CLASSIFICATION OF LARGE MICROARRAY DATA SETS USING FAST RANDOM FOREST CONSTRUCTION

E. A. Manilich [1,2,*] , Z. M. Ozsoyoglu [1], V. Trubachev [2] ,T. Radivoyevitch [3]

[1]*Computer Science Department, Case Western Reserve University, Cleveland, Ohio 44106, USA*

[2]*Digestive Disease Institue, Cleveland Clinic Cleveland, Ohio 44195, USA*
[*]*Email: manilie@ccf.org*

[3]*Department of Epidemiology and Biostatistics, Case Western Reserve University, Cleveland, Ohio 44106, USA*

Random forest is an ensemble classification algorithm. It performs well when most predictive variables are noisy and can be used when the number of variables is much larger than the number of observations. The use of bootstrap samples and restricted subsets of attributes makes it more powerful than simple ensembles of trees. The main advantage of a random forest classifier is its explanatory power: it measures variable importance or impact of each factor on a predicted class label. These characteristics make the algorithm ideal for microarray data. It was shown to build models with high accuracy when tested on high dimensional microarray data sets. Current implementations of random forest in the machine learning and statistics community, however, limit its usability for mining over large datasets as they require that the entire dataset remains permanently in memory. We propose a new framework, an optimized implementation of a random forest classifier, which addresses specific properties of microarray data, takes computational complexity of a decision tree algorithm into consideration, and shows excellent computing performance while preserving predictive accuracy. The implementation is based on reducing overlapping computations and eliminating dependency on the size of main memory. The implementation's excellent computational performance makes the algorithm useful for interactive data analyses and data mining.

1. INTRODUCTION

The fact that microarrays allow investigations of thousands of genes at the same time is an advantage from a data generation perspective, but a challenge from a data analysis perspective. Analyses of microarray data pose a challenge due to the extremely large number of predictors, their possible interactions, and the comparatively small number of samples. As a result, classical statistical techniques cannot be applied directly to microarray datasets and novel methods are necessary. Random forest is one such method.

Random forest is a classification ensemble algorithm developed by Leo Breiman that uses multiple binary decision trees. Each of the classification trees is built using a sample of data and at each node a randomly chosen set of variables is considered for the best split.[1] Random forest has become a major data analysis tool. It has been applied to large-scale tissue microarray data and genome-wide association studies for complex diseases. [2-9]

The random forest algorithm is suitable for microarray data mining for several reasons. Firstly, classification trees are non-parametric and do not make any assumptions about the underlying distribution. Second, it performs excellently even when most of the predictive variables are noisy. Third, it can be used when the number of variables is much larger than the number of observations. And lastly, it returns measures of variable importance that can be used for gene selection. [1]

The accuracy of random forest is comparable or superior to alternative state-of-the-art microarray data-

* Corresponding author.

based prediction methods.[3] Comprehensive evaluation of random forest applications to real microarray data classification problems have shown high predictive accuracy.[3,4,9] This evaluation included an assessment of the effects of changes in the parameters of random forest on its classification performance. Changes in the two important parameters, the number of input variables selected at each split and the number of trees in a forest, have negligible effects, i.e. the algorithm is robust to the extent that there is no need to fine-tune default parameters or pre-select variables.

2. EXISTING RANDOM FOREST FRAMEWORKS

High quality random forest implementations include : the original Fortran code from L. Breiman and A. Cutler, the R package port of this code, `randomForest` by A. Liaw and M. Wiener (this code is available through the Comprehensive R Archive Network, CRAN), and Weka, a collection of machine learning algorithms for data mining tasks implemented in Java.[10-12] A problem with each of these frameworks, however, is that their applications to large microarray datasets are limited due to high computational requirements.

Performance and scalability have not been the primary design objectives of the presently available frameworks; instead, their emphasis has been on the accuracy of the implementation. Efficiency and scalability are major issues of concern when random forest is applied to very large datasets. The existing implementations mentioned above require that the entire training data remains permanently in main memory and this limits their usability.

