Parallel Bayesian Additive Regression Trees, using Apache Spark

Sigurdur Geirsson
Abstract

Parallel Bayesian Additive Regression Trees, using Apache Spark

Sigurdur Geirsson

New methods have been developed to find patterns and trends in order to gain knowledge from large datasets in various disciplines, such as bioinformatics, consumer behavior in advertising and weather forecasting.

The goal of many of these new methods is to construct prediction models from the data. Linear regression, which is widely used for analyzing data, is very powerful for detecting simple patterns, but higher complexity requires a more sophisticated solution.

Regression trees split up the problem into numerous parts but they do not generalize well as they tend to have high variance. Ensemble methods, a collection of regression trees, solves that problem by spreading the model over numerous trees.

Ensemble methods such as Random Forest, Gradient Boosted Trees and Bayesian Additive Regression Trees, all have different ways to constructing prediction model from data. Using these models for large datasets are computationally demanding.

The aim of this work is to explore a parallel implementation of Bayesian Additive Regression Trees (BART) using Apache Spark framework. Spark is ideal in this case as it is great for iterative and data intensive jobs.

We show that our parallel implementation is about 35 times faster for a dataset of pig’s genomes. Most of the speed improvement is due to serial code modification that minimizes scanning of the data.

The gain from parallelization is a speedup of 2.2x, gained by using four cores on a quad core system. Measurements on a computer clusters consisting of four computers resulted in a maximum speedup of 2.1x for eight cores.

We should emphasize that these gains are heavily dependent on size of datasets.
## Contents

1 Introduction ................................. 6

2 Regression Trees ......................... 7

3 Ensemble Methods ......................... 9

4 Bayesian Additive Regression Trees ........ 10
   4.1 Algorithm .............................. 11
       4.1.1 Priors .............................. 12
       4.1.2 Posterior (MCMC backfitting) .... 13

5 Spark ........................................ 14

6 Implementation ............................. 16
   6.1 Serial code .............................. 16
       6.1.1 Original software ................. 16
       6.1.2 Profiling ............................ 18
       6.1.3 Code modification ................. 18
       6.1.4 Serial code optimization ......... 19
   6.2 Parallelization ......................... 20
       6.2.1 Background ......................... 20
       6.2.2 Implementation in Spark .......... 20
       6.2.3 Method #1: Find predictors ..... 22
       6.2.4 Method #2: Prediction .......... 24

7 Results ...................................... 27
   7.1 Data set ................................. 27
   7.2 Performance metrics ................... 27
   7.3 Measurements ........................... 28

8 Discussion .................................. 32
1. Introduction

With the arrival of information age, statistical problems have increased in both size and complexity. This has led to development of new methods to data analysis, as the challenge is different from classical statistics. The goal of these methods is to find patterns and trends in order to make sense of data. Large amount of data is being generated by various areas such as finding patterns between genome and disease in bioinformatics, consumer behavior in advertising and weather forecasting.

The objective of supervised learning is to predict value based on number of input measure. The input could be for example a genomic data and the predicted value a disease. The prediction model can be considered as an adapter that converts data into prediction. One simple model is linear regression, which is only useful for basic patterns. In order to allow more flexibility, a regression tree, discussed in chapter 2, can be constructed. The benefit of regression trees is that they split up the problem into smaller more manageable tasks, and they allow for more complicated relationship between variables. The issue with regression trees is that they have high variance, and the prediction model can completely change with minor changes in the data. A prediction model with high variance does not generalize well. One way to counteract this, is to use a collection of regression trees that aggregate to a single predictor in order to reduce variance. These are ensemble methods, discussed in chapter 3, whereas each tree handles a portion of the overall prediction.

The main focus for this thesis is the ensemble method Bayesian Additive Regression Trees, which will be discussed in chapter 4. BART is an iterative backfitting Markov chain Monte Carlo (MCMC) algorithm that generates samples from a posterior. It utilizes various priors that limit the contribution of each regression tree to the overall prediction, limits the tree size and prevents overfitting. In model construction, one of four possible tree modification are done in every iteration.

The aim of the thesis is to parallelize BART using Apache Spark, discussed in chapter 5, open-source computing framework. It is ideal for iterative algorithms as computations are done in memory which increases performance. It also allows for efficient data sharing among slaves, as the data only needs to be sent once to each worker for all iterations. The code and parallel implementation is discussed in chapter 6, and results in chapter 7.
2. **Regression Trees**

Regression trees are used for creating prediction models from data. The feature space, n-dimensional vector for the data, is recursively partitioned into rectangular areas, where each area contains points with similar values. The advantage of splitting up the feature space is that the predictor variable can then be based on any number of variables. Whereas linear regression is limited by just two. The space can also be split up by different values in a particular variable [18]. After partitioning is done, then each area contains a constant prediction value that represents that particular area [18]. The basic terminology of the trees will first be explained, followed by how they are constructed.

**Terminology**

A regression tree consists of a root node, internal nodes and leaf nodes. In the beginning of tree construction, the trees only have one node, the root node. This node represents all the data. Then to grow a tree we must split the data into two parts based on some rule. This rule is based on a predictor and consequently a splitting value. These new nodes are called leaf nodes as they are the bottom nodes in a tree and they contain the prediction. Parent node of the child nodes have a splitting rule which determines how to split up the data by predictor and its splitting value. It is best to showcase this with an example. Following is a simulated sample that shows the general idea behind regression trees. This is a simple regression tree that consists of only two variables.

![Simulated sample showing regression tree based on two variables](image)

Figure 2.1: Simulated sample showing regression tree based on two variables [6]
The regression tree can be seen on the left hand side of the image. Data is recursively partitioned starting with the root node on the X2 variable. Graphical representation of the data points can be seen on the right hand side. Each rectangular area contains a prediction value in the leaf node. Then when we have a new individual and finally can predict what the response value is based on this model.

**Tree construction**

There are different ways to construct the trees, that is how to select the splitting variable and the value. Two very well-known algorithms, CART and C4.5, use impurity reduction for selecting the predictor and its cutpoint [18]. It determines if the prediction model is improving. Another algorithm, AID, used exhaustive search in order to find the split that gives the minimum total impurity score of the child nodes after split. It tries all predictors and cutpoints repeatedly, until a stopping criterion is reached [14]. It is prohibitively expensive to try all options and it does not necessarily give better results. A suboptimal split can lead to a superior split further down the tree.

