Journal of Statistical Software
*March 2017, Volume 77, Issue 3.* *doi: 10.18637/jss.v077.i03*

# wsrf: An R Package for Classification with Scalable Weighted Subspace Random Forests

**He Zhao**
SIAT

**Graham J. Williams**
SIAT

**Joshua Zhexue Huang**
Shenzhen University

**Abstract**

We describe a parallel implementation in R of the weighted subspace random forest algorithm (Xu, Huang, Williams, Wang, and Ye 2012) available as the **wsrf** package. A novel variable weighting method is used for variable subspace selection in place of the traditional approach of random variable sampling. This new approach is particularly useful in building models for high dimensional data – often consisting of thousands of variables. Parallel computation is used to take advantage of multi-core machines and clusters of machines to build random forest models from high dimensional data in considerably shorter times. A series of experiments presented in this paper demonstrates that **wsrf** is faster than existing packages whilst retaining and often improving on the classification performance, particularly for high dimensional data.

*Keywords*: **wsrf**, weighted random forests, scalable, parallel computation, big data.

## 1. Introduction

The random forest algorithm (Breiman 2001) is an ensemble method based on building an ensemble of decision trees. In many situations it is unsurpassed in accuracy compared to many other classification algorithms (Caruana and Niculescu-Mizil 2006) and is widely used in data mining (Díaz-Uriarte and De Andres 2006; Bosch, Zisserman, and Muoz 2007; Cutler *et al.* 2007; Williams 2011; Kandaswamy *et al.* 2011; Rodriguez-Galiano, Ghimire, Rogan, Chica-Olmo, and Rigol-Sanchez 2012; Touw *et al.* 2013). However, when facing high-dimensional data, and especially sparse data, the method of randomly selecting variables for splitting nodes (i.e., subspace selection) requires attention. Using a purely random selection from among very many variables (features), as in the traditional random forest algorithm, can mean that informative variables have too little opportunity for being selected from amongst the large number of less informative variables available. This will result in weak trees constituting the forest and consequently the classification performance of the random forest will be adversely affected.

Amaratunga, Cabrera, and Lee (2008) proposed a variable weighting method for subspace sampling, where the weight of a variable is computed from the correlation between the variable and the class using a $t$-test for the analysis of variance. This weight is treated as the probability of that variable being chosen for inclusion in a subspace. Consequently, informative variables are more likely to be selected when growing trees for a random forest, resulting in an increase in the average strength of the trees making up the forest. Xu *et al.* (2012) generalized Amaratunga's method using information gain ratio for calculating the variable weights, so as to be applicable to multi-class problems.

The R software environment for statistical computing and graphics (R Core Team 2016b) provides several packages for building random forests among which **randomForest** (Liaw and Wiener 2002) and **party** (Hothorn, Bühlmann, Dudoit, Molinaro, and Van Der Laan 2006a; Strobl, Boulesteix, Zeileis, and Hothorn 2007; Strobl, Boulesteix, Kneib, Augustin, and Zeileis 2008) are the more commonly referenced. **randomForest** is derived from the original Fortran code implementing the algorithm described by Breiman (2001). **party** includes `cforest` as an implementation of Breiman's random forest based on conditional inference trees (Hothorn, Hornik, and Zeileis 2006b).

Extensive experience in using these packages for building classification models from high dimensional data with tens of thousands of variables and hundreds of thousands or millions of observations has demonstrated both computation time and memory issues. Yet, such big data continues to become increasingly common with our growing ability to collect and store data. Neither supports any variation to variable selection, nor methods for improving on their computational requirements for building random forests.

In this paper we present the implementation of **wsrf** – a random forest algorithm based on the weighted subspace random forest algorithm for classifying very high-dimensional data (Xu *et al.* 2012). As well as being applicable to multi-class problems this algorithm will build models comparable in accuracy to Breiman's random forest when using a fixed subspace size of $\lfloor \log_2(M) + 1 \rfloor$ where $M$ is the total number of variables (Breiman 2001). However the time taken to build random forests using these sequential algorithms can be excessive.

In order to minimize the time taken for model building and to fully make use of the potential of modern computers to provide a scalable, high performance, random forest package in R, **wsrf** adopts several strategies. Taking advantage of R's capabilities for parallel computation we make use of multiple cores on a single machine as well as multiple nodes across a cluster of machines. Experimental comparisons presented in this paper demonstrate significant performance gains using **wsrf**.

During the development of **wsrf** similar packages with support for parallel execution have become available, including **bigrf** (Lim 2014), **ParallelForest** (Ieong 2014) and **Rborist** (Seligman 2016). Whilst these developments provide useful additions to the landscape we identify time and memory issues with them as we note later in the paper.

The paper is organized as follows: In Section 2 we briefly introduce Breiman's random forest algorithm and provide a detailed description of the weighted subspace random forest algorithm. Section 3 describes our parallel implementation of the algorithm. In Section 4 various experiments are carried out on several datasets used by Xu *et al.* (2012) to verify our implementation's validity and scalability. The usage of **wsrf** is presented in Section 5 with a simple example providing a template for users to quickly begin using **wsrf**. Comments on installation options are provided in Section 6. We conclude with a short review and identify future work in Section 7.

## 2. Weighted subspace random forest

In this section we briefly introduce Breiman's original random forest algorithm. The weighted subspace random forest algorithm is then described and presented in comparison to the original random forest algorithm. We assume a basic knowledge of decision tree building and related terminology (see Williams 2011, Chapter 11).

### 2.1. The random forest algorithm

The original random forest algorithm (Breiman 2001) is based on building an ensemble (i.e., a collection) of decision tree models. The concept of building multiple decision trees and combining them into a single model originates from the research of Williams (1988) in multiple inductive learning and the MIL algorithm. When combining decision trees into an ensemble to be deployed as a classification model the class assigned to a new observation is the class that the majority of trees assign to the observation.

The key development of the random forest algorithm is the random selection of both observations and variables throughout the tree building process. For each tree we slightly vary the training dataset and quite dramatically vary the variables used:

- The training dataset for constructing each tree in the forest is obtained as a random sample with replacement of the original dataset;
- Instead of using all $M$ variables as candidates when selecting the best split for any node during tree building, a subset (i.e., a subspace) $m \ll M$ is chosen at random. An empirical value for $m$, used by Breiman (2001), is $\lfloor \log_2(M) + 1 \rfloor$;
- There is no pruning in the process of tree building and so the resulting trees will, individually, over-fit the training dataset.

We see that the random forest algorithm samples both observations and variables from the entire training data. Therefore, all trees within the forest are potentially quite different and each tree potentially models the training dataset from quite different points of view, resulting in high diversity across the trees. Diversity works to our advantage in increasing the overall accuracy of the resulting ensemble model.

### 2.2. Variable weighting for subspace selection

When presented with very high dimensional data where typically only a small number of variables are informative of the class, the random subspace sampling method of random forests will randomly select those informative variables relatively rarely. Consequently a variable subspace of size larger than $\lfloor \log_2(M)+1 \rfloor$ is often needed and chosen. This has computational implications on the algorithm and potentially reduces the variety in the resulting trees.

In order to build decision tress with improved performance Xu *et al.* (2012) proposed a variable weighting method for subspace selection. In the method the informativeness of a variable with respect to the class is measured by an information gain ratio. The measure is used as the probability of that variable being selected for inclusion in the variable subspace when splitting a specific node during the tree building process. Therefore, variables with higher values by the measure are more likely to be chosen as candidates during variable selection and a stronger tree can be built.

Specifically, given a dataset $D$ with $val$ observations, let $Y$ be the target variable with distinct class labels $y_j$ for $j = 1, \ldots, q$. Consider a categorical variable $A$ in the dataset $D$ with $p$ distinct values $a_i$ for $i = 1, \ldots, p$. The number of observations in $D$ satisfying the condition that $A = a_i$ and $Y = y_j$ is denoted $val_{ij}$. Then, a contingency table of $A$ against $Y$ can be obtained, where we denote the marginal totals for $A$ by $val_{i.} = \sum_{j=1}^{q} val_{ij}$ for $i = 1, \ldots, p$, and the marginal totals for $Y$ by $val_{.j} = \sum_{i=1}^{p} val_{ij}$ for $j = 1, \ldots, q$.

The information measure of dataset $D$, which measures the class purity in $D$, is defined as:

$$Info(D) = -\sum_{j=1}^{q} \frac{val_{.j}}{val} \log_2 \frac{val_{.j}}{val} \tag{1}$$

while the information measure of a subset $D_{A=a_i}$ is defined as:

$$Info(D_{A=a_i}) = -\sum_{j=1}^{q} \frac{val_{ij}}{val_{i.}} \log_2 \frac{val_{ij}}{val_{i.}} \tag{2}$$

and the weighted sum of the information entropy for all subsets based on $A$ is:

$$Info_A(D) = -\sum_{i=1}^{p} \frac{val_{i.}}{val} Info(D_{A=a_i}) \tag{3}$$

The information gain ratio is then defined as:

$$IGR(A, Y) = \frac{Gain(A)}{SplitInfo(A)} \tag{4}$$

where

$$Gain(A) = Info(D) - Info_A(D) \tag{5}$$

and

$$SplitInfo(A) = -\sum_{i=1}^{p} \frac{val_{i.}}{val} \log_2 \frac{val_{i.}}{val} \tag{6}$$

Assume dataset $D$ has $M$ variables $A_1, A_2, \ldots, A_M$. The weighted subspace random forest method uses $IGR(A, Y)$ (Equation 4) to measure the informativeness of these variables and considers them as variable weights. These are normalized to become the probability of a variable being selected:

$$w_i = \frac{\sqrt{IGR(A_i, Y)}}{\sum_{i=1}^{M} \sqrt{IGR(A_i, Y)}} \tag{7}$$

Compared with Breiman's method we again note that the difference here is the way in which the variable subspace is selected at each node. Breiman's method selects $m \ll M$ variables at random from which a variable is then chosen for splitting the node. The weighted subspace random forest method extracts the same size subspace but using $w_i$ (Equation 7) as the probabilities for the inclusion of each of the variables. Also for the implementation of **wsrf** the trees are based on C4.5 (Quinlan 1993) rather than CART (Breiman, Friedman, Olshen, and Stone 1984) and so binary splits are applied to continuous variables while categorical variables are $k$-way split.

## 3. A parallel implementation

Today's computers are equipped with CPUs with multiple cores. To fully utilize the potential for high performance computing **wsrf** has adapted existing mature multi-threaded and distributed computing techniques to implement the weighted subspace random forest algorithm. This can significantly reduce the time for creating a random forest model from high dimensional large datasets.

