George Mason University

CSI 702 Project Report

Comparison of Random Forest and Ranger in R for Letter Recognition dataset

Submitted by

Ajay Kulkarni G01024139

Table of contents

Abstract

The main objective of the project was to analyze the Random Forest and Ranger in R. For analyzing the Random Forest and Ranger Letter Recognition Dataset has been used. R provides different options for parallelism but for this project shared memory parallelism is used. It is achieved using “foreach” and “doMC” libraries in R. To analyze the performance of Random Forest and Ranger two cases were studied. In one case 1000 decision trees were built and in another case, 5000 decision trees were built for both the algorithms. The performance matrices used for this project were execution time, speedup, efficiency, and cost. Further, profiling is also done using “Rprof” command in R to understand more about the functions.

Chapter 1: Introduction

1.1 Letter Recognition Dataset

Letter Recognition dataset has been used in this project to analyze the performance of Random Forest and Ranger. Letter Recognition dataset has been taken from the UCI Machine Learning Repository. Dataset consist of 20 different fonts and each letter within these 20 fonts were randomly distorted to produce file consist of 20,000 unique stimuli. Each stimulus was converted into 16 numerical attributes that were in turn submitted to classifier system. The 20 different fonts were designed by Dr. Allen V. Hershey, a mathematical physicist at U.S. Naval Weapons Laboratory.

Each character image was scanned pixel by pixel, to extract 16 numerical attributes. These attributes represent primitive statistical features of the pixel distribution. To achieve compactness, each attribute was then scaled linearly to a range of integer values from 0 to 15. This final set of values was adequate to provide a perfect separation of the 26 classes. The 16 attributes are given below,

```
> head(mydata)
  Letter X1 X2 X3 X4 X5 X6 X7 X8 X9 X10 X11 X12 X13 X14 X15 X16
1      T  2  8  3  5  1  8 13  0  6   6  10   8   0   8   0   8
2      I  5 12  3  7  2 10  5  5  4  13   3   9   2   8   4  10
3      D  4 11  6  8  6 10  6  2  6  10   3   7   3   7   3   9
4      N  7 11  6  6  3  5  9  4  6   4   4  10   6  10   2   8
5      G  2  1  3  1  1  8  6  6  6   6   5   9   1   7   5  10
6      S  4 11  5  8  3  8  8  6  9   5   6   6   0   8   9   7
```

Figure 1: Letter Recognition Dataset

X1: The horizontal position, counting pixels from the left edge of the image, of the center of the smallest rectangular box that can be drawn with all "on" pixels inside the box.

X2: The vertical position, counting pixels from the bottom, of the above box.

X3: The width, in pixels, of the box.

X4: The height, in pixels, of the box.

X5: The total number of "on" pixels on the character image.

X6: The mean horizontal position of all "on" pixels relative to the center of the box and divided by the width of the box. This feature has a negative value if the image is "left heavy" as would be the case for the letter L.

X7: The mean vertical position of all "on" pixels relative to the center of the box and divided by the height of the box.

X8: The mean squared value of the horizontal pixel distances as measured in X6 above. This attribute will have a higher value for images whose pixels are more widely separated in the horizontal direction as would be the case for the letters W or M.

X9: The mean squared value of the vertical pixel distances as measured in X7 above.

X10: The mean product of the horizontal and vertical distances for each "on" pixel as measured in X6 and X7 above. This attribute has a positive value for diagonal lines that run from bottom left to top right and a negative value for diagonal lines from top left to bottom right.

X11: The mean value of the squared horizontal distance times the vertical distances for each "on" pixel. This measures the correlation of the horizontal variance with the vertical position.

X12: The mean value of the squared vertical distance times the horizontal distance for each "on" pixel. This measures the correlation of the vertical variance with the horizontal position.

X13: The mean number of edges encountered when making systematic scans from left to right at all vertical positions within the box. This measure distinguished between letters like "W" or "M" and letters like "I" or "L".

X14: The sum of the vertical positions of edges encountered as measured in X13 above. This feature will give a higher if there are more edges at the top of the box, as in the letter "Y".

X15: The mean number of edges encountered when making systematic scans of the image from bottom to top over all horizontal positions within the box.

X16: The sum of horizontal positions of edges encountered as measured in X15 above.

