The maximum depth of the tree. If None, then nodes are expanded until all leaves are pure or until all leaves contain less than <code>min_samples_split</code> samples.	50, 100, None
min_samples_split	integer, optional (default=2)	The minimum number of samples required to split an internal node.	2, 10, 20

Random Forest			
Parameter	Type	Description (from scikit-learn documentation)	Parameter values for cross-validation
n_estimators	integer, optional (default=10)	The number of trees in the forest.	10, 50, 100
max_features	int, float, string or None, optional (default="auto")	The number of features to consider when looking for the best split. If "auto", then $\text{max\_features} = \sqrt{n\_features}$ . If "log2", then $\text{max\_features} = \log_2(n\_features)$ . If None, then $\text{max\_features} = n\_features$ .	auto, log2
max_depth	integer or None, optional (default=None)	The maximum depth of the tree. If None, then nodes are expanded until all leaves are pure or until all leaves contain less than <code>min_samples_split</code> samples.	50, 100, None
min_samples_split	integer, optional (default=2)	The minimum number of samples required to split an internal node.	2, 10, 20

Serial BCART	
Parameter	Parameter values for cross-validation
$\alpha, \beta$	(0.95, 0.5), (0.95, 1.0), (0.95, 1.5)
$\delta, \gamma$	1, 1
minimum split value	0, 10, 20
Iterations	5000
Repeats	500

Parallel BCART	
Parameter	Parameter values for cross-validation
$\alpha, \beta$	(0.95, 0.5), (0.95, 1.0), (0.95, 1.5)
$\delta, \gamma$	1, 1
minimum split value	0, 10, 20
Iterations	1000
No. of samples	512, 1024

# Appendix B

## Code

### B.1 Code Listing

All scripts can be found in the Scripts folder. This also contains a user guide detailing how to execute the scripts.

#### **Data Preprocessing**

These scripts were developed using the IPython Notebook, which have been downloaded as html files to provide an easy to follow demonstration.

Preprocessing Pancreatic Data.html

Preprocessing Twitter Data.html

#### **Decision Tree and Random Forest**

DecisionTree.py

RandomForest.py

#### **Serial BCART**

SerialBCART.py

BayesianTree.py

## Parallel BCART

ParallelBCART.py

Included in the BCART package:

- ParallelBCART\_BayesianTree.py
- ParallelBCART\_initialSample.py
- ParallelBCART\_proposal.py
- ParallelBCART\_evaluate.py
- \_init\_.py (creates the BCART package)

## Parallel BCART with Caching

ParallelBCART\_withCache.py

The key differences to the non-cache script are:

- An additional map transformation is required - `partition_count`. The `resample` transformation needs to know which row of the RDD it is dealing with, to determine how many times it should be resampled. `partition_count` counts the number of rows in each partition. Without this the tuple held in each row of the RDD would have to have an ID number and it would only be efficient to update the ID number in serial by creating a new RDD, as in the non-cache code;
- The Spark configuration includes `.set("spark.executor.heartbeatInterval", "80s");`
- The checkpoint directory must be defined;
- The random numbers are generated within the map transformations. A different random function is required to ensure all workers have a different thread, see below.

```
seed = int(os.urandom(4).encode('hex'), 16)
rs = np.random.RandomState(seed)
```

Note: the cache versions of the BCART package scripts have not been included as the only difference is the random numbers.