There are many packages offering high performance computing solutions in R. These include **multicore** (Urbanek 2011)[1] for running parallel computations on machines with multiple cores or CPUs; **snow** (Tierney, Rossini, Li, and Sevcikova 2016) that can use PVM, MPI, and NWS as well as direct networking sockets to distribute computation across several nodes; and **RHadoop** (Piccolboni 2013), a collection of three R packages that allow users to manage and analyze data with Hadoop. A list of similar packages is collected by Eddelbuettel (2017).

The random forest algorithm inherently lends itself to being implemented in parallel, in that all trees can be built independently of each other. Our implementation of **wsrf** provides two opportunities for parallelism: multi-threaded and distributed. When run on a single machine with multiple cores the multi-threaded version of **wsrf** is the default for model building. In a cluster of servers the distributed version can be used to distribute the tree building tasks across the multiple nodes. Below we provide a detailed description of the options.

We also note that our algorithm for building one tree in **wsrf** is implemented in C++ with a focus on the effective use of time and space. This ensures we can optimize computational time compared to an implementation directly in R.

### 3.1. Multi-threaded

The traditional random forest method grows one tree at a time, sequentially. However, whilst the trees within a single random forest are built independently of each other, on machines with multiple cores only one core is allocated while others are idle. If we can use multiple cores, building one tree at a time on each core, then multiple trees can be built simultaneously. This can significantly reduce the model building time.

For a multi-threaded implementation two questions need to be answered: how to engage multiple cores and how to ensure each tree building process is independent. For the first question we use multiple threads and for the second we find that we need to deal with the issue of independent random number generation.

By default, the multi-threaded version of **wsrf** creates the same number of threads as available cores to build trees. When there is only one core it degrades to a sequential random forest algorithm. All but one of the threads are used to build one tree at a time in parallel. The remaining thread is reserved as the master thread tasked with distributing the tree building,

[1] **multicore** is removed from CRAN during the review process of this paper, and now part of R via the **parallel** package (R Core Team 2016b).

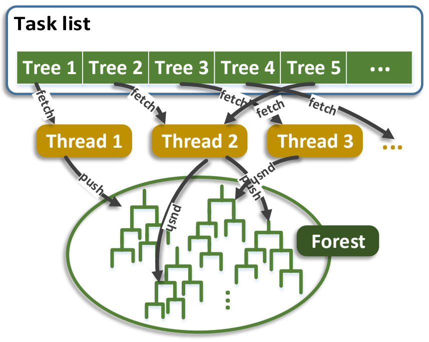


Figure 1: Multi-threaded **wsrf** will dispatch each tree build to an available core on a single machine/node. The resulting trees are accumulated by the master into the final forest.

generating training sets, providing random seeds, as well as handling exceptions and user interruptions. Once a tree build is finished the tree building thread returns that tree to the master and fetches another available training set to build the next tree, until all required trees are built. Figure 1 visualizes this process.

Many contributed R packages use multiple threads at the C programming language level via OpenMP (The OpenMP Architecture Review Board 2016) or Pthreads (IEEE and The Open Group 2004). Either approach requires additional installation setup for the users of the packages and present issues around portability. Modern C++ compilers supporting the C++11 standard with multi-threaded functionality are now available[2].

We use the **Rcpp** package (Eddelbuettel and François 2011; Eddelbuettel 2013) to access the multi-thread functionality of C++ and to ensure the functionality is easily portable without user intervention. **Rcpp** provides R functions as well as C++ library access which facilitates the integration of R and C++. The C++11 standard library provides a set of classes and functions for multi-threaded synchronization and error handling, of which **wsrf** uses `async` for creating threads, `future` for querying thread status, and `mutex` for dealing with race conditions.

When compiling **wsrf** newer versions of the C++ standard library are required. Unlike GNU/Linux, on Microsoft Windows the build environment and toolchain do not come standard with the operating system. The **Rtools** package (R Core Team 2016a) is the only tool that is supported and recommended by R Core Team (2016b) to build R packages from source. Unfortunately there is no support for multi-threading in the version of g++ provided by **Rtools**[3] and so the multi-threading functionality of **wsrf** is not readily available on Windows[4]. For older versions of g++ (before 4.8.1) **wsrf** provides an option for multi-threading

[2]See the Internet news items: *GCC 4.8.1 released, C++11 feature complete* (`https://isocpp.org/blog/2013/05/gcc-4.8.1-released-c11-feature-complete`) and *LLVM 3.3 is released* (`https://isocpp.org/blog/2013/06/llvm-3.3-is-released`).

[3]Because GCC 4.6.x is used on Windows as mentioned in *Writing R Extensions – Using C++11 code* `https://CRAN.R-project.org/doc/manuals/r-release/R-exts.html#Using-C_002b_002b11-code`

[4]Support for compiling C++11 code in packages is experimental in R-devel but not yet finished, see *Daily News about R-devel on 2013-12-02* (`https://developer.R-project.org/blosxom.cgi/R-devel/NEWS/2013/12/02#n2013-12-02`) and *MinGW-w64 Notes by Duncan Murdoch* (`https://rawgit.com/kevinushey/RToolsToolchainUpdate/master/mingwnotes.html`).

using the Boost C++ library (Boost Community 2016).

Another issue is random number generation (RNG) in C++ when there are multiple threads running in parallel. Functions such as `rand` from C++ `<cstdlib>` behave differently in different operating systems. In **wsrf** the `random` library in C++11 or Boost is used to obtain independent pseudo-random number sequences. The task of making sure different trees have different initial RNG seeds is handled in R. When invoked with a specification of the number of trees **wsrf** generates the same number of random different seeds – the tree seeds. The tree building procedures then use their own seed as passed from the master in R in order to generate their own random number sequences. Thus, as desired, for a specific initial seed, initialized in R using `set.seed()`, the same model will always be built.

## 3.2. Distributed computation

We implement a distributed version of **wsrf** by building trees on different machines (nodes) across a cluster. The multi-threaded version of **wsrf** is employed on each node, thus increasing the level of parallelism.

In a distributed environment the interaction between nodes becomes a problem and there are many ways for dealing with it: through a direct connection using sockets; indirect communication using MPI; or through using Hadoop for data-intensive computations. As for our multi-threaded version of **wsrf** we implement a simple and robust approach to deploying distributed computation using R's **parallel** package.

The **parallel** package (R Core Team 2016b) provides parallelism at many different levels. It is adopted by **wsrf** for distributed computing in R. The crucial point when using **parallel** is that every chunk of computation should be independent and should not communicate with each other. Noting that each tree-building task is independent of any other tree-building task the natural chunk of computation is one tree-building task.

The main steps for the distributed version of **wsrf** are then:

1. According to the parameters passed by the user the master node calculates the number of trees to be built on each worker node of the identified cluster;
2. The related data, parameters and number of trees is communicated to the corresponding node to initiate the tree-building procedure on each node of the cluster;
3. The master node collects the results from all worker nodes to reduce the result into a single random forest model.

The process is illustrated in Figure 2.

There are two alternatives to controlling the number of threads to be used on each node in the cluster. By default, as in Section 3.1, the user only needs to specify which nodes of a cluster to use and **wsrf** will use as many threads as there are available cores to build the model. Alternatively, the number of threads on each node can be specified as a named vector as in `c(hostname1 = 10, hostname2 = 15)` which requests 10 threads on host `hostname1` and 15 threads on `hostname2`.

Once again the generation of random numbers is an issue across distributed nodes. The same approach as presented for the multi-threaded version is used except that now the seeds are passed to the nodes.

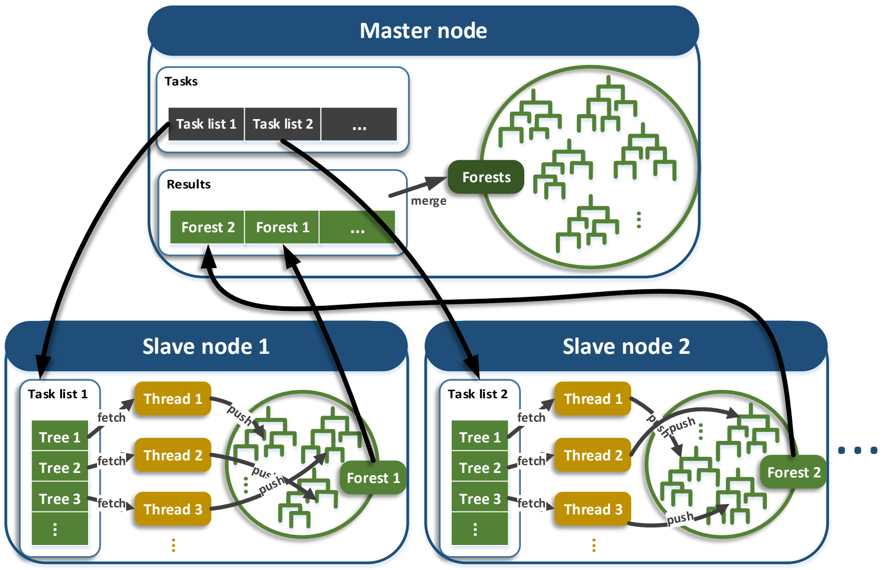


Figure 2: Distributed **wsrf**.

## 4. Experimental comparisons

We now present various experiments with three purposes in mind – to ascertain accuracy, elapsed-time performance and memory usage. We first determine that similar results are obtained from our scalable version of **wsrf**, confirming the validity of our scalable **wsrf**. We then compare the elapsed-time performance of **wsrf** to related packages, and explore its scalability.

### 4.1. Datasets

The datasets for the experiments are the same as used by Xu *et al.* (2012). All are high dimensional datasets from image recognition, bioinformatics and text analysis. Table 1 provides an overview of the datasets with the size for the corresponding CSV (comma-separated values) file of each dataset and ordered by the proportion of observations to variables.