In response to this limitation, preprocessing steps that reduce the dimensionality of the data have been applied to large datasets before their submission to the random forest algorithm. The process of variable reduction has, however, drawbacks that include changes of the resulting classification model, reduced accuracy, and loss of the potentially useful and interesting prediction variables. Thus, it is important to understand the computational complexity of random forest, investigate the use of the algorithm for classification of large microarray datasets, and provide a scalable version of random forest that produces exactly the same decision trees.

3. DECISION TREE CLASSIFIERS

In this section, we discuss computational resources required by random forest and examine previously known scalable algorithms for construction of classification trees.

3.1. Computational resources

Algorithms used to construct ensembles of classification trees can be characterized by their computational complexity, the distribution of execution times of specific operations, and identification of operations that are the most computationally expensive.[13]

Algorithm learning times are dominated by two operations: 1) the sorting of training data on each candidate variable selected for the split and, to a lesser extent, 2) the random selection of sample and variable subsets. At every step, a classification tree uses exhaustive search of all possible combinations of variables and split points to achieve the global minimum in impurity. To select the best splitting variable, the algorithm requires sequential access to all numeric variables in sorted order. The procedure of sorting requires many comparisons and the procedure of sampling requires intensive memory access. Respectively, these operations consume approximately 60% and 20% of of the execution time.

3.2. Previous work in the database literature

Several software frameworks have addressed scalability requirements of tree construction algorithms without modifying the result. SLIQ, one of the first decision tree algorithms for large datasets, uses a technique where after presorting the sort order is maintained in the tree growth phase to avoid resorting and to reduce the cost of numeric variable evaluations.[14] SLIQ separates the input dataset into attribute lists at the beginning of the algorithm and it thus requires an in-memory data structure that grows linearly with the number of records in the training dataset.

A second framework, SPRINT, removes the in-memory requirement of SLIQ and instead proposes a new data model that runs with a minimal amount of main memory.[15] Similar to SLIQ, SPRINT also minimizes sorting by creating a sorted attribute lists at the beginning of the algorithm, but it uses a different

data structure wherein the initial sorted attributes lists are associated with the root of the classification tree and as trees grow, the attribute lists are partitioned and distributed among children. The costly operation here is the distribution of attribute lists at each node of the decision tree. This operation involves significant rewriting of the disk resident dataset with large numbers of attributes and it also triples the size of the training dataset.

As a third framework, RainForest can be viewed as an optimized version of SPRINT.[16] The optimization is based on the observation that the distribution of distinct class values for each predictor variable contains sufficient statistics to evaluate all possible split points. RainForest proposes a new structure, a set of aggregate lists of all predictor variables, that can be viewed as a compressed version of attribute lists used in SPRINT. For categorical variables, this structure is proportional to the number of distinct values of predictor variables and is highly likely to fit in main memory; the reduced main memory requirements lead to significant performance improvements over SPRINT for datasets with categorical variables. For continues numeric variables, however, the size of the required in-memory data structure will again be equal to that of an attribute list in SPRINT. Thus, to take advantage of RainForest, numeric variables have to be discretized, and that may impact the classification results.

SPRINT and RainForest are the fastest scalable classification tree construction algorithms proposed previously that focus on scalability issues and which do not modify results of a classification method. The main contribution of this paper is a scalable random forest framework for high dimensional real valued datasets (e.g. microarray data), also with no adverse impact on the quality of the classification model.

4. FAST RANDOM FOREST

In this section, we introduce a framework that results in a scalable random forest algorithm and we discuss how it encompasses previous work. The emphasis of this work is a fast implementation of the algorithm for microarray datasets that are too large to fit in memory. The goal here is to speed up the tree growth phase, as this is the most computationally expensive component of the algorithm. Observations of the nature of microarray data enable significant computational saving.

4.1. Algorithm description

Random forest builds an ensemble of classification trees. Each of the classification trees is built using a bootstrap sample of the data, and at each node of the tree a random subset of the variables is examined for the best split. The algorithm uses the impurity-based Gini index as an attribute selection measure used to assess the splitting criterion.[17] To evaluate the impurity function and decide how to split a node based on a numeric attribute, the algorithm requires access to each randomly selected attribute in sorted order. The number of searches for the best split point is proportional to the number of samples and attributes in the training dataset. Note that each randomly selected attribute is examined independent of the other predictor attributes.

