Parallel Numerics '05, 119-125 M. Vajteršic, R. Trobec, P. Zinterhof, A. Uhl (Eds.)
Chapter 5: Optimization and Classification ISBN 961-6303-67-8

Reimplementation of the Random Forest Algorithm

Goran Topić*, Tomislav Šmuc, Zorislav Šojat, Karolj Skala*

Ruđer Bošković Institute
Bijenička 54, 10000 Zagreb, Croatia

The random forests algorithm is one of the more versatile data classification algorithms currently known to data mining community. It can classify huge amounts of data, with large number of attributes. Unfortunately, it was originally written in Fortran 77, a cumbersome language not suited for structured programming, and without any support for parallel execution. However, the algorithm itself is very well suited to parallelisation – the random forest consists of random decision trees, which can be generated (and evaluated) quite independently. Therefore, we have undertaken to reimplement the algorithm in a higher programming language, and to allow it to run in a cluster environment. To keep the algorithm accessible to the mathematicians we decided to stick to Fortran, but Fortran 90 – a structured programming language with many features that were missing in the earlier standard.

1 Introduction

The random forest algorithm is one of the more versatile data classification algorithms currently known to data mining community. It can work on big datasets, with large number of attributes; it can provide many other incidental pieces of information about the dataset besides the class tags; and, it is loosely coupled, and can be easily parallelised.

Unfortunately, it was originally written in Fortran 77, a cumbersome language not suited for structured programming, and without any support for parallel execution. Fortran was chosen because it is traditionally employed in the mathematician circles, being well suited to expressing numerical and

*Corresponding author. E-mail: goran.topic@irb.hr
*Corresponding author. E-mail: skala@irb.hr

computational processes – and the 77 standard because at the time only commercial compilers handled the newer Fortran specifications. However, it is very hard to understand the code, even when the algorithm is known, because of the language limitations – the most notable of which being the lack of composite types, recursion and dynamic allocation. Also, the very strict source code format specification does not facilitate legibility. To make the matters worse, the code was written in a very concise, mathematical style, with terse and insufficiently descriptive variable names – coupled with the sheer number of variables used in the program, and the general scarcity of comments, there was very little possibility for the program to serve as an illustration of the algorithm, for those who wished to gain first-hand knowledge of it by directly studying the implementation. Furthermore, Fortran 77 does not provide the functionalities that would enable a comfortable user interface, and certain matters (like formatted input and output) were sacrificed for speed of execution and simplicity of code. Consequently, the original implementation's source code needed to be edited and recompiled for each run, and the input and output data were not legible by humans, and thus needed additional processing.

For these three reasons – sequentialism, illegibility and user-unfriendliness – we have decided to reimplement the algorithm to alleviate these problems. We have chosen to do so using Fortran 90. In this way, the code remains close to the mathematics community, and on the other hand it can be written in a structured way. Also, we shall enable parallelism, incorporate dynamic allocation (removing the need for defining the data dimensions within the code) and implement a better user interface.

2 Description of the algorithm

To briefly describe the algorithm: the random forests are based on decision trees. We take a seed – so to speak – from the training dataset by pulling out at random a collection of samples (the bootstrap). A random set of attributes is then chosen. The attribute from this set that can create the best possible split is selected, as well as the value criterion that yields this split, and the samples are separated into the "yes" branch and the "no" branch. The process repeats for each of the branches, until our "seed" grows into a proper tree – the termination condition stating that leaves are the nodes that are either too small to split, homogeneous, or where no splitting criterion can be found in the set of the randomly chosen attributes. One tree by itself is thus incomplete – only a part of the dataset participates in the tree growth, and many of the nodes are split by a suboptimal criterion, simply because not all attributes are considered. However, grow ten of them, or a hundred, and the situation changes. When the forest is employed in classification, each tree casts votes, assigning a class to each sample, and the votes generally differ from tree to tree. If we poll the results, the class that received the most votes "wins".

The classification is very easy and quick, but still it might be desirable to distribute it for extremely large datasets. Each tree makes the decision by itself – the only interaction is the final counting of the votes for each of the samples.