The datasets `gisette` (Guyon 2013) and `mnist` (Roweis 2013) are an encoding of handwritten

| Dataset | Observations | | Variables | $\sqrt{Vars}$ | $\lfloor \log_2(Vars) + 1 \rfloor$ | Obs/Vars (train) | Classes | Size (MB) |
|---|---|---|---|---|---|---|---|---|
| | Train | Test | | | | | | |
| `wap` | 1,104 | 456 | 8,460 | 91 | 14 | 0.13 | 20 | 18.0 |
| `la1s` | 1,963 | 887 | 13,195 | 114 | 14 | 0.15 | 5 | 50.0 |
| `la2s` | 1,855 | 845 | 12,432 | 111 | 14 | 0.15 | 5 | 45.0 |
| `re1` | 1,147 | 510 | 3,758 | 61 | 12 | 0.31 | 25 | 8.3 |
| `fbis` | 1,711 | 752 | 2,000 | 44 | 11 | 0.86 | 17 | 6.6 |
| `gisette` | 5,000 | 999 | 5,000 | 70 | 13 | 1.00 | 2 | 54.0 |
| `newsgroups` | 11,268 | 7,504 | 5,000 | 70 | 13 | 2.25 | 20 | 108.0 |
| `tis` | 5,200 | 6,875 | 927 | 30 | 10 | 5.61 | 2 | 9.3 |
| `mnist` | 48,000 | 10,000 | 780 | 27 | 10 | 61.54 | 2 | 84.0 |

Table 1: Summary of the datasets used in the experiments.

| Package | Version | Published date | Model building function |
|---|---|---|---|
| **wsrf** | 1.5.14 | 2014-06-09 | `wsrf` |
| **rpart** | 4.1.8 | 2014-03-28 | `rpart` |
| **randomForest** | 4.6.10 | 2014-07-17 | `randomForest` |
| **party** | 1.0.19 | 2014-12-18 | `cforest` |
| **bigrf** | 0.1.11 | 2014-05-16 | `bigrfc` |
| **doParallel** | 1.0.8 | 2014-02-28 | — |
| **bigmemory** | 4.4.6 | 2013-11-18 | — |

Table 2: Summary of R packages.

digit images. Each observation of `gisette` describes either the handwritten digit "4" or "9". All observations of mnist are grouped into two classes corresponding to the digits 0 to 4 and 5 to 9 respectively. The datasets `fbis`, `la1s`, `la2s`, `re1`, `wap` (Karypis 2013) and `newsgroups` (Rennie 2013) are all text data that can be grouped into multiple categories. Dataset `tis` (Li 2013) is biomedical data which records the translation initiation sites (TIS) at which the translation from a messenger RNA to a protein sequence was initiated.

The original datasets can be obtain from the links provided in the references. Note that the datasets `fbis`, `la1s`, `la2s`, `re1` and `wap` are available as a single archive file download. The actual datasets used here are subsets of the originals and can be downloaded from Xu (2015).

All datasets are split into two parts, one for training the model and the other for testing and from which the classification error rates are derived.

### 4.2. Experimental setting

Experiments are conducted to compare **wsrf** with the other common random forest packages in R: **randomForest** and **party**. Comparisons with one of the parallel random forests package **bigrf** are also included[5]. Table 2 lists the corresponding versions of these packages with their dependant packages which may have influence over the performance, plus the functions for building models in the experiments. Specifically, two extra packages are included, **doParallel** (Analytics and Weston 2015) and **bigmemory** (Kane, Emerson, and Weston 2013), on which **bigrf** depends. As a base comparison, the results of a single decision tree built from the R package **rpart** (Therneau, Atkinson, and Ripley 2015) will also be provided.

The experiments were carried out on servers having an Intel Xeon CPU (E5620) with 16 cores running at 2.40 GHz, with 32GB RAM, running Ubuntu 12.04 LTS, GCC 4.9.2 and R 3.1.2. The distributed experiments were performed on a cluster of 10 nodes with the same configuration for each node.

[5]The other two packages mentioned in Section 1 cannot fulfil the needs of our experiments. **ParallelForest** (Version 1.1.0) can only be used for binary classification, and fails on `tis` – the smallest two-class dataset in our experiments; **Rborist** (Version 0.1.0) fails even on the small example of **ParallelForest** at *Introduction to the* ***ParallelForest*** *Package* (`https://CRAN.R-project.org/web/packages/ParallelForest/vignettes/ParallelForest-intro.html`). We should also note that **bigrf** (Version 0.1.11) does not pass all the experiments as mentioned in Section 4.3, and during the review process of this paper **bigrf** was removed from CRAN due to check problems on 2015-11-21.

The experiments are separated into 4 parts:

1. Build a single decision tree to benchmark basic tree building performance for each package;
2. Build forests varying the numbers of trees from 30 to 300 with a step of 30 and so 10 trials in total;
3. Build forests of 100 trees but varying the size of the subspace (i.e., number of variables) from 10 to 100 with a step of 10 and so 10 trials in total;
4. Using **wsrf** build a forest of 100 trees whilst varying the number of threads from 1 to 19 and so 10 trials in total and then the number of computer nodes from 1 to 10 and so 10 trials in total.

All parameters use the default values of the corresponding model building functions in those packages unless otherwise stated. The size of the subspace is controlled by the parameter `mtry` in `randomForest` and `bigrfc` (`mtry`= $\sqrt{M}$ by default in the packages where $M$ is the number of variables within the data) and `cforest` (`mtry`= 5 by default in the package), with `nvars` as a synonym available in `wsrf` (which is `mtry`= $\lfloor \log_2(M) + 1 \rfloor$ by default). For `bigrfc` we use the recommended **doParallel** as the back end for its parallelism and `big.matrix` is used by default. The number of cores used in parallel computing by `wsrf` is the number of available cores minus 2 (which is 14 in our experiments) so we set the same on `bigrfc` for comparison.

In presenting the results, `wsrf`_t represents `wsrf` running on a single machine with multi-threading and `wsrf`_c represents `wsrf` distributed over a cluster of 10 nodes. For a more reasonable comparison with `randomForest` and `bigrfc` we include the results for `wsrf`_nw and `wsrf`_seq_nw which represent `wsrf` without weighting and `wsrf` without weighting and computed serially rather than in parallel, respectively. Both are with `mtry`= $\sqrt{M}$. Unless otherwise stated `wsrf` represents our weighted subspace random forests running on a single machine with multi-threading.

All experiments for **rpart**, **wsrf** and **bigrf** over the datasets `fbis`, `mnist` and `tis` were repeated 10 times with different random seeds to improve the stability of the estimates of performance. The rest were carried out only once because of the time taken to complete a single run. As Table 5 shows `cforest` from **party** was taking over 2 days to build one tree on dataset `la1s` compared to less than 1 minute for `wsrf`. Hence, the curves corresponding to these particular experiments exhibit more variance.

## 4.3. Experimental results

Extensive experimentation has been carried out and in this section we report on three measures: classification accuracy, elapsed time and memory consumption. We briefly overview the results first and then drill down into the details for each of the three measures.

*Overview*

Table 5 presents the results for the comparative single tree building mentioned in Section 4.2. The purpose here is to compare the accuracy, run time and memory usage when building a

| | `cforest` | `randomForest` | `bigrf` | `wsrf_t` | `wsrf_c` | `wsrf_nw` | `wsrf_seq_nw` |
|---|---|---|---|---|---|---|---|
| *Accuracy* | | | | | | | |
| `wap` | 22.1–22.1 | 78.3–**84.2** | **78.3**–81.6 | 77–79.6 | — | 74.9–77.9 | — |
| `la1s` | 32.2–32.2 | 84.8–86.1 | 84.2–85.6 | **85.9**–**86.8** | — | 81.4–83.4 | — |
| `la2s` | 31.7–31.7 | 85.1–**88.2** | 86.8–87.9 | **87.2**–88.2 | — | 84–85.4 | — |
| `re1` | 21.2–21.2 | 81.1–82.4 | 81.7–82.2 | **83.3**–**83.9** | — | 81.4–82 | — |
| `fbis` | 46.1–48.4 | 79.9–81.8 | 80.1–81.7 | **83.1**–**84.4** | — | 82.1–83.3 | — |
| `gisette` | 84.1–88.1 | 96.2–**97** | **96.4**–96.6 | 96–96.4 | — | 95.8–96.1 | — |
| `newsgroups` | 6.9–14.6 | 66.3–70.6 | 66.6–69 | **69.4**–**71.2** | — | 68.5–70.4 | — |
| `tis` | 75.5–75.5 | 89.2–90.3 | 89.1–90.2 | **90.2**–**90.4** | — | 84.8–85.9 | — |
| `mnist` | 79.2–83.4 | 97–97.5 | 96.9–97.2 | **97.1**–**97.5** | — | 96.8–97.4 | — |
| *Time* | | | | | | | |
| `wap` | 8h–8h | 8h–8h | 15s–1m | 42s–6m | 29s–1m | **2s**–**5s** | 4s–29s |
| `la1s` | 2d–2d | 2d–2d | 35s–4m | 57s–8m | 48s–1m | **5s**–**9s** | 8s–39s |
| `la2s` | 1d–1d | 1d–2d | 30s–2m | 50s–7m | 44s–1m | **4s**–**8s** | 7s–35s |
| `re1` | 21m–23m | 22m–26m | 8s–2m | 19s–3m | 15s–28s | **1s**–**3s** | 2s–19s |
| `fbis` | 3m–3m | 3m–5m | 5s–2m | 10s–1m | 11s–18s | **0s**–**2s** | 2s–15s |
| `gisette` | 1h–1h | 1h–2h | 21s–19m | 47s–7m | 42s–1m | **2s**–**7s** | 5s–48s |
| `newsgroups` | 1h–1h | 2h–6h | 8m–2h | 12m–2h | 4m–14m | **11s**–**1m** | 1m–12m |
| `tis` | 27s–50s | 43s–6m | 5s–6m | 8s–1m | 12s–18s | **0s**–**3s** | 2s–20s |
| `mnist` | 5m–44m | 6m–53m | 38s–1m | 3m–26m | 1m–4m | **5s**–**43s** | 34s–5m |
| *Memory* | | | | | | | |
| `wap` | 9G–20G | 2G–2G | 1015M–3G | **254M**–**288M** | — | 291M–326M | 291M–326M |
| `la1s` | 87G–390G | 6G–7G | 2G–4G | 617M–665M | — | **595M**–643M | 597M–**624M** |
| `la2s` | 76G–331G | 4G–6G | 2G–3G | 544M–585M | — | 533M–**582M** | **529M**–606M |
| `re1` | 712M–3G | 263M–359M | 493M–3G | **44M**–**77M** | — | 59M–84M | 60M–83M |
| `fbis` | 560M–3G | 118M–181M | 360M–2G | **1M**–**26M** | — | 20M–49M | 19M–49M |
| `gisette` | 16G–202G | 1G–2G | 2G–2G | **10M**–**34M** | — | 25M–56M | 13M–54M |
| `newsgroups` | 78G–334G | 3G–4G | 5G–6G | 122M–259M | — | 108M–249M | **104M**–**248M** |
| `tis` | 1014M–3G | 148M–212M | 347M–660M | **1M**–**25M** | — | 7M–38M | 5M–38M |
| `mnist` | 60G–251G | 2G–3G | 3G–3G | 16M–**154M** | — | 19M–215M | **8M**–215M |

Table 3: Minimum and maximum accuracy, elapsed time and memory used in building a forest of trees whilst varying the number of trees (30, 60, 90, . . . , 300) to include in the forest. The best results for each dataset are in boldface, the worst are underlined. Since with the same set of random seeds the results on accuracy of `wsrf_c` and `wsrf_t` are the same, the results for `wsrf_c` are not presented, so are `wsrf_seq_nw`.

single decision tree with any one of the algorithms. The original single decision tree algorithm (`rpart`) generally performs the best in terms of accuracy. Note that this is compared to a single un-pruned decision tree from the random forest algorithms – forests with many decision trees are generally more accurate than a single decision tree. The primary comparison is for time taken and memory used where we see the benefit of parallel model building and the efficient use of memory, particularly with **wsrf**.