The proposed fast random forest algorithm avoids repeated sorting. It uses a one-time sort and separate lists for each predictor attribute. The pre-processing step of one-time sorting reduces the computational burden at each node. The algorithm makes one scan over the dataset and constructs a *list of sorted indices* for each predictor attribute in the dataset. Entries into a *list of sorted indices* contain record identifiers of the training dataset sorted by the value of the corresponding attribute. Unlike usual sorting methods, which store the sorted values, our algorithm stores the original indices of the sorted records instead. Additionally, a *class list* of length equal to the number of records is used to reference a class label of each record and a pointer to a leaf node of the classification tree. At the beginning of the tree construction process, pointers for records included in a bootstrap sample are initially contained in the root node.

Due to the nature of a typical microarray dataset with a relatively small number of samples, we assume that a *class list* and at least one *list of sorted indices* always fit into main memory. Lists of sorted indices of each predictor attribute are needed in main memory, one at a time, to be given as arguments to the Gini coefficient function as a metric of impurity.

	R1	R2	R3	R4	R5	...
Gene 1	5.513	8.576	10.523	9.717	12.533	...
Gene 2	4.025	11.015	10.169	7.651	12.058	...
Gene 3	7.636	11.914	10.802	12.043	9.889	...
Gene 4	11.769	12.428	10.974	9.889	4.774	...
...	...	...	...	...	...	...
Class	1	2	1	1	2	...

Gene 1	1	-2	4	-3	5	...
Gene 2	1	-4	3	-2	5	...
Gene 3	1	-5	3	-2	4	...
Gene 4	5	-4	3	-1	2	...
...	...	...	...	...	...	...

Fig. 1. Sample dataset and corresponding lists of sorted indices. The gene expression levels in the top row are representative of robust multi-chip analyses (RMA) wherein units are on a $\log_2$ scale.

Calculation of the Gini index is based on the relative frequency, or distribution, of class labels and does not require access to actual values. Note that a *list of sorted indices* and *a class list* have all the necessary information to calculate the Gini index. However, one subtle refinement to our implementation has to be taken into consideration as the split point is evaluated at the midpoint between consecutive **distinct** data values. Thus, the Gini index will not be evaluated if values corresponding to the two adjacent sorted indices are equal. This requirement is not naturally fulfilled by a list of sorted indices as an index of the original value cannot be used to determine if two adjacent values are equal. To overcome this limitation, we implemented the following adjustment. Each index is incremented by one and represented in a form of positive non-zero integers. Thereafter, the sign of each index is adjusted to reflect changes in the value. If the two adjacent values are equal, the sign will remain the same; otherwise, the sign will change. As a result of this, changes or lack thereof in the values can be detected. Figure 1 represents a sample dataset and corresponding lists of sorted indices as input to the fast random forest algorithm (Algorithm 1).

4.2. Discussion

Our implementation of random forest employs a decision tree algorithm that adopts a middle ground between SLIQ and SPRINT, the fastest previously proposed scalable classification tree construction algorithms for datasets with continuous numeric values. Similar to SLIQ and SPRINT, the algorithm avoids sorting at each node by using pre-sorting techniques. The main difference between previously described scalable decision tree algorithms and the proposed implementation is the use of a novel data structure, a *list of sorted indices*.

Fast random forest is based on the observation that a list of sorted indices for each predictor attribute and a class list contain sufficient statistics to calculate the Gini index and select the best split point. The fast random forest algorithm requires a minimal amount of memory equal to the size of a list of sorted indices and a class list. The algorithm offers significant performance improvement over SPRINT for datasets with large numbers of attributes. Firstly, it does not triple the size of the training dataset and, therefore, utilizes main memory more efficiently. And secondly, it does not require a costly operation to partition and distribute attribute lists among children.

Algorithm 1. Fast random forest induction schema and optimization.