Multiple Linear Regression and Boruta were used for the variable selection. Both the methods have given similar results i.e. all the sixteen variables are important. So for building model, all the sixteen variable are selected and selected model is given as follows,

Selected Model

Letter ~ X1 + X2 + X3 + X4 + X5 + X6 + X7 + X8 + X9 + X10 + X11 + X12 + X13 + X14 + X15 + X16

1.2 Random Forest

Random Forest is one of the most widely used machine learning algorithm for classification. It can also be used for regression model but it mainly performs well on classification model. Random Forests are an ensemble learning method for classification and regression that operate by constructing a lot of decision trees at training time and outputting the class that is the mode of the classes output by individual trees. Each tree is grown as,

1. Each tree is trained on roughly 2/3rd of the total training data. Cases are drawn at random with replacement from the original data. This sample will be the training set for growing the tree.

2. Some predictor variables (m) are selected at random out of all the predictor variables and the best split on these m is used to split the node. By default, m is square root of the total number of all predictors for classification.

3. For each tree using the leftover data calculate the misclassification rate – Out Of Bag (OOB) error rate. Aggregate error from all tress to determine overall OOB error rate for the classification.

4. Each tree gives a classification and we say the tree "votes" for that class. The forest chooses the classification having the most votes over all the trees in the forest. This is the RF score and the percent votes received is the predicted probability.

1.3 Ranger

Ranger is the fast implementation of Random Forest for high dimensional data. The problem with Random Forest is that it is not optimized for the high dimensional data so it is time consuming as well as memory consuming to use Random Forest on high dimensional data. The core of ranger is implemented in C++ and uses standard libraries only. The R package "Rcpp" was employed to make the new implementation available as R packages, reducing the installation to a single command and simplifying its usage.

Ranger is optimized for high dimensional data by the extensive runtime and memory profiling. For different types of input data bottlenecks were identified and optimized the relevant algorithms. The most crucial part was the node splitting and two different algorithms were used. The first one sorts the feature values beforehand and accesses them by their index. In the second algorithm, the raw values are retrieved and sorted while splitting. In the runtime optimized code, the first version is used in large nodes and the second one in small nodes. In memory-efficient mode, only the second algorithm is used. Major improvements were also achieved by using the algorithm by Knuth (1985 P. 137) for sampling without replacement.

To analyze the performance of the Random Forest and Ranger R is used. The details about the libraries for parallelization and performance matrices are given in chapter 2. Analysis about the performance is explained in chapter 3 and the findings of this analysis are concluded in chapter 4 of the report.

Chapter 2: Parallelism in R and Performance Matrices

R provides different options for parallelism. For this project shared memory parallelism is implemented using doMC and foreach libraries. Also after parallelizing the code on a different number of cores some performance matrices were used to compare the performance and to decide which algorithm is better. The details about libraries and performance matrices are as follows,

2.1 doMC and foreach

The foreach package provides a new looping construct for executing R code repeatedly. The main reason for using the foreach package is that it supports parallel execution. It means that it can execute those repeated instructions on multiple processors/cores. The doMC package is a "parallel backend" for the foreach package. It provides a mechanism needed to execute foreach loop in parallel. The foreach package must be used in conjunction with a package such as doMC in order to execute code in parallel. The user must register a parallel backend to use otherwise foreach will execute sequentially even when the %dopar% operator is used.

To register doMC to be used with foreach one must call the registerDoMC function. This function takes only one argument named "cores". This specifies the number of cores on the machine. By default, the multicore package will use the value of the "cores" option, as specified with the standard "options" function. If it isn't set, then multicore will try to detect the number of cores and use approximately half that many workers. The example of the use of doMC and foreach is given below,

```
library(randomForest)
library(foreach)
library(doMC)

newdata <- read.csv('letter-recognition.data')

registerDoMC(2)

rf <- foreach(ntree=rep(500,2), .combine = combine, .packages = 'randomForest') %dopar%
randomForest(Letter~X1+X2+X3+X4+X5+X6+X7+X8+X9+X10+X11+X12+X13+X14+X
15+X16,data = newdata,ntree=ntree,confusion=TRUE,importance=TRUE)
```

The above code is for executing Random Forest on two cores. Parallelism is achieved by building 500 trees on each core simultaneously. Two cores are used for the execution, therefore, "*registerDoMC(2)"* is used. After execution of the code, the output will get combine by using "*.combine = combine"* command and it will get stored in the variable "rf". "*%dopar%*" is used to indicate R that the following function should be executed on every core/processor.

2.2 Performance Matrices

It is very important to study the performance of the parallel program with a view of determining the best algorithm. A number of metrics have been used based on the desired outcome of performance analysis. For this analysis, four matrices were used which are as follows,

1. Execution Time

The serial runtime T_S of a program is the time elapsed between the beginning and the end of its execution on a sequential computer. The parallel runtime T_P is the time that elapses from the moment a parallel computation starts to the moment the last processing element finishes execution.

