University of Cape Town

Small-Scale Distributed Machine Learning in R

Student:
Brenden Taylor
TYLBRE007

Supervisor:
Mr Stefan S Britz
Co-supervisor:
Dr Etienne Pienaar

A dissertation for the degree of Master of Science specialising in Data Science

at the

Department of Statistical Sciences

June 26, 2022

Abstract

Machine learning is increasing in popularity, both in applied and theoretical statistical fields. Machine learning models generally require large amounts of data to train and thus are computationally expensive, both in the absolute sense of actual compute time, and in the relative sense of the numerical complexity of the underlying calculations. Particularly for students of machine learning, appropriate computing power can be difficult to come by. Distributed machine learning, which involves sending tasks to a network of attached computers, can offer users access to significantly more computing power than otherwise by leveraging more processors than in a single computer.

This research outlines the core concepts of distributed computing and provides brief outlines of the more common approaches to parallel and distributed computing in R, with reference to the specific algorithms and aspects of machine learning that are investigated. One particular parallel backend, `doRedis`, offers particular advantages as it is easy to set up and implement, and allows for the elastic attaching and detaching of computers from a distributed network. This paper will describe core features of the `doRedis` package and show, by means of applying certain aspects of the machine learning process, that it is both viable and beneficial to distribute these machine learning aspects.

There is the potential for significant time savings when distributing machine learning model training. Particularly for students, the time required for setting up of a distributed network in which to use `doRedis` is far outweighed by the benefits. The implication that this research aims to explore, is that students will be able to leverage the many computers often available in computer labs to train more complex machine learning models in less time than they would otherwise be able to when using the built-in parallel packages that are already common in R. In fact, certain machine learning packages that already parallelise model training can be distributed to a network of computers, thereby further increasing the gains realised by parallelisation. In this way, more complex machine learning is more accessible.

This research outlines the benefits that lie in the distribution of machine learning problems in an accessible, small-scale environment. This small-scale 'proof of concept' performs well enough to be viable for students, while also creating a bridge, and introducing the knowledge required, to deploy large-scale distribution of machine learning problems.

Contents

Chapter 1

Introduction

1.1 Background

For the last four years, Stanford University's Institute for Human-Centered Artificial Intelligence (AI) has published its AI Index Report which aims to "provide unbiased, rigorously vetted, and globally sourced data for policymakers, researchers, executives, journalists, and the general public to develop intuitions about the complex field of AI" (Zhang et al., 2021). The report indicates two findings that are applicable here. Firstly, that the percentage of North American students graduating in 2019 with PhDs in AI who went into industry increased by a little over twenty percentage points from 2010, and secondly that the number of AI-based publications on arXiv, an open-access repository for scholarly publications, has grown by more than six times in as many years (Zhang et al., 2021). Figure 1.1 below shows that the relative importance of AI, at least in the academic space, has been increasing over the last two decades (and fairly substantially in the last few years).

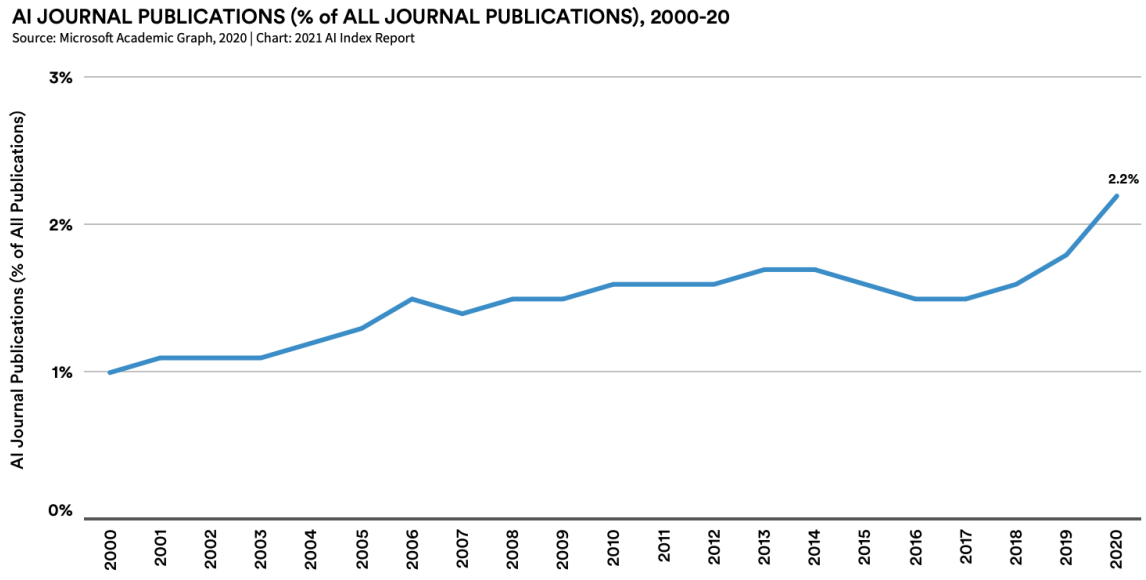

Figure 1.1: The percentage of all journal publications that are AI-focused since 2000 (Zhang et al., 2021).

The two points above relate specifically to AI. However, machine learning (ML), as a subset of AI, has seen substantial growth in the last few years. In industry, AI jobs related to ML saw the "fastest growth in online AI job postings in the United States" (Zhang et al., 2021). Zhang et al. (2021) also found that ML had the second fastest growth in number of publications on arXiv among the AI-related fields of study, increasing tenfold between 2015 and 2020 (second only to the number of robotics publications, which had increased by eleven times in

the same time frame). Clearly then, ML is a growing field in both the business environment as well as the academic environment, and a potential driving factor for this growth, at least in the business context, is the increasing quantity of data that companies are collecting.

With the advancement of the global village, data are being more and more readily collected and analysed. Naturally, modelling large datasets can be quite computationally expensive. This is possible for medium- to large-sized businesses and organisations, who can deploy the training of their models and algorithms on remote clusters (as discussed in Section 2.2), but is more difficult and inaccessible for smaller organisations and individuals, particularly students, who do not have access to the same resources as their more well-funded counterparts. A distributed statistical computing cluster in R is a potential solution to this problem and would allow greater access to more advanced machine learning techniques by allowing users to leverage a number of otherwise 'dormant' computers to run calculations simultaneously.

Distributed computing has already proven its worth in the scientific community. For example, the Folding@Home project became the world's first exaflop computing system during the Covid-19 pandemic when the project saw a twenty-fold increase in traffic as more users lent their computers to process protein folding calculations that were aimed at aiding the discovery of a Covid-19 therapy (Patrizio, 2020). A distributed statistical computing cluster that is accessible to all would therefore allow its users access to significantly more computing power than otherwise.

`doRedis` (Lewis, n.d.) provides a parallel backend (similar to packages like `doParallel`) for the `foreach` function in R by sending a queue of tasks ("jobs") to a Redis server. Other computers can then retrieve jobs from the Redis server, compute them, and send the results back to the Redis server. In this way, `doRedis` allows for distributed statistical computing in R and this can be leveraged to allow for faster computation of complex machine learning techniques. Additionally, Redis servers offer key advantages over the currently popular parallel backends available for R. These include fault tolerance and the ability to dynamically add or remove machines from the cluster without a hitch (Lewis, 2021b).

Many machine learning algorithms are parallelisable problems, however many of the popular machine learning packages in R are not currently able to leverage the `doRedis` parallel backend effectively. This research will attempt to demonstrate 'proof of concept' by making use of the `doRedis` package to distribute machine learning problems.

1.2 Scope and Outline of the Research

This research aims to answer a few questions about distributed machine learning in R, and its feasibility on a small-scale.

Firstly, what are parallel and distributed computing systems? The research will begin with an outline of the two topics, their similarities, differences, advantages, disadvantages, and considerations in chapter 2. The concept of distributed computing as a sub-category of parallel computing will be explained in slightly more detail. This chapter will then briefly outline some of the more common packages and backends for parallel and distributed computing in R and discuss their usability, advantages and disadvantages in order to answer the question about what exists in R at the moment. Finally, the case for using `doRedis` as a parallel backend to facilitate distribution on a small-scale will be put forward.

Chapter 3 aims to answer the question of why machine learning is a prime candidate for distribution, and which aspects of ML are 'distributable'. This will largely be done by qualitative research and explanations of certain aspects of ML models and algorithms. We will

see that certain ML models are perfect candidates for distribution but largely exist as 'black box' packages that do not offer the user the right controls to allow distribution. However, we will also see that certain aspects of ML are prime candidates for distribution, regardless of the choice of model or algorithm.

Next, the questions of "How does one use `doRedis`?" and "What controls exist in `doRedis` that one can leverage for more effective distribution?" will be discussed in Chapter 4. This will begin with a basic outline of the common terminology used in distribution, and `doRedis` specifically, and continues with an outline of the useful options available in the package that can be leveraged while distributing ML problems. The chapter ends with a discussion of the considerations and trade-offs one must be aware of when using `doRedis`.

Finally, Chapter 5 aims to demonstrate exactly how much time can be saved when running different types of computations on a distributed network, with a particular focus on the use case of a student whose primary device is very limited in terms of computational power, but who could have temporary access to another machine to use as a node (even in a limited capacity). The chapter begins with a simple t-test as a 'proof of concept' example, and then demonstrates the machine learning use case by distributing the fitting of random forests using R's `ranger` package, as well as demonstrating that cross validation is easily distributed.

1.3 Relevance

Parallel and distributed computing is a fairly well-researched area (at least, in general). Many companies exist that offer paid-for access to distributed, readily scalable computing clusters that users can use. However, the focus throughout this research will remain on the use case outlined in this section.

The primary use case for using `doRedis` as a parallel `foreach` backend to distributed computations at a high level is that of users who simply do not have regular access to relatively powerful computing hardware. For example, a student studying and applying ML techniques to large datasets may only have regular access to an under-powered laptop or notebook. In that case, this research aims to show that it is indeed viable for the user to write their code in such a way that when they do have access to another more powerful machine (or multiple machines), for example in a computer lab, their computations can be distributed to those machines.

The same approach can also be used to allow users access to more computational power than they might otherwise have access to. Many universities have access to high-performance computing clusters (HPCs) but their interface is generally quite different to the user interface most users are familiar with (RStudio, for example), and getting time on these HPCs is often a difficult process. It is likely to be significantly easier to get access to a number of computers that can then be attached to a Redis server where one can then distribute the computations locally.

For example, consider Figure 1.2, which shows the average daily number of jobs submitted to the University of Cape Town's HPC cluster over the last year. The average daily number of queued jobs is 29.85, indicating a certain popularity (University of Cape Town's ICTS High Perfomance Computing Team, 2022). Furthermore, a quick look at the University of Cape Town's "Introduction to Linux for HPC 2019" training material (University of Cape Town's ICTS High Perfomance Computing Team, 2019) indicates that it is not a simple case of running scripts through R or RStudio.

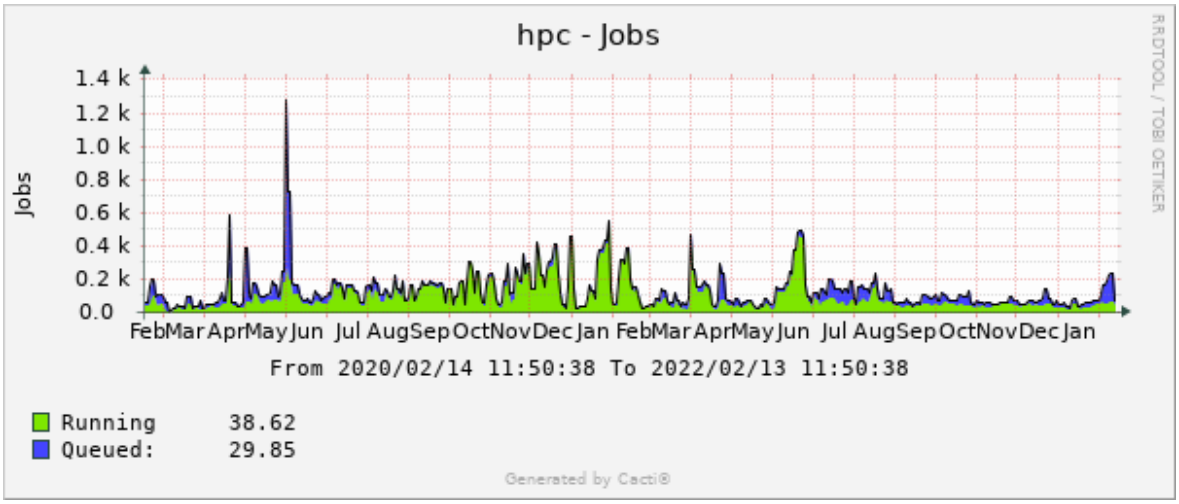

Figure 1.2: The average number of daily jobs on the University of Cape Town's HPC cluster over the last year.