Input: matrix W $s \times g$, with s samples and g genes, number of trees *nTrees*
Output: ensemble of *nTrees* classification trees

Preprocessing (sorting):
1. Create g lists of sorted indices.
2. Increment each index by 1.
3. Run through values, changing sign of the index if adjacent values are not equal
4. Write each list of sorted indices to a new line in the sort file.
5. Index files: write byte offsets for each line in the data and sort files.

```
BuildForest(sample W):
  for ( i = 1 , i <= nTrees, i ++ )
    BuildTree(I)
  end for
```

```
BuildTree(sample W):
  if (class list is homogeneous)
    #branch complete
    Save outcome as leaf node.
  else
    Pick a subset g' consisting of √g attributes to examine.
    FindBestSplit(g') using the Gini coefficient.
    Read line corresponding to the split attribute from data file into memory.
    Use split value to partition records into W'1 and W'2
    BuildTree(W'1)
    BuildTree(W'2)
  end if
```

```
FindBestSplit(attributes g'):
  for ( each attribute in g')
    for ( j = 2 , j <= s , j++ )
      if (sign of (j - 1)th sorted index == sign of (j)th sorted index)
        skip to next iteration
      else
        calculate Gini index
      end if
    end for
  end for
  select best split criterion based on Gini index
```

The critical difference of our data model is that entries into a list of sorted indices contain record identifiers only. Thus, the algorithm makes more efficient use of available memory. For most microarray datasets, we expect that a randomly selected set of sorted indices at each node of the tree will fit in main memory. The assumption that a set of sorted indices of the root node fits in memory does not imply that the complete dataset fits in memory, since random forest selects a random subset of attributes at each node. If not, it is highly likely that at least a list of sorted indices of each individual predictor attribute can fit in main memory.

5. EXPERIMENTAL RESULTS

The gap between the size of real-life datasets and scalability of available data-mining applications that implement the random forest classifier is especially visible when analyzing microarray data. Both the Weka software framework and the R package randomForest failed to process a dataset with 200 samples and 20,000 gene probes.

The scalable and fast implementation of random forest was used to look for differences between the genomes of patients with a recurrent colon cancer and those without. This method allowed us to find genetic markers that were previously not correlated with colorectal cancers. [18]

As part of the algorithm testing, both our memory-based and fast file-based implementations of the random forest algorithm were timed. The tests were run on an Intel Core 2 Duo E6550 CPU (2.33 GHz), with 1.95 GB of RAM. The operating environment was an unmodified Java 6 SE (update 16) environment running under Windows XP SP3.

A comparison of our memory- and fast file-based implementations of the random forest algorithm is provided in Table 1 to highlight the origin of the excellent performance and scalability of fast random forest. Our new implementation not only makes it possible to analyze large microarray datasets on personal computers, but it also makes the algorithm available for efficient and interactive data analyses.

As expected, the memory-based implementation is much faster for small data sets. For large datasets (e.g.

Table 1. Description of computing operation for file-based fast random forest implementation vs. memory-based implementation.

Part of Process	**File-Based Fast Random Forest**	**Memory-Based**
Reading	n/a	reading the entire file into main memory
Sorting	reading the data file one line at a time and writing lists sorted indices	sorting the data by a given attribute
Picking Attributes	picking the attributes to examine at each node in the tree; reading lists of sorted indices for selected attributes into main memory	picking the attributes to examine at each node in the tree
Setting Split Point	picking the best split criterion, based on sorted indices and class attribute	picking the best split criterion
Performing Spilt	separating the remaining indices into new index masks for each branch of the tree.	separating the remaining data into two subsets, one for each branch.

20,000 attributes and 1,000 records), however, the memory-based implementation will run out of memory, while the file-based implementation will successfully generate the given number of trees.

In the file-based implementation, increasing the number of records and number of trees has a large impact on performance, as both increase the total number of sorted indices that need to be read from the sort file (Figures 2-7). The file read/write operations take up the biggest chunk of time, with the exception of sorting with large numbers of trees (see Figure 7). In this case, the sorting time will remain constant and take up an increasingly smaller percentage of time as the number of trees increases. The other file-intensive operation is picking the attributes to examine, as the selected attributes' sorted indices must be read into memory from the sort file each time. This is the single most expensive operation in the entire process (see Figures 5-7). Due to the special way the sorted indices are modified before storage, setting the split point requires no interaction with the file system and takes up a very small portion of execution time. Performing the split, on the other hand, does require reading the data file, but only one line. Thus, it does not take up a very large portion of the execution time.