Table 3 provides a broad view of the three measures for each of the random forest algorithms being compared as the number of trees built increases. The corresponding accuracy and elapsed time are plotted in Figure 3. Note that memory usage for `wsrf_c` is not provided in the table as it is not comparable to the others which only run on a single machine and the results of `wsrf_t` are indicative. Due to the performance issues with `cforest`, as will be mentioned in the following sub-sections, we do not report its results in the plots. The time performance of `randomForest` is also significantly inferior to that of the others and so it is not included in the plots for elapsed time. Since we use the same set of random seeds the accuracy of `wsrf_c` and `wsrf_t` are identical and thus not repeated in the table and so for

| | cforest | randomForest | bigrf | wsrf_t | wsrf_c | wsrf_nw | wsrf_seq_nw |
|---|---|---|---|---|---|---|---|
| *Accuracy* | | | | | | | |
| wap | 22.1–22.1 | 66.9–80.7 | 66.3–**80.9** | **76.9**–80.3 | — | 49.5–77.2 | — |
| la1s | 32.2–45.7 | 72.2–85.7 | 71.2–85.1 | **84.8**–**86.7** | — | 32.2–82.1 | — |
| la2s | 31.7–49.1 | 73.4–87.2 | 73.2–87.4 | **85.9**–**88.2** | — | 31.7–84.2 | — |
| re1 | 21.2–28 | 62.1–**84.1** | 60.9–83.7 | **82**–83.7 | — | 45.3–82.5 | — |
| fbis | 48.8–50.2 | 75.6–83.2 | 76.1–82.9 | **79.9**–**84** | — | 75–84 | — |
| gisette | 90.9–93.2 | 95.8–**96.9** | 95.8–96.8 | **96**–96.5 | — | 95–96 | — |
| newsgroups | 11.3–24.2 | 68.1–70.6 | **68.4**–68.4 | 68.3–**71.2** | — | 52.9–70.3 | — |
| tis | 75.6–89.8 | 85.7–**91.1** | 85.1–91.1 | **90.1**–90.4 | — | 75.6–90.4 | — |
| mnist | 88.8–92.7 | **97**–**97.5** | 97–97 | 96.7–97.4 | — | 96.6–97.4 | — |
| *Time* | | | | | | | |
| wap | 8h–8h | 8h–8h | 28s–31s | 2m–2m | 33s–35s | **2s**–**3s** | 2s–12s |
| la1s | 2d–2d | 2d–2d | 1m–1m | 2m–3m | 53s–55s | **4s**–**6s** | 4s–14s |
| la2s | 1d–1d | 1d–1d | 1m–1m | 2m–2m | 48s–49s | **4s**–**5s** | 4s–13s |
| re1 | 22m–23m | 23m–24m | 14s–16s | 45s–54s | 16s–18s | **0s**–**2s** | 1s–11s |
| fbis | 3m–4m | 3m–3m | 11s–11s | 28s–31s | 12s–13s | **0s**–**2s** | 1s–12s |
| gisette | 1h–1h | 1h–1h | 48s–53s | 2m–2m | 47s–48s | **1s**–**4s** | 3s–23s |
| newsgroups | 1h–2h | 3h–3h | 5h–5h | 37m–38m | 5m–5m | **3s**–**44s** | 17s–5m |
| tis | 39s–2m | 2m–2m | 12s–13s | 24s–25s | 13s–13s | **0s**–**3s** | 2s–22s |
| mnist | 17m–1h | 17m–25m | 2m–2m | 9m–11m | 2m–2m | **5s**–**1m** | 35s–9m |
| *Memory* | | | | | | | |
| wap | 9G–22G | 1G–2G | 2G–2G | **260M**–**274M** | — | 295M–301M | 292M–301M |
| la1s | 91G–386G | 6G–7G | 3G–3G | 599M–**632M** | — | 598M–645M | **596M**–646M |
| la2s | 72G–296G | 4G–6G | 3G–3G | 532M–571M | — | **529M**–554M | 529M–**549M** |
| re1 | 805M–3G | 252M–350M | 1G–1G | **58M**–**63M** | — | 62M–67M | 61M–67M |
| fbis | 783M–1G | 93M–153M | 953M–1G | **1M**–**5M** | — | 26M–28M | 26M–28M |
| gisette | 18G–107G | 1G–2G | 2G–2G | **9M**–**24M** | — | 18M–36M | 17M–49M |
| newsgroups | 77G–321G | 2G–3G | 8G–8G | 148M–166M | — | **108M**–155M | 111M–**155M** |
| tis | 2G–2G | 132M–178M | 525M–556M | **1M**–**6M** | — | 8M–16M | 7M–16M |
| mnist | 93G–130G | 2G–2G | 3G–3G | **43M**–**53M** | — | 56M–84M | 55M–84M |

Table 4: Minimum and maximum accuracy, elapsed time and memory used in building a forest of trees whilst varying the number of variables (10, 20, 30, . . . , 100) randomly selected at each node. The best results for each dataset are in boldface, the worst are underlined. Since with the same set of random seeds the results on accuracy of `wsrf_c` and `wsrf_t` are the same, the results for `wsrf_c` are not presented, so are `wsrf_seq_nw`.

`wsrf_nw` and `wsrf_seq_nw`.

Table 4 and Figure 4 provide similar results but varying the number of variables to randomly select at each node whilst building a decision tree.

Figure 5 considers just **wsrf** and plots the run-time performance using different number of threads and nodes for model building as described in Section 4.2.

In order to highlight the best and the worst performing of the modellers over a specific datasets the tabular results will emphasize the best performance (i.e., the minimum elapsed time and memory used and the maximum accuracy) using a bold face. We also underline the worst performances (i.e., the maximum elapsed time and memory usage, and the minimum accuracy) for each specific dataset.

## *Classification accuracy*

The results for building a single tree using **rpart** are included in Table 5 for comparison. Of course, **rpart** is generally used to build a single tree model rather than multiple trees to be

| | `rpart` | `cforest` | `randomForest` | `bigrf` | `wsrf` | `wsrf_nw` |
|---|---|---|---|---|---|---|
| *Accuracy* | | | | | | |
| `wap` | **63** | 22 | 48 | 51 | 56 | 53 |
| `la1s` | **75** | 32 | 63 | 63 | 71 | 59 |
| `la2s` | **78** | 33 | 63 | 64 | 73 | 62 |
| `re1` | **77** | 23 | 57 | 56 | 70 | 60 |
| `fbis` | **70** | 38 | 56 | 56 | 62 | 59 |
| `gisette` | **92** | 56 | 90 | 90 | 89 | 88 |
| `newsgroups` | 32 | 9 | 37 | 39 | **52** | 50 |
| `tis` | **89** | 76 | 79 | 79 | 85 | 78 |
| `mnist` | 79 | 81 | 88 | 88 | **90** | 88 |
| *Time* | | | | | | |
| `wap` | 48s | 8h | 8h | 9s | 12s | **2s** |
| `la1s` | 2m | 2d | 2d | 18s | 16s | **4s** |
| `la2s` | 2m | 1d | 1d | 16s | 14s | **4s** |
| `re1` | 19s | 23m | 21m | 4s | 5s | **0s** |
| `fbis` | 14s | 3m | 2m | 3s | 2s | **0s** |
| `gisette` | 1m | 1h | 59m | 10s | 12s | **1s** |
| `newsgroups` | 6m | 1h | 1h | 30s | 3m | **3s** |
| `tis` | 15s | 25s | 12s | 2s | 2s | **0s** |
| `mnist` | 2m | 2m | 54s | 12s | 41s | **1s** |
| *Memory* | | | | | | |
| `wap` | 1G | 10G | 2G | 867M | **288M** | 288M |
| `la1s` | 3G | 97G | 6G | 2G | **586M** | 605M |
| `la2s` | 3G | 77G | 5G | 2G | 550M | **549M** |
| `re1` | 157M | 2G | 253M | 393M | **43M** | 45M |
| `fbis` | 53M | 477M | 112M | 274M | 17M | **17M** |
| `gisette` | 1G | 16G | 1G | 2G | 32M | **11M** |
| `newsgroups` | 2G | 78G | 3G | 4G | 102M | **102M** |
| `tis` | 29M | 697M | 176M | 309M | **2M** | 3M |
| `mnist` | 890M | 33G | 2G | 3G | **1M** | 1M |

Table 5: Comparison of accuracy/time/memory for a ***single decision tree*** from the tree/forest model building algorithms. The time is in seconds, minutes, hours, and days while accuracy is a percentage. The best results for each dataset are in boldface, the worst are underlined.