Finally, the options of `doRedis`, as outlined later in Section 4.2, allow for users to attach machines to a Redis server to run computations without using so much of the attached machine's computing power that the machine is unusable while the computations run. Because of this, it is possible that a user can still make use of a number of machines to distribute calculations while the machines' owners can still use their machines for other purposes. Simply put, `doRedis` can be used to distribute calculations that are run 'in the background' of a number of machines.

In summary, the use case of this research is largely directed at 'at-home' users. This research aims to show that is is simple and beneficial enough for the average student to distribute their ML calculations to leverage faster computers than they may otherwise have access to.

Chapter 2

Parallel and Distributed Computing

2.1 Parallel Computing

Parallel computing is a well-established field that allows for "many calculations or processes [to be] carried out simultaneously" (Almasi and Gottlieb, 1994). Parallel computers, therefore, are computers that use "multiple processing elements simultaneously in a cooperative manner to solve a computational problem" (Malony, n.d.). The use of the phrase "processing elements" used by Malony (n.d.) is important because there are two main elements and concepts that both need to be present in order for a computer to process calculations in parallel.

First, we need the concepts of dependencies and concurrency. A computational task is said to be dependent if the task requires the results of another task or computation (with the required tasks being referred to as dependencies). For example, each step in the Fibonacci sequence is dependent on the previous two steps (apart from the first two steps, of course). Concurrent tasks are then those tasks that have no dependencies and therefore can be executed simultaneously.

Secondly, and even more obviously, we need computer hardware that is capable of parallel computing. The specifics of parallel computing hardware are well beyond the scope of this research, but certain important aspects may be introduced as necessary. Furthermore, additional hardware is often required to facilitate parallel computing over and above the hardware that will actually run the computations. For example, a 'conductor' may be needed to ensure calculations are distributed to the computing hardware efficiently, results are correctly combined etc. We will see in Section 2.4 that different 'conductors' offer different advantages.

Parallel execution of computations is then the term used to describe the actual running of concurrent tasks simultaneously (Malony, n.d.), and is a generalisation from sequential execution of tasks (which would need to be the case if dependencies between tasks exist) (Eddelbuettel, 2021). Another generalisation from sequential execution is that of concurrent execution, not to be confused with concurrent tasks mentioned above [1]. Briefly, parallel computing is made up of concurrency combined with the parallel hardware that facilitates parallel computing.

Particularly for machine learning problems, which often require vast amounts of data to train models adequately, the ability to run many calculations simultaneously has the potential to improve model training times.

Almost all modern computers have multiple central processing unit (CPU) cores that allow the computer to run multiple tasks simultaneously. Each core can run a number of threads simultaneously, and the product of the number of cores and the number of threads each core

[1] Concurrent execution is a generalisation from sequential execution as well but does not actually involve running tasks in parallel. Instead, tasks are run in non-overlapping time intervals and are switched between regularly. This can look like parallel execution but is not quite. (Eddelbuettel, 2021).

can run gives the number of logical processors. In R, there are many packages available that allow the user to leverage these logical processors and run their code in parallel to increase the speed of the calculations, and some of these will be further investigated later on in Section 2.4. Naturally, the more logical processors a computer has, the more parallel 'threads' can be run simultaneously.

Distributed computing builds on the idea of parallel computing, over an even larger 'network' of computers.

2.2 Distributed Computing

A limitation of the traditional approach to parallel computing arises when we consider that some users might not have access to computers with many logical processors since, as with almost anything, 'better' and 'more expensive' go hand-in-hand. Thus, distributed computing has begun to gain popularity as it allows even more access to logical processors for a fraction of the equivalent cost of purchasing one's own computational power (Steen and Tanenbaum, 2016).

In the traditional sense, 'vanilla' parallel computing allows one to run calculations simultaneously on one machine, while distributed computing allows one to 'distribute' calculations to many machines and therefore many CPUs. This, coupled with the ever-growing 'global village', has resulted in services that allow one to effectively rent computing power over the internet, such as Amazon Web Services (AWS), Google Cloud and Microsoft Azure (Venters and Whitley, 2012). The cost effectiveness of this solution can be considered in the same way that we compare renting and purchasing property (although generally on a smaller scale than with property). If you do not have the initial capital to purchase your own equipment, you can still rent it as needed.

At the most basic level, a distributed network requires some sort of server to coordinate the tasks (akin to the 'conductor' mentioned earlier), and some worker nodes to actually run the computations (Steen and Tanenbaum, 2016). The cloud computing solutions mentioned above are of course more complicated, but do not fully address the use case that we are focused on – the average student – for a number of reasons, the first and foremost being cost. While cloud computing solutions are significantly more affordable than setting up an equivalent distributed network and can be rented as needed (Venters and Whitley, 2012), there is still a cost involved. This research aims to show that setting up one's own (small) network for distributed computing is in fact relatively simple and beneficial, particularly in ML.

Further justification for small-scale distributed machine learning in R is given by way of the author's personal example:

> During the coursework component of my master's degree in data science, there were many occasions when my student laptop would take hours to train various machine learning models. I had an even older laptop and a relatively ancient desktop machine sitting unused and I thought that there must be a way to make use of their computing resources to aid my struggling laptop. It was then that I stumbled upon distributed computing, and `doRedis` in particular.

2.3 Parallel and Distributed Computing Concepts

There are various concepts within parallel and distributed computing that one must understand in order to fully unpack the packages that exist in R. Furthermore, there exist different approaches to parallel computing, and the various parallel computing packages in R leverage these approaches in different ways. While these concepts and approaches do tend to go beyond the scope of this research, the important aspects will be discussed to provide a foundation with which the packages will be analysed.

2.3.1 Multicore and Cluster Computing

A multicore processor is "a single integrated circuit ... that contains multiple core processing units, more commonly known as cores" (Firesmith, 2017). In essence, it is comparable to having many separate processors, or CPUs, in one machine. However, because the two cores are directly connected, the connection between the cores is even faster than it would be if there were multiple separate processors (Rouse, 2007). A multicore processor receives ordinary instructions to execute, however the processor can run instructions on the many cores simultaneously. In this way, calculations can be run in parallel on a single machine.

Multicore computing, then, offers parallelisation by using the multiple cores to simultaneously run computations. In R, multicore parallelisation is synonymous with forked processing (Bengtsson, 2021b).

In contrast with multicore computing is cluster computing. Traditionally, cluster computing refers to multiple computers (nodes) that are connected together by some network infrastructure, normally a local area network (LAN) (Schmidberger et al., 2009). However, it is possible to simulate a cluster on a single machine by creating multiple sessions or instances of a program, along with a 'master' to facilitate the distribution of computations. In this way, computations can be run in parallel on a single machine (or indeed on many machines).

In R, cluster computing is implemented by means of sockets (see Section 2.4 below).

2.3.2 Parallelisable Algorithms

It must be noted that not all algorithms can be run in parallel. In order to run an algorithm in parallel, the algorithm must not need to be carried out sequentially. Parallel computing allows multiple computations to occur simultaneously, however if these calculations depend on one another, then the algorithm cannot be run in parallel. We will call these algorithms non-parallelisable.

As an example of a non-parallelisable algorithm consider the following algorithm:

Algorithm 1: Basic non-parallelisable algorithm

Input : Data (with ≥ 1 explanatory variable(s) and 1 response variable, y), and a vector of (random) initial weights, v

Output: Vector of trained weights

1 **for** *row r in data* **do**
2 Use v on the explanatory variables in r to predict the response variable, $\hat{y}$
3 Calculate some error measure based on the difference between y and $\hat{y}$
4 Use r to update v by minimising the error measure
5 Set v to the newly updated weights;
6 **end**

Clearly this algorithm cannot be run in parallel as each step depends on the previous step's result. Thus, each calculation must be completed before any further calculations are under-

taken. This algorithm must therefore be run sequentially, or serially.

However, there are algorithms that are said to be 'embarrassingly parallel'. These algorithms allow calculations to be carried out by processors completely independently, without any communication (Neiswanger, Wang, and Xing, 2013). As an example, consider Monte Carlo simulation.

Monte Carlo simulation is a method of simulating random variables by using repeated sampling (Raychaudhuri, 2008). In Monte Carlo simulation, 'input' values are generated based on a statistical distribution. For each input value, a calculation can be done and a result noted. For example, the input could be the return on a share for a given period and the calculated value could be an investor's loss or gain, which clearly depends on the return of that share which they have invested in. By repeatedly sampling the return of the share and calculating the loss or gain, a simulated distribution for the investor's loss or gains can be generated. Clearly these calculations can be done completely independently of one another and so they can be done by a network of processors simultaneously. Thus, this example is embarrassingly parallel.

2.4 Parallel and Distributed Computing in R

By default, R is a sequential language, that is R executes code in a specific order (Eddelbuettel, 2021). Any package in R that allows for parallelisation will involve "starting new R processes, and the means for communicating between these processes" (Rosenblatt, 2019). It is therefore important to understand the two most common methods of creating new R processes: forking (based on multicore computing) and sockets (based on cluster computing). Forking involves creating each new R session as a duplicate of the original R session while a socket is in fact a mechanism that allows different processes on a computer to communicate with one another (and the resulting processes are sometimes called spawn processes) (Rosenblatt, 2019).

Forking in R involves sending the parallel computations to be run on the different cores simultaneously by creating forked processes. By doing this, global variables are inherited automatically. On the other hand, using sockets in R involves starting many independent processes that are given instructions by the main R session and so global variables need to be explicitly passed to the nodes, yielding a greater communications overhead (Eddelbuettel, 2021).

While forking may seem more appealing due to its lower communications overhead, a fairly significant drawback is that it is not fully supported on non-Unix-based machines (i.e. it is not fully supported on Windows machines) (Eddelbuettel, 2021). If a function like `mclapply` (`mc` being shorthand for 'multicore') is used on a Windows machine, it will revert to sequential processing (Eddelbuettel, 2021). Cluster computing using sockets is fully supported on Windows-based machine, and possibly the bigger advantage in our case, is that it is conceptually easy to see how cluster computing can be extending to allow external ('remote') workers (or nodes) to act as additional R processes that connect to the main R session and run computations.

The remainder of this chapter will outline some of the numerous packages available in R that exist to allow computations to be parallelised.

2.4.1 `parallel`

The `parallel` package has been included in base R since version 2.14.0 (R Core Team, 2021). This package allows one to replace `lapply()` and `mapply()` with `mclapply()` and `mcmapply()` respectively to allow for parallelisation (with a few more options, for example `mc.cores` sets the number of cores to use). As mentioned previously, `mcmapply` implies a multicore/forking approach, which is not available on Windows machines. However, the `parallel` package provides the user with a workaround. One can create a cluster of R sessions (using sockets) with the `makeCluster()` function from `parallel`. One can then run calculations in parallel using that cluster of R session.

There are then a number of packages that build on `parallel`. One of the more common ones is `doParallel` (Wallig et al., 2020), which provides the user with a parallel backend that they can leverage on their own machine, specifically for `foreach`. Using `doParallel` in combination with `foreach` and `%dopar%` allows the user to easily run traditional `for` loops in parallel, which is beneficial given that many calculations, especially parts of ML model training, require many iterations (i.e. `for` loops).

2.4.2 `future`

According to the package's documentation, "[t]he purpose of the `future` package is to provide a very simple and uniform way of evaluating R expressions asynchronously using various resources available to the user" (Bengtsson, 2021a). The `future` package is based on the distributed computing concept of futures (and promises), where futures can be thought of as values that, while not presently available, will become available at some stage in the future (Prasad, Patil, and Miller, 2016). Essentially, a future can be thought of as a placeholder that will be resolved at a later stage.