2. Speedup

Speedup is a measure that captures the relative benefit of solving a problem in parallel. When evaluating a parallel system, we are often interested in knowing how much performance gain is achieved by parallelizing a given application over a sequential implementation. It is defined as the ratio of the time taken to solve a problem on a single processing element to the time required to solve the same problem on a parallel computer with p identical processing elements. In ideal parallel system speedup is equal to p but in practice speedup is less than p. Speedup is denoted by symbol S and it is given as,

$$S = \frac{T_S}{T_{Pi}}$$

3. Efficiency

Ideal behavior is not achieved because while executing a parallel algorithm, the processing elements cannot devote 100% of their time to the computation of the algorithm. Part of the time required by the processing elements for computation is spent in idling. So, efficiency is a measure of the fraction of time for which a processing element is usefully employed. It is defined as the ratio of speedup (S) to the number of processing elements. In ideal parallel system efficiency is equal to one but in practice efficiency is between 0 and 1. Efficiency is denoted by E and it is given as,

$$E = \frac{S}{P_i}$$

4. Cost

We define the cost of solving a problem on a parallel system as the product of runtime and the number of processing elements used. Cost reflects the sum of the time that each processing element spends solving the problem. Cost is sometimes referred to as work or processor-time product and a cost-optimal system.

$$C = T_p * P_i$$

Chapter 3: Results

Random Forest and Ranger has been used for training the model for 5000 trees and 1000 trees. To analyze the results both the algorithms were executed serially as well as parallelly on 2, 4, 8, 10 and 16 cores on Stampede supercomputer. The performance of the Random Forest and Ranger has been compared using execution time, speedup, efficiency, and cost. Results of the above analysis are explained below in detail.