## B.2 Code Structure

### B.2.1 Serial BCART

#### Description of the attributes and methods in the BayesainTree.py

Class / Function	Attribute / Method	Name	Description
Node	A	isLeft	Describes whether the node is a left child or right child (for the root note this will be None).
	A	parent	Pointer to the parent of the node (for the root note this will be None).
	A	leftChild	Pointer to the left child of the node (for a terminal node this will be None).
	A	rightChild	Pointer to the right child of the node (for a terminal node this will be None).
	A	depth	Depth of the node, starting with the root node having a depth of 0.
	A	data_indices	Indicates which rows of the training data have filtered into the node.
	A	split_feature	The split feature of the node (for a terminal node this will be None).
	A	split_value	The split value of the node (for a terminal node this will be None).
	A	class_label	Indicates the class label of the node (terminal nodes only).
	A	test_data_indices	Indicates which rows of the testing data have filtered into the node.
Tree	A	nodeID	Unique number. Used in printing the trees.
	A	root_node	The root node of the tree.
	A	terminal_nodes	List of all terminal nodes.
	A	internal_nodes	List of all internal nodes
	A	alpha	Parameter for Psplit.
	A	beta	Parameter for Psplit.
	A	X	The training data.
	A	Y	The training labels.
	A	likelihood	Likelihood of the tree.
	A	qp_acceptance	Part of the acceptance calculation (see explanation below)
	M	calc_likelihood	Calculates the likelihood of a tree. This involves going through each terminal node and counting the labels, so the class label of the terminal nodes are also updated.
	M	grow	Grows a node on the tree. First, it randomly selects one of the terminal nodes, creates the left and right child, updates the terminal_nodes and internal_nodes attributes and calculates qp_acceptance.
	M	create_children	Used by the grow method. Check that it is a valid split (e.g. all terminal nodes have minimum number of instances and, if so, returns to new nodes.
	M	prule	Prule uniformly selects a split feature and split value for a node (used by the create_children and change methods.
	M	split_data	Returns the data indices for a node given its parent's split feature and value (used in the create_children, reset_indices and check_indices methods).
	M	psplit	Psplit calculates the probability of a node splitting (used by the grow and prune methods in calculating qp_acceptance).
	M	prune_list	Checks which nodes can be pruned, e.g. a node those children are both terminal nodes (used by the prune methods and also the grow method to calculate qp_acceptance).
	M	prune	Prunes a node in the tree. It randomly selects a node to prune, updates the terminal_nodes and internal_nodes attributes, the attributes of the nodes being pruned and calculates qp_acceptance.
	M	change	Changes the split value of a node. It randomly selects a node, generates a new split value and reset the data indices in all the nodes below this node. Finally, it checks that this produced a valid split, and if not, reverse the change. If the change is not reversed, the qp_acceptance is set to 1.
	M	reset_indices	Resets the data indices of all nodes below the given node.
	M	swap	Swaps the split feature and value of 2 internal nodes. It selects a pair of node to swap, resets the data indices of all the node below the pair, checks this is a valid split and if not reverse the swap. If the swap is not reversed, the qp_acceptance is set to 1.
	M	implement_swap	Used to implement the swap and reverse it if necessary.
	M	train_auc	Calculates the AUC of the training data.
	M	test_auc	Calculates the AUC of the testing data. First, the test data must be filter down the tree.
	M	filter_test_data	Similar to split_data but for the test data.
	M	set_test_indices	Similar to reset_indices but for the test data.
	M	check_indices	Debugging method to check the data indices are correct.
	M	printTree	Prints out a diagram of the tree using the Graphviz library.
acceptance	n/a	n/a	Calculates the acceptance given the current and candidate trees.

### Explanation of the `qp_acceptance` attribute

The acceptance step of the BCART algorithm (line 6 of the algorithm in Appendix A.1) requires the following calculation:

$$\frac{q(T^i|T^*)p(Y|X, T^*)p(T^*)}{q(T^*|T^i)p(Y|X, T^i)p(T^i)}$$

For the SWAP and CHANGE steps, the structure of the tree is unchanged, hence  $p(T^*) = p(T^i)$ , and we if we have generated the tree  $T^*$  via a SWAP or CHANGE then we can generate  $T^i$  by repeating the same step, hence  $q(T^i|T^*) = q(T^*|T^i)$ . Hence,

$$\frac{q(T^i|T^*)p(T^*)}{q(T^*|T^i)p(T^i)} = 1$$

Similarly, for the GROW and PRUNE steps, many of the terms in the acceptance step cancel such that:

$$\begin{aligned} \frac{q(T^i|T^*)p(T^*)}{q(T^*|T^i)p(T^i)} &= \frac{|T_n^i|}{|T_p^*|} \frac{p_{SPLIT}(d_N, T)}{1 - p_{SPLIT}(d_N, T)} \\ \frac{q(T^i|T^*)p(T^*)}{q(T^*|T^i)p(T^i)} &= \frac{|T_p^i|}{|T_n^*|} \frac{1 - p_{SPLIT}(d_N, T)}{p_{SPLIT}(d_N, T)} \end{aligned}$$

respectively, where  $T_p$  is the set of node which could be pruned,  $T_n$  is the set of terminal nodes and  $N$  is the node either grown or pruned.

To avoid unnecessary calculations, the `qp_acceptance` stores the above expression, to be combined with the tree likelihoods in the acceptance step.