In the memory-based implementation, the initial reading of the file takes up the bulk of the execution time, with the exception of an increasing number of trees (see Figures 3and 7). In that case, the reading time is constant (due to the constant file size), regardless of the number of trees, and the sorting time increases proportional to the number of attributes, records, and trees, as each has a similar impact on the number of sorting operations required. Selecting the attributes to examine takes very little time, as does performing the split. Setting the split point, however, takes the majority of the time (besides reading), mainly due to the sorting that occurs so often.

In both implementations file input and output operations take up most of the execution time. The only exception is for the memory-based implementation with an increasing number of trees, where the time taken to read the file stays constant, and with over 25 trees, reading the file does not take up most of the execution time.

It is not valid to directly compare our file-based and memory-based implementations quantitatively. While the memory-based implementation has enough memory to build the classifier successfully, it will always be faster, due to less access to the file system (which is considerably slower). Further, when the memory-based implementation runs out of memory, and the file-based version completes successfully, again, a comparison cannot be made, in this case because one failed completely. Therefore, an indirect comparison is necessary.

To compare the two implementations we broke them down into separately comparable components. One aspect immediately visible is how setting the split point scales. The file-based implementation scales very well, while the computationally-heavy memory-based implementation scales poorly, especially when the number of trees increases (see Figure 7). On the other hand, the file-based implementation scaled poorly in the read-intensive attribute selection stage, while the memory-based version performed well.

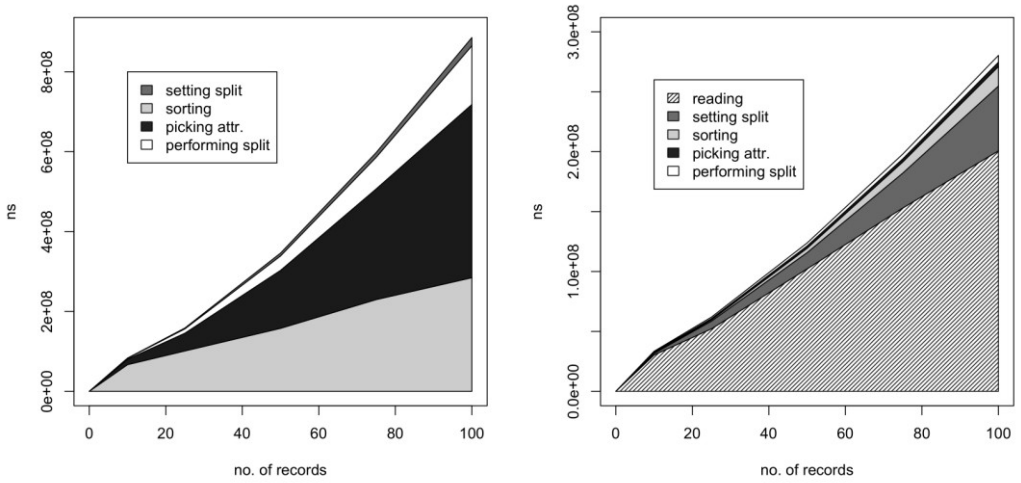

Fig. 2. Running time for increased number of records: comparison of file-based fast random forest (left) and memory-based (right) implementation with 1,000 attributes and 5 trees.