The full explanation of futures and promises falls outside the scope of this research, however an important feature of the `future` package in R is that it allows for parallel/distributed computing. In `future`, one must specify a plan (how the futures will be evaluated) before evaluating the futures. Some of the options include 'sequential' (the default), 'multisession' (which resolves futures in parallel in separate R sessions on the local machine) and 'cluster' (similar to multisession but futures are resolved on typically more than one machine). 'cluster' is then the option that would allow for distributed computing in R.

2.4.3 Docker

Docker is a program that creates containers (virtual operating systems) on a machine (Fay, 2019). Traditionally, the use case for Docker is to avoid dependency issues: one can build a container with all the correct versions of packages that one needs. In this way, there will never be dependency issues when new versions of packages or software are published. However, it is simply enough to see how this could be leveraged for parallelisation: one could create a number of containers and then send calculations to each of the different containers to be run simultaneously.

Indeed, this could be done locally (but creating more containers than your computer has cores would be redundant) or one could create a number of containers on a number of external computers. This network of connected machines is called a Docker Swarm, and calculations could then be distributed among the nodes in the Swarm.

2.4.4 `sparklyr`

`sparklyr` (Li, 2021) provides an interface to R for Apache Spark, an open-source engine aimed at big data analytics. Without going into too much detail, Apache Spark "is a general-purpose cluster computing framework with language-integrated APIs in Scala, Java, Python and R" (Salloum et al., 2016).

Furthermore, Apache Spark makes use of Resilient Distributed Datasets (RDDs), which aim to close the gap left by MapReduce's lack of an efficient data-sharing system across nodes in a distributed network (Zaharia, 2016). RDDs provide an immutable, fault-tolerant data storage system that allows segments of the RDD to be distributed over a cluster (Zaharia, 2016). However, it is the cluster computing aspect of Spark that makes it worth investigating in the use-case of this research.

In experimenting with `sparklyr`, several reasons were discovered for preferring the `doRedis` package. While considering these points that follow, it is important to keep the use case of this research (outlined above in Section 1.3) in the back of one's mind: to allow greater access to distributed machine learning for users who otherwise might not have the resources needed.

Firstly, `sparklyr` requires setting up an Apache Spark cluster. While this is of course possible, it proved significantly more challenging than setting up a Redis server. One possible redemptive quality of Apache Spark, however, is that it allows clusters to be run on a variety of systems (locally and cloud-based) and data can be accessed from a variety of data sources.

Secondly, the learning curve with `sparklyr` is far steeper than for `doRedis`. `sparklyr` makes use of many of its own functions and requires things like conversion (and declaration) of data into a particular `sparklyr` format. While learning should not be a barrier to adopting something, it does mean that alternatives to `sparklyr` are more attractive, especially alternatives that require little setup and new learning.

2.4.5 `doRedis`

`doRedis` is an R package written by Bryan W. Lewis (2021) that "defines a `foreach` parallel computing adapter using Redis and the `redux` package". In essence, the package provides a parallel backend much like `doParallel` does except that it instructs R to send jobs to a Redis server. `doRedis` can then be used with `foreach` to distribute computations to nodes attached to the Redis server.

While `doRedis` is not as popular as some of the other methods and packages mentioned above, there are certain benefits that make it a prime candidate for the use case outlined in Section 1.3 above.

Firstly, `doRedis` allows for an elastic pool of workers. This means that the nodes attached to the Redis server need not be fixed, or specified in advance (as is the case with some other packages/approaches mentioned above). `doRedis` allows the pool of workers to be dynamic, growing or shrinking as more nodes are added or removed, and the tasks are redistributed accordingly. The author of `doRedis` describes this approach as being "geared for modern cloud computing environments" (Lewis, 2021b). One complication of this elasticity is how the server manages with nodes that 'drop'. Luckily, this is also an advantage of `doRedis`.

Secondly, `doRedis` is "partially fault-tolerant" (Lewis, 2021a). Should a node go down while working on a task, `doRedis` acknowledges the fault and redistributes the affected tasks. As already mentioned, this is a key feature when combined with `doRedis`'s elasticity. As more nodes are added, there are more opportunities for a node to fail a task (Bansal, Sharma, and

Trivedi, 2011) and therefore some level of fault tolerance is crucial.

Lastly, `doRedis` allows the user to distribute tasks across different operating systems (Windows, Mac OS X and Linux). Naturally the nodes will require an R instance on each of them but beyond that, tasks can be completed by nodes running any of the major operating systems. The only limitation in this regard is that it is recommended that the Redis server be run on Linux (as opposed to Windows). Carlson (2013), in his freely available e-book on the Redis website, mentions a number of drawbacks to running a Redis server on a Windows machine, the most prominent of which is that without "the ability to fork, Redis is unable to perform some of its necessary database-saving methods without blocking clients until the dump has completed."

This chapter defined parallel and distributed computing, both of which are available to users in R, and highlighted the most commonly used packages in R. In the next chapter, we will place specific focus on ML problems, the aspects of them that are suitable for distribution, and why it may be beneficial to distribute these aspects.

Chapter 3

Distributed Machine Learning

Mitchell (1997) describes machine learning (ML) as being "concerned with the question of how to construct computer programs that automatically improve with experience." Naturally, an algorithm that is expected to 'learn' will require data to learn from, and it is also intuitive that more data will result in a better result (nuances such as data quality notwithstanding). In this day and age, it is generally not a lack of data that affects most users wishing to train an ML model, as data is collected more frequently and made more readily available online. However, training these models on such large datasets, or even training very complex models on relatively smaller datasets, requires computing power that the users described in the use case in Section 1.3 above may not have access to.

Furthermore, certain machine learning algorithms are perfect candidates for distributed computing as they are parallelisable. For example, random forests can be parallelised (and therefore distributed) (Genuer et al., 2017). Of course there are other ML models that are also candidates for parallel computing, however random forests will be investigated as a primary example. In truth, any model that involves independently-built sub-models or parts should be a candidate for parallelisation. It is important to note that distribution is often difficult when using the common ML packages in R because the packages are built to be easily used and are their calculations are thus done 'behind the scenes'. The iterative processes in these packages, which is where distribution would be implemented, are generally not editable.

However, many ML models require cross validation (Berrar, 2019) and/or hyper-parameter tuning. These two aspects can often be implemented automatically by a package but they can also easily be implemented manually and then distributed to a cluster. The purpose of this section is to first demonstrate that where a ML model is coded 'from scratch', it is possible to distribute, and then to demonstrate that distributing hyper-parameter tuning and cross-validation using `doRedis` are indeed beneficial.

3.1 Random Forests

Random forests are a class of ML models built on the foundation of bootstrap aggregated ('bagged') trees, which are in turn built on the foundation of decision trees (James et al., 2017). In random forests, a number of decision trees are built on bootstrapped training samples, as in bagging. The difference, however, is that random forests allow only $m < p$ predictors to be considered when building the decision trees. The rationale for this, to decorrelate the trees, is not important at this point, but is unpacked in *An Introduction to Statistical Learning with Applications in R* (2017). What is important, is that the decision trees are built independently of one another and thus can be distributed, instead of being built sequentially.

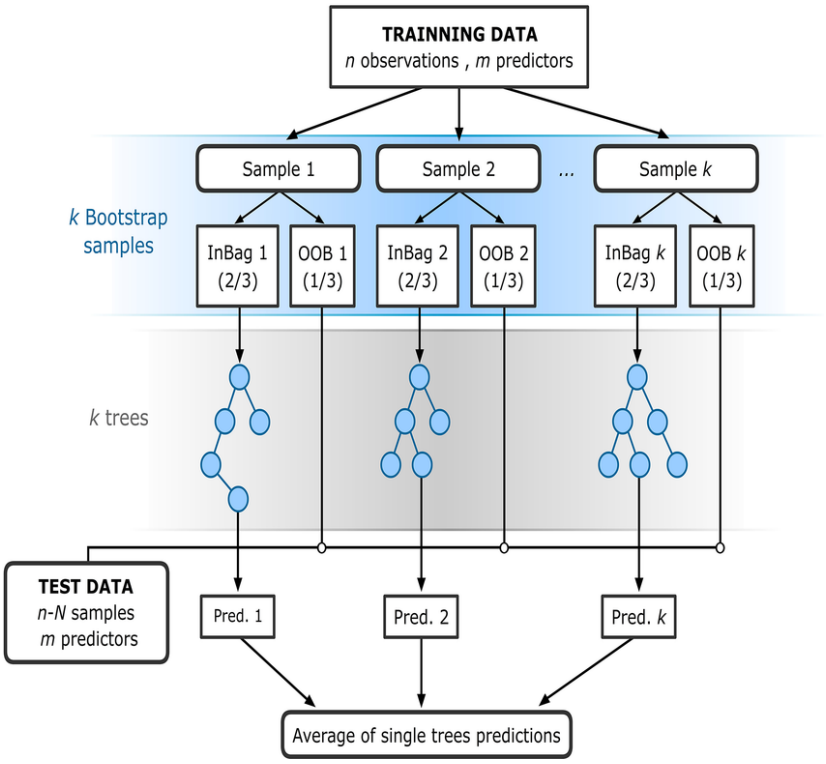

Figure 3.1: Flowchart depicting the process of fitting a random forest for a regression problem (Rodriguez-Galiano et al., 2016). Licensed under CC BY 3.0.

Figure 3.1 above outlines the process of fitting a random forest regression model. First, k samples are generated by bootstrapping the observations (i.e. sampling with replacement). A decision tree is constructed on each sample independently, using $m < p$ predictor variables, and these decision trees are then used to generate predictions based on the input variables. The final step involves averaging the predictions (for regression problems) to arrive at a final prediction. As with any ML algorithm, a proxy for the testing error is very useful. In the case of random forests, cross validation (Section 3.3) is unnecessary. Instead, one can use the 'out-of-bag' (OOB) error.

For random forests, each bagged tree will use approximately two-thirds of the available data on average (James et al., 2017). The remaining data, the 'out-of-bag' observations, can be used as a sort of validation dataset (where the process of predicting and averaging the predictions is the same as for the 'in-bag' observations described above). The OOB error can then be calculated and used as an estimate of the test error.

Because the decision trees are grown on each of the bootstrapped samples completely independently, random forests are prime candidates for parallelisation, and therefore distribution. However, many of the currently popular implementations of random forests in R are 'black boxes' that cannot easily be distributed themselves. The purpose of this section and its description of random forests is to indicate that it would theoretically be possible to extend a particular random forest algorithm/package to make use of a parallel backend such as `doRedis` to allow for the distribution of a random forest package. In fact, some random forest packages in R such as `ranger`, which is described below, already make use of a random forest's ability to be parallelised, but only locally.

Furthermore, random forests involve a number of hyper-parameters that affect the model, both in terms of accuracy and in terms of the time it takes to train a random forest. It is common practice to 'tune' these hyper-parameters in random forests (and many other ML models). This is discussed in more detail in Section 3.2 below.

A commonly-used R package to implement random forests is `ranger`. `ranger` is "a fast implementation of random forests for high dimensional data" (Wright and Ziegler, 2017) and is widely known for its speed advantage as it is implemented in C++. A specific feature of `ranger` that will prove useful is that it is, by default, run in parallel on the user's local machine. While a `ranger` model cannot easily be distributed itself, it is still a prime candidate for distribution with `doRedis` when combined with hyper-parameter tuning (Section 3.2 below).

Each distributed model trained on a combination of the hyper-parameters is its own `ranger` model. This means that when a node picks up a task to train a `ranger` model, it will also train that model in parallel locally. In this way, there is a sort of 'double-parallelisation': the models arising from the combinations of hyper-parameters are distributed to the nodes, and each of those jobs are then also run in parallel on the nodes themselves. This is a particular advantage available to `ranger` because of its inherent parallelisation of random forests.