## B.2.2 Parallel BCART

### Description of the attributes and methods in the `ParallelBCART_BayesianTree.py`

Class / Function	Attribute / Method	Name	Description
Node	A	isLeft	Describes whether the node is a left child or right child (for the root node this will be None).
	A	parent	Pointer to the parent of the node (for the root node this will be None).
	A	leftChild	Pointer to the left child of the node (for a terminal node this will be None).
	A	rightChild	Pointer to the right child of the node (for a terminal node this will be None).
	A	depth	Depth of the node, starting with the root node having a depth of 0.
	A	data_indices	Indicates which rows of the training data have filtered into the node.
	A	split_feature	The split feature of the node (for a terminal node this will be None).
	A	split_value	The split value of the node (for a terminal node this will be None).
	A	class_label	Indicates the class label of the node (terminal nodes only).
	A	test_data_indices	Indicates which rows of the testing data have filtered into the node.
	A	terminal_id	Unique number. Used in printing the trees.
Tree	A	root_node	The root node of the tree.
	A	terminal_nodes	List of all terminal nodes.
	A	internal_nodes	List of all internal nodes
	A	alpha	Parameter for Psplit.
	A	beta	Parameter for Psplit.
	A	likelihood	Likelihood of the tree.
	A	qp_acceptance	Part of the acceptance calculation.
	M	printTree	Prints out a diagram of the tree using the Graphviz library.

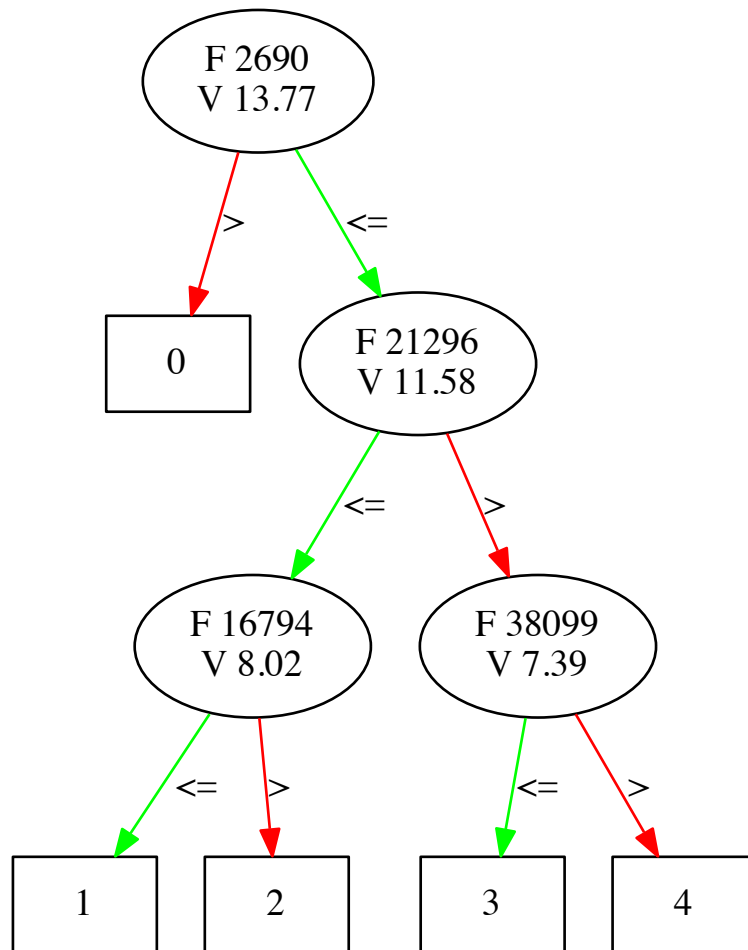
The Node class is only marginally different from the serial BCART scripts. The `nodeID` is replaced by `terminal_id`. Both of these are used when printing out the trees. A slight alteration was needed for the parallel script as it was not possible to generate a unique ID number.

## The `printTree()` Method

This is an example of one of the trees generated by the BCART algorithm which has been converted into a diagram using the `printTree` method. This method was written using the Graphviz library.

The internal nodes are shown with an oval and the terminal nodes a rectangle. The arrows represent the pointer between nodes, with a red arrow to a right child and a green arrow to a left child. The internal nodes detail their split feature and value.

When the RDD is collected it returns an array which includes the Tree objects. The `printTree` method can then be used to generate a diagram of the tree.