**Error measure**

The prediction error is usually measured by the squared difference between the observed and predicted values. The total model complexity is shared between the tree structure and the set of node models. The complexity of tree structure decrease with increasing complexity of the node models. Therefore a user can choose the tradeoff. Piecewise constant models such as AID is mainly used for insight as they are easy to interpret. These models tend to have a low prediction accuracy unless it yields a tree with many nodes. It can be harder to get insight from bigger trees and there is also a concern for overfitting [14]. Overfitting means that the sample fits very well to the learned data but the performance is poor because it does not generalize well. Having more variables in the model makes it more adjustable and thus increases the fit of the prediction model [18].

**Disadvantages**

The issue with tree model is that it has high variance, that is a several randomly selected sets of data points could have a completely different tree structure [18]. As a result, the split in the parent node has major effect on subsequent splits in the child nodes. This is a problem as it could lead to big changes to the prediction model, which makes it less reliable.
3. **Ensemble Methods**

Using Ensemble methods is one way to reduce the variance of regression tree model. The idea is to construct multiple predictors, whereas each predictor handles a different part of the prediction. This is done by the sum of tree model which aggregates the prediction across all trees. [4]. The benefit of sum of tree model is that it reduces the variance of the model.

For validating the model, data is split into training and testing data. Then the model is constructed and the training data is used to validate the model in order to test the performance. Following is a summary of common ensemble methods.

**Bagging**

Bootstrap aggregation, or bagging, is a procedure proposed by Breiman [3]. It generates multiple samples from the dataset drawn with replacement. Each sample is then used to construct a tree and the collection of these trees are averaged for prediction. When a node splits the model finds best split possible from all features of the data.

**Random forest**

Random Forest, proposed by Breiman [5], is similar to Bagging with the exception that the best split is found from a subset of features. This means that a fraction of features will be tested until it finds the best one. This effectively reduces correlation between the trees. Then each sample constructs a sum of tree model for aggregated prediction model.

**Boosting**

Boosting is a method, proposed by Freund & Schapire [10], that combines many weak classifiers into a strong one. A weak classifier is slightly better than random guessing but when they are aggregated to form a single predictor, the performance is much better. There are various algorithms that utilize this mechanic such as AdaBoost [10] and Gradient boosted trees [11].
4. Bayesian Additive Regression Trees

Bayesian additive regression trees is an ensemble method proposed by Chipman [7]. It utilizes a backfitting MCMC algorithm that generates samples from a posterior. The basics of algorithm will be first discussed followed by a more detailed discussion of the algorithm.

Probabilities

Posterior is a conditional probability that takes into account some knowledge. A simple example would be picking marbles from a bag one by one. The probability of picking a marble of a certain color either decreases or increases after removing marble from the bag one after another. The P(X|Y) is the probability of X happening if we know that Y happened. The posterior probability is used as weights for averaging the predictions from individual trees.

Prior is the probability without taking into account evidence. The method encompasses of several regularization priors that serve various purposes. One prior regularizes the tree construction to make sure that the individual trees don’t have too much contribution to the final signal being modelled and another prior prevents the trees from growing too big.

Iterations

The method consists of predefined number of iterations and trees. The recommended number of trees is two hundred as proposed by Chipman [7]. The method generally requires over a thousand iterations with two hundred burn-in samples. The model evolves like sculpting a figure from clay, it exposes a residual value being the difference between the prediction model and the dataset.

Variable selection

BART can also be used for variable selection by choosing the variables that appear most often in the models. It tends to get less effective when the number of trees is big as the trees tends to mix many irrelevant predictors with the relevant ones. This is less effective for numerous trees as they tend to utilize many insignificant predictors.
4.1 Algorithm

In this section we will go through steps of the algorithm in an illustrative manner. Below is a pseudo code for the method where each step in the algorithm will be explained. The algorithm has two for loops. The outer for loop is the iteration and inner the regression tree.

**Algorithm 1 BART**

1: Inputs: Training data (X,Y)
2: BART hyperparameters (v,q,k,m,α,β)
3: Initialization: For all j set $Tree_j^{(0)} = \{ ε \}$ and sample $μ_j^{(0)}$
4: for $i = 1$ to max iteration do
5:   Sample error variance
6:   for $j = 1 : m$ do
7:     Compute Residual
8:     MCMC backfitting
9:     Sample leaf parameters

Hyperparameters

There are hyperparameters control various properties of the trees, for example they can dictate how big the trees can get, prevent overfitting etc. They will be discussed where applicable in the coming steps.

Initialization

The root nodes $ε$ are created for all $m$ regression trees. These then sample a leaf parameter $μ_j^{(0)}$ from the Each sample a starting prediction sampled from the signal.

In the initialization the root nodes $ε$ are created for all the trees. In the initialization the is sampled into the tree nodes $ε$ for all the $m$ regression trees. They are sampled in a way that each root node consists of different subset of the dataset.

Compute residual

In this step the residual is computed by traversing through all the trees and aggregate the predictions. The trees are then exposed to a residual response that remains unfitted in the previous iteration. The prediction model is constantly evolving and will on average improve over time. As an error measure the mean square error of the prediction and the real data is often used.
4.1.1 Priors

In the algorithm there are three priors that regulate the tree constructions. They have various purposes such as reducing tree size, prevent overfitting and sample leaf parameter.

Tree structure regularization

The prior enforces shallow tree structures. In sums of trees models, numerous trees can collectively represent a complex model whereas a single tree cannot. These models generally produce a prediction model that is easier to interpret and learn from as opposes to having a single enormous tree. It is also easier to grasp how certain attributes can dictate changes in the signal. By enforcing shallow trees we are effectively reducing the influence of individual trees for the prediction model. For example, if several out of two hundred trees end up being enormous and representing most of the data then the results will be difficult to work with. The rest of the trees would therefore have insignificant contribution to the model.