3.2 Hyper-parameter Tuning

Many ML models involve parameters that influence a model's accuracy, ability to generalise to unseen data, and training time. These are called hyper-parameters to distinguish them from the parameters of a model, which are the numerical values associated with the output of the model. Weerts, Mueller, and Vanschoren (2020) describe parameters simply as being "derived through training", and hyper-parameters as "parameters that need to be fixed before running [the models]". For example, consider a simple two-dimensional linear regression with regularisation applied. The y-intercept and gradient of the linear regression would be parameters since they are output-related values – they are calculated by the model with reference to the data. On the other hand, the α value supplied to the regularisation component (whether LASSO or ridge regularisation) of the linear regression model would be a hyper-parameter – it is supplied beforehand and impacts the output and/or training of the model.

It is common practice to create a grid of various values and the associated (exhaustive) combinations of these hyper-parameters, and to train the model on these combinations to identify which specific combination best suits the data, based on some metric (Bergstra and Bengio, 2012). This involves training the model numerous times, which is computationally costly. However, the models are trained with the various combinations of hyper-parameters completely independently and so can be distributed. This can greatly improve model training times, as we will see later.

Many ML models make use of this hyper-parameter tuning approach including, but not limited to, k-nearest neighbours models, Support Vector Machines (SVMs), neural networks and even regression (with regularisation, that is). In fact, hyper-parameter tuning is sometimes built into the R package for a specific model. However, it is often done in a fairly straightforward way: the user supplies a grid of the combinations of hyper-parameters and the model will iterate through the grid and train the model using the various hyper-parameter values. This process can easily be implemented manually in R with only a few lines of code and can easily be distributed.

3.3 Cross Validation

A model's test error rate is "the average error that results from using a statistical learning method to predict the response on a new observation" (James et al., 2017). Since models are built to be applied to 'unseen' data (i.e. data other than the data used to train the model), it then follows that a lower test error rate implies a model that is likely to generalise better. Of course, it is not possible to know the test error of a model because one cannot have the

'unseen' data that the model would be applied to. Thus, data scientists and statisticians make use of an approximation of the test error to provide an indication of the ability of the model to generalise.

A naive approach would be to randomly split the data, creating a 'training' set and a 'validation' set. The model could be trained on the training set and evaluated on the validation set to provide an estimate of the test error [1]. However, this may not provide the best estimate as it is only based on a sample size of one. An extension of this is cross validation (CV). CV still makes use of the 'training'/'validation' split approach above but does this multiple times so as to arrive at a test error estimate that is more powerful. In this way, CV gives a view of the variation in the validation performance of the model. Clearly, the repeated fitting and evaluating (on a validation set) of a model increases the computational overhead, but it is often a necessary aspect of ML.

k-fold CV begins by randomly partitioning the data into k sets. The model is then trained on the first $k-1$ partitions and evaluated on the k^{th} partition to get a validation error for that partition. The model is then trained on all the partitions except partition 2 and is evaluated on partition 2. This process is repeated k times, until all partitions have been 'left out' of the model training and used as validation sets. The CV error is then the mean of the validation errors for each of the k validation errors. A common approach is to use $k = 10$ (where there are enough data to allow for this).

CV is a good candidate for distribution because each of the k processes of training and then evaluating the trained model is independent of the others, and it is so common in many ML models. While many R packages allow for CV within their options, CV is an inherently straight-forward concept that is simple to manually distribute with the use of `foreach` loops and `doRedis` in order to gain the benefits of distribution.

The next chapter considers `doRedis` and `foreach` in more detail.

[1] A test set can still be created from the original data, however it must only be used once as a measure of the ability of the model to generalise to unseen data. The validation set can be used to inform the model and the hyper-parameters can then be adjusted accordingly, but this is not the case with the test set. If the test set is used more than once, it loses its value as an indication of the ability of the model to adapt to unseen data.

Chapter 4

`doRedis` as a `foreach` Parallel Backend

This chapter will outline the more specific details of the `doRedis` package. The terminology will be outlined briefly but the focus will be on functions, specifications, and options (Section 4.2). The majority of the information in this section comes directly from the package documentation (Lewis, 2021b) and vignette (Lewis, 2021a), but will have a particular focus on using `doRedis` to distribute ML tasks, where applicable. The package documentation can be easily accessed for more specific details.

4.1 Terminology

A *task* is "a collection of one or more loop iterations" (Lewis, 2021a). The number of iterations (i.e. `foreach` loops) per task can be set with the `setChunkSize()` function (see 4.2 below).

A *job* is "a collection of one or more tasks that covers all the iterations required by a single `foreach` loop" (Lewis, 2021a).

Jobs are then submitted to a *queue* on the Redis server as a Redis list made up of the tasks. *Nodes* (R sessions) are then attached by pointing the R session to the Redis server, where they are "assigned work by task" (Lewis, 2021b).

The figure below provides a brief overview of a typical `doRedis` setup.

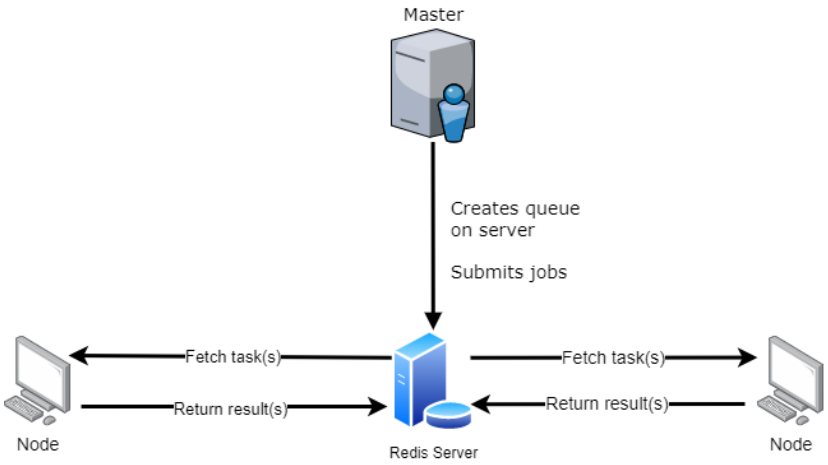

Figure 4.1: Network diagram describing the task submission/retrieval process. Created using *diagrams.net*.

Consider the example mentioned in Appendix B.3 for a more detailed description. If the

chunkSize has been set to one, then each iteration of the foreach loop will be its own task. This is the default. The whole collection of ten tasks is then the job that is submitted to the work queue on the server. As a node attaches to the server, it would pick up a task (a single iteration of the loop, in this case), run it, and return the results to the server. Once all tasks have been completed and returned, the master R session will compile them using R's sum function. Changing the chunkSize to five, for example, would mean that only two tasks are submitted as a job to the work queue. Of course different applications may benefit from different chunkSizes, and the benefits and drawbacks in certain cases are described in Section 4.2.

4.2 Functions, Specifications and Options

doRedis has a few functions and options that can be used or set to allow more control over the distribution of jobs. There are other functions available but those described below are the most frequently used.

4.2.1 redisWorker and startLocalWorkers

Both redisWorker and startLocalWorkers can be used to attach nodes to a Redis server. The key difference lies in whether background R processes are created, or if the main R session is used instead.

redisWorker "enrols the current R session" (Lewis, 2021b). This means that once redisWorker is used, the R session is 'blocked' until the queue is removed from the server (using the removeQueue function). Thus, only one worker is effectively fetching, calculating, and returning tasks at a time.

On the other hand, startLocalWorkers starts a specified number of R sessions in the background. Naturally this uses more processing power the more R sessions are started, but it can provide a significant benefit to calculation time. Modern computers are often computationally strong enough to handle multiple R sessions simultaneously and so it makes sense to leverage this available computing power on each computer (provided it is not needed, of course). In order to gain the most benefit from distributing a calculation, one would want to use as much computing power as is available. Doing this depends on a few things, including whether a calculation can be parallelised locally (as in Section 3.1 above) and the number of iterations that make up each task. This is specified using setChunkSize.

4.2.2 setChunkSize

The number of foreach iterations that are grouped together as a task is set by setChunkSize, with the default being one. Setting a higher chunkSize will generally improve performance for simpler calculations by reducing the time spent fetching and returning results via the network (i.e. reducing the communication overhead). However, setting chunkSize too large can result in worse load balancing across workers (Lewis, 2021b).

Furthermore, a large chunkSize can negate the benefits of distribution entirely. For example, if chunkSize is set to the total number of foreach iterations (the maximum allowed value) then only a single node will be involved in the calculations because the job is made up of only one task, which is made up of all of the iterations. It is, however, possible that setting the chunkSize to its maximum is beneficial.

Consider the case where only a single node is available, for whatever reason. In this case, reducing the communication overhead to as small as possible makes sense, and this is done by setting chunkSize to the total number of foreach iterations. A complication arises because,

as mentioned above, most modern computers can handle many R sessions and so could act as multiple nodes. However, if a calculation cannot be locally parallelised, and the potential node is more powerful than the potential 'master', then distribution becomes beneficial (assuming the computation benefit outweighs the communication overhead). This approach can essentially be thought of as outsourcing a calculation to another machine entirely.

Essentially, `setChunkSize` allows the user to decide how much of a calculation to distribute to each node. Clearly, using `setChunkSize` can have a significant impact on one's computation time. Proper thought should therefore go into the `chunkSize` and the computation that one intends to distribute so as to maximise the benefit of distribution.

4.2.3 `removeQueue`

`removeQueue` is simply used to remove the work queue from the Redis server. Both `redisWorker` and `startLocalWorkers` have an argument, `linger`, with a default of thirty, that specifies how long, in seconds, after the work queue is removed a node will terminate itself. Because of `linger`, removing a job queue with `removeQueue` is a convenient way to terminate all nodes. It is also important to remove the work queue so that, if one was to re-run the code, there are no errors that arise because of trying to create a queue that already exists. Lastly, removing the queue also keeps the Redis server slightly tidier.

4.3 Considerations

Based on the options and functions mentioned above, certain considerations need to be thought through so that a user gets the most out of distributing their computations. A particular case arises when one is using a package like `Ranger`, described in Section 3.1. This case provides a logical division with which to analyse the considerations when using `doRedis`: ML models that can be locally parallelised and models that cannot.

An ML model that can be locally parallelised (a random forest using the `Ranger` package, for example) can inherently use all available logical processors in a machine (or a user-specified number). When run locally, this allows a user to use more of the computing power available to them than other models that are not parallelised. Distributing a parallelisable ML model offers similar benefits, albeit on multiple nodes as opposed to on a single machine. However, a hybrid approach can still be used to offer even greater benefits. This was discussed in Section 3.1. In this case, the `chunkSize` chosen is relevant but the choice between `startLocalWorkers` and `redisWorker` is less important. Because the training of the actual random forest will be locally parallelised, one is unlikely to need to start multiple R sessions; a single R session will make use of multiple logical processors anyway (unless specified otherwise). The optimal `chunkSize`, on the other hand, is likely to depend on the number of hyper-parameter combinations and the number of nodes available.

ML models that are not locally parallelised but are parallelisable require a fair amount of forethought as to the number of background R sessions to start with `startLocalWorkers`, and the value to pass to `setChunkSize`. Since the model will not be trained in parallel on each machine attached to the server, the opportunity to make each machine run multiple nodes to fully utilise the computing power of each machine presents itself. The number of R sessions that will lead to fully utilising each machine will depend on both the computing power in that machine and the complexity of the calculations being run. Similarly, the optimal `chunkSize` will also depend on the computing power available, the complexity of the calculations and the number of nodes.

The general idea is to balance the number of R sessions and the `chunkSize` so that no R session is idle while others still have more tasks to complete. A naive approach, albeit better than leaving the `chunkSize` at its default of one, would be to set the `chunkSize` to be the total number of `foreach` iterations divided by the number of nodes. In this way, each node will pick up a task from the server. Obviously this is not ideal if the number of nodes changes (which is one of the benefits of `doRedis` – elasticity).

The next chapter aims to present some examples of the benefits of distributing computations via a Redis server and `doRedis`. It begins with a simple 'proof of concept' example, and then moves on to more focused ML problems.

Chapter 5

Benchmark of the Time Improvement in ML Training when using `doRedis`

For this chapter, a series of 'benchmarking' examples were run. This benchmarking exercise aims to determine a) if using `doRedis` to distribute computations in a limited case does indeed save compute time, and b) if using `doRedis` in a ML context is viable.