However, the more interesting part of the problem is the "cultivation" of the forest. Growing a tree is rather costly, in both processor time and memory, since a large number of split possibilities need to be examined to find the best one – but it is noteworthy that a tree is grown in isolation, too. From the moment we pick the bootstrap, the "seed", until the tree is "mature", no communication or interaction of any kind is necessary. We can thus hire as many worker processes as we are to have trees, and assign each worker a seed to tend to. They shall report back to the overseer when their tree is fully grown.

In this way the whole forest grows simultaneously, and the only communication is passing the training data to the workers, and collecting the trees back to the controlling process if necessary.

3 Implementation

3.1 Coding

The implementation was split into modules, which are the closest one can get to the concept of an "object class" in Fortran.

The module instancesets defines a set of instances, whether from a training dataset or testing dataset. It contains a routine that loads the data from an ARFF file, and a matching routine to write it out in the same format. The ARFF specification was slightly modified for this program – date types are not supported, integers are treated either as a numeric or as category values, and class weights are supported, to name the most important changes.

The module bootstraps defines a data structure that holds a bootstrap sample collection.

The module trees defines a single tree as a recursive composite type, and provides the methods to build a tree and to classify data on it, as well as basic input and output operations.

The module forests brings together a collection of trees, and enables the saving and loading of whole forests. It also has methods for data classification, and it orchestrates the tree building – and it is here where the coordination of worker processes happens.

In addition to these modules that implement the random forests algorithm itself, there are several other supporting modules. The module bitvectors implements packed arrays of logical values. Besides the obvious reason of memory conservation, a bitvector can be easily and quickly incremented, as well as randomised – two operations necessary for the generation of the combinations of categories used in searching for the best split on category values.

The module options parses the command line options given by the user, and puts them in a globally accessible structure where all other modules can straightforwardly access them.

There is also the module compatibility, done in several variants – one for each supported compiler, collecting the implementations for functions that are non-standard and compiler-dependent, to allow the code to be used on several popular compilers.

3.2 Parallelisation issues

The algorithm has two distinct phases, training and classification. Classification is very fast, algorithmically extremely simple, and would not benefit from parallel execution. However, depending on the data set, the training phase can last from seconds to hours or more. The exact execution time is a function of many parameters, both those specified on the command line (e.g. number of trees to grow, number of missing value fill iterations, etc.) and those inherent in the data set (such as the number of instances, the number of attributes, the number of categorical attributes and their ranges and the distribution of the continuous attributes).

This process can be readily parallelised in two ways. The first one is trivial, and was already described in the Introduction: assign the trees to workers, and grow the forest in parallel. In actuality it is not practical to have a worker for each tree, but rather for each processor we have available for computation. Thus each worker sequentially grows its share of trees, and "subforests" grow in parallel.

The second way is the parameter sweep option. The forest construction has several parameters that influence the prediction accuracy (primarily the number of random attribute picks to test at each node); however, the optimal values cannot be determined *a priori*. To avoid the manual fiddling with the parameter values, one could in parallel construct several whole forests, each with a different parameter value, then automatically pick the one that yields the least number of misclassified instances.

3.3 Speed-up estimation

At this moment the code is still under development and parallelization performance and benefits could only be estimated indirectly. We have set up an experiment to measure the speed-up of calculations in a most simple form of parallelisation, i.e. partitioning of the forest to p nodes. We have produced two sets of classification problems, first with only categoric variables and second with only numeric variables. Details of this sets are given in Table 1.

Table 1: Properties of datasets used for sequential calculations and speed-up estimation. It is important to notice that categorical set has 50 variables with 20 categories each. All datasets were randomly generated.

Dataset	No. of variables	No. of classes	No. of samples
10k	50	5	10000
20k	50	5	20000
50k	50	5	50000
100k	50	5	100000
200k	50	5	200000