Tree sizes are dictated by the hypoparameters $\alpha$ and $\beta$ which have recommended values of 0.95 and 2 respectively [7]. The parameters can of course be altered and enforcing bigger trees can be done by changing these parameters if the dataset calls for it. [7] The recommended values have shown to work well with various datasets that were tested. The prior is defined as the probability that a node at depth $d$ is nonterminal with the formula $\alpha(1 + d)^{-\beta}$ and the ranges for the parameters are $\lambda \in (0, 1)$ and $\beta \in [0, \infty)$. The depth of a node is defined as being the number of splits from the root node. The root node has depth 0 and the children of that node have the depth 1 and so on.

Sample leaf parameter

The prior serves the purpose of reducing the contribution of individual trees to the overall fit of the model. It also prevents overfitting of the model for low values of $v$ as it concentrates too much on small error values. It contains the best guess of the parameter for this particular predictor space. The predictor space refers to the area that are developed by splitting of the parent nodes. The leaf parameter is a value drawn from a inverse-gamma $(\nu, \frac{\nu\lambda}{2})$ distribution. Variable $\lambda$ is determined from the data so that there is a $q = 90\%$ chance that the BART model will improve.
Sample error variance

The purpose of sampling from the error variance is to ensure that the model is improving. It is drawn from an inverted Gamma distribution. The default hyperparameters $v$ and $\alpha$ give a 90% chance that the model will improve. Additionally, this also has the effect of preventing overfitting of the model.

4.1.2 Posterior (MCMC backfitting)

The algorithm proposes a change on the current tree, see step 9 in 1, using a Metropolis within Gibbs sampler. It proposes one of four possible changes on the current tree. These proposals are grow, prune, change and swap. Growing a node means that a leaf node is chosen randomly and grown into two child nodes. If this happens then both the variable and the splitting point are chosen randomly. Pruning is the opposite of growing, it collapses two leaf nodes into a single node. Changing is as the name implies an internal node changes its decision rule. Finally, swapping means that two internal nodes swap their decision rules, so the parent node gets the decision rule for the child node and vica versa. All these proposals have different probabilities, grow and prune steps have 25% chance, changing 40% and swapping 10%. It is also possible that the method rejects the proposals and no changes are made in the given iteration. Every iteration can change the number of leaf nodes by 1 or changing one or two decision rules. [7]

The trees can evolve in a way that it grows big and collapse back down to a single node as the algorithm iterates. With each iteration it makes small changes to the trees and with time will improve the fit of the model. When using only a single big tree it can get stuck in a local neighborhood of that tree, meaning that it stops evolving to improvement. [7]
5. Spark

*Apache Spark* is an open source computing platform that offers a simple way to parallelize diverse range of applications. It is especially powerful for data intensive jobs. It consists of various components and offers many common algorithms in machine learning and various statistical methods. Spark Core contains the basic functionality of Spark as it does task scheduling, memory management and fault recovery. Additional components of Spark add features such as streaming, databases, graph processing and machine learning. The benefit of having all these methods in one package is that it makes it easier to parallelize complicated tasks. [20]

**Distributed datasets**

Distributed file system breaks down the data into smaller pieces, or blocks, and distributes them throughout the cluster. An example would be HDFS (Hadoop Distributed File System) [19]. Data is stored with redundancy as data can be recovered in case of failure. It also allows for parallel processing whereas each machine has portion of the data. Spark utilizes resilient distributed datasets or RDDs, which are read only collection of elements that are portioned and can be operated on in parallel.

**Benefits of Spark**

**Computation in memory**

Computations are done in memory which increases performance by an order of magnitude, and as a result the efficiency is higher than some other frameworks. There is a performance penalty in having to reload data from disk with each iteration. Notable tasks that benefit greatly from this are iterative algorithms and interactive data mining tools [20].

**Data sharing**

The dataset is only sent once and used numerous times by the worker, in some other frameworks it had to be sent with every iteration which is expensive. The driver program partitions the datasets into RDDs and workers can do operations on them in parallel. The worker can refer to an individual CPU in a single machine or a whole cluster of computers, depending on the scale of the system. Broadcast variables are used to distribute large read only data between workers. For example, if the workers need to work with a lookup table. This reduces the communication costs as the variable is only sent once to each worker by efficient broadcast algorithms [20].
Driver program launches and broadcasts data to multiple workers. Each worker stores data in memory if there is sufficient space available. Local disk can be utilized if memory requirements exceed available memory. [20].

**Fault tolerance**

RDDs have built-in fault tolerance in case something fails. Given that an individual RDD fails, there is sufficient information to recompute it on demand. This process can be done efficiently in a parallel manner [20]. There are various settings that can be set to control how they operate, persistence is a setting that can ensure the RDD stays alive and can be used for further computations. This is beneficial as the framework does not need to recreate them again. It is also possible to partition these RDDs and form a storage plan. In large clusters with many machines, the data can be split across them by using keys in the data. [20]

**How parallelization works**

As an example of how parallelization is done, let's discuss the MapReduce process. MapReduce is a two-step process that distributes workload among workers with mapping functions that generate a key value pair. Finally, it uses a reduce function to get the final result. As an example, let’s consider we are interested in finding the frequency of certain words in a book. Mapping function distributes the book among workers, and each worker processes a certain range of pages, whereas the word count is computed. Finally, the reduce function aggregates the results to gather word count for the whole book.

**Machine learning library**

Sparks Mlib is a framework that supports many machine learning methods. These methods are made to run in parallel on clusters and are accessible to all the programming languages that Spark supports. The most basic are statistical methods such as correlation, sampling, hypothesis testing, for example. For classification and regression, it supports support vector machines, decision trees, and Bayes classification. It also has functionality for the ensemble methods Random Forest and Gradient-Boosted trees [2].

![Diagram of driver program launching and broadcasting data to multiple workers](image)
6. Implementation

6.1 Serial code

In this section the original code will be explained which entails data structure, algorithm and the parameters used for running it. Then profiling will be done on serial code using a generated sample data. In the last part there will be some discussions about paralysing the code.

6.1.1 Original software

The original software was written for a paper that introduces a new sampling method for BART, so called *Particle Gibbs* filter [13]. The software also contains the sampling method proposed by Chipman [7].