Whenever the computations in this chapter were run sequentially (i.e. no parallel backend) or using `doParallel` as a backend, they were run on the same machine. This is also the same machine that Redis was installed on to identify if it is indeed viable for someone with (relatively) limited resources to install Redis and use `doRedis` to distirbute tasks whenever next they have access to slightly more advanced hardware. The computations were distributed to a separate machine when using `doRedis` as a parallel backend.

Section 5.1 compares three parallel backends to one another using a series of repeated t-tests.

Section 5.2 applies `doRedis` to distribute random forest model training and hyper-parameter tuning using the `ranger` package (as outlined in sections 3.1 and 3.2).

Section 5.3 compares cross validation using different parallel backends. See Section 3.3 for more detail on cross validation.

5.1 Three Parallel Backends

To begin with, a non-ML task was set up in order to test the viability of distributing simple tasks. The calculation chosen is outlined by the pseudo-code below.

Simply put, the algorithm generates n random standard Normal values twice and performs a t-test between the two samples, testing whether the population means differ. This process is done *count* times using a `foreach` loop so that each of the *count* iterations could be run in parallel or be distributed. The time taken to complete each set of *count* iterations is recorded, and the whole process is repeated r times to create a distribution of the time taken per iteration.

The process was repeated three times using three different parallel backends for `foreach`: a sequential 'backend', a local parallel backend, and `doRedis`. A detailed list of the hardware used can be found in Appendix A, however, the use case for this example was the ability to send computations from a (relatively) slow computer to a (relatively) fast one. Particularly for students working on netbooks or under-powered laptops, it may be the case that they

Algorithm 2: Basic data generation and t-test.

Input : Number of t-tests to perform, *count*. Number of random normal values sampled, n. The number of repetitions (i.e. times to record), r.

Output: Vector of time taken (in seconds) for each iteration

```
for i in 1 up to r do
    Begin timer
    for j in 1 up to count do
        Generate a vector of n random normal values
        Generate a 2nd vector of n random normal values
        Perform a t-test between the 2 vectors
    end
    End timer
    Calculate difference between beginning and ending timer and record
end
Return vector of times
```

can code the algorithm themselves and run it on a faster machine. Of course the benefit of `doRedis` lies in its elasticity, and so more computers can easily be attached to the distributed network of machines.

Figure 5.1 shows the distribution of time taken per iteration with each parallel backend for $r = 7500$, $count = 2500$ and $n = 50$, along with their associated means. For ease of visibility, the longest one per cent of times per iteration were excluded for each of the backends.

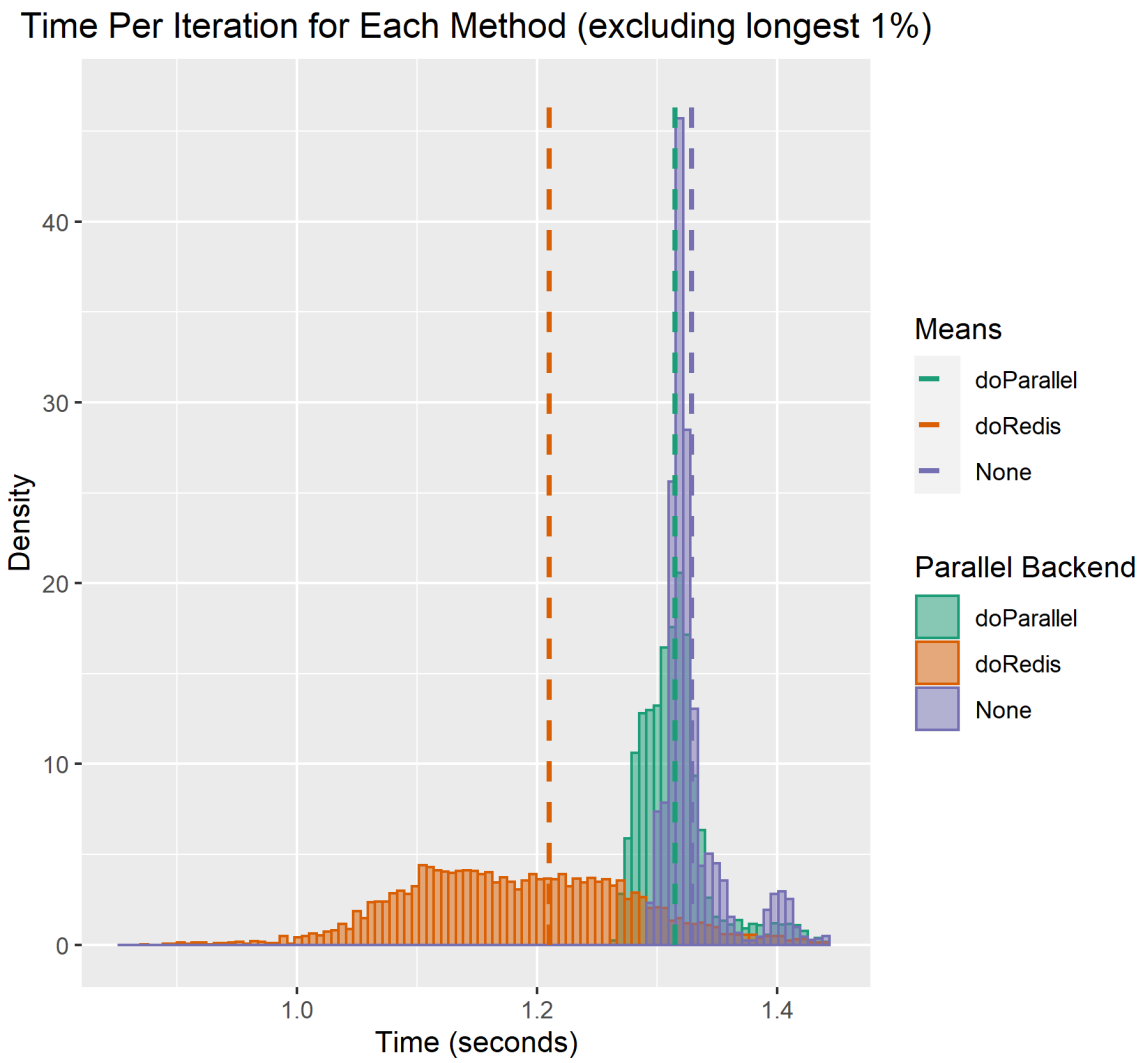

Figure 5.1: Densities of the time taken per iteration for each of the three parallel backends when running a sample t-test, shown along with their respective means.

Clearly, `doRedis` outperforms the other two backends. This is not especially surprising given that the setup was essentially distributing calculations to a faster machine. However, it is interesting to note the spread among the times per iteration. `doRedis` appears to have a significantly larger variation in time per iteration than either of the other two backends (with a standard deviation of 0.4355 for `doRedis`, 0.0300 for `doParallel`, and 0.0233 when run sequentially). This is likely due to the communication overhead that is inherent when distributing tasks via a network as tasks need to be sent to the server, fetched from the server, computed, and returned to the server to be compiled. Any of those steps could introduce the slightest time difference, resulting in a greater variation of times.

It also appears that the time taken per iteration when the calculations were run sequentially, was bi-modal. This is interesting, but is likely due to other processes on the machine taking up slightly more computational power at different times. The machine running the computations sequentially still has various other processes running at the same time, and so it is possible that any number of other processes required more computational power for a short while.

5.2 Random Forests and Hyper-parameter Tuning

As mentioned in Section 3.1, the `ranger` package in R runs in parallel by default. For this section, a hyper-parameter grid was set up and `ranger` was used to fit random forests to a dataset (with 2930 observations and 80 explanatory variables) based on the values in the hyper-parameter grid. In this case, the benefit of distribution is that one can try more combinations of hyper-parameters and distribute the fitting of a random forest for each combination.

The basic algorithm followed is below. The hyper-parameter grid is set up in such a way that each row is a single combination of the hyper-parameter values specified.

Algorithm 3: Random forests and hyper-parameter tuning.

Input : Hyper-parameter grid, g. Number of repetitions, r.
Output: Vector of time taken (in seconds) for each iteration.

```
for i in 1 up to r do
    Begin timer
    for j in 1 up to number of rows in g do
        Obtain hyper-parameter values from row j in g
        Fit a random forest with ranger, using the hyper-parameters from the step above
        Obtain the root-mean-square error (RMSE) of the model
    end
    End timer
    Calculate difference between beginning and ending timer and record
end
Return vector of times
```

Of course, hyper-parameter tuning can be used with a variety of ML models and so the value of being able to perform hyper-parameter tuning more quickly or to be able to try more combinations of hyper-parameter tuning quickly becomes obvious. A common package in R that allows for hyper-parameter grid searches is `caret` (Kuhn, 2021). `caret` also allows for the user to register a parallel backend and so would be easily implemented in conjunction with `doRedis` (Kuhn, 2021). The details of `caret` are beyond the scope of this dissertation.

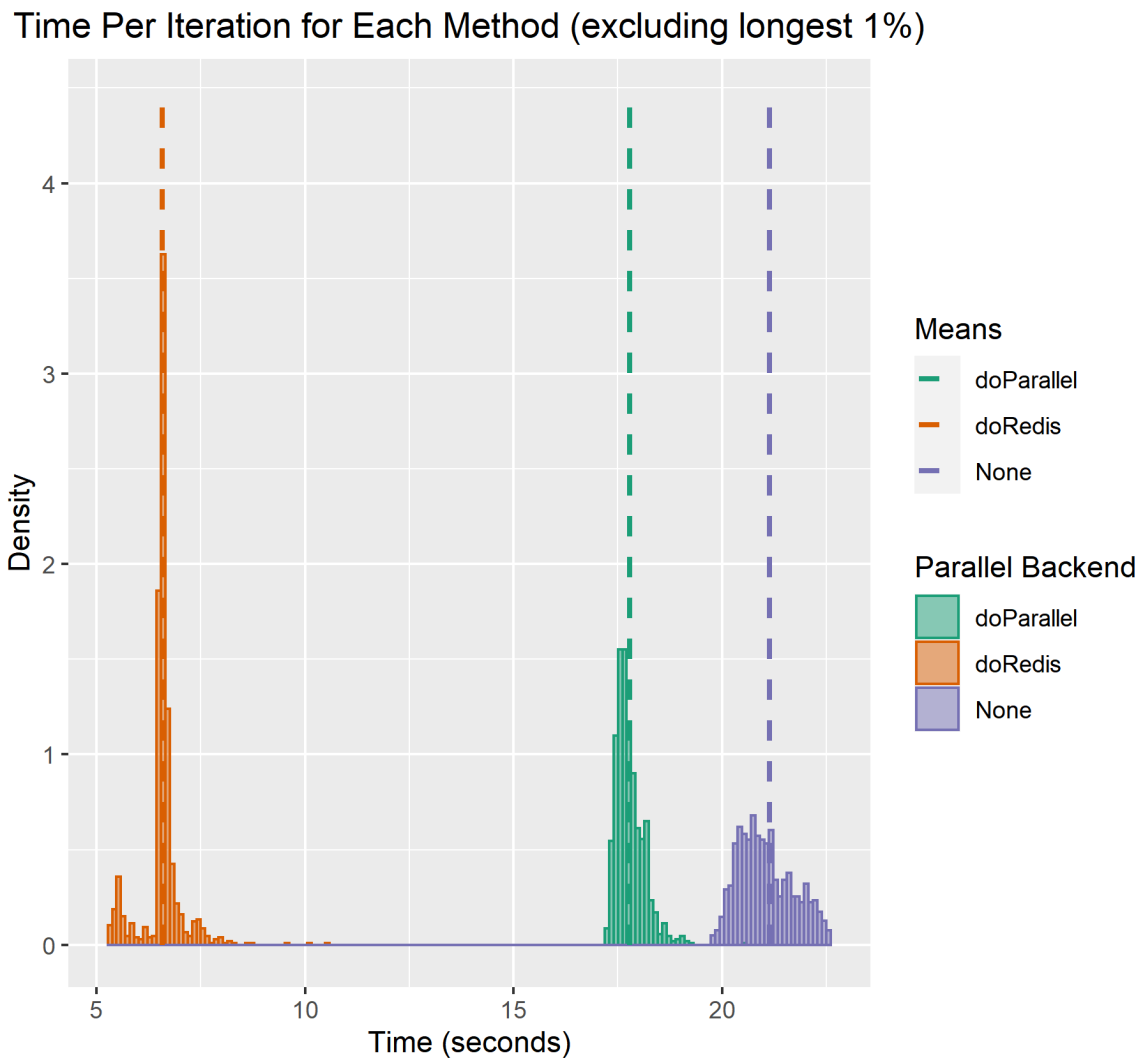

Figure 5.2: Densities of the time taken per iteration for each of the three parallel backends when fitting a random forest using the `ranger` package, shown along with their respective means.