## Error on Hartree Cluster

The screen shot below shows the error obtained when trying to run the parallel BCART scripts on the Hartree cluster.

```
of each attempt.
Diagnostics: File does not exist: hdfs://bdb1b19-ib0.dawson.hartree.stfc.ac.uk:8020/user/fxc90-kkr04/.sparkStaging/application_1455017837997_3294/pyspark.zip
java.io.FileNotFoundException: File does not exist: hdfs://bdb1b19-ib0.dawson.hartree.stfc.ac.uk:8020/user/fxc90-kkr04/.sparkStaging/application_1455017837997_3294/pyspark.zip
    at org.apache.hadoop.hdfs.DistributedFileSystem$22.doCall(DistributedFileSystem.java:1309)
    at org.apache.hadoop.hdfs.DistributedFileSystem$22.doCall(DistributedFileSystem.java:1301)
    at org.apache.hadoop.fs.FileSystemLinkResolver.resolve(FsLinkResolver.java:81)
    at org.apache.hadoop.hdfs.DistributedFileSystem.getFileStatus(DistributedFileSystem.java:1301)
    at org.apache.hadoop.yarn.util.FSDownload.copy(FSDownload.java:253)
    at org.apache.hadoop.yarn.util.FSDownload.access$800(FSDownload.java:63)
    at org.apache.hadoop.yarn.util.FSDownload$2.run(FSDownload.java:361)
    at org.apache.hadoop.yarn.util.FSDownload$2.run(FSDownload.java:359)
    at java.security.AccessController.doPrivileged(Native Method)
    at javax.security.auth.Subject.doAs(Subject.java:422)
    at org.apache.hadoop.security.UserGroupInformation.doAs(UserGroupInformation.java:1657)
    at org.apache.hadoop.yarn.util.FSDownload.call(FSDownload.java:358)
    at org.apache.hadoop.yarn.util.FSDownload.call(FSDownload.java:62)
    at java.util.concurrent.FutureTask.run(FutureTask.java:266)
    at java.util.concurrent.Executors$RunnableAdapter.call(Executors.java:511)
    at java.util.concurrent.FutureTask.run(FutureTask.java:266)
    at java.util.concurrent.ThreadPoolExecutor.runWorker(ThreadPoolExecutor.java:1142)
    at java.util.concurrent.ThreadPoolExecutor$Worker.run(ThreadPoolExecutor.java:617)
    at java.lang.Thread.run(Thread.java:745)

Failing this attempt. Failing the application.
ApplicationMaster host: N/A
ApplicationMaster RPC port: -1
queue: default
start time: 1472234212834
Final status: FAILED
Tracking URL: http://bdb1b23-ib0.dawson.hartree.stfc.ac.uk:8088/cluster/app/application_1455017837997_3294
user: fxc90-kkr04
Exception in thread "main" org.apache.spark.SparkException: Application application_1455017837997_3294 finished with failed status
    at org.apache.spark.deploy.yarn.Client.run(Client.scala:855)
    at org.apache.spark.deploy.yarn.Client$.main(Client.scala:881)
    at org.apache.spark.deploy.yarn.Client.main(Client.scala)
    at sun.reflect.NativeMethodAccessorImpl.invoke0(Native Method)
    at sun.reflect.NativeMethodAccessorImpl.invoke(NativeMethodAccessorImpl.java:57)
    at sun.reflect.DelegatingMethodAccessorImpl.invoke(DelegatingMethodAccessorImpl.java:43)
    at java.lang.reflect.Method.invoke(Method.java:606)
    at org.apache.spark.deploy.SparkSubmit$.org$apache$spark$deploy$SparkSubmit$$runMain(SparkSubmit.scala:665)
    at org.apache.spark.deploy.SparkSubmit$.doRunMain$1(SparkSubmit.scala:178)
    at org.apache.spark.deploy.SparkSubmit$.submit(SparkSubmit.scala:193)
    at org.apache.spark.deploy.SparkSubmit$.main(SparkSubmit.scala:112)
    at org.apache.spark.deploy.SparkSubmit.main(SparkSubmit.scala)
16/08/26 18:56:57 INFO util.Utils: Shutdown hook called
16/08/26 18:56:57 INFO util.Utils: Deleting directory /tmp/spark-f17b1e9c-0b32-49ab-b046-dcac156e27b4
```

# Appendix C

## Results

### C.1 Pancreatic Dataset

#### The Data Splits

This table provides a breakdown five data splits.