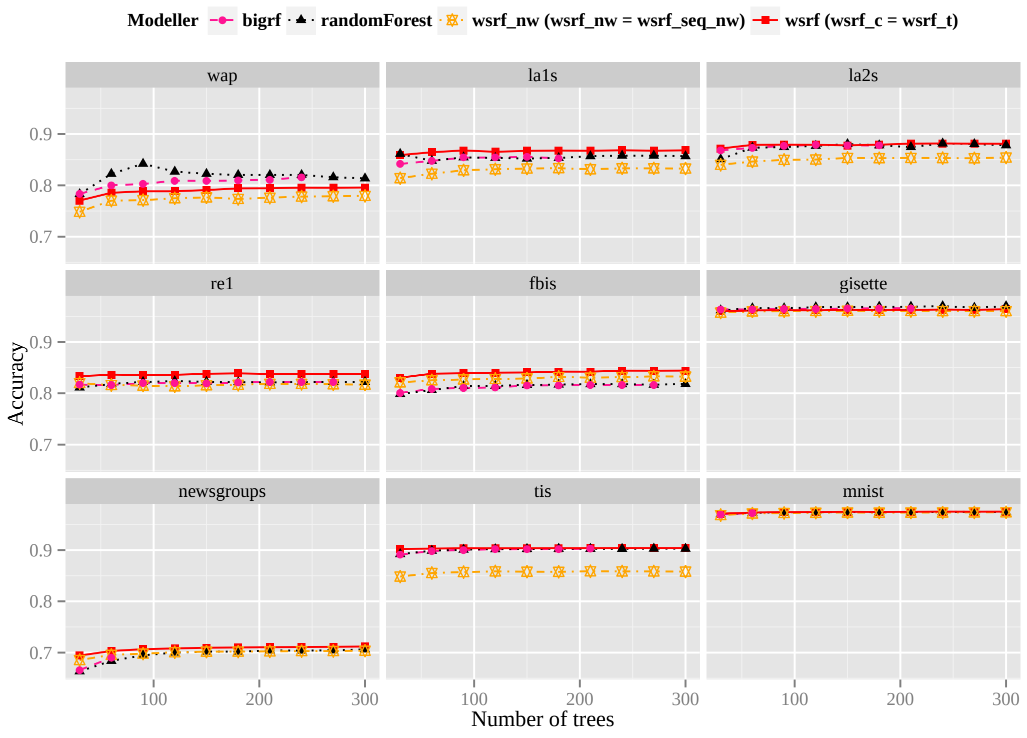
combined into an ensemble. It is important to note that, as mentioned in Section 2.1, no pruning is performed in the process of tree building for random forests. Therefore, a single tree built without pruning will usually suffer from over-fitting. Besides, all variables are selected in the node-splitting process by **rpart**, so we observe that **rpart** generally produces more accurate single tree models, with the exceptions for datasets `newsgroups` and `mnist`.

From Table 5 we also see that for a single tree from `cforest` the accuracy is quite poor, being less than 50% on all datasets except for `gisette`, `tis` and `mnist`. We concede that there may be opportunity for tuning the tree building parameters in `cforest`, but have not done so. Also from the results of varying the number of variables in Table 4 similar conclusions can be made.

The accuracy of a single decision tree from **wsrf** over all nine datasets is often quite comparable to the best accuracy of the other single decision tree builders.

We observe that **bigrf** has similar classification accuracy to **randomForest** from Table 5, Figure 3 and Figure 4. Both are based on Breiman's original random forests algorithm.

From Table 3 and the top collection of plots in Figure 3 we can see that **wsrf** have slightly better classification accuracy over all the datasets except `wap` as the number of trees changes.



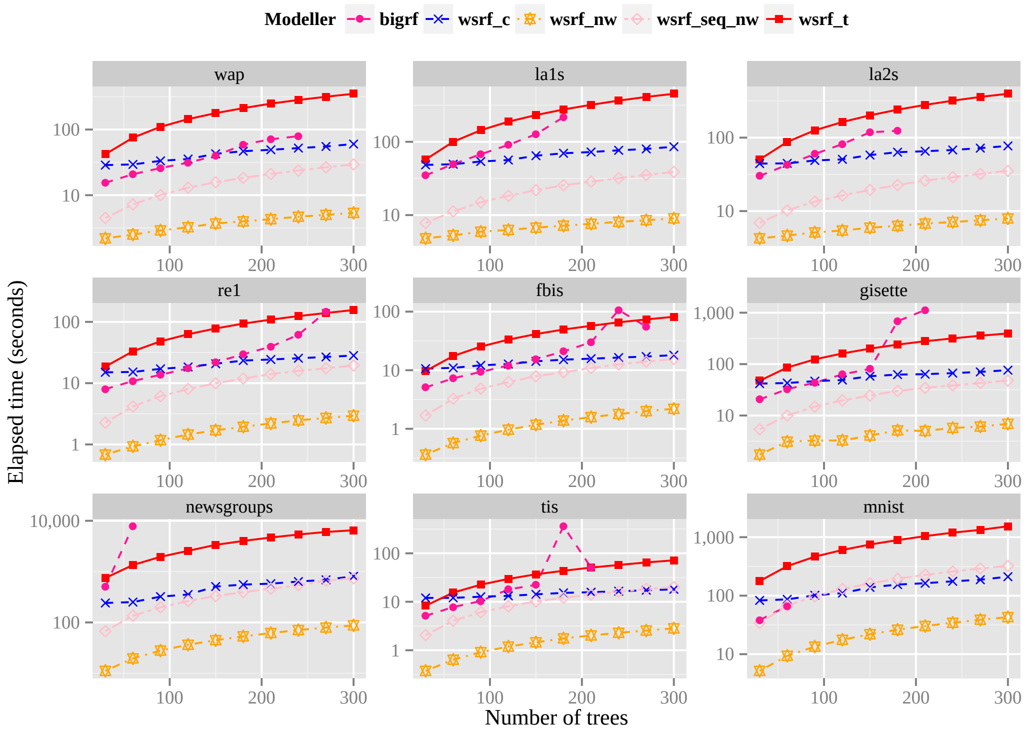


Figure 3: Comparison of accuracy and elapsed time across number of trees. Since with the same set of random seeds the results on accuracy of `wsrf`_c and `wsrf`_t are the same, `wsrf`_c and `wsrf`_t are collectively represented by **wsrf** in the legend of the top panel, so are `wsrf`_nw and `wsrf`_seq_nw.

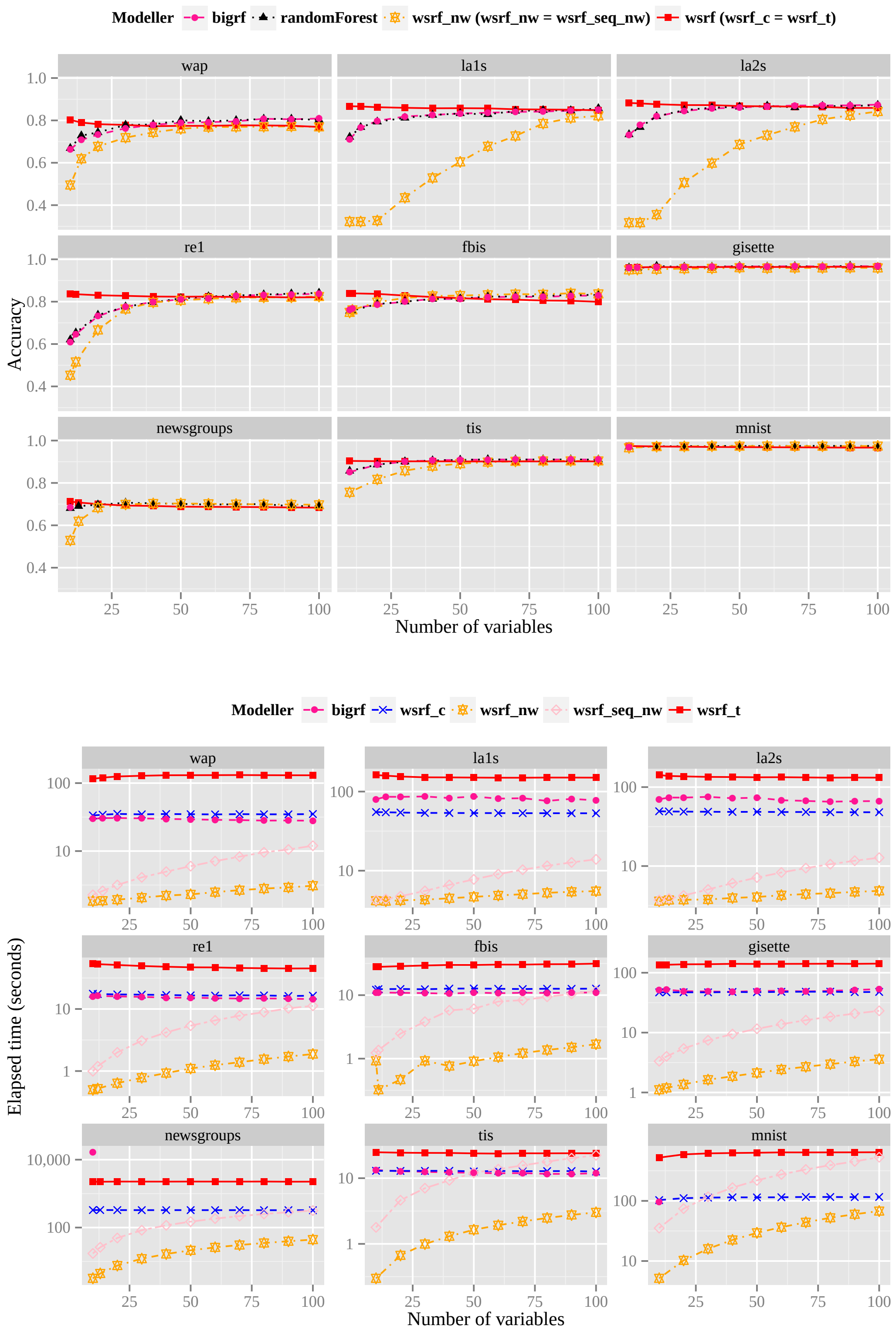


Figure 4: Comparison of accuracy and elapsed time across number of variables. Since with the same set of random seeds the results on accuracy of **wsrf**_c and **wsrf**_t are the same, **wsrf**_c and **wsrf**_t are collectively represented by **wsrf** in the legend of the top panel, so are **wsrf**_nw and **wsrf**_seq_nw.

Figure 4 and Table 4 similarly presents the experimental results based on varying the number of randomly selected variables. In short we can again observe that **wsrf** obtains high accuracy with fewer variables (and hence reduced effort) compared to other packages. Note the relatively poor performance results for **randomForest** and `wsrf_nw` when a smaller number of variables are chosen. This supports the observation we made early in the paper relating to the exclusion of informative variables when uniform random sampling is used. Informative variables are not regularly being included in the sample of variables selected by **randomForest** and `wsrf_nw` when we have a large number of variables to select from. The weighted subspace approach delivers a performance gain with a reduced amount of effort.

The accuracy plots presented in Figures 3 and 4 present many further opportunities to be explored in understanding the nature of building ensembles. The marginally, but consistently better performance of **randomForest** over **wsrf** for `wap` could shed some light on where weighted subspace selection may not offer any benefit. Most of the other datasets have more, and sometimes many more, variables than observations. The interplay between the number of observations and the number of variables has not been investigated here.