Figure 5.2 above shows the value to be gained by distributing the fitting of a random forest for each combination of hyper-parameters. One can see that there is a significant time difference between not using a parallel backend and using `doParallel`, and using `doRedis`.

The size of the difference can largely be explained by the fact that `ranger` runs in parallel by default. This means that each machine that picks up a task from the Redis server will run that task in parallel locally. And so, by distributing to a faster computer with more cores (eight versus the four of the master node), the effects of distribution are amplified. Naturally it is then clear that the same benefits would likely not be realised if one were to distribute the training of the model to a slower, less efficient machine (for example, if the roles of the machines in this example were swapped). In that case, it would probably be faster to simply run the model fitting locally. However, distributing the model fitting to a number of machines is likely to still offer benefits, and an added bonus of `doRedis` is that one can always use the master node to run calculations as well by using the `startLocalWorkers` function mentioned earlier.

What is interesting about the results displayed in Figure 5.2 is that there is still a difference between fitting the `ranger` model using `doParallel` compared to without a parallel backend, considering that `ranger` runs in parallel by default. The explanation likely involves considering what is being parallelised in each case. The example was set up to distribute the fitting of a random forest for each line (i.e. combination of hyper-parameters) of the hyper-

parameter grid, and so using `doParallel` would distribute each line to a different core on the machine. On the other hand, not using a parallel backend means that each combination of hyper-parameters was tried sequentially, but that the random forest was still fit in parallel. In this case, it appears that the relative lack of complexity in the `ranger` models meant that parallelising each combination of hyper-parameters still offered benefits.

It also appears that all three of the distributions may be bi-modal. A possible explanation of this is that the minimum node sizes supplied to the hyper-parameter grid were three and six. Since the minimum node size is "probably the most common [hyper-parameter] to control tree complexity" (Boehmke and Greenwell, 2020), it implies that the fitting of some of the models was more complex than the fitting of others. This may have led to an increase in the time taken per iteration (when the model was more complex).

5.3 Cross Validation

`xgboost` is a package in R "which is an efficient implementation of the gradient boosting framework" (He, 2021). Gradient boosting is an ensemble method of growing trees where each tree is based on the output of the previous tree (Singh, 2018). Firstly, a tree is fit to the data. The next tree is then fit to the data but using the residuals of the first tree as the second tree's response variable. This process is repeated, ultimately resulting in a number of trees that are each trained to account for some of the variation in the previous tree (Singh, 2018). `xgboost` was used as the base model for the comparison of cross validation in this section, for no particular reason other than simplicity of application.

The basic process of cross validation, as explained in section 3.3, was applied in the following way (to a dataset of 2930 observations and 34 explanatory variables):

Algorithm 4: Cross validation of `xgboost` model.

Input : Number of CV folds, n. Number of repetitions, r. List of row indices for training samples for each fold, $folds$. Various other parameters passed to `xgboost`.

Output: Vector of time taken (in seconds) for each iteration.

```
for i in 1 up to r do
    Begin timer
    for j in 1 up to n do
        Split the data into training and validation sets using folds
        Fit an xgboost model on the training data
        Obtain predictions using the validation data
        Calculate the root-mean-square error (RMSE) of the predictions
    end
    End timer
    Calculate difference between beginning and ending timer and record
end
Return vector of times
```

Essentially, the data is split into n folds. The CV process is then run on the n folds, and the process timed. The whole process is repeated r times to obtain a distribution of times. The number of CV folds, n also ultimately controlled the level of distribution since there were n `foreach` iterations and therefore n tasks sent to the Redis server. It is also worth noting that the specific parameters of the `xgboost` model (as well as calculating the RMSE of the predictions on the validation set) are inconsequential. The focus was on whether or not distribution is viable, and indeed beneficial, and so the trade-off between problem complexity and number of timed iterations was weighted towards more iterations.

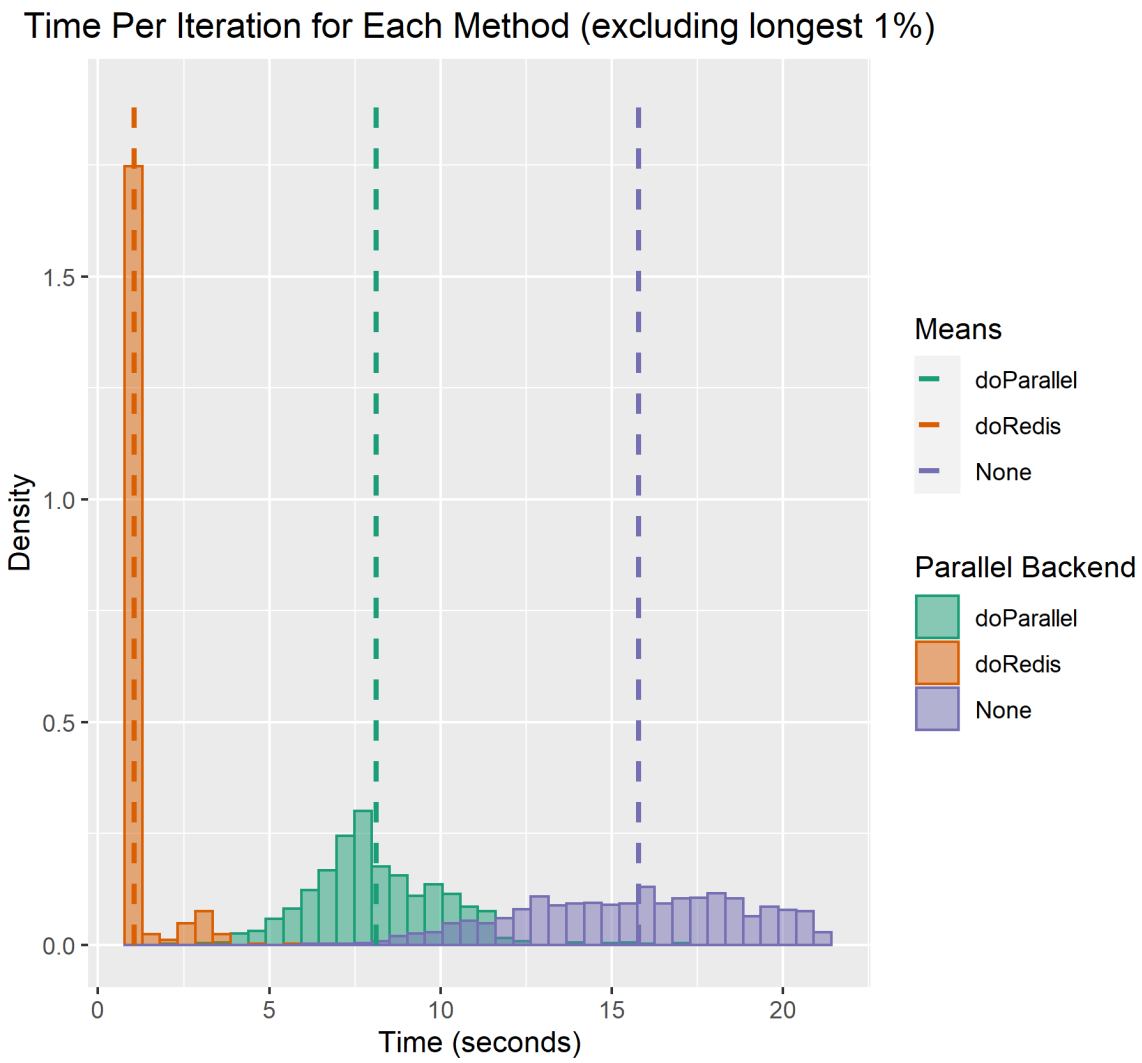

Figure 5.3: Densities of the time taken per iteration for each of the three parallel backends when applied to cross validation, shown along with their respective means.

Figure 5.3 shows the densities of the time taken per iteration of the CV process, again excluding the longest one per cent. Again, and not surprisingly, `doRedis` offered an improvement in time taken overall, against both `doParallel` and running the CV process sequentially. Interestingly though, the `doRedis` times were noticeably less variable than either of the other two parallel backends, which is in fairly stark contrast to the outcome of the t-tests depicted in Figure 5.1. The `doRedis` backend resulted in a standard deviation of 0.6132, the `doParallel` a standard devitaion of 1.9052, and the sequential computations a standard deviation of 3.3493. There are a few potential reasons for this, including the simple fact that distributing calculations involves sending tasks to another computer to run and so any number of differences between the 'master' node and the 'slave' node could introduce this variability.

Whatever the reason, one can clearly see that using `doRedis` was beneficial and saved time. Again, this is to be expected since the calculations were distributed to a more powerful machine, however this section mostly serves to indicate that using `doRedis` is viable for, and can be applied to, ML problems and contexts. CV is a frequently used tool in ML problems and Figure 5.3 shows that it is indeed viable to distribute CV calculations.

Furthermore, it is not a particularly difficult exercise to program CV calculations so that they can be distributed. While many R packages come with CV built in, it is very simple to do it manually, making use of `foreach` loops to allow for distribution.

5.4 Elasticity of `doRedis` Demonstration

This section aims to demonstrate one of the key advantages of using `doRedis` to distribute ML problems: its elasticity. As mentioned in Section 2.4.5 above, `doRedis` allows for an elastic pool of nodes such that the user can attach and detach nodes as needed. Figure 5.4 below depicts it in action.

Cross validation was chosen to demonstrate the elasticity. In the same way as in Section 5.3 above, the training of an `xgboost` model for each of the n folds was distributed to the Redis server. However, this time n was set to ten folds and `xgboost`'s `nrounds` parameter, which sets the number of decision trees in the final model, was set to 1000 (as opposed to fifty in Section 5.3). n and `nrounds` were increased to better simulate the application of distributing ML problems. Whereas Section 5.3 was more a 'proof of concept', attaching more nodes to the network allows for the user to introduce greater complexity into the model as there are more machines available to pick up tasks from the server.

At random times throughout the running of the code, additional nodes were attached to the network. These nodes were started on different machines using the `startLocalWorkers` function from `doRedis`. Furthermore, the `chunkSize` was set to three with the `setChunkSize` function.