Data Split	Training		Testing	
	Mild	Severe	Mild	Severe
1	33	41	5	15
2	31	45	7	11
3	28	48	10	8
4	30	46	8	10
5	31	45	7	11

## Speedup

These tables detail the average runtime (in seconds) and standard deviation of 10 runs of the parallel BCART script, as the number of threads increases from 1 to 4.

Number of samples		512							
Number of threads		1		2		3		4	
Step	Measurement	Mean	$\sigma$	Mean	$\sigma$	Mean	$\sigma$	Mean	$\sigma$
	Total runtime	85.82	7.81	69.15	7.12	73.31	6.52	77.13	7.55
	Spark Setup	2.66	0.60	2.31	0.31	2.51	0.31	2.36	0.25
1	Reading the data	6.04	0.41	6.09	0.44	6.03	0.49	6.10	0.37
1	Broadcast the data	0.41	0.05	0.41	0.05	0.42	0.05	0.42	0.06
2	Build RDD with root node trees	0.25	0.02	0.26	0.04	0.26	0.03	0.26	0.03
4	Generate random number for initial sample	0.04	0.01	0.04	0.00	0.04	0.00	0.04	0.00
4	Generate the initial sample	1.61	0.14	1.63	0.15	1.76	0.18	1.90	0.19
5	Calculate the AUC	0.43	0.04	0.29	0.03	0.30	0.03	0.33	0.03
	Initialisation Total	11.45	1.06	11.02	0.78	11.30	0.88	11.42	0.82
6	Sum the weights and calculate N_eff	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
7	Calculate trees to resample	0.01	0.00	0.01	0.00	0.01	0.00	0.01	0.00
8	Resample transformation	0.26	0.03	0.25	0.03	0.27	0.02	0.28	0.02
9	Normalise weights	0.05	0.01	0.05	0.01	0.05	0.01	0.04	0.01
10	Generate random number for proposal step	0.01	0.00	0.01	0.00	0.01	0.00	0.01	0.00
11	Proposal Step	0.13	0.01	0.12	0.02	0.13	0.01	0.15	0.02
12	Calculate the AUC	0.40	0.04	0.25	0.02	0.25	0.03	0.26	0.03
	Average iteration runtime	0.74	0.07	0.58	0.06	0.62	0.06	0.66	0.07
	Total runtime of all iterations	74.37	6.98	58.13	6.48	62.01	5.73	65.70	6.81

Number of samples		4096							
Number of threads		1		2		3		4	
Step	Measurement	Mean	$\sigma$	Mean	$\sigma$	Mean	$\sigma$	Mean	$\sigma$
	Total runtime	594.56	54.71	409.49	25.22	445.78	32.78	376.69	41.03
	Spark Setup	2.49	0.49	2.56	0.56	2.45	0.40	2.41	0.33
1	Reading the data	6.14	0.38	6.15	0.46	6.19	0.27	6.22	0.31
1	Broadcast the data	0.43	0.05	0.41	0.05	0.43	0.03	0.41	0.05
2	Build RDD with root node trees	0.64	0.06	0.62	0.04	0.63	0.04	0.63	0.03
4	Generate random number for initial sample	0.33	0.03	0.33	0.03	0.34	0.02	0.33	0.02
4	Generate the initial sample	2.50	0.21	2.23	0.20	2.56	0.21	2.54	0.20
5	Calculate the AUC	2.89	0.25	1.58	0.19	1.89	0.13	1.32	0.16
	Initialisation Total	15.41	1.19	13.89	1.10	14.48	0.88	13.85	0.52
6	Sum the weights and calculate N_eff	0.02	0.00	0.02	0.00	0.02	0.00	0.02	0.00
7	Calculate trees to resample	0.01	0.00	0.01	0.00	0.01	0.00	0.01	0.00
8	Resample transformation	2.03	0.18	1.78	0.11	1.82	0.14	1.71	0.19
9	Normalise weights	0.38	0.05	0.36	0.05	0.37	0.04	0.37	0.06
10	Generate random number for proposal step	0.03	0.00	0.03	0.00	0.03	0.00	0.03	0.00
11	Proposal Step	0.96	0.09	0.68	0.06	0.73	0.06	0.65	0.08
12	Calculate the AUC	2.92	0.21	1.54	0.08	1.82	0.12	1.30	0.13
	Average iteration runtime	5.79	0.54	3.96	0.25	4.31	0.32	3.63	0.41
	Total runtime of all iterations	579.15	53.81	395.60	24.65	431.30	32.13	362.84	40.61