It is written in Python and consists of three files: *bart*, *treemcmc* and *bart_utils*. The main script that handles all the iterative computation is *bart*, and it utilizes functions from *treemcmc* that adds the tree constructing functionality and *bart_utils* that has a collection of various functions that both *bart* or *treemcmc* call upon.

On to the *bart* code. Tree generation involves numerous functions that are inside two for loops. The outer loop being the iteration and inner loop the particular tree. Code is written in an object oriented way, a class instance called *bart* is created that contains all the trees and functions.

The algorithm below shows how functions constructing the trees are related and how the objects evolve over time.

**Algorithm 2 Overview of bart**

1: for i = 1 : max iterations do
2:   sample lambda bart (bart, param, data, settings) → updates param
3:   for i = 1 : tree do
4:   update residual(bart, data) → updates data
5:   update loglik node all(bart.trees[i], data, param, cache, settings) → updates bart.trees
6:   run mcmc single tree(bart.trees[i], settings, data, param, cache, change, mcmc counts, cache tmp, bart.pmcmc objects[i]) → updates bart.trees
7:   sample param(bart.trees[i], settings, param) → updates bart.trees
8:   update pred val(bart, i, data, param, settings) → updates bart
9:   update depth of trees (bart.trees[i]) → updates bart.trees

The input of functions are shown in brackets and the output are shown with an arrow on the right hand
side. Changes are not obvious as the functions update objects that are then passed on. Getting an overview of the changes being made makes it clear how to handle the communication in parallelization. The timings of model creation and prediction are measured separately. This has the benefit of being able to see how the mean square error of the prediction evolves over time without affecting the timing of model creation. If store every is enabled in the parser, then the prediction will be made after each iteration and stored. It always displays for example the mean square error of the testing and training data at runtime. The run mcmc single tree is where most of the work lies as it contains the tree changing step.

Data structure

The following is an overview of the various variables used by the functions and determine which is applicable to use as a broadcast variable.

In the initialization of the program an object called bart is created which contains all the necessary functions for BART. The trees are stored in bart class attribute called trees and is updated every iteration.

The objects data and cache comprises of the data that is being modelled. It should be mentioned that some of the functions only use a small part of the objects at a given time.

The objects mentioned above, bart, data and cache, are by far the largest objects passed on to the functions. Of those only bart updates and grows with each iteration as the trees get bigger. The other ones, data and cache, do not change over time. There is one exception and that is the function update residual stores a temporary variable in data that is used by the function update loglik node all.

The relatively small objects param and settings are inexpensive to pass on to functions. param gets updated in the function bart.sample lambda bart and settings is containing the parameters being used. The variable change is a boolean expression that expresses if changes were made to a given tree.

Parameters

The software can run with various parameters. They are added as script parameter when running the program. The parameters are:

- alpha_s and beta_s are the $\alpha$ and $\beta$ values,
- k_bart, dataset, m_bart is the number of trees,
- n_iter is the number of iterations,
- save is wehter to save the results in a file,
- v is to enable printing for debugging,
- mcmc_type is the mcmc settings, possible values are cgm and growprune

The mcmc type cgm refers to the algorithm proposed by Chipman [7] and growprune as the name expresses only allows grow and prune steps in the mcmc backfitting.
6.1.2 Profiling

For parallelization it is important to have an overview of relative runtimes of the most demanding tasks of the program. Profiling was done in cProfile profiler in PyCharm 2016.1.4 [17] using a generated dataset. It was done on the following system.

MacBook Pro (Retina, 15-inch, Mid 2014)
2.2 GHz Intel Core i7
16 GB 1600 MHz DDR3

The following runtime settings were used –alpha_s 0.95 –beta_s 0.5 –k_bart 2 –dataset friedman-5000 –m_bart 100 –n_iter 100 –save 0 -v 0 –mcmc_type=cgm and using 2000 training ids for each predictor. In the results of profiling it became clear that the most demanding task is run mcmc single tree which has 95.9% of the runtime. It calls upon sample and one of the underlying function Find valid dimensions corresponds a massive 93.3% out of those 95.9%. Genomic data is high dimensional so this is not an unrealistic example. The proportional runtimes are heavily dependent on the size of dataset as the computational costs grows rapidly, especially for the function Find valid dimensions.

6.1.3 Code modification

Before we detail how the parallelization was done, a listing of modification that was done on the code will be listed and it includes changes in timing measurements, loading the dataset and some modification of the algorithm itself.

In the serial code timing were done by utilizing time.clock() to measure time. This is highly inaccurate if it entails parallel operations as can be the case here. It returns the current processor cycles in seconds, as there are multiple processors being utilized this can be misleading. The measurements were changed to time.time() which measures wall time, or elapsed time measured by a stopwatch.

Importing dataset was already implemented but it relied on loading a Pickle dump file. The data processing itself was done in R and it exports four different files. In our case, using our data we have DNA sequence as the raw data and phenotype value as the signal. These need to be split into two parts, one for training the trees and other to test how effective the predictive model is. A function was created, load pigs, to load these files directly without the intermediate step of saving it as a Pickle file. It is basically a way to save objects in Python on disk. The option pigs were added to the parser enabling loading data this way. Furthermore, data path option handles location of the files.

In the proposal step of making a modification of the trees, where the tree can grow, prune, change
or swap the probabilities of change and swap was changed. In the serial code, all these steps have 25% probability of happening. The probability of change was changed from 25% to 40% and swap from 25% to 10% to match the theory in [7]. In the sample of tree construction, a custom non uniform sampling function \texttt{random pick} from Python cookbook [15] was used.