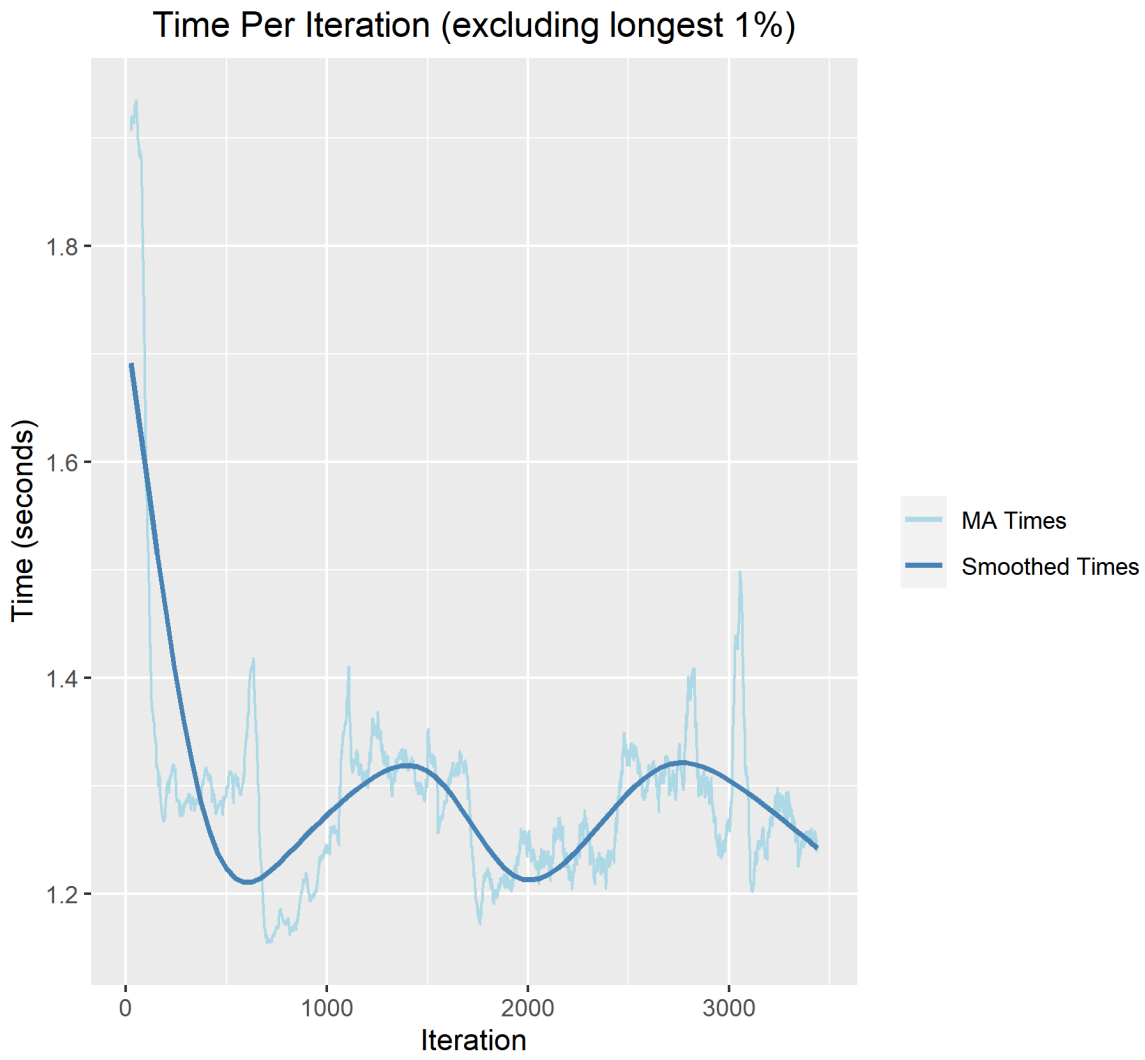

Figure 5.4: Time taken for each iteration. Show here are the moving average times, overlayed with a smoothed curve of the moving averages.

Figure 5.4 shows the time taken for each of the 3 500 iterations. In order to make the figure more interpretable, the moving average (with a window of fifty iterations) and a smoothed curve of the moving averages are shown. Again, to avoid outliers making the figure difficult to interpret, the longest one per cent of times are removed.

There is clearly some variation among the time per iteration data. The first noticeable 'drop' in time taken per iteration roughly coincides with when an additional node was attached to the server, and so we expect the time taken per iteration (i.e. the whole CV process) to decrease as it is the training of the `xgboost` model on each fold that is distributed.

Since the `chunkSize` is set to three, each node will pick up three folds to fit an `xgboost` model to. This provides a clear example of the consideration needed when distributing problems (mentioned in Section 4.3). Each timed iteration involves ten `foreach` loops that will be distributed, and a group of three of them will form a task. Now, since the timing process of this benchmarking was sequential, that means that there are only four tasks (three tasks of three `foreach` loops and one task of one `foreach` loop) available on the server per iteration. Thus, attaching more than four nodes is unlikely to prove significantly beneficial [1].

Although only two 'wavelengths' appear visible, it does appear that the time taken per iteration is cyclical in nature. A potential reason for this is that the nodes are different machines with different capabilities, and so some nodes are more powerful than others. In this case, looking at the moving average times provides more clarity. We can see that there appears to be intervals with relatively longer times per iteration, followed by intervals with relatively shorter times per iteration. This supports the argument that a particular combination of nodes is faster than a different combination of nodes, and the faster combination of nodes had picked up the majority (if not all) of the tasks from the server when the times appear to 'dip'.

[1] It could be argued that more nodes may still provide benefit but that is only likely to be the case if the computational power of the nodes varies, in which case greater benefit would be had by simply attaching the 'stronger' nodes only.

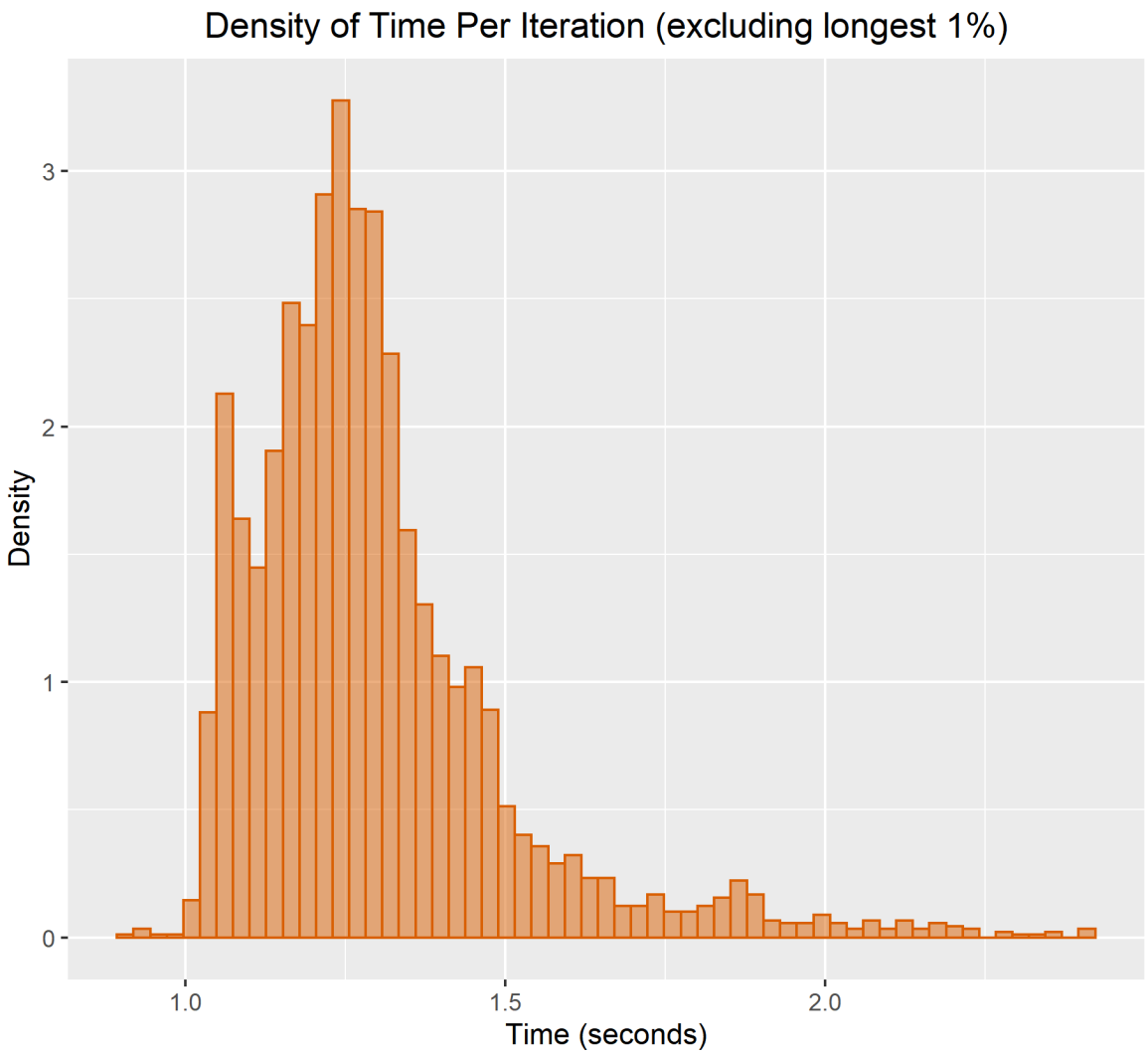

Figure 5.5: Density of the time taken for each iteration.

Figure 5.5 shows the density of the times taken for each iteration. This allows us to get a better idea of the variation present in the times, and there clearly is some variation present (the collection of times taken per iteration has a standard deviation of 0.5275 seconds). This is one reason that Figure 5.4 above shows the moving average and not the raw times.

Of course, some level of variation is to be expected. Attaching different nodes to a distributed network inherently implies some sort of variation because each node will handle the calculations that it picks up at a slightly different rate. Attaching a number of different nodes also results in different combinations of nodes picking up the tasks that make up a single timed iteration, and so there really are a number of different aspects of the distribution process that can introduce variation.

Chapter 6

Conclusion

This research ultimately aimed to show that distributed computing is more accessible than generally perceived to be. It does not require a particularly steep learning curve, nor does it incur significant financial costs (for example, to rent computing power online).

Furthermore, the ongoing growth and popularity of machine learning techniques has increased the number of students who are studying these models and algorithms. Naturally, one would need to practice applying them to data while studying them, and this often requires a certain level of computational power. This research showed that it is possible to leverage already-available computational resources, in the form of computer labs or the like, to create a small-scale distributed network of nodes.

Particularly in R, there are a number of packages that allow for calculations to be distributed, however some offer distinct advantages over others. `doRedis` is one such package that provides a parallel backend to `foreach` calls in R, and sends them to a Redis server. Nodes can then be attached to the server to fetch, compute, and return tasks. Two advantages of using `doRedis` are that it is relatively easy to set up and intuitive to adapt one's algorithms to leverage it, and that it is elastic, handling nodes attaching and detaching well. Since `doRedis` sends tasks to a Redis server to be distributed, it stands to reason that one could open up the Redis server online, allowing nodes to be attached and detached from anywhere with an internet connection (although this is inherently risky from an internet safety perspective, and so should not be undertaken lightly). Furthermore, the controlling functions of `doRedis` allow for one to attach nodes in a limited capacity so as not to fully use their computing power, and therefore someone could continue to use these nodes while they fetch and run tasks from the Redis server in the background.

The next step is recognising that some common aspects of ML algorithms and models are suitable for distribution. Indeed, certain whole ML algorithms can be distributed, as is this case with random forests. However, the well-known random forest packages in R are often 'black-boxes' and so one might have to code their own random forests from scratch in order to distribute them fully. Other aspects of ML can also be easily implemented, such as cross validation. CV is perfectly parallelisable and very commonly used, and is a use case for distributing machine learning.

Finally, the clear benefits of distributing computations were demonstrated in Chapter 5. It is evident that there can be a clear time saving when distributing computations but it is also clear that there are certain considerations that one must be aware of before using `doRedis` to distribute computations. In particular, thought must go into the `chunkSize` specified as it is the primary control for just how many of the computations are distributed at a time. This is a potential area for future projects as finding the optimal `chunkSize` might not always be straightforward. There is also space to analytically determine the communications overhead present when distributing with `doRedis`, although that is likely to be highly dependent on

the hardware and network infrastructure being used.

6.1 Future Work and Recommendations

More research into the optimal `chunkSize` for a variety of ML problems would prove valuable. `chunkSize` is a useful option in `doRedis` that, effectively, allows the user to specify the frequency with which each node fetches a new task from the server. Decreasing this frequency may reduce the communications overhead, but could also leave some nodes idle while others are overworked. As mentioned in Section 4.3, there are other considerations when setting the `chunkSize`. Research into the specifics of these considerations, with a focus on choosing the `chunkSize` that leads to the greatest improvement in model training times in a variety of ML applications, would prove useful.

A further step in using `doRedis` as a parallel backend would be allowing the Redis server to connect to the internet. This would allow the user to attach nodes from anywhere with a stable internet connection, removing the limitation present in this research that the nodes and server all be connected to the same local network. Indeed, this is possible since Redis allows for connections via the internet. However, allowing connections from the internet creates significant safety concerns and, if not done correctly, and with the necessary knowledge and caution, opens the server, and therefore the entire network that the server is connected to, up to malicious activity. However, the ability to attach any computer with an internet connection as a node has clear value. Future work on opening the server up to connections from the internet in a safe, secure manner would be beneficial.