### Final AUC results

Final AUC results for RF, DT and parallel BCART. The parameter setting are:

**RF** : `n_estimators = 100`, `n_estimators = log2`, `max_depth = None`,  
`min_samples_split = 20`;

**DT** : `criterion = entropy`, `max_features = log2`, `max_depth = 50` ,  
`min_samples_split = 20`;

**Parallel BCART** :  $\beta = 1.5$ , minimum split value = 20

The BCART results are the average of 10 runs. The DT and RF results are the average of 500 runs.

Data Split	Algorithm	AUC	Standard Deviation
1	Random Forest	0.74	0.03
	Decision Tree	0.63	0.12
	Parallel BCART	0.72	0.05
2	Random Forest	0.87	0.03
	Decision Tree	0.67	0.11
	Parallel BCART	0.70	0.08
3	Random Forest	0.51	0.06
	Decision Tree	0.52	0.10
	Parallel BCART	0.47	0.10
4	Random Forest	0.68	0.03
	Decision Tree	0.62	0.10
	Parallel BCART	0.65	0.06
5	Random Forest	0.77	0.04
	Decision Tree	0.65	0.11
	Parallel BCART	0.66	0.04

## C.2 Twitter Dataset

### The Data Splits

This table provides a breakdown five data splits.

Data Split	Training		Testing	
	Pain	Control	Pain	Control
1	1,329	1,351	348	326
2	1,348	1,336	329	341
3	1,356	1,328	321	349
4	1,335	1,349	342	328
5	1,340	1,344	337	333

## Speedup

These tables detail the average runtime (in seconds) and standard deviation of 10 runs of the parallel BCART script, as the number of threads increases from 1 to 4.

Number of samples		512							
Number of threads		1		2		3		4	
Step	Measurement	Mean	$\sigma$	Mean	$\sigma$	Mean	$\sigma$	Mean	$\sigma$
	Total runtime	129.68	7.84	99.15	7.76	97.31	7.94	100.01	7.95
	Spark Setup	2.55	0.50	2.71	0.48	2.51	0.50	3.04	0.50
1	Reading the data	0.63	0.10	0.62	0.07	0.62	0.08	0.68	0.20
1	Broadcast the data	0.37	0.05	0.34	0.03	0.36	0.04	0.37	0.05
2	Build RDD with root node trees	0.34	0.03	0.34	0.03	0.35	0.03	0.34	0.03
4	Generate random number for initial sample	0.08	0.01	0.08	0.01	0.08	0.01	0.08	0.01
4	Generate the initial sample	2.19	0.15	2.06	0.15	2.27	0.41	2.29	0.19
5	Calculate the AUC	0.92	0.07	0.57	0.05	0.52	0.04	0.56	0.07
	Initialisation Total	7.09	0.66	6.71	0.47	6.70	0.85	7.35	0.61
6	Sum the weights and calculate N_eff	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
9	Normalise weights	0.13	0.01	0.12	0.01	0.11	0.01	0.12	0.00
10	Generate random number for proposal step	0.01	0.00	0.01	0.00	0.01	0.00	0.01	0.00
11	Proposal Step	0.45	0.04	0.38	0.04	0.39	0.04	0.39	0.03
12	Calculate the AUC	0.64	0.03	0.41	0.03	0.40	0.03	0.41	0.04
	Average iteration runtime	1.23	0.07	0.92	0.08	0.91	0.07	0.93	0.08
	Total runtime of all iterations	122.59	7.49	92.45	7.53	90.61	7.20	92.66	7.59