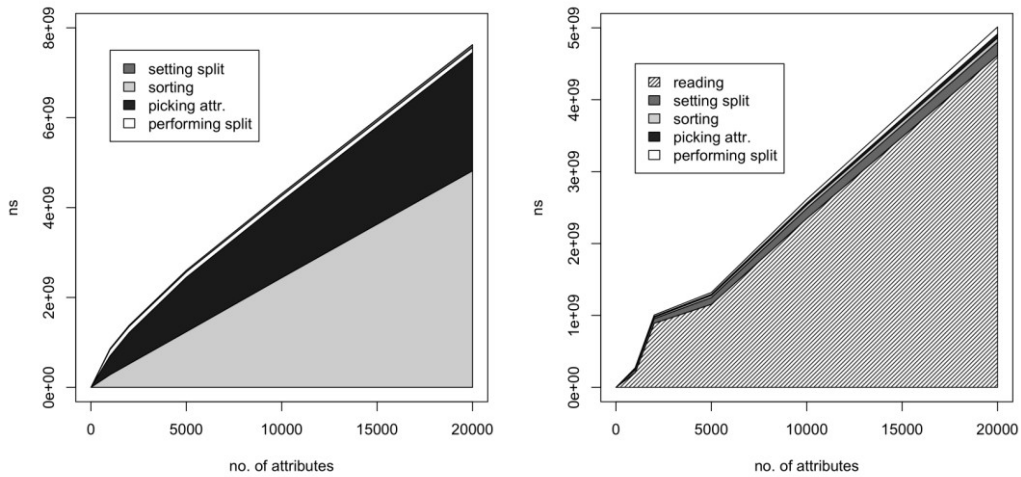

Fig. 3. Running time for increased number of attributes: comparison of file-based fast random forest (left) and memory-based (right) implementation with 100 records and 5 trees. The y-axis units are nanoseconds (ns).

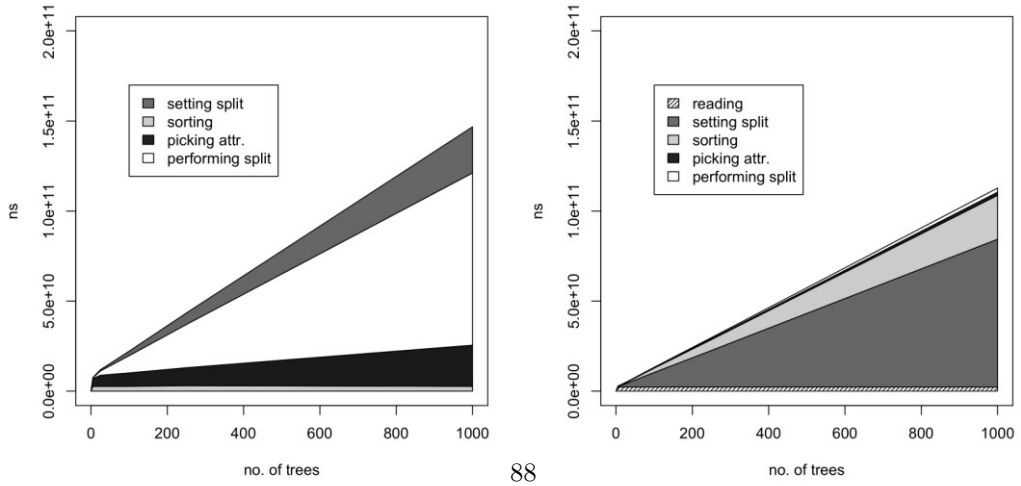

Fig. 4. Running time for increased number of trees: comparison of file-based fast random forest (left) and memory-based (right) implementation with 5,000 attributes and 200 records.

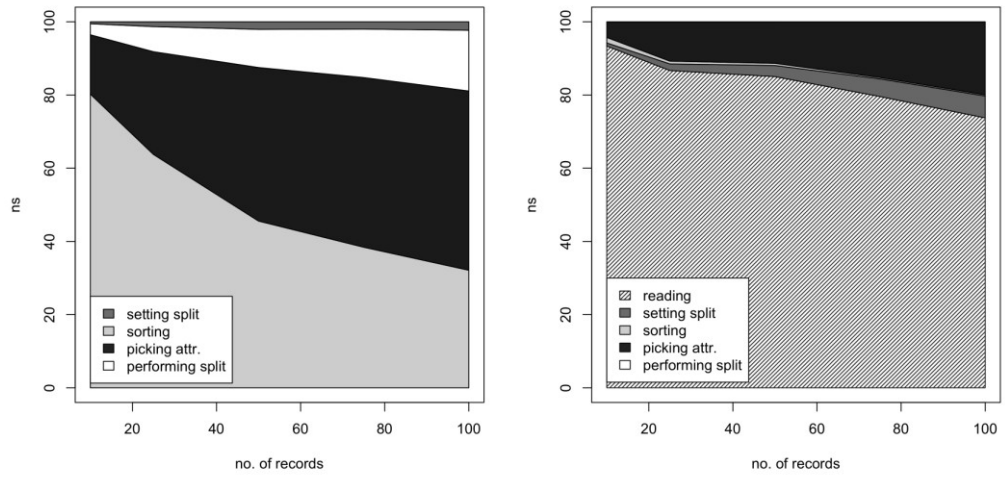

Fig. 5. Distribution of total running time for increased number of records: comparison of file-based fast random forest (left) and memory-based (right) implementation with 1,000 attributes and 5 trees.