3.1 Execution time

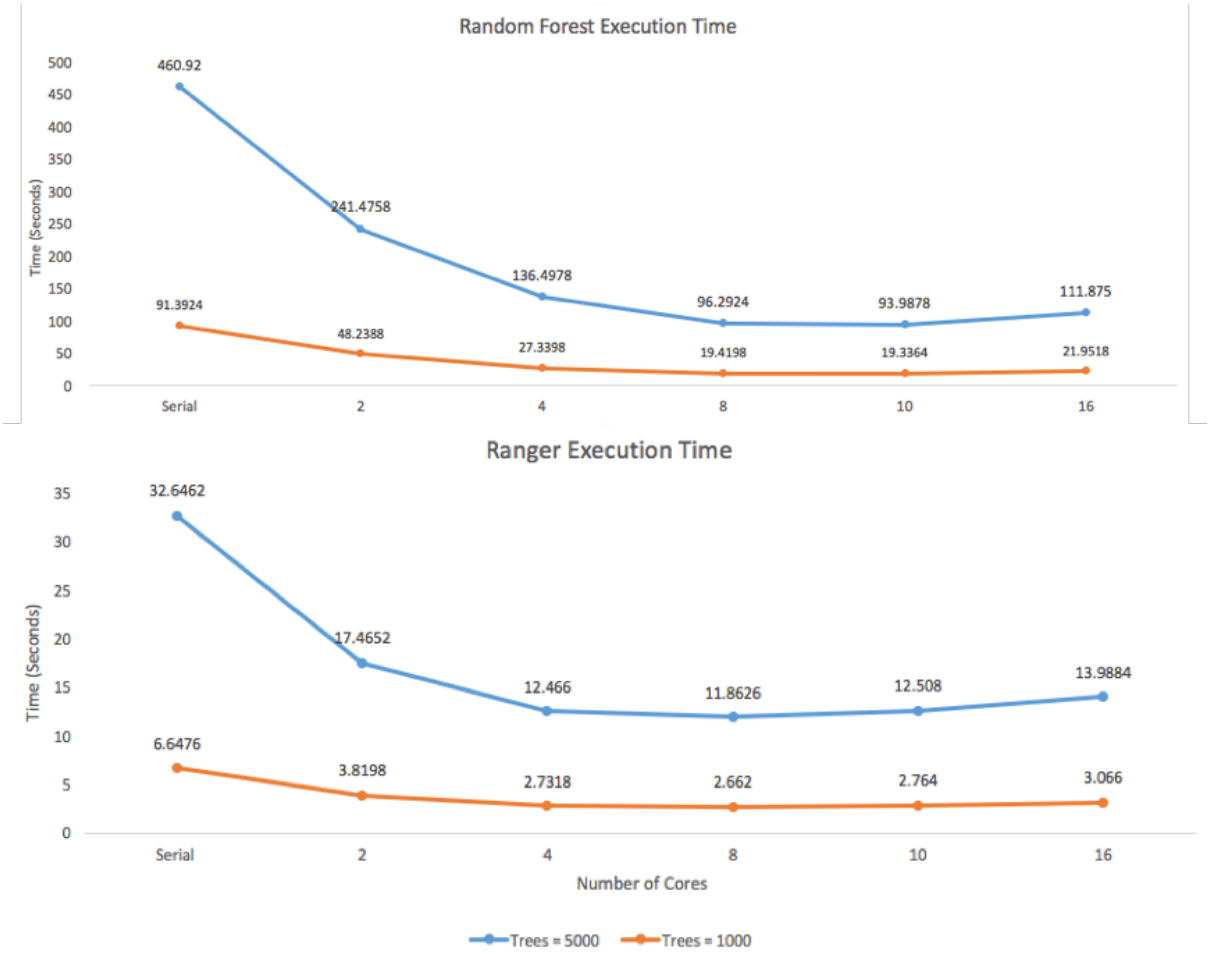


Figure 2: Execution Time for Random Forest and Ranger

The above plots show the execution time for Random Forest and Ranger. In both the plots, blue color indicates the 5000 trees and orange color indicates the 1000 trees. It can be observed from the above plot that for Random Forest execution time is increasing as the number of cores increasing but after 10 cores execution time is decreasing. So it may not be ideal to use more than 10 cores for Random Forest. In the case of Ranger similar results can be observed but after 8 cores the execution time is decreasing. So in the case of Ranger, it is not

ideal to use more than 8 cores. It is also observed that execution time is very less for Ranger as compared to Random Forest. So, the Ranger is really a fast implementation of Random Forest.

	Random Forest			Ranger		
Number of cores	Time for Trees = 5000 (Sec)	Time for Tress = 1000 (Sec)	Percentage increase	Time for Trees = 5000 (Sec)	Time for Tress = 1000 (Sec)	Percentage increase
1	460.900	91.392	404.31%	32.6462	6.6476	391.1%
2	241.758	48.239	401.17%	17.4652	3.8198	357.23%
4	136.498	27.340	399.26%	12.4660	2.7318	356.33%
8	96.292	19.420	395.84%	11.8626	2.6620	345.63%
10	93.988	19.336	386.08%	12.5080	2.7640	352.53%
16	111.575	21.952	408.27%	13.9884	3.0660	356.24%

Table 1: Execution time for Random Forest and Ranger

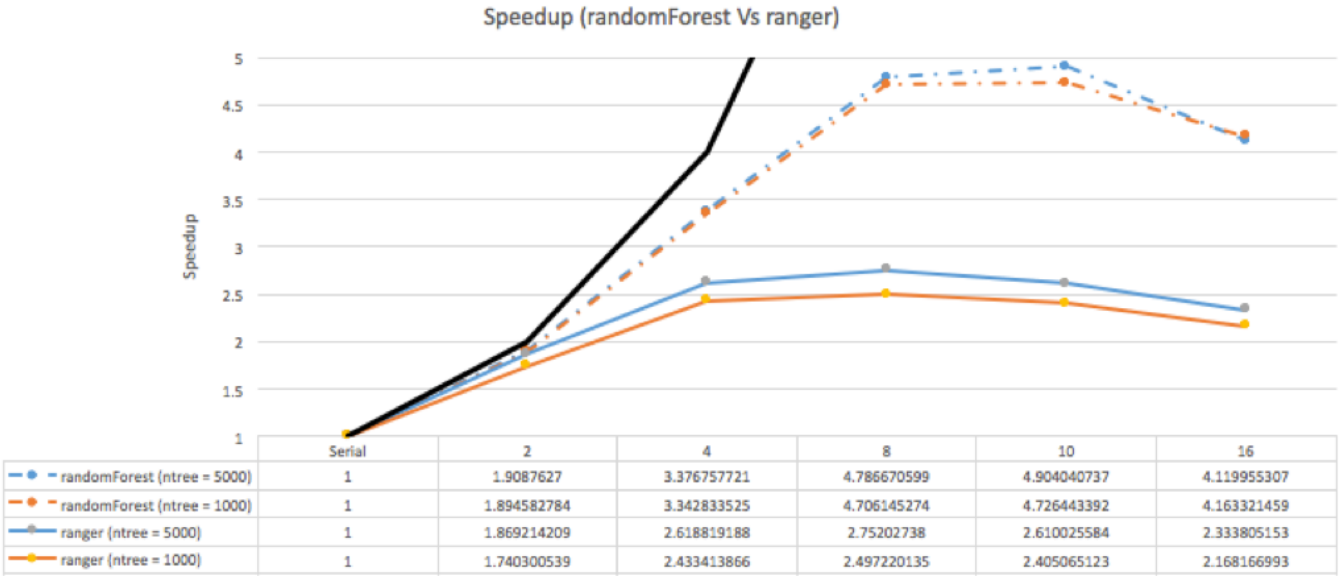
Above table indicates the percentage increase in execution time. It can be observed that in the case of Random Forest for serial execution if we want to switch from 1000 trees to 5000 trees percentage increase in execution time is about 404.31%. For parallel execution for different cores, the percentage change in execution time is in-between 380% to 410%. In the case of Ranger for serial execution, percentage time increase is about 391.1% and in the case of parallel execution, the percentage increase in execution time is in-between 350% to 360%. Also, it can be observed that change in percentage in Random Forest is less as compared to Ranger.