Number of samples		1024							
Number of threads		1		2		3		4	
Step	Measurement	Mean	$\sigma$	Mean	$\sigma$	Mean	$\sigma$	Mean	$\sigma$
	Total runtime	262.06	20.79	187.69	19.16	176.95	18.57	178.88	17.47
	Spark Setup	2.74	0.53	2.64	0.60	2.38	0.40	2.44	0.42
1	Reading the data	0.62	0.08	0.62	0.08	0.61	0.08	0.62	0.07
1	Broadcast the data	0.36	0.03	0.36	0.03	0.36	0.03	0.37	0.04
2	Build RDD with root node trees	0.50	0.05	0.45	0.04	0.45	0.03	0.44	0.04
4	Generate random number for initial sample	0.08	0.01	0.11	0.01	0.11	0.01	0.11	0.01
4	Generate the initial sample	2.87	0.20	2.53	0.22	2.58	0.22	2.74	0.26
5	Calculate the AUC	1.72	0.11	1.00	0.09	0.95	0.07	0.93	0.10
	Initialisation Total	8.88	0.75	7.71	0.94	7.43	0.66	7.64	0.64
6	Sum the weights and calculate N_eff	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
9	Normalise weights	0.23	0.02	0.24	0.03	0.22	0.02	0.23	0.02
10	Generate random number for proposal step	0.01	0.00	0.01	0.00	0.01	0.00	0.01	0.00
11	Proposal Step	1.00	0.10	0.79	0.10	0.75	0.09	0.76	0.08
12	Calculate the AUC	1.29	0.08	0.76	0.06	0.71	0.06	0.71	0.07
	Average iteration runtime	2.53	0.20	1.80	0.18	1.70	0.18	1.71	0.17
	Total runtime of all iterations	253.19	20.23	179.98	18.34	169.53	18.03	171.24	17.04



## Parameter Tuning

Results of parameter tuning the pancreatic data.

No of samplers	Data Split	Min Split value	Beta	AUC	Standard Deviation
512	1	0.5	0	0.6097	0.0013
512	1	1	0	0.6115	0.0009
512	1	1.5	0	0.6125	0.0012
512	1	0.5	10	0.6120	0.0021
512	1	1	10	0.6101	0.0013
512	1	1.5	10	0.6097	0.0015
512	1	0.5	20	0.6192	0.0004
512	1	1	20	0.6176	0.0027
512	1	1.5	20	0.6182	0.0004
512	All	0.5	20	0.6218	0.0051
512	All	1.0	20	0.6212	0.0052
512	All	1.5	20	0.6220	0.0046
1024	All	1.5	20	0.6240	0.0049

### Final AUC results

Final AUC results for RF, DT and parallel BCART. The parameter setting are:

**RF** : `n_estimators = 100`, `n_estimators = auto`, `max_depth = 50`,  
`min_samples_split = 2`;

**DT** : `criterion = gini`, `max_features = None`, `max_depth = 100` ,  
`min_samples_split = 20`;

**Parallel BCART** :  $\beta = 1.5$ , minimum split value = 20

The BCART results are the average of 3 runs. The DT and RF results are the average of 500 runs.

Data Split	Algorithm	AUC	Standard Deviation
1	Random Forest	0.95	0.00360
	Decision Tree	0.96	0.00182
	Parallel BCART	0.62	0.00037
2	Random Forest	0.94	0.00404
	Decision Tree	0.96	0.00089
	Parallel BCART	0.63	0.00134
3	Random Forest	0.94	0.00461
	Decision Tree	0.95	0.00200
	Parallel BCART	0.62	0.00145
4	Random Forest	0.92	0.00458
	Decision Tree	0.94	0.00135
	Parallel BCART	0.62	0.00169
5	Random Forest	0.94	0.00410
	Decision Tree	0.97	0.00000
	Parallel BCART	0.63	0.00039

## C.3 Fish Dataset

This table lists the results from running a DT model on the fish dataset.