### 6.1.4 Serial code optimization

In this section an optimization of the serial code is proposed which minimizes the data scanning needed. Benefit of this modification is largely dependent on the number of individuals in the sample, ranging from nil for small datasets to considerably for very large ones. The absolute minimum information that is needed to determine a valid predictor is if it contains at least two unique values. After we have a complete list of valid predictors, we randomly select one and find its minimum and maximum values. The proposed modification to the algorithm is detailed in a pseudo code below.

\begin{algorithm}
\caption{Improved \texttt{Get info dimension}}
1: \texttt{Input: data, cache, train ids, settings, predictor}
2: \texttt{if length of training ids is less than two}
3: \hspace{1cm} \texttt{return -1}
4: \texttt{IndexA} $\leftarrow$ index for first training id
5: \texttt{for all others training ids}
6: \hspace{1cm} \texttt{IndexB} $\leftarrow$ index for training id
7: \hspace{1cm} \texttt{If} \texttt{IndexA is not equal to IndexB}
8: \hspace{1.5cm} \texttt{return predictor}
9: \texttt{return -1}
\end{algorithm}

As mentioned above, this reduces the amount of scanning needed and could theoretically make a significant impact on the runtime depending on the data being processed. The serial code processes this in a different manner. It finds the minimum and maximum values for all predictors in determination of valid ones which is unnecessary. The function in the code that handles this process is \texttt{get info dimension}. There are some modifications of the function \texttt{find valid dimension} that need to be done, for instance as it does no longer has minimum and maximum values for all predictors it must be recomputed for the chosen predictor.

As the software generates a unique sorted list of all predictors it can select a splitting point fast. If minimum and maximum values are known, we simply find their index value in unique sorted list for the predictor to select a splitting point from that. An example would be if possible values in a predictor is 0-9 and we find minimum and maximum value is 4 and 7. These values are looked up in a unique sorted list
from 0,1,2..9 and we get the sublist 4,5,6,7. Then it is pretty clear that there are only two possible splits 5 and 6 which can be chosen. The parallelization of modified serial code is done in a similar manner as explained in Method #1 (See 6.2.3).

6.2 Parallelization

For parallelization the goal is to efficiently distribute the workload between workers. It is important to reduce communication costs and the number of barriers. In the next section a short overview of previous work will be detailed followed by the implementation in Spark.

6.2.1 Background

There are few parallel implementations of BART written in both R and C using MPI. Following is a short summary of those implementations.

Implementation done by Pratola [16] was done in C++ using MPI. There were several modifications done to the main algorithm. First, it only allows for growing and pruning steps in the MCMC sampler. Method works in a similar way but the reason for having the other steps, change and swap, is that it can improve the exploration of model space. They argue that this grow and prune steps are sufficient as the trees are small and therefore easily explored.

The main idea behind parallelization is that the data is parallelized. The data is partitioned among workers and each worker returns a partial residual that is summed up. As the method iterates, whenever a tree modification is proposed and accepted then the changes are propagated across all workers. As data is not shared then a minimal model was used to represent the regression trees. In this simplified model computations are often not stored but rather recomputed when needed. The speedup they get is almost linear for the 48 processor cores for a dataset of size 200,000x40 using 20,000 MCMC iterations. For further detail see [16].

Another approach is to parallelize the Gibbs sampler like is done in bartMachine [12] R package. The implementation allows for three steps: grow, prune and change. In the parallelization, one independent Gibbs chain is created per core. The downside is that the burn-in samples are not parallelized. For example, having 250 burn-in and 1000 post burn-in samples and four workers. Each worker would process 250 burn-in and 250 post burn-in samples. The end result is approximately 30% speed-up for model creation when using four cores instead of one.

6.2.2 Implementation in Spark

The parallel implementation was done using Apache Spark 1.62 framework in Python 2.69. Based on previous work the way to parallelize will favor the Pratola implementation. There are notable differences
though as the current code is expensive memory wise. Here, there will be emphasis on parallelizing the data processing without partitioning the data across workers. This is not necessary in *Apache Spark* as all the data can be efficiently shared among all the workers. Variables `data` and `cache` can be used as a broadcast variable to minimize communication and improve performance. There is one thing to be kept in mind, there is a small subset of `data` is updated in every iteration and that is the column `y-train`, the current signal to be modelled. Special provision must be provided if parallelized need that particular part of the data, which was not determined to be necessary in this case.

**Data processing parallelization**

As has been mentioned before in 6.2.1 the idea is to parallelize the data processing of the method. There were three different parts that were attempted. The serial part of parallelized tasks will be kept to enable or disabling parallelization based on the dataset being used. The effectiveness of parallelization is largely dependent by the datasets.

In [16] the parallelization is done by data partitioning. Each worker works with part of the data, if we have 1000 training ids and 10 workers, then each worker would only process 100 training ids and compute the partial residual value for it. There is no need to split the data this way using *Spark* as it has efficient shared memory. All workers have read only access to the data by using broadcast variables. Before looping occurs both `data` and `cache` is broadcasted to all the workers.

**Broadcast variables**

Broadcast variable are created with `databc = sc.broadcast(data)` and is then passed on as a input variable to functions and its content accessed inside the parallel function with the `.value` extension. Broadcast variables used in this implementation are `data`, `cache` and `lists`.

As was discussed in 6.1.1 the `data` and `cache` consists of the data that is to be processed. `Lists` is a custom variable created to minimize the communication regarding parallelizing the data by rows. It is utilized by parallelization of the prediction (See 6.2.4). The general idea is in order to partition the data by rows we must let each worker process a sublist of all training ids. Instead of sending the list to process every time we split a list of training ids into partitions and send those lists to all workers. Then the workers can easily access the sublists by an integer. Example for this would be: if we have 1000 training ids that must be processed by 10 workers. Then each worker would need to process 100 and the first one 0,1,2...99. Instead we simply split the list of 1000 into 10 lists and send the worker list integer 0 to access the sublist 0,1,2..99.
Parallel operations

Furthermore, cache() extension was used on all parallel operations as then they don’t have to be recreated all the time. This is important as it takes resources to recreate workers, and it also ensures that the broadcasted data is only sent once to the worker. This is crucial in iterative jobs as it minimizes communication costs.

In the following subsections, the parallelization that was done will be explained.

6.2.3 Method #1: Find predictors

The particular serial code being parallelized here, was optimized. First, the parallelization of original code will be discussed. Followed by parallelization of the improved serial code.

Original serial implementation

The first data processing function to be parallelizes is Find Valid Dimensions. Of all the methods to be parallelized this is the most important one as its computational cost grows fastest with larger datasets. It is called upon by three steps in the tree modification proposals: grow, change and swap. The function sample split prior calls upon it in the grow and change steps. In swap, recompute prob split uses it. The purpose of this function is as the name suggests to find valid predictors to split on. First the serial code implementation will be explained and then what changes were made to parallelize it.