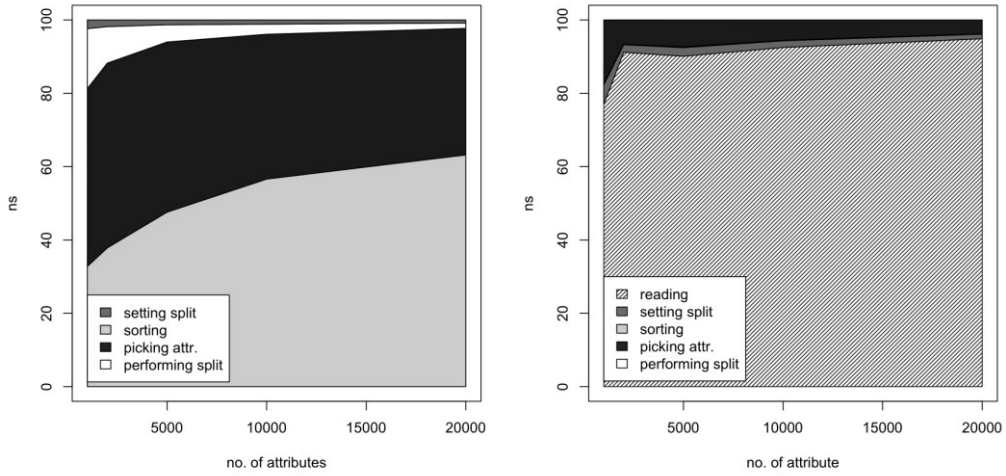

Fig. 6. Distribution of total running time versus number of attributes: comparison of file-based fast random forest (left) and memory-based (right) implementation with 100 records and 5 trees.

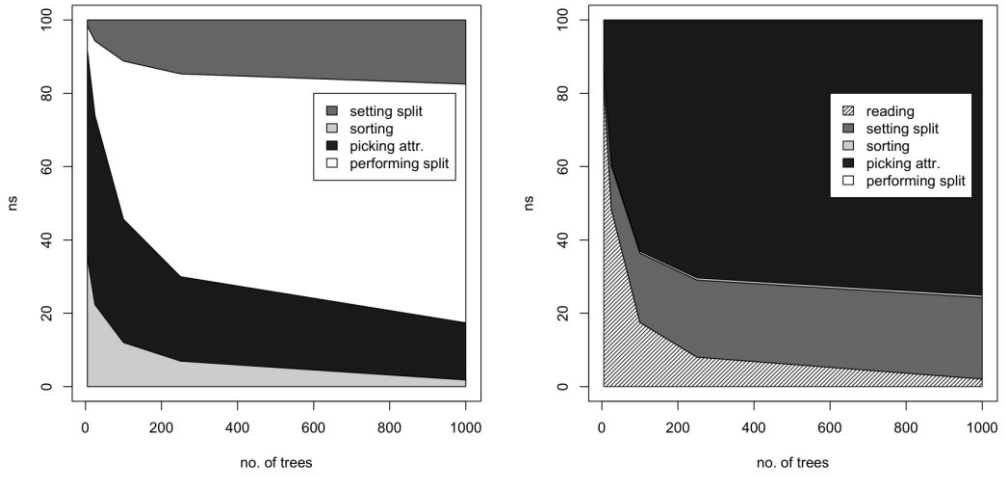

Fig. 7. Distribution of total running time versus number of trees: comparison of file-based fast random forest (left) and memory-based (right) implementation with 5,000 attributes and 200 records. The units ns on the y-axis are the percentages of running time.

The algorithm was executed on larger input data. For a data set with 30,000 attributes and 400 records, the file-based implementation of the algorithm constructed 5 trees within 2 minutes and 1,000 trees within 34 minutes. For a larger data set with 30,000 attributes and 800 records, the file-based implementation built 5 trees and 1,000 trees within 7 minutes and 140 minutes, respectively.