Appendix A

Hardware

This section in the appendix outlines the hardware used for this research:

Table A.1

Use	**Machine**	**Hardware Specifications**	**Operating System**
Redis server; 'master' node	Lenovo ThinkCentre M90p	Intel Core i5-650; 6GB RAM	Ubuntu 20.04
Node 1	ASUS Zenbook 14	Intel Core i7-8565U; 16GB RAM	Windows 10 Pro - 64-bit
Node 2	MacBook Air (M1, 2020)	Applie M1; 8GB RAM	macOS 12.1
Node 3	Lenovo ThinkPad X1 Yoga	Intel Core i7-6600U; 16 GB RAM	Windows 10 Pro - 64 bit
Network hardware	tp-link Archer C20	–	–

Appendix B

Using `doRedis`

Having described the various options for distributed computing (and machine learning) in R, this section will outline the process of implementing a small-scale distributed network in R using `doRedis`. There will then be a brief benchmarking exercises to demonstrate the viability. Lastly, this section will discuss some examples of distributed machine learning in R.

B.1 Setting Up a Redis Server

The first step in setting up a small-scale distributed network is setting up a Redis server. As mentioned previously, it is recommended that Redis is run on a Linux machine. Ubuntu is a free, open source Linux operating system (Debian Installer Team, 2020) that is easy to dual-boot with Windows. There are many different installations but just as many resources available online to assist with installation of Ubuntu.

Following the installation of Ubuntu, Redis needs to be installed and correctly set up on Ubuntu. The simplest way to do this is to run `sudo apt-get install redis-server` from a terminal. Having installed Redis, there are a few options that need to be changed in Redis to allow `doRedis` to function optimally. These are mentioned in the `doRedis` documentation (Lewis, 2021b) but involve changing the following in Redis's *redis.conf* file:

- Change *daemonize no* to *daemonize yes*
- Change *timeout 300* to *timeout 0*

B.2 Installing R, RStudio and `doRedis`

Next, an installation of R (and optionally, RStudio) is needed. Thankfully, R is very popular and so is easy to install. The process is outlined well by Asaad (2013).

`doRedis` also requires the `libhiredis-dev` package to be installed on Ubuntu. `libhiredis-dev` is a "minimalistic C client library for the Redis database" (Lamb, 2021). This package can be installed in an Ubuntu terminal using the command `sudo apt-get install libhiredis-dev`.

Lastly, we are ready to install `doRedis` in R. It is recommended to install this package directly from its GitHub repository (which requires the `remotes` package to do) using `remotes ::install_ github(''bwlewis/doRedis'')`.

B.3 A Basic Example

In order to test the setup described above, we can run a simple Monte Carlo approximation of π. This is a common example used to demonstrate parallel computing and can easily be

distributed since the calculations are independent of one another. As a high-level overview, one machine will be the 'master' node and another machine will be the 'slave' node. The calculation is initiated on the master node and the slave node is simply attached to the Redis server using `doRedis`'s built-in functions. The master will distribute each `foreach` loop to the server and the slave will retrieve them, run the calculations, and then return the results to the server. The master then combines the results in the manner specified in the `foreach` function.

First, the job queue needs to be set up on the Redis server with the `registerDoRedis()` function:

```
library(doRedis)
registerDoRedis(queue = "jobs",
    host = "<machine name of server>",
    progress = T)
```

If the Redis server was set up with a password, the `pass` argument must also be supplied to `registerDoRedis()`.

A useful command in Redis is to run `redis-cli --stat` in a terminal on the machine that is running the Redis server. This will display some information such as number of keys, number of clients and number of connections, and a new line is printed every second. It is particularly useful in checking whether the `registerDoRedis` function has successfully created the queue (and whether a new node has attached or not).

What remains is to a) run the calculation, and b) direct any slave nodes to fetch jobs from the server. These can be done in any order (but cannot be done before registering the queue as above).

The calculation, which is well explained by Easterday and Smith (1991), is as follows:

```
foreach(i=1:10,.combine=sum,.multicombine=TRUE,.inorder=FALSE) %dopar% {
    4*sum((runif(1000000)^2 + runif(1000000)^2)< 1)/10000000
}
```

Directing a node to retrieve jobs from the server and calculate them can be done using `startLocalWorkers()` or `redisWorker()`. The differences between these methods is outlined briefly in Section 4.2 further on. For now, we will use `redisWorker()`:

```
redisWorker(queue = "jobs",
    host = "<machine name of server>")
```

If all has gone well, the progress bar in the master's R console should increase as the calculations are returned by the nodes to the server (and the approximation of π should show in the console of RStudio, of course).

Bibliography

Almasi, George S and Allan Gottlieb (1994). *Highly parallel computing*. Benjamin-Cummings Publishing Co., Inc.

Asaad, Al-Ahmadgaid (Mar. 2013). *Download and Install R in Ubuntu*. Online. URL: https://www.r-bloggers.com/2013/03/download-and-install-r-in-ubuntu/.

Bansal, Sanjay, Sanjeev Sharma, and Ishita Trivedi (2011). "A Detailed Review of Fault-Tolerance Techniques in Distributed System". In: *International Journal on Internet and Distributed Computing Systems* 1.1. ISSN: 2327-7157.

Bengtsson, Henrik (Aug. 2021a). *Package 'future'*. URL: https://cran.r-project.org/web/packages/future/future.pdf.

– (Dec. 2021b). *supportsMulticore function (parallelly)*. URL: https://cran.r-project.org/web/packages/sparklyr/sparklyr.pdf.

Bergstra, James and Yoshua Bengio (Feb. 2012). "Random Search for Hyper-Parameter Optimization". In: *Journal of Machine Learning Research* 13, pp. 281–305.

Berrar, Daniel (2019). *Cross-validation*. Online.

Boehmke, Bradley and Brandon Greenwell (Feb. 2020). *Hands-On Machine Learning with R*. URL: https://bradleyboehmke.github.io/HOML/.

Carlson, Josiah (2013). *Redis In Action*. E-book. Simon and Schuster. URL: https://redis.com/ebook/appendix-a/a-3-installing-on-windows/a-3-1-drawbacks-of-redis-on-windows/.

Debian Installer Team (2020). *What is Ubuntu?* Online. URL: https://help.ubuntu.com/lts/installation-guide/s390x/ch01s01.html.

Easterday, Kenneth and Tommy Smith (1991). "A Monte Carlo application to approximate pi". In: *The Mathematics Teacher* 84.5, pp. 387–390.

Eddelbuettel, Dirk (2021). "Parallel computing with R: A brief review". In: *Wiley Interdisciplinary Reviews: Computational Statistics* 13.2, e1515.

Fay, Colin (Jan. 2019). *An Introduction to Docker for R Users*. Online. URL: https://colinfay.me/docker-r-reproducibility/.

Firesmith, Donald (Aug. 2017). *Multicore Processing*. Software Engineering Institute. URL: https://insights.sei.cmu.edu/blog/multicore-processing/.

Genuer, Robin et al. (2017). "Random Forests for Big Data". In: *Big Data Research* 9, pp. 28–46. ISSN: 2214-5796. DOI: https://doi.org/10.1016/j.bdr.2017.07.003. URL: https://www.sciencedirect.com/science/article/pii/S2214579616301939.

He, Tong (Nov. 2021). *Package 'xgboost'*. URL: https://cran.r-project.org/web/packages/xgboost/xgboost.pdf.

James, Gareth et al. (2017). *An Introduction to Statistical Learning with Applications in R*. 8th ed. Springer.

Kuhn, Max (Oct. 2021). *Package 'caret'*. URL: https://cran.r-project.org/web/packages/caret/caret.pdf.

Lamb, Chris (2021). *Package: libhiredis-dev (0.14.1-2)*. Online. URL: https://packages.debian.org/sid/libhiredis-devPackage:libhiredis-dev(0.14.1-2).

Lewis, Bryan W. (n.d.). *doRedis Repository*. [Online repository]. URL: github.com/bwlewis/doRedis.

– (Aug. 2021a). *Elastic computing with R and Redis*. URL: https://cran.r-project.org/web/packages/doRedis/vignettes/doRedis.pdf.

– (Aug. 2021b). *Package 'doRedis'*. URL: https://cran.rstudio.com/web/packages/doRedis/doRedis.pdf.

Li, Yitao (Mar. 2021). *Package 'sparklyr'*. URL: https://cran.r-project.org/web/packages/sparklyr/sparklyr.pdf.

Malony, Allen D (n.d.). *CIS 410/510: Introduction to Parallel Computing*. URL: https://ipcc.cs.uoregon.edu/curriculum.html.

Mitchell, Tom M. (1997). *Machine Learning*. McGraw-Hill Education.

Neiswanger, Willie, Chong Wang, and Eric Xing (2013). "Asymptotically exact, embarrassingly parallel MCMC". In: *arXiv preprint arXiv:1311.4780*.

Patrizio, Andy (2020). *The coronavirus pandemic turned Folding@Home into an exaFLOP supercomputer*. URL: arstechnica.com/science/2020/04/how-the-pandemic-revived-a-distributed-computing-project-and-made-history/. [2021, February 21].

Prasad, Kisalaya, Avanti Patil, and Heather Miller (2016). "Progamming Models for Distributed Computing". In: Unpublished. Chap. Futures and Promises.

R Core Team (2021). *R: A Language and Environment for Statistical Computing*. R Foundation for Statistical Computing. Vienna, Austria. URL: https://www.R-project.org/.

Raychaudhuri, Samik (2008). "Introduction to monte carlo simulation". In: *2008 Winter simulation conference*. IEEE, pp. 91–100.

Rodriguez-Galiano, Victor et al. (June 2016). "Modelling interannual variation in the spring and autumn land surface phenology of the European forest". In: *Biogeosciences* 13, pp. 3305–3317. DOI: 10.5194/bg-13-3305-2016.

Rosenblatt, Jonathan D (Oct. 2019). *R (BGU course)*. URL: http://www.john-ros.com/Rcourse/.

Rouse, Margaret (Mar. 2007). *Definition: multi-core processor*. Online. URL: https://searchdatacenter.techtarget.com/definition/multi-core-processor.

Salloum, Salman et al. (2016). "Big data analytics on Apache Spark". In: *International Journal of Data Science and Analytics* 1.3, pp. 145–164.

Schmidberger, Markus et al. (Aug. 2009). "State of the Art in Parallel Computing with R". In: *Journal of Statistical Software* 31.1. URL: https://www.jstatsoft.org/article/view/v031i01.

Singh, Harshdeep (Nov. 2018). "Understanding Gradient Boosting Machines". In: *Towards Data Science*. Online. URL: https://towardsdatascience.com/understanding-gradient-boosting-machines-9be756fe76ab.

Steen, Maarten van and Andrew S Tanenbaum (2016). "A brief introduction to distributed systems". In: *Computing* 98.

University of Cape Town's ICTS High Perfomance Computing Team (2019). *Introduction to LInux for HPC*. Online. URL: http://hpc.uct.ac.za/db/Introduction_to_Linux_for_HPC.pdf.

– (Feb. 2022). *Performance Graphs*. Online. URL: http://hpc.uct.ac.za/cacti/.

Venters, Will and Edgar A Whitley (2012). "A critical review of cloud computing: researching desires and realities". In: *Journal of Information Technology* 27.

Wallig, Michelle et al. (Oct. 2020). *Package 'doParallel'*. URL: https://cran.r-project.org/web/packages/doParallel/doParallel.pdf.

Weerts, Hilde J. P., Andreas C. Mueller, and Joaquin Vanschoren (2020). *Importance of Tuning Hyperparameters of Machine Learning Algorithms*. DOI: 10.48550/ARXIV.2007.07588. URL: https://arxiv.org/abs/2007.07588.

Wright, Marvin N. and Andreas Ziegler (2017). "ranger: A Fast Implementation of Random Forests for High Dimensional Data in C++ and R". In: *Journal of Statistical Software* 77.1, 1–17. DOI: 10.18637/jss.v077.i01. URL: https://www.jstatsoft.org/index.php/jss/article/view/v077i01.

Zaharia, Matei (2016). *An architecture for fast and general data processing on large clusters*. Morgan & Claypool.

Zhang, Daniel et al. (Mar. 2021). *The AI Index 2021 Annual Report*. AI Index Steering Committee, Human-Centered AI Institute, Stanford University. URL: https://aiindex.stanford.edu/wp-content/uploads/2021/11/2021-AI-Index-Report_Master.pdf.