The serial code is named find valid dimensions serial in the code and it has the input variables data, cache, train ids and settings. It runs through all the predictors of the data and finds the corresponding minimum and maximum values for the training ids in that particular node. As was explained in previous section, training ids is basically the row vectors of the data whereas predictor corresponds to column vectors. As the nodes have different subset of training ids the minimum and maximum values of the predictors are constantly changing. This expensive operation of scanning the data must therefore be done in every iteration. The reason is that it is not possible to split an array with a single unique value. This is actually more probably with genetic data as it is often represented by a binary format. This increases the chance of having no valid splits by a great deal.
**Algorithm 4** Overview of Find Valid Dimension Serial Code

1: Find valid dimensions (Input: data, cache, train ids, settings)

2: score feat ← copy from cache

3: for predictor = 1 : number of predictors do

4: xmin, xmax, idxmin, idxmax ← get info dimension(data, cache, train ids, settings, predictor)

5: if idxmin is equal to idxmax do

6: score feat[predictor] = 0

7: else

8: Feat split info = [idxmin, idxmax, xmin xmax, featscore]

9: Valid predictors ← all predictors with score feat > 0

10: Split not supported ← True if no valid predictors

11: Return: Valid predictors, Score Feat, Feat split info, Split not supported

The variable score feat contains an array of precomputed probability of a given predictor. In the case where it is not possible then the score feat for that predictor is set to zero. The function iterates through all the predictors and find its minimum and maximum values. In the precomputation there is some things done to speed up computations. For all predictors the unique values are sorted, this means that for binary dataset there are only two values stored and one possible split. The index value can be looked up in cache converting a value to an index. This can be for example done by the command:

```python
idxmin = cache[feat_val2idx][feat_id][xmin]
```

This gets the precomputed index of the sorted unique list. This is frequently returned by the get info dimension function and consequently the indexes are compared to check if they are the same or not. The probability of splits in predictors is saved in score feat. The minimum, maximum and their given index values are saved in a variable called feat split info.

It is important for parallelization to minimize communication. As was mention above the inputs are data, cache, train ids and settings. Obviously, we would like to broadcast data and cache and store the RDDs by cache() extension on the parallel directive. This means that we will send these variables to all workers only once for all iterations. Another approach that is taken is to split training ids into equal sized lists and broadcast them to the workers. Then it not necessary to send the lists but instead just an integer to access the sublists. This process is done by the function split list and number of lists are based on set partitions for parallelization. These sublists are parallelized by c. parallelize (lists,settings,partitions) and mapped to featvalid which is a modified version of find valid dimension serial (See 6.2.3) which only returns valid predictors.
The obvious thing is to broadcast data and cache and store the RDDs by using the cache() extension on the parallel directive. This means that these variables will only be sent once to the workers for all the iterations. Training ids is constantly evolving and needs to be used as a regular input variable. The minimal amount of information that the function needs to return is just the valid predictors. Other variables can easily be recomputed again when a predictor has been chosen.

These lists are parallelized by c. parallelize (lists,settings,partitions) and mapped to featvalid which is a modified version of find valid dimension serial (See 6.2.3) which only returns valid predictors.

These lists are parallelized by

```
feat id valid2 = i t .flatMap( lambda i t : self . featvalid ( data bc , cache bc , train ids ,
    settings , i t ) ) .cache()
```

Then we must use collect() extension on the feat id valid2 RDD to get the valid predictors. In order to get the necessary information to sample a splitting point, the predictor chosen must be scanned again for minimum and maximum values. This gives us information on exactly which splitting points to sample from.

**Improved serial implementation**

The parallelization of improved serial implementation is done in a similar manner. There was one altercation done on the improved Get info dimension function (See 6.1.4). Instead, we return the predictors which are not possible to split. This should improve runtime of the code as it is significantly less likely than a valid one. Then a list of valid predictors is simply generated by the following command.

```
feat_id_valid = set(cachebc.value[range_n_dim]) - set(feat_id_notvalid)
```

### 6.2.4 Method #2: Prediction

The second method parallelizes computation of new prediction after tree modification step (See 11. in 6.1.1). As was mentioned in BART algorithm 4, the leaf nodes contain the best guess of a response. The general idea behind this method is to find and sum up these response values to get the current prediction. In order to get the response value, each individual must traverse through the trees based on splitting rules in order to determine which leaf node it ends up in. These leaf nodes contain the best guess of the response. Current prediction for each individual is thus a sum of these best guess values for all trees. This is a relatively fast process as the trees are generally small. The prediction can then be compared to the real value which we are modelling after. Then we sum up the prediction values for all those leaf nodes and compare it to the phenotype value.

An example of this would be, let’s say we have p trees. Then each individual must gather and sum up the prediction for the leaf nodes in those p trees. This is a relatively fast computation but it may be effective
for datasets which are very numerous.

As the training ids are independent, this process can be parallelized. The gathering of node ids is parallelized per individual as they traverse down the trees. These node ids can then easily be converted to prediction values by list comprehension. This is the general idea behind the parallelization, implementation of it will be further explained below.

The function being parallelized is \texttt{update pred val} (See 11. in 6.1.1). The broadcasted variable used by this function are \texttt{data} and \texttt{lists}. The first step is parallelizing the lists by

\begin{verbatim}
it = sc.parallelize(range(settings.partitions))
\end{verbatim}

The partitions settings here, determines into how many lists the whole range of training ids are split into. At minimal we need as many as the number of workers but having too many increases the communications between workers.

Function being parallelized is \texttt{traverse parallelbc} and it returns tuples of training id and it corresponding leaf node id. The minimal communication we need to the workers is \texttt{data}, \texttt{lists}, \texttt{integer} and the current tree as input. The \texttt{data} and \texttt{lists} are broadcast variables so they are only sent once. The integer corresponds to the range of training ids to process.

\begin{verbatim}
node ids = it.flatMap(lambda it: traverse parallelbc(databc, it, trees tmp, listsbc)).
cache()
node ids = node ids.values()
node ids = node ids.collect()
\end{verbatim}

Then the prediction for leaf nodes are gathered by list comprehension. The trees are constantly changing so we must send them to the worker in every iteration.