CONCLUSIONS

Random forest shows excellent predictive accuracy for high dimensional genomic data but efficiency and scalability are issues of concern. In this work we addressed scalability by restricting our attention to a classification problem for large microarray datasets containing thousands of numeric gene expression predictors. We presented a new framework for scaling up random forest to larger datasets. We described its design and its performance. The fast random forest implementation improved the learning time of the algorithm without loss of accuracy, and it allowed the classification to be performed on large disk resident datasets. Experiments on gene expression data have demonstrated its efficiency

Project home page

Optimized implementation of Random Forest in Java is available for download at www.colorectal.ccf.org/randomforest .

Acknowledgments

Part of this work was supported by NOIA Fund for Data Mining.

References

1. Breiman L. Random forest. *Machine Learning* 2001; 45:5-32.
2. Bureau A., Dupuis J, Falls K, Lunetta KL, Hayward B, Keith TP, Van Eerdewegh P. Identifying SNPs predictive of phenotype using random forests. *Genetic Epidemiology* 2005; 28:171-82.
3. Diaz-Uriarte R, Alvarez de Andres S. Gene selection and classification of microarray data using random forest. *BMC Bioinformatics* 2006; 7(3).
4. Garcia-Magarinos M, Lopez-de-Ullibarri I, Cao R, Salas A. Evaluating the ability of tree-based methods and logistic regression for the detection of SNP-SNP interaction. *Annals of Human Genetics* 2009; 73(3):360-9.
5. Han P, Zhang X, Feng ZP. Predicting disordered regions in proteins using the profiles of amino acid indices. *BMC Bioinformatics* 2009; 10.
6. Kim BK, Lee WJ, Park PJ, Shin YS, Lee WY, Lee KA, Ye S, et al. The multiplex bead array approach to identifying serum biomarkers associated with breast cancer. *Breast Cancer Research* 2009; 11(2).
7. Kumar KK, Pugalenthi G, Suganthan PN. DNA-prot: Identification of DNA binding proteins from protein sequence information using random forest. *Journal of Biomolecular Structure and Dynamics* 2009; 26(6):679-86.
8. Lunetta KL, Hayward LB, Segal J, Van Eerdewegh P. Screening large-scale association study data: Exploiting interactions using random forests. BMC *Genetics* 2004; 5(1):32.
9. Shi T, Seligson D, Belldegrun AS, Palotie A, Horvath S. Tumor classification by tissue microarray profiling: Random forest clustering applied to renal cell carcinoma. *Modern Pathology* 2005; 18(4):547-57.
10. Breiman L. Random forest. *Machine Learning* 2001; 45(1):5-32.
11. Liaw A, Wiener M. Classification and regression by random forest. *Rnews* 2002; 2:18-22.
12. Witten IH, Frank E. *Data mining: Practical machine learning tools and techniques*. 2nd ed. Morgan Kaufmann, San Francisco, CA. 2005.
13. Borisov A., Chikalov I, Eruhimov V, Tuv E. Performance and scalability analysis of tree-based models in large-scale data- mining. *Intel Technology Journal* 2005; 9(02):143-50.
14. Mehta M, Agrawal R, Rissanen J. SLIQ: A fast scalable classifier for data mining. In Proc. of *Extending Database Technology* 1996; 1057:18-32.
15. Shafer J, Agrawal R, Mehta M. SPRINT: A scalable parallel classifier for data mining. In Proc. of *Very Large Data Bases* 1996; 544-55.
16. Gehrke J, Ramakrishnan R, Ganti V. RainForest: A framework for fast decision tree construction of large datasets. In Proc. of *Very Large Data Bases* 1998; 416-27.

17. Breiman L, Friedman J, Stone CJ, Olshen RA. *Classification and regression trees*. Chapman Hall, New York. 1984.
18. Jiang Y, Casey G, Lavery IC, Zhang Y, Talantov D, Martin-McGreevy M, Skacel M, Manilich E, Mazumder A, Atkins D, Delaney CP, Wang Y. Development of a clinically feasible molecular assay to predict recurrence of stage II colon cancer. *J Mol Diagn*. 2008; 10(4):346-54.