3.2 Speedup

Figure 3 indicates the speedup observed for Random Forest and Ranger. In figure 3, dotted lines represent Random Forest and solid lines represent Ranger. Also, blue color indicates 5000 trees and orange color indicates 1000 trees. In both the cases, the sublinear speedup is observed. In the case of Random Forest, it is observed that speedup is increasing as the number of cores are increasing but after 10 cores speedup is decreasing. The maximum speedup is obtained for 10 cores but the difference between speedup for 8 cores and 10 cores is considerably small. In the case of speedup pattern for both the number of trees (5000 & 1000), a similar pattern can be observed.

In the case of Ranger, the speedup is less as compared to Random Forest. It can be seen that up to 2 cores both Random Forest and Ranger have approximately similar speedups but after 2 cores the ranger is giving less speedup as compared to Random Forest. It is also observed that for 8 cores highest speedup is achieved and after that, the speedup is decreasing.

In the case of a pattern of speedup for 1000 and 5000 trees, 5000 trees are showing more speedup as compared to 1000 trees in Ranger.



	Serial	2	4	8	10	16
randomForest (ntree = 5000)	1	1.9087627	3.376757721	4.786670599	4.904040737	4.119955307
randomForest (ntree = 1000)	1	1.894582784	3.342833525	4.706145274	4.726443392	4.163321459
ranger (ntree = 5000)	1	1.869214209	2.618819188	2.75202738	2.610025584	2.333805153
ranger (ntree = 1000)	1	1.740300539	2.433413866	2.497220135	2.405065123	2.168166993

Figure 3: Speedup for Random Forest and Ranger

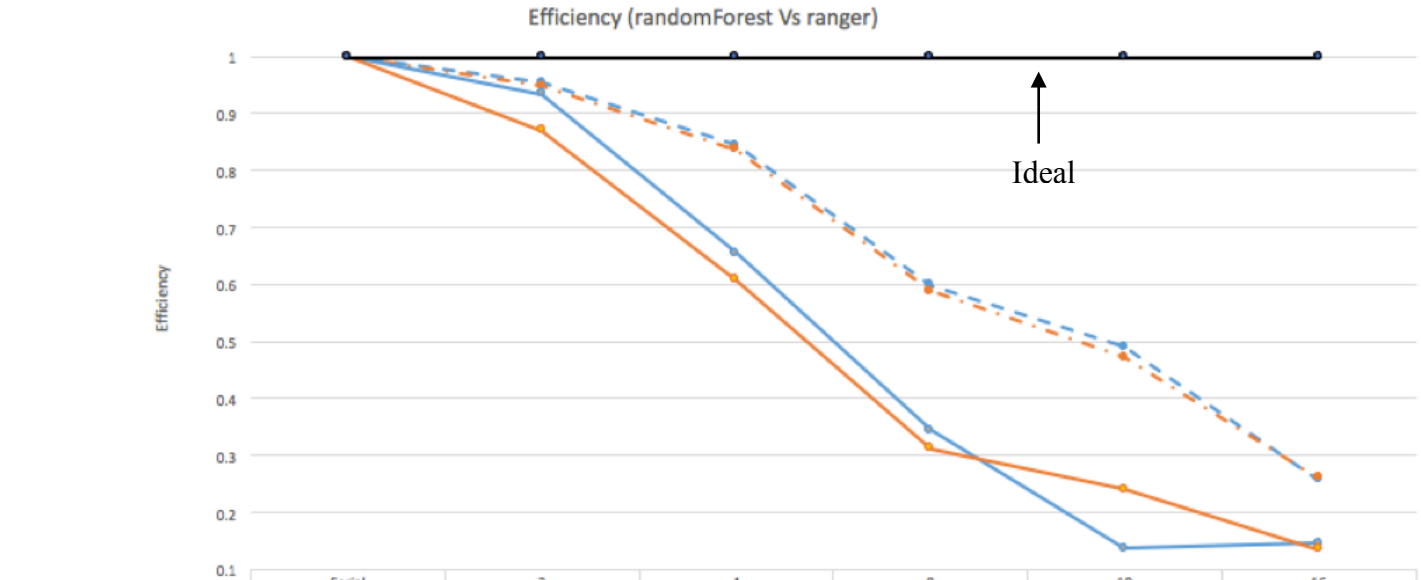
	Random Forest				Ranger			
Number of cores	Speedup for Trees = 5000	Percentage change in speedup	Speedup for Trees = 1000	Percentage change in speedup	Speedup for Trees = 5000	Percentage change in speedup	Speedup for Trees = 1000	Percentage change in speedup
1	1	-	1	-	1	-	1	-
2	1.90876	90.88%	1.89458	89.46%	1.86921	86.92%	1.74030	74.03%
4	3.37676	237.68%	3.34283	234.28%	2.61882	161.88%	2.43341	143.34%
8	4.78667	378.67%	4.70615	370.62%	***2.75203***	***175.2%***	***2.49722***	***149.72%***
10	***4.90404***	***390.4%***	***4.72644***	***372.64%***	2.61003	161%	2.40507	140.51%
16	4.11996	312%	4.16332	316.33%	2.33381	133.38%	2.16817	116.82%