\textbf{Method #3: Compute statistics after split}

The third method to parallelize is regarding the computation of residual values after split occurs in the grow phase of the method. This is actually one of the parts that is parallelized in the Pratola implementation [16]. It is important to note that they only allow grow and prune steps in the Gibbs sampler. Then there is 50\% chance of this step taking place as opposed to 25\% when all steps are allowed. As their parallelization is based on data partitioning, this process must be parallelized. In the data processing parallelization, we can choose which processes are worth parallelizing. This process becomes increasingly insignificant with larger datasets. Preliminary timing measurements indicates that this parallelization is ineffective for all datasets. The full implementation will not be discusses further as it turned out to be ineffective. For interested, see function \texttt{compute left right statistics parallel} in the code.
Validation of methods

In order to test the correctness of the methods the variable $\beta$ used in sampling for error variance was used for comparison. This variable is determined from the mean square error of the prediction and it is directly dependent on the tree structure. The trees must be the same in order to produce the exact same value. This is the case for improved serial code (6.1.4), Method #1 (6.2.3) and Method #2 (6.2.4). Method #3 produces some minor error but that parallelization was deemed to be ineffective as it is slower in all cases. We will only consider the first two methods in our implementation.

Regulation of methods

It is important to note that the effectiveness of methods depend on the dataset being used. Considering the data is two dimensional, the columns being predictors and rows individuals.

The first method is especially beneficial for genetic data where the predictors or markers are numerous. In prediction, the parallelization is done on individuals which are all independent in the data. That is the rows are independent in prediction. To allow for flexibility there are options to enable or disable these parallelization methods as well as set the number of partitions to use. This is done by adding the following script parameters to the run time of the code

-`partitions=4 -p fvp =1 -p upv=0`

Partitions is self-explanatory as it determines into how many parts a particular job should be divided for. The acronyms `fvp` is find valid predictor method and `upv` for prediction parallelization
7. Results

This section is split into three parts. First, we discuss the dataset being used, then how effective the implementation is and then finally we do time measurements for the most optimal settings for the dataset being tested. These measurements will be done on both real and virtual cores to see how well it scales.

7.1 Data set

A genetic dataset was used, which consists of phenotypes for 3534 pigs [8]. In this case, we are constructing a model that predicts one of the phenotype based on the genetic makeup of the pigs. After cleaning up the data in R for missing values for the third phenotype, we end up with 3141 pigs which is further split into training (75%) and testing (25%) data. Rows of the data represent pigs while columns represent the predictor or marker. Each pig has 50276 markers which are represented by the digits 0, 1 and 2. There is therefore only two possible cutting points.

7.2 Performance metrics

Speedup refers to the relative performance gain is achieved by parallelized implementation. It is defined as the ratio of execution time of serial code divided by parallelized implementation. This basically tells us how much faster it is after parallelization. The efficiency is defined as speedup divided by number of cores used in the parallelization.

\[
\text{Speedup} = \frac{T_{\text{serial}}}{T_{\text{parallel}}}
\]

Speedup basically tells us how much faster it is after parallelization. The efficiency is defined as speedup divided by number of cores used in the parallelization.

\[
\text{Efficiency} = \frac{\text{Speedup}}{\text{no.of cores}}
\]

Amdahl’s law is useful to estimate theoretical speedups of parallel programs [1]. It is defined by the following fraction

\[
\text{Amdahl} = \frac{1}{r_s + \frac{r_p}{n}}
\]

In this formulation \(r_s\) and \(r_p\) represent the ratio of serial and parallel port of the program. It is important to note that the parallel code is limited by the serial code. For example, if serial part of code is 10% then the speedup cannot be more than 10. Regardless how many cores are used.


## 7.3 Measurements

Measurements were done on the pig’s dataset (See 7.1) on the system shown in the Profiling section 6.1.2 unless otherwise specified. In all cases the method allows for all four steps: grow, prune, change and swap. On measurements using real cores, the job is split into as many parts as there are workers. In other words, if there are 4 workers then the job is split into four parts. This gave best preliminary results as it reduces the parallel overhead as the workload is evenly distributed.

### Improved serial code

Optimization was done on the serial code to minimize scanning needed to find valid predictors (See 6.1.4). This drastically reduces the work depending on the dataset being modelled as this is the most demanding task of the software for datasets of this size. Total execution time was compared by modelling with 50 trees and iterations.

<table>
<thead>
<tr>
<th></th>
<th>Execution time (sec)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Serial code</td>
<td>21024</td>
</tr>
<tr>
<td>Improved Serial Code</td>
<td>1310</td>
</tr>
</tbody>
</table>

Table 7.1: Comparison of total execution time, with 50 trees and iterations, for serial code before and after optimization.

The is a considerable change for this dataset as it is approximately 16 times faster. The effectiveness is depending on the dataset as it becomes better with more rows. This also has the effect of reducing the effectiveness of parallelization Method #1 (See 6.2.3) as the process is not as dependent on rows of the dataset. As will be shown below (See 7.3) the function still has a significant 92% part of total execution time. This optimization will be enabled in all further measurements.
Parallel methods

The purpose of this section is to investigate how effective the parallelization is on the dataset being used. As we are interested in processing real genetic data, we run the software and do a trial run to see how effective these methods are. The parallelization was done using four cores with the job split into four parts.

<table>
<thead>
<tr>
<th>Parallelized Method</th>
<th>#1</th>
<th>#2</th>
<th>Execution time (sec)</th>
</tr>
</thead>
<tbody>
<tr>
<td>×</td>
<td>×</td>
<td>443</td>
<td></td>
</tr>
<tr>
<td>✓</td>
<td>×</td>
<td>240</td>
<td></td>
</tr>
<tr>
<td>×</td>
<td>✓</td>
<td>3592</td>
<td></td>
</tr>
</tbody>
</table>

Table 7.2: Some preliminary measurements to test effectiveness of methods on the dataset. The table shows the total execution time with either parallel methods enabled or both disabled. The checkmark symbol (✓) indicates that the method is enabled and other symbol (×) indicates it is disabled. It is clear that only Method #1 is effective.