*Elapsed time*

In terms of elapsed time Table 5 demonstrates that `wsrf_nw` is the most efficient and **wsrf** generally takes less time than the other algorithms with the exceptions for `mnist` and `newsgroups`. We note that **wsrf** re-calculates the weights for each candidate variable for every node while growing a tree. As the number of observations increases the time taken will increase as the re-calculation of the measure is taking more observations into the formula. As mentioned in Section 3.1 the smallest parallel unit of **wsrf** is the single tree building process, so the timing for **wsrf** for single tree building in Table 5 is the minimum for model building.

Tables 3 and 4 demonstrate that `wsrf_nw` consistently delivers models more quickly. Rather than model building taking many minutes, hours or days, `wsrf_nw` takes only seconds in most cases. Even `wsrf_seq_nw` primarily takes seconds. Whilst `wsrf_t` takes up to 2 hours for `newsgroups` when building 300 trees (Table 3) the distributed version (`wsrf_c`) demonstrates the scalability of **wsrf**. The `cforest` and **randomForest** algorithms generally take considerably longer to build their models.

We can particularly compare the time taken for the `la1s` dataset. Here **wsrf** completes the task within a few minutes compared to `randomForest` and `cforest` taking 2 days to complete. Because a single tree build is the unit for parallel computation within **wsrf** we note that the least time required for growing a forest is the time taken for building one tree, plus a small overhead for distributing the work. The comparison between `wsrf_t` and `wsrf_nw` illustrates the cost of calculating weights in **wsrf**.

The lower collection of plots in Figures 3 and 4 demonstrate the considerable time advantage obtained by going parallel. Be aware that these plots have varying y-axis ranges and it should be noted that the y-axis of each plot uses a log scale so as to differentiate the lower timings. Hence time differences are even more dramatic than they may at first appear. Note also that the plots for **bigrf** are truncated as **bigrf** appears to not terminate for larger number of trees. The abrupt increases of elapsed time in the figures are also a testament to this. Its fail is also at times due to memory limitations.

Figure 5 provides insights into the benefit of parallelism over multiple threads and multiple nodes. A series of experiments were carried out by increasing the number of nodes (compute

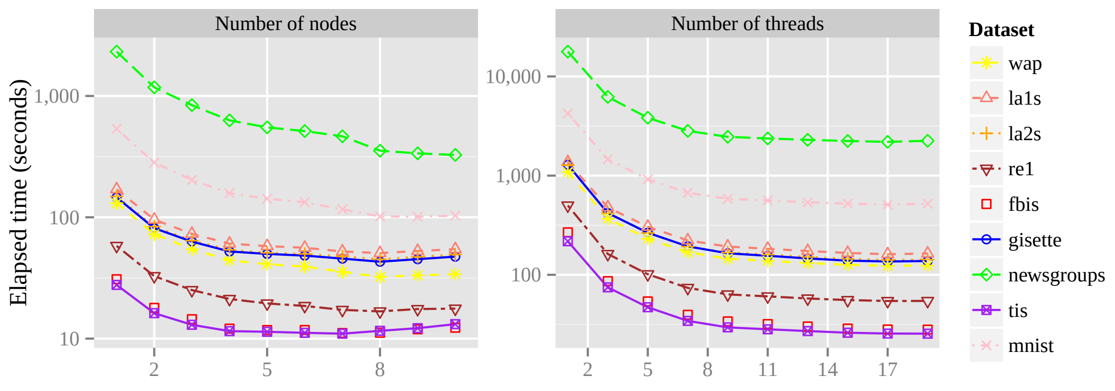


Figure 5: A comparison of the elapsed time for building **wsrf**-based random forests of 100 trees for each of the 9 datasets. Note the log scale on the y-axis.

servers) deployed and separately by increasing the number of threads deployed on a single node. Each point represents the time taken to build 100 decision trees for one forest averaged over 10 random forests. We can see that as the number of threads or nodes increases the running time of course reduces. We note again the log scale of the y-axis.

### *Memory consumption*

Table 5 suggests that although **rpart** can be a good choice in terms of classification accuracy and elapsed time, as we observed in Sections 4.3 and 4.3, the memory requirement for a single tree is quite significant.

Tables 5, 3 and 4 record that `cforest` requires hundreds of gigabytes and is a significantly higher requirement than other algorithms. This adversely impacts compute time of course as once available memory is exhausted the server will begin swapping to virtual memory. This underlies the significant elapsed compute time for `cforest`.

The measurements also record that the memory consumption of **randomForest** is an order of magnitude lower than that of `cforest`. Nonetheless **randomForest** takes a similar amount of time as `cforest` with no apparent virtual memory swapping. We observe that **wsrf** generally uses the least amount of memory among the algorithms of our experiments. It peaks at around 660MB across all 9 datasets and is well below the 32GB of RAM available on each node.

In the experiments each dataset is loaded into R before being passed on to the modellers. The memory recorded includes the memory required by the modeller and any additional copies of the datasets made by the implementations. We have used R's `Rprof`[6] for memory profiling and the sample timing interval will affect timings. A small interval provides more precise memory summaries but at the cost of compute time whilst a larger interval has minimal affect on elapsed time. Unfortunately the results suggest a smaller memory footprint than actually consumed. We experimented to find appropriate timing intervals for the different datasets and algorithms. Larger intervals were employed for the longer running algorithms and smaller

[6]See also *Writing R Extensions – Memory statistics from* `Rprof` (`https://CRAN.R-project.org/doc/manuals/r-release/R-exts.html#Memory-statistics-from-Rprof`).

intervals for smaller datasets and quicker algorithms. We can see slight inconsistency in the memory usage on smaller datasets as in the memory used by `wsrf_nw` (20M–49M) and `wsrf_t` (1M–26M).

## 5. Package overview and usage

We now demonstrate the use of **wsrf** by illustrating the utilization of both a single node with multiple cores and a cluster of machines each using multiple cores. We will use the small weather dataset from **rattle** (Williams 2011) to ensure the examples are readily repeatable. See the help page of **rattle** in R (`?weather`) for more details on the dataset.

We begin by loading the required packages into R, assuming they have been installed (using `install.packages()`).

```
R> library("rattle")
```

The first step is then to set up the dataset, identifying the target variable. The approach we use simplifies repeated running of any modelling experiments by recording general information in variables (parameters) that we can change in one place. Experiments are repeated by running the remainder of the code with minimal if any change. This streamlines both interactive and iterative model development.

```
R> ds <- weather
R> target <- "RainTomorrow"
R> nobs <- nrow(ds)
```

Once we have established the dataset it is always advisable to summarize the basic dataset information.

```
R> dim(ds)

[1] 366  24

R> names(ds)

 [1] "Date"          "Location"      "MinTemp"       "MaxTemp"
 [5] "Rainfall"      "Evaporation"   "Sunshine"      "WindGustDir"
 [9] "WindGustSpeed" "WindDir9am"    "WindDir3pm"    "WindSpeed9am"
[13] "WindSpeed3pm"  "Humidity9am"   "Humidity3pm"   "Pressure9am"
....
```

We would normally also review the `summary()`, `head()` and `tail()` and `str()` of the dataset, but we skip that here for brevity.

Before building the model we need to prepare the training dataset, specifically identifying the variables to ignore in the modelling. Identifiers and any output variables (RISK_MM is an output variable) should not be used (as independent variables) for modelling.

```
R> vars <- setdiff(names(ds), c("Date", "Location", "RISK_MM"))
R> dim(ds[vars])

[1] 366  21
```

Next we deal with missing values. Currently the C++ component of the implementation of **wsrf** does not support data with missing values. We take care of this in the R component of the implementation through the `na.action=` option of `wsrf()`. Alternatively we can take care of them in some way, for example using `na.roughfix()` from **randomForest** to impute values. We do so here for illustrative purposes but in general care needs to be taken in dealing with missing data.

```
R> ds[vars] <- randomForest::na.roughfix(ds[vars])
```

It is useful to review the distribution of the target variable.

```
R> table(ds[target])

 No Yes
300  66
```

We now construct the formula that describes the model to be built. The aim of the model is to predict the target based on all other variables.

```
R> (form <- formula(paste(target, "~ .")))

RainTomorrow ~ .
```

Randomly selected training and test datasets are constructed, setting a seed so that the results can be exactly replicated.

```
R> set.seed(49)
R> train <- sample(nobs, 0.7 * nobs)
R> test  <- setdiff(1:nobs, train)
R> length(train)

[1] 256

R> length(test)

[1] 110
```

We are now ready to build the model.

```
R> library("wsrf")
R> system.time(model.wsrf <- wsrf(form, data = ds[train, vars]))
```

```
   user  system elapsed
  1.345   0.027   0.132
```

```
R> model <- model.wsrf
```

A single tree, or indeed all 500 trees, can be printed using the usual `print()` command, with a summary on the first line providing the tree size and its specific error rate. Which tree(s) to print is managed by the `tree=` parameter.

```
R> print(model, tree = 1)

Tree 1 has 20 tests (internal nodes), with OOB error rate 0.1458:

 1) Temp9am <= 19.7
 .. 2) Temp9am <= 11.1
 .. .. 3) Humidity9am <= 88
 .. .. .. 4) Temp3pm <= 17.4   [No] (1 0) *
 .. .. .. 4) Temp3pm >  17.4
 .. .. .. .. 5) Sunshine <= 8.7   [No] (0.5 0.5) *
 .. .. .. .. 5) Sunshine >  8.7   [No] (1 0) *
 .. .. 3) Humidity9am >  88
 .. .. .. 6) Humidity3pm <= 72   [No] (1 0) *
 .. .. .. 6) Humidity3pm >  72   [Yes] (0.33 0.67) *
 .. 2) Temp9am >  11.1
 .. .. 7) Temp9am <= 17.5
 .. .. .. 8) WindGustSpeed <= 48
....
```

Compare this to the 500th decision tree.

```
R> print(model, tree = 500)

Tree 500 has 17 tests (internal nodes), with OOB error rate 0.2079:

 1) Sunshine <= 2
 .. 2) Cloud9am <= 7
 .. .. 3) Pressure9am <= 1013.8   [No] (0.5 0.5) *
 .. .. 3) Pressure9am >  1013.8   [No] (1 0) *
 .. 2) Cloud9am >  7
 .. .. 4) WindSpeed9am <= 6   [Yes] (0 1) *
 .. .. 4) WindSpeed9am >  6   [Yes] (0.33 0.67) *
 1) Sunshine >  2
 .. 5) Pressure3pm <= 1016
 .. .. 6) Sunshine <= 8.4
 .. .. .. 7) Sunshine <= 4.7   [No] (1 0) *
 .. .. .. 7) Sunshine >  4.7
 .. .. .. .. 8) Sunshine <= 6.1   [Yes] (0 1) *
....
```

Without the `tree=` parameter, an summary of the forest is presented.

```
R> print(model)
```

```
A Weighted Subspace Random Forest model with 500 trees.

  No. of variables tried at each split: 5
                 Out-of-Bag Error Rate: 0.14
                              Strength: 0.59
                           Correlation: 0.18

Confusion matrix:
     No Yes class.error
No  212   3        0.01
Yes  33   8        0.80
```

Notice how the error rate for the individual decision trees is higher than that for the overall ensemble model which we see below. This is a general observation of the performance of ensembles – the overall model is generally more accurate than any individual component model.