Table 2: Speedup for Random Forest and Ranger

Table 2 indicates the percentage change in the speedup for Radom Forest and Ranger. It is observed that in the case of Random Forest maximum speedup is observed for 10 cores which is about 390.4% for 1000 trees and in the case 5000 trees it is about 372.64%. Also for 16 cores, the speedup is reduced about 78.4% for 1000 trees and 56.31% for 5000 trees. In the case of Ranger, it is observed that for 8 cores maximum speedup is observed which is about 175.2% for 1000 trees and for 5000 trees it is about 149.72%. The reduction in speedup is about 14.2% and 9.21% for 1000 and 5000 trees respectively. In both the cases, it is observed that as

the number of trees are increasing speedup is also increasing but Random Forest is giving more speedup than Ranger.

3.3 Efficiency



	Serial	2	4	8	10	16
randomForest (ntree = 5000)	1	0.95438135	0.84418943	0.598333825	0.490404074	0.257497207
randomForest (ntree = 1000)	1	0.947291392	0.835708381	0.588268159	0.472644339	0.260207591
ranger (ntree = 5000)	1	0.934607104	0.654704797	0.344003423	0.137369768	0.145862822
ranger (ntree = 1000)	1	0.87015027	0.608353467	0.312152517	0.240506512	0.135510437

Figure 4: Efficiency for Random Forest and Ranger

Figure 4 indicates the efficiency observed for Random Forest and Ranger. In figure 4, dotted lines represent Random Forest and solid lines represent Ranger. Also, blue color indicates 5000 trees and orange color indicates 1000 trees. It is observed that Random Forest have more efficiency as compared to Ranger. For Random Forest it is observed that for 5000 and 1000 trees efficiency is decreasing as the number of cores are decreasing. Also, the pattern of efficiency is same for 5000 and 1000 trees in Random Forest. In the case of Ranger, it can be observed that as the cores are increasing efficiency is decreasing. The efficiency is slightly more for 5000 trees as compared to 1000 trees for Ranger. For 10 cores efficiency is minimum and for 16 cores both 1000 and 5000 trees have approximately similar efficiency.

Table 3 indicates the percentage decrease in efficiency for Random Forest and Ranger. From the table 3, we can say that for Random Forest in the case of 5000 trees it has slightly more efficiency than 1000 trees for all cores except 16 cores. Also, in the case of Ranger for 5000 trees, it is reflecting slightly more efficiency than 1000 trees till 8 cores. For Ranger, it is also observed that for 1000 trees the least efficiency is observed for 16 cores which is about 86.45% decrease in efficiency and for 5000 trees least efficiency is observed for 8 cores which is about 86.26% decrease in efficiency. By analyzing both, the table as well as the plot we can say that Random Forest is more efficient as compared to Ranger for serial as well as for parallel execution.