criterion	max_features	max_depth	min_samples_split	AUC	$\sigma$
gini	log2	None	20	0.62	0.23
gini	log2	100	20	0.62	0.23
gini	auto	None	20	0.75	0.23
gini	log2	50	20	0.64	0.22
entropy	auto	None	20	0.74	0.22
entropy	log2	None	20	0.63	0.22
gini	auto	100	2	0.73	0.22
entropy	auto	100	20	0.74	0.22
entropy	log2	None	2	0.64	0.22
gini	auto	50	20	0.74	0.22
gini	auto	100	20	0.75	0.22
gini	log2	50	2	0.65	0.22
gini	None	None	20	0.81	0.22
gini	None	50	10	0.82	0.22
entropy	auto	None	2	0.75	0.22
entropy	auto	100	10	0.75	0.22
gini	None	50	2	0.82	0.22
gini	auto	50	10	0.75	0.22
entropy	log2	100	20	0.64	0.22
entropy	auto	None	10	0.73	0.22
entropy	auto	50	2	0.73	0.22
gini	auto	50	2	0.73	0.21
entropy	log2	50	20	0.65	0.21
gini	auto	100	10	0.73	0.21
gini	log2	50	10	0.64	0.21
gini	log2	None	10	0.64	0.21
entropy	log2	50	10	0.68	0.21
gini	auto	None	2	0.73	0.21
gini	auto	None	10	0.75	0.21
gini	None	None	2	0.83	0.21
entropy	None	100	2	0.82	0.21
entropy	None	None	10	0.83	0.21
entropy	None	50	2	0.83	0.21
gini	log2	100	2	0.64	0.21
entropy	None	None	2	0.83	0.21
gini	None	100	2	0.83	0.21
entropy	log2	50	2	0.68	0.21
entropy	auto	50	20	0.75	0.21
entropy	log2	100	2	0.66	0.21
gini	log2	100	10	0.65	0.21
gini	log2	None	2	0.64	0.21
entropy	log2	None	10	0.65	0.21
entropy	auto	100	2	0.74	0.21
entropy	None	50	10	0.84	0.21
gini	None	100	10	0.84	0.20
gini	None	50	20	0.83	0.20
entropy	None	100	20	0.84	0.20
entropy	log2	100	10	0.66	0.20
entropy	None	50	20	0.83	0.20
gini	None	None	10	0.83	0.20
entropy	None	100	10	0.84	0.20
entropy	None	None	20	0.84	0.20
entropy	auto	50	10	0.74	0.20
gini	None	100	20	0.85	0.19

## Appendix D

# Learning Points

This demonstrates of the difficulty in using the serial Tree and Node classes in Spark. An RDD of root node trees is created. A map transformation is applied which attempts to prune the trees. We would want the transformation to just return a root node tree (as it cannot be pruned). However, instead it returns 'None' as it reads the `return` command in 5th line of the prune function. See the Scripts folder for html version.

```
In [1]: from __future__ import division
        from BayesianTree import Node, Tree
        from pyspark.sql import SQLContext
        import numpy as np
        import pandas as pd
        import copy

        Read in the data.

In [2]: data = pd.read_csv("/Users/Frankie/Documents/Dissertation/code/BCART/test_data.csv")
        label = data['label'].as_matrix()
        features = data.drop('label', axis=1).as_matrix()

        Broadcast the data.

In [3]: X_train = sc.broadcast(features)
        y_train = sc.broadcast(label)
        min_split = sc.broadcast(0)

        Create the starting tree with just a root node.

In [4]: starting_indices = np.arange(features.shape[0])
        rootNode = Node(data_indices = starting_indices)

        tree = Tree(root_node=rootNode, alpha = 0.95, beta = 0.5, X=features, Y=label)
        tree.calc_likelihood()

        Copy the tree ready to make the RDD.

In [5]: no_of_trees = 8
        forest = []
        for i in range(0, no_of_trees):
            tree_copy = copy.deepcopy(tree)
            forest.append((i, tree_copy))

In [6]: forest_RDD = sc.parallelize(forest)

In [7]: for row in forest_RDD.take(forest_RDD.count()):
        print row

(0, <BayesianTree.Tree object at 0x11326a0d0>)
(1, <BayesianTree.Tree object at 0x11326a2d0>)
(2, <BayesianTree.Tree object at 0x11326a350>)
(3, <BayesianTree.Tree object at 0x11326a3d0>)
(4, <BayesianTree.Tree object at 0x113258e10>)
(5, <BayesianTree.Tree object at 0x113258a10>)
(6, <BayesianTree.Tree object at 0x113258a50>)
(7, <BayesianTree.Tree object at 0x113258ad0>)
```

Now try to prune the tree. The method in the BayesianTree class starts:

```
def prune(self):  
    #find the number of nodes which could be pruned  
    possible_prunes = self.prune_list()  
    no_possible_prunes = len(possible_prunes)  
  
    if no_possible_prunes == 0:  
        return None
```

So, we would expect the pruning to be unsuccessful as a root node cannot be prune.

```
In [8]: forest_RDD = forest_RDD.map(lambda x: (x[0],x[1].prune()))
```

When the resulting RDD is printed, we see that the prune methods has returned None.

```
In [9]: for row in forest_RDD.take(forest_RDD.count()):  
        print row  
  
(0, None)  
(1, None)  
(2, None)  
(3, None)  
(4, None)  
(5, None)  
(6, None)  
(7, None)
```