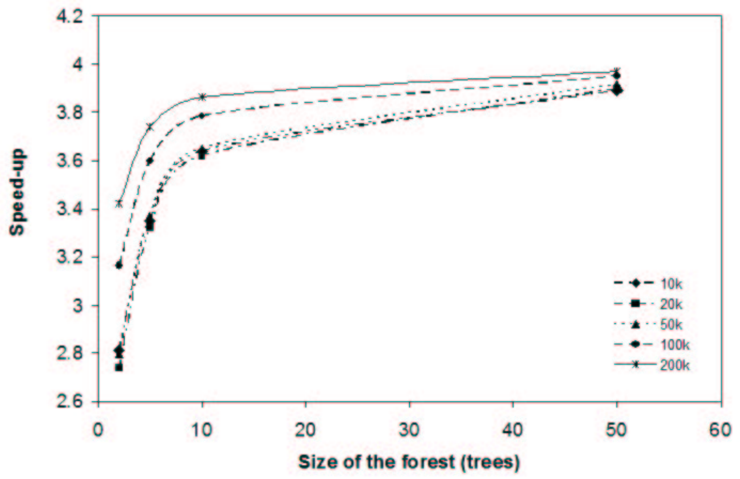

Figure 1: Speed-up for datasets with numeric variables.

Experimental runs of the new implementation of Random Forests algorithm were performed on a mini-cluster of four, two 64-bit processor machines (2.8 GHz) under Linux operating system, mutually connected via 1Gb ethernet. We have estimated realistic bandwidth of the local connection experimenting

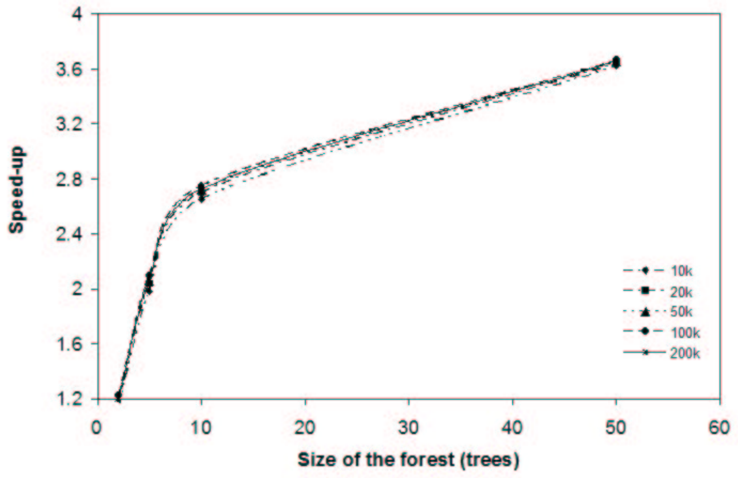

Figure 2: Speed-up for datasets with categorical variables.

with sending varible packets of the size between 1MB – 256MB. An average of 25MB/sec was used in our speed-up estimate. In this calculations we have added other computations (such as parsing, sorting) into a sequential overhead of the parallel estimate. Figures 1 and 2 depict speed-up estimates for numerical datasets and categorical ones. It is important to emphasize that performance between the two cannot be directly compared, since different option regarding the minimumu split node size of the tree was used for these two instances. For numerical datasets we have used minimum size of the split node to be 0.1 percent of the number of samples, while for the categorical datasets, fixed absolute size (10) was used. The reason for this is that performance in growing the tree with categoricals is much higher if higher values for the minimum size of the split node are used. In that case speed-up would grow slower than shown in Figure 2.

Real problems are usually a mixture of the two types of varible sets used in this experiment. This means that each new problem would have its own speed-up properties. However, it is clear that even for relatively modest sizes of datasets (20-50000 samples), time savings from parallelization of Random Forests are considerable. They will be even more pronounced in the case of parameter sweep, since in that case forests are grown in parallel rather than parts of the forest, while the communication overhead is not changed significantly.

4 Summary

This article summarizes current state of the new implementation of the Random Forests algorithm. Most of the important changes to the original code are related to coding and design: use of Fortran 90 standard, use of OOP concepts, adressing portability issues, constant testing and code optimization. Reasons for the parallelization of the code are explained, and an estimate computational savings are roughly estimated on a set of calculations using current version of the code.

References

[1] L. Breiman, Random Forests, *Machine Learning* (2001), 45, 5-32.

[2] L. Breiman and A. Cutler, *Random Forests* (2004), http://stat-www.berkeley.edu/users/breiman/RandomForests/cc_home.htm

[3] A. Gramma, A. Gupta, G. Karypis and V. Kumar, *Introduction to Parallel Computing*, Pearson Education Ltd.,(2003).