	Random Forest				Ranger			
Number of cores	Efficiency for Trees = 5000	Percentage decrease in efficiency	Efficiency for Trees = 1000	Percentage decrease in efficiency	Efficiency for Trees = 5000	Percentage decrease in efficiency	Efficiency for Trees = 1000	Percentage decrease in efficiency
1	1	-	1	-	1	-	1	-
2	0.95438	4.56%	0.94729	5.27%	0.93461	6.54%	0.87015	12.99%
4	0.84419	15.58%	0.83571	16.43%	0.65470	34.53%	0.60835	39.17%
8	0.59833	40.17%	0.58827	41.17%	0.34400	65.6%	0.31215	68.79%
10	0.49040	50.96%	0.47264	52.74%	***0.13737***	***86.26%***	0.24050	75.95%
16	***0.25750***	***74.25%***	***0.26021***	***73.98%***	0.14586	85.41%	***0.13551***	***86.45%***

Table 3: Efficiency for Random Forest and Ranger

3.4 Cost

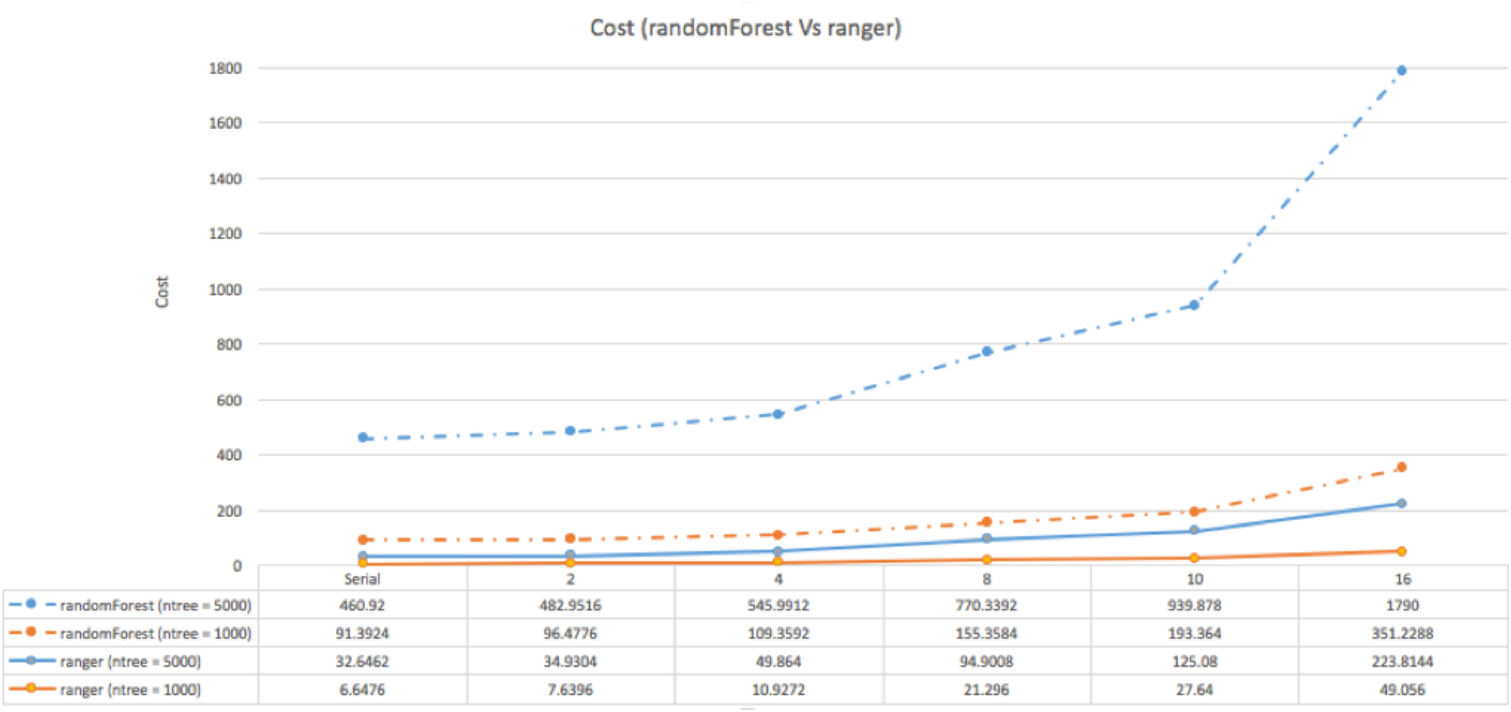


Figure 5: Cost for Random Forest and Ranger

Figure 5 indicates the efficiency observed for Random Forest and Ranger. In figure 5 dotted lines represent Random Forest and solid lines represent Ranger. Also, blue color indicates 5000 trees and orange color indicates 1000 trees. It can be observed from the figure 5 that Random Forest is very costly for both 5000 and 1000 trees. It also can be observed that

cost is increasing with the increasing number of cores as well as with the number of trees. We can also observe that for Ranger in case 5000 trees the cost is less as compared to Random Forest for 1000 trees. So Ranger is a cost efficient algorithm to use.