The summary also include a measure of the *Strength* (Equation 9 below) and the *Correlation* (Equation 10 below) as introduced by Breiman (2001) for evaluating a random forest model. *Strength* measures the collective performance of individual trees in a random forest and *Correlation* measures the diversity of the trees. A goal for ensemble model building is to maximize the strength of the component models and to minimize the correlations between the component models.

The *Strength* and *Correlation* are two measures we use to gain insight into the performance of the modeller. For a particular random forest let $h_k$ be the $k$th tree classifier built from the $k$th training dataset $D_k$. Each $D_k$ is sampled from the full dataset $D$ with replacement. The random forest model contains $K$ trees. The out-of-bag proportion of votes for predicted class $j$ for observation $d_i \in D$ is then the number of trees (the $h_k$) for which $d_i$ is out-of-bag (i.e., $d_i$ is not in the training dataset $D_k$ used to build the specific tree $h_k$) and the tree predicts the class to be $j$ for that observation $d_i$, divided by the number of times $d_i$ is out-of-bag across the $K$ decision trees. Essentially this is the out-of-bag score over the ensemble for predicting class $j$ for a specific observation $d_i$.

$$Q(d_i, j) = \frac{\sum_{k=1}^{K} I(h_k(d_i) = j, d_i \notin D_k)}{\sum_{k=1}^{K} I(d_i \notin D_k)} \tag{8}$$

We then define *Strength* as the average difference between the out-of-bag score for $d_i$ and the true class $y_i$, and the maximum out-of-bag score for all other classes predicted for $d_i$, over all $n$ observations in $D$.

$$Strength = \frac{1}{n}\sum_{i=1}^{n}(Q(d_i, y_i) - \max_{j \neq y_i} Q(d_i, j)) \tag{9}$$

*Correlation* is computed as:

$$Correlation = \frac{\frac{1}{n}\sum_{i=1}^{n}(Q(d_i, y_i) - \max_{j\neq y_i} Q(d_i, j))^2 - Strength^2}{(\frac{1}{K}\sum_{k=1}^{K}\sqrt{p_k + \bar{p}_k + (p_k - \bar{p}_k)^2})^2} \tag{10}$$

where

$$p_k = \frac{\sum_{i=1}^{n} I(h_k(d_i) = y_i, d_i \notin D_k)}{\sum_{i=1}^{n} I(d_i \notin D_k)} \tag{11}$$

and

$$\bar{p}_k = \frac{\sum_{i=1}^{n} I(h_k(d_i) = \hat{j}(d_i), d_i \notin D_k)}{\sum_{i=1}^{n} I(d_i \notin D_k)} \tag{12}$$

and

$$\hat{j}(d_i) = \arg\max_{j \neq y_i} Q(d_i, j) \tag{13}$$

The values for *Strength* and *Correlation* are calculated and stored in the model object, and are accessed with their respective accessor functions:

```
R> strength(model)

[1] 0.5938936

R> correlation(model)

[1] 0.1839344
```

The standard `predict()` function is also available to apply the model to new data. Here we apply it to the holdout test dataset to obtain an expected measure of accuracy.

```
R> cl <- predict(model, newdata = ds[test, vars], type = "response")
R> actual <- ds[test, target]
R> (accuracy.wsrf <- sum(cl == actual) / length(actual))

[1] 0.8545455
```

We have a model that is close to 85% accurate on the unseen testing dataset.

As we have seen, other packages provide random forest models and we now compare the accuracy of a resulting model to `cforest` and `randomForest`, out-of-the-box, so to speak.

```
R> library("randomForest")
R> system.time(model.rf <- randomForest(form, data = ds[train, vars]))

   user  system elapsed
  0.270   0.008   0.278

R> model <- model.rf
R> model


Call:
 randomForest(formula=form, data=ds[train, vars])
               Type of random forest: classification
....
```

```
R> cl <- predict(model, newdata = ds[test, vars], type = "response")
R> (accuracy.rf <- sum(cl == actual) / length(actual))

[1] 0.8727273
```

For this dataset we note that `randomForest()` has delivered better accuracy. This is not necessarily unexpected as `wsrf()` is best suited to datasets with very many variables. And with different random seeds, different results would be obtained. Also note that for small dataset like weather, the user time of **wsrf** is larger than **randomForest**, which means running **wsrf** without parallelism will be slower than **randomForest**.

```
R> library("party")
R> system.time(model.cf <- cforest(form, data = ds[train, vars]))

   user  system elapsed
  0.698   0.000   0.697

R> model <- model.cf
R> cl <- predict(model, newdata = ds[test, vars], type = "response")
R> (accuracy.cf <- sum(cl == actual) / length(actual))

[1] 0.8363636
```

Next, we will specify building the model on a cluster of servers. By default, `wsrf()` has `parallel=TRUE` and will use two less threads than the number of cores available on the host node for building the collection of decision trees, one tree per thread/core at a time. We can use `detectCores()` to identify the number of available cores on the host.

```
R> detectCores()

[1] 16
```

If we want to use all cores then we can specify the number as the value of `parallel=`. Using all cores will, for example, leave no compute cycles spare and thus interactive usage of the computer will be affected.

```
R> system.time(model.wsrf <- wsrf(form, data = ds[train, vars],
+    parallel = 4))

   user  system elapsed
  1.008   0.032   0.275

R> print(model.wsrf)
```

```
A Weighted Subspace Random Forest model with 500 trees.

  No. of variables tried at each split: 5
               Out-of-Bag Error Rate: 0.14
                            Strength: 0.60
                         Correlation: 0.18

Confusion matrix:
     No Yes class.error
No  210   5        0.02
Yes  30  11        0.73
```

Notice that the elapsed time is increased because only 4 cores are allocated.

To run the process over a cluster of servers we can name the servers as the value of `parallel=`. This can be a simple character vector naming the host, or it can be named integer vector, whose integer values nominate the number of cores to use on each node. We illustrate the simple example first.

```
R> servers <- c("node33", "node34", "node35", "node36")
R> system.time(model.wsrf.1 <- wsrf(form,
+    data = ds[train, vars], parallel = servers))

   user  system elapsed
  0.041   0.045   2.256

R> print(model.wsrf.1)

A Weighted Subspace Random Forest model with 500 trees.

  No. of variables tried at each split: 5
               Out-of-Bag Error Rate: 0.14
                            Strength: 0.59
                         Correlation: 0.19

Confusion matrix:
     No Yes class.error
No  210   5        0.02
Yes  30  11        0.73
```

Notice the timing here. There is an overhead in initiating an R session on each server and in transferring the dataset to each server. For a small task this is a relatively high overhead. For a larger task the overhead is marginal.

Now we specify the number of cores to use per node:

```
R> servers <- c(node33 = 2, node34 = 4, node35 = 8, node36 = 10)
R> system.time(model.wsrf.2 <- wsrf(form,
+    data = ds[train, vars], parallel = servers))
```

```
   user  system elapsed 
  0.038   0.040   2.189 
```

```
R> print(model.wsrf.2)
```

```
A Weighted Subspace Random Forest model with 500 trees.

  No. of variables tried at each split: 5
                 Out-of-Bag Error Rate: 0.14
                              Strength: 0.60
                           Correlation: 0.18

Confusion matrix:
     No Yes class.error
No  210   5        0.02
Yes  30  11        0.73
```

**wsrf** also provides `subset.wsrf()` for manipulating part of the forest and `combine.wsrf()` for merging multiple models into a larger model.

```
R> model.subset  <- subset.wsrf(model.wsrf, 1:200)
R> model.combine <- combine.wsrf(model.wsrf.1, model.wsrf.2)
```

In summary, the signature of the function to build a weighted random forest model in **wsrf** is:

```
wsrf(formula, data, nvars = log2(n) + 1, ntrees = 500,
  weights = TRUE, parallel = TRUE, na.action = na.fail)
```

In the simplest invocations we have demonstrated above, all parameters except `formula=` and `data=` use their default values. The parameter `nvars=` (or `mtry=`) is the number of variables to choose from for each node in each tree, with the default being the largest integer less than the log base 2 of the number of variables being modelled, plus 1. For `ntrees=`, in common with **randomForest** and **party**, the default number of trees to build is 500. The parameter `weights=`, defaulting to `TRUE`, requests using weighted sub-spaces for random sampling, rather than traditional un-weighted random sampling. The level of parallelism is controlled by `parallel`, indicating whether to run sequentially, in parallel on a single node (multi-cores) or parallel across a number of nodes, each possibly with multiple cores. Finally, we can use `na.action=` to specify a function to deal with missing data. For example, we could have used `na.action=na.roughfix` instead of fixing the missing values ourselves.

## 6. Installation options

The released version of the **wsrf** package is available from CRAN[7] which by default is a sequential only version. The type of parallel computing compiled into the package is reported

[7]CRAN is the Comprehensive R Archive Network from where R packages can be installed. See `https://CRAN.R-project.org/`.

when the package is loaded (using `library("wsrf")`) to the current R session. The choice is made when the source code is compiled and the available options depend on what is available at the time of compilation.

To make use of parallel computations a source install is required together with a C++ compiler with the C++11 standard support for threads or else version 1.54 or above of the Boost C++ library (Boost Community 2016). To install from source and include the preferred parallel computation capability we run the following command within R:

```
R> install.packages("wsrf", type = "source",
+    configure.args = "--enable-c11=yes")
```

C++11 is not currently supported by CRAN[8] and so we provide three options for installing **wsrf**. The first and default is the implementation without parallelism where trees are built sequentially. This is often the only way to install on a Windows platform.

```
R> install.packages("wsrf", type = "source",
+    configure.args = "--enable-c11=no")
```

The second and recommended install method uses the parallelism provided by the modern C++11 compilers. This is suitable for GNU/Linux installations with recent versions of the g++ compiler, as with Debian and Ubuntu.

```
R> install.packages("wsrf", type = "source",
+    configure.args = "--enable-c11=yes")
```

The third install method uses the older Boost library for multi-threading. This is useful when the available version of C++ does not support C++11 but has the Boost libraries installed.

```
R> install.packages("wsrf", type = "source",
+    configure.args = "--with-boost-include=<Boost include path>
+                      --with-boost-lib=<Boost lib path>")
```

For more details see the reference manual or the vignette for the **wsrf** package.