It is clear from these preliminary measurements that Method #1 is only effective for this dataset. It was expected as Method #2 would require far more individuals in order to be effective, if at all. It is expensive to send the trees repeatedly in every iteration. The parallel overhead is just too high in this case. It is important to note that the effectiveness of parallel methods is largely dependent on the size of the dataset and it should perform better with bigger datasets in Spark. Next, we investigate how well it scales with both real and virtual cores.
Real cores

In these measurements we compare the total execution time of construction a 100 tree model with 200 iterations. In the serial runtime the parallelized portion consists of 91% of run time. According to Amdahl’s law we can expect maximum speedup of 11. Following is the results from executing the code on system specified in profiling section (See 6.1.2). The serial code took approximately 4.2 hours to run.

<table>
<thead>
<tr>
<th>CPU</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>8</th>
</tr>
</thead>
<tbody>
<tr>
<td>Speedup</td>
<td>1</td>
<td>1.6</td>
<td>2</td>
<td>2.2</td>
<td>2</td>
<td>2.1</td>
<td>1.8</td>
</tr>
<tr>
<td>Efficiency</td>
<td>1</td>
<td>0.82</td>
<td>0.67</td>
<td>0.55</td>
<td>0.41</td>
<td>0.35</td>
<td>0.23</td>
</tr>
</tbody>
</table>

Figure 7.1: Speedup measurements for construction of 100 tree model with 200 iterations on a quad core machine specified in Profiling section. 6.1.2.

Here we get a decent speedup of 2.2x for four cores. According to Amdahl’s law the best possible speedup is 3.15x which is not realistic as it does not take into account communication costs and parallel overhead. The computer has four processor cores and can operate with four additional virtual cores by using hyper-threading which can improve performance in some cases. This is mostly for processes where the processors have to wait for the main memory bus, while it is waiting the other virtual core can execute code. It is clear that the algorithm does not benefit from hyperthreading as the code is constrained by computation. Having more workers than real cores only adds communication costs without any tangible benefit, and leads only to decreased performance.
Cluster cores

In this part we do the same measurements on a computer cluster to investigate the speedup on a larger, more powerful system. We compare the total execution time of construction a 100 tree model with 200 iterations. The serial runtime takes 3.8 hours and the parallelized portion is 87% of the total runtime.

The computer cluster is running in HPC2N region in SNIC Science Cloud [9]. It consists of 4 computers each having the specification: Intel Xeon E312xx 2.3GHz and 16 GB memory. There is no sharing of resources so it can be considered as a virtualized dedicated environment.

Following is the results from running the software on the system.

![Speedup measurements for construction of 100 tree model with 200 iterations on a computer cluster.](image)

<table>
<thead>
<tr>
<th>CPU</th>
<th>1</th>
<th>2</th>
<th>4</th>
<th>6</th>
<th>8</th>
<th>12</th>
<th>16</th>
</tr>
</thead>
<tbody>
<tr>
<td>Speedup</td>
<td>1</td>
<td>1.4</td>
<td>1.8</td>
<td>2</td>
<td>2.1</td>
<td>2.1</td>
<td>2</td>
</tr>
<tr>
<td>Efficiency</td>
<td>1</td>
<td>0.71</td>
<td>0.45</td>
<td>0.33</td>
<td>0.26</td>
<td>0.17</td>
<td>0.12</td>
</tr>
</tbody>
</table>

Here we get the maximum speedup of 2.1x for eight cores. Having additional cores only reduces the performance. The lower speedup, compared to real cores, is probably a combination of two factors. Added communication costs as the transmission now runs across computers and lower threshold by Amdahl’s law due to faster run time of the parallel code. Communication also weighs more as workload per worker decreases. Consequently, effectiveness of parallelization for additional worker is reduced.
8. Discussion

The most significant modification is the serial improvement of the code, which reduces workload considerably as the scanning of data is minimized. In our measurements, the result is a code that is 16 times faster for the pig’s genome dataset. Effectiveness is largely dependent on the number of rows the dataset has, as after modification it only scans until it finds two unique values for each predictor. Consequently, this reduces the effectiveness of first parallel method as the workload is much less. The discovery of this improvement was made at the end of the process, which changed results greatly.

The first parallel method is effective with speedup of 2.2 for four cores by parallelizing find valid predictors, in this case out of 50276 predictors. The second parallel method was not effective as the computations are too fast for only 2355 individuals. As is the case for all methods, the effectiveness is largely dependent on the size of data. The end result, a code that is approximately 35 times faster by using four cores with first parallel method enabled. The algorithm is tricky to parallelize as it consists of series of fast computations. Effectiveness would certainly be better for larger datasets.

The run time of the code can be improved by approximately 10-15% by modifying parallel method #1. Instead of having the function returning not valid predictors, return a random valid predictor. Consequently, the master can then randomly select one returned from the workers. This eliminates the need to generate the list of valid predictors that can add a few seconds to each iteration. This has the effect of changing the random number generator so the verification of results has to be done in some other way than comparing the predictions.

It is difficult to get a fair comparison for other parallel BART implementations. Pratola implementation [16], allows grow and prune steps, and bartmachine, allows grow, prune and change steps. While our implementation uses all four steps grow, prune, change and swap. The parallel schema of Pratola implementation [16] is probably superior as it minimizes communication costs, the cost of sending the tree structure is less than training ids. It also encapsulates a larger part of the process than our implementation. It is not possible to adapt this schema without completely rewriting the code. In terms of speed, Pratola is probably faster due to it using, C++, a faster programming language. Direct comparison of speedup is not feasible as the Pratola implementation does the same computation repeatedly, while ours does no computation twice.

The benefit of using Spark in Python is reliability due to fault tolerance and ease of use. Implementation in Spark works on a wide range of systems as it handles all communication. It is easy to add workers to shared resources, which can be processor on a machine or cluster of computers.
Acknowledgement

I would like to thank my supervisor Behrang Mahjani for his patience, motivation and assistance with the work of my thesis project for an extended period of time. I would also like to thank Salman Toor for the help with the computer cluster and Patrik Waldmann with the help in understanding Bayesian Additive Regression Trees. Finally, I thank the IT department of Uppsala University for all the support.
Bibliography