Our examples here work for any of the installations though run time performance is of course dependent on use of the parallel options.

## 7. Summary and future work

The **wsrf** package is the result of integrating C++ and R with multi-threaded and distributed functionality for our specific algorithm. It implements a scalable weighted random forest algorithm which can be used on a single high performance server or a cluster of such servers to offer very accurate models more quickly than the traditional random forest algorithms.

A significant remaining task for future work is to deal with memory usage, especially with bigger data becoming available. Further work is required to deal with the in-memory and out-of-memory requirements of the current implementation. Research is under way to consider various big-data implementations, including the use of Hadoop type paradigms.

[8]This is likely to change in the future and the version of **wsrf** from CRAN will then directly support the parallel implementation.

## Acknowledgments

We would like to thank the reviewers and the editors for their constructive comments and valuable suggestions, which considerably improved this paper and **wsrf**. This work was supported by grants from the National Natural Science Foundation of China (61473194 and 61305059).

Corresponding author: Joshua Zhexue Huang.

## References

Amaratunga D, Cabrera J, Lee YS (2008). "Enriched Random Forests." *Bioinformatics*, **24**(18), 2010–2014. doi:10.1093/bioinformatics/btn356.

Analytics R, Weston S (2015). ***doParallel**: Foreach Parallel Adaptor for the **parallel** Package.* R package version 1.0.10, URL https://CRAN.R-project.org/package=doParallel.

Boost Community (2016). *Boost C++ Libraries.* URL http://www.boost.org/.

Bosch A, Zisserman A, Muoz X (2007). "Image Classification Using Random Forests and Ferns." In *IEEE 11th International Conference on Computer Vision 2007 (ICCV 2007)*, pp. 1–8. IEEE.

Breiman L (2001). "Random Forests." *Machine Learning*, **45**(1), 5–32. doi:10.1023/a:1010933404324.

Breiman L, Friedman JH, Olshen RA, Stone CJ (1984). *Classification and Regression Trees.* Wadsworth.

Caruana R, Niculescu-Mizil A (2006). "An Empirical Comparison of Supervised Learning Algorithms." In *Proceedings of the 23rd International Conference on Machine Learning*, pp. 161–168. ACM.

Cutler DR, Edwards Jr TC, Beard KH, Cutler A, Hess KT, Gibson J, Lawler JJ (2007). "Random Forests for Classification in Ecology." *Ecology*, **88**(11), 2783–2792. doi:10.1890/07-0539.1.

Díaz-Uriarte R, De Andres SA (2006). "Gene Selection and Classification of Microarray Data Using Random Forest." *BMC Bioinformatics*, **7**(1), 3. doi:10.1186/1471-2105-7-3.

Eddelbuettel D (2013). *Seamless R and C++ Integration with **Rcpp**.* Springer-Verlag, New York.

Eddelbuettel D (2017). *CRAN Task View: High-Performance and Parallel Computing with R.* Version 2017-02-16, URL https://CRAN.R-project.org/view=HighPerformanceComputing.

Eddelbuettel D, François R (2011). "**Rcpp**: Seamless R and C++ Integration." *Journal of Statistical Software*, **40**(8), 1–18. doi:10.18637/jss.v040.i08.

Guyon I (2013). *Gisette Data Set Download Link.* URL http://archive.ics.uci.edu/ml/machine-learning-databases/gisette/GISETTE/.

Hothorn T, Bühlmann P, Dudoit S, Molinaro A, Van Der Laan M (2006a). "Survival Ensembles." *Biostatistics*, **7**(3), 355–373. doi:10.1093/biostatistics/kxj011.

Hothorn T, Hornik K, Zeileis A (2006b). "Unbiased Recursive Partitioning: A Conditional Inference Framework." *Journal of Computational and Graphical Statistics*, **15**(3), 651–674. doi:10.1198/106186006x133933.

IEEE, The Open Group (2004). *The Open Group Base Specifications Issue 6, IEEE Std 1003.1, 2004 Edition.* URL http://pubs.opengroup.org/onlinepubs/007904975/.

Ieong B (2014). ***ParallelForest**: Random Forest Classification with Parallel Computing.* R package version 1.1.0, URL https://CRAN.R-project.org/package=ParallelForest.

Kandaswamy KK, Chou KC, Martinetz T, Möller S, Suganthan PN, Sridharan S, Pugalenthi G (2011). "AFP-Pred: A Random Forest Approach for Predicting Antifreeze Proteins from Sequence-Derived Properties." *Journal of Theoretical Biology*, **270**(1), 56–62. doi:10.1016/j.jtbi.2010.10.037.

Kane MJ, Emerson J, Weston S (2013). "Scalable Strategies for Computing with Massive Data." *Journal of Statistical Software*, **55**(14), 1–19. doi:10.18637/jss.v055.i14.

Karypis G (2013). *5 Text Data Set Download Link.* URL http://glaros.dtc.umn.edu/gkhome/fetch/sw/cluto/datasets.tar.gz.

Li J (2013). *Tis Data Set Download Link.* URL http://levis.tongji.edu.cn/gzli/data/TIS.zip.

Liaw A, Wiener M (2002). "Classification and Regression by **randomForest**." *R News*, **2**(3), 18–22. URL https://CRAN.R-project.org/doc/Rnews/.

Lim A (2014). ***bigrf**: Big Random Forests: Classification and Regression Forests for Large Data Sets.* R package version 0.1-11, URL https://CRAN.R-project.org/package=bigrf.

Piccolboni A (2013). ***RHadoop** Project.* URL https://github.com/RevolutionAnalytics/RHadoop/wiki.

Quinlan JR (1993). "C4.5: Programs for Machine Learning." *The Morgan Kaufmann Series in Machine Learning*, **1**. doi:10.1007/bf00993309.

R Core Team (2016a). *Building R for Windows.* URL https://CRAN.R-project.org/bin/windows/Rtools/.

R Core Team (2016b). *R: A Language and Environment for Statistical Computing.* R Foundation for Statistical Computing, Vienna, Austria. URL https://www.R-project.org/.

Rennie J (2013). *Newsgroups Data Set Download Link.* URL http://qwone.com/~jason/20Newsgroups/20news-bydate-matlab.tgz.

Rodriguez-Galiano VF, Ghimire B, Rogan J, Chica-Olmo M, Rigol-Sanchez JP (2012). "An Assessment of the Effectiveness of a Random Forest Classifier for Land-Cover Classification." *ISPRS Journal of Photogrammetry and Remote Sensing*, **67**, 93–104. doi:10.1016/j.isprsjprs.2011.11.002.

Roweis S (2013). *Mnist Data Set Download Link.* URL http://www.cs.nyu.edu/~roweis/data/mnist_all.mat.

Seligman M (2016). ***Rborist**: Extensible, Parallelizable Implementation of the Random Forest Algorithm.* R package version 0.1-3, URL https://CRAN.R-project.org/package=Rborist.

Strobl C, Boulesteix AL, Kneib T, Augustin T, Zeileis A (2008). "Conditional Variable Importance for Random Forests." *BMC Bioinformatics*, **9**(307). doi:10.1186/1471-2105-9-307.

Strobl C, Boulesteix AL, Zeileis A, Hothorn T (2007). "Bias in Random Forest Variable Importance Measures: Illustrations, Sources and a Solution." *BMC Bioinformatics*, **8**(25). doi:10.1186/1471-2105-8-25.

The OpenMP Architecture Review Board (2016). *The OpenMP API Specification for Parallel Programming.* URL http://openmp.org/.

Therneau T, Atkinson B, Ripley B (2015). ***rpart**: Recursive Partitioning.* R package version 4.1-10, URL https://CRAN.R-project.org/package=rpart.

Tierney L, Rossini AJ, Li N, Sevcikova H (2016). ***snow**: Simple Network of Workstations.* R package version 0.4-2, URL https://CRAN.R-project.org/package=snow.

Touw WG, Bayjanov JR, Overmars L, Backus L, Boekhorst J, Wels M, van Hijum SAFT (2013). "Data Mining in the Life Sciences with Random Forest: A Walk in the Park or Lost in The Jungle?" *Briefings in Bioinformatics*, **14**(3), 315–326. doi:10.1093/bib/bbs034.

Urbanek S (2011). ***multicore**: Parallel Processing of R Code on Machines with Multiple Cores or CPUs.* R package version 0.1-7, URL https://CRAN.R-project.org/package=multicore.

Williams GJ (1988). "Combining Decision Trees: Initial Results from the MIL Algorithm." In JS Gero, RB Stanton (eds.), *Artificial Intelligence Developments and Applications: Selected Papers from the First Australian Joint Artificial Intelligence Conference, Sydney, Australia, 2-4 November, 1987*, pp. 273–289. Elsevier.

Williams GJ (2011). *Data Mining with **rattle** and R: The Art of Excavating Data for Knowledge Discovery.* Springer-Verlag, New York.

Xu B (2015). *GitHub Download Link of the Experimental Data Sets for **wsrf**.* doi:10.5281/zenodo.48691. URL https://github.com/SimonYansenZhao/wsrf-test-data/raw/master/dataFromBaoxun.zip.

Xu B, Huang JZ, Williams GJ, Wang Q, Ye Y (2012). "Classifying Very High-Dimensional Data with Random Forests Built from Small Subspaces." *International Journal of Data Warehousing and Mining (IJDWM)*, **8**(2), 44–63. doi:10.4018/jdwm.2012040103.

## Affiliation:

He Zhao, Graham J. Williams
Shenzhen College of Advanced Technology, University of Chinese Academy of Sciences
Shenzhen Institutes of Advanced Technology (SIAT), Chinese Academy of Sciences
1068 Xueyuan Avenue
Shenzhen University Town, Shenzhen, People's Republic of China
E-mail: Simon.Yansen.Zhao@gmail.com, Graham.Williams@togaware.com

Joshua Zhexue Huang
College of Computer Science & Software Engineering
Shenzhen University
Nanhai Ave 3688
Shenzhen, People's Republic of China
E-mail: zx.huang@szu.edu.cn

*Journal of Statistical Software* http://www.jstatsoft.org/
published by the Foundation for Open Access Statistics http://www.foastat.org/
March 2017, Volume 77, Issue 3 *Submitted:* 2015-01-12
doi:10.18637/jss.v077.i03 *Accepted:* 2015-12-04