3.5 Profiling

Profiling helps to identify the bottlenecks and pieces of codes that need more efficient implementation. In R profiling is performed using "Rprof" function and it implemented for both Random Forest and Ranger. The results of profiling are given below,

```
> newdata <- read.csv('letter-recognition.data')
> Rprof()
> rf <- randomForest(Letter~X1+X2+X3+X4+X5+X6+X7+X8+X9+X10+X11+X12+X13+X14+X15+X16,data = newdata,ntree=1000,importance=TRUE)
> Rprof(NULL)
> summaryRprof(rf)
Error in file(filename, "rt") : invalid 'description' argument
> summaryRprof()
$by.self
                      self.time self.pct total.time total.pct
".C"                      87.90    94.46      87.90     94.46
"matrix"                   1.86     2.00       1.86      2.00
"array"                    0.88     0.95       0.88      0.95
"randomForest.default"     0.74     0.80      93.04     99.98
"integer"                  0.66     0.71       0.66      0.71
"aperm.default"            0.62     0.67       0.62      0.67
"double"                   0.26     0.28       0.26      0.28
"apply"                    0.04     0.04       0.08      0.09
"/"                        0.02     0.02       0.02      0.02
"cbind"                    0.02     0.02       0.02      0.02
"data.matrix"              0.02     0.02       0.02      0.02
"na.omit.data.frame"       0.02     0.02       0.02      0.02
"t.default"                0.02     0.02       0.02      0.02

$by.total
                      total.time total.pct self.time self.pct
"randomForest"             93.06    100.00      0.00     0.00
"randomForest.formula"     93.06    100.00      0.00     0.00
"randomForest.default"     93.04     99.98      0.74     0.80
".C"                       87.90     94.46     87.90    94.46
"matrix"                    1.86      2.00      1.86     2.00
"aperm"                     1.50      1.61      0.00     0.00
"array"                     0.88      0.95      0.88     0.95
"integer"                   0.66      0.71      0.66     0.71
"aperm.default"             0.62      0.67      0.62     0.67
"double"                    0.26      0.28      0.26     0.28
"t"                         0.10      0.11      0.00     0.00
"apply"                     0.08      0.09      0.04     0.04
"/"                         0.02      0.02      0.02     0.02
"cbind"                     0.02      0.02      0.02     0.02
"data.matrix"               0.02      0.02      0.02     0.02
"na.omit.data.frame"        0.02      0.02      0.02     0.02
"t.default"                 0.02      0.02      0.02     0.02
"do.call"                   0.02      0.02      0.00     0.00
".External2"                0.02      0.02      0.00     0.00
"FUN"                       0.02      0.02      0.00     0.00
"is.na"                     0.02      0.02      0.00     0.00
"is.na.data.frame"          0.02      0.02      0.00     0.00
"model.frame"               0.02      0.02      0.00     0.00
"model.frame.default"       0.02      0.02      0.00     0.00
"na.omit"                   0.02      0.02      0.00     0.00

$sample.interval
```

Figure 6: Random Forest Profiling Result

Figure 6 shows the profiling result for Random Forest. In the above figure, $by.self represents the timing required for the function to execute while $by.total represents total time taken by the function including time taken by the function for making calls etc. It can be observed that Random Forest have many functions and the majority of the time is consumed by ".C" function. ".C" function takes about 94.46% of the total time for execution. So we may

say that if we want to optimize the Random Forest then we may need to optimize the function ".C".

```
> library(ranger)

> library(tictoc)
> newdata <- read.csv('letter-recognition.data')
> Rprof()
> rf <- ranger(Letter~X1+X2+X3+X4+X5+X6+X7+X8+X9+X10+X11+X12+X13+X14+X15+X16,data = newdata,num.tree=1000)
> Rprof(NULL)
> summaryRprof()
$by.self
               self.time self.pct total.time total.pct
".Call"            59.26    99.97      59.26     99.97
"as.character"      0.02     0.03       0.02      0.03

$by.total
                  total.time total.pct self.time self.pct
"ranger"               59.28    100.00      0.00     0.00
".Call"                59.26     99.97     59.26    99.97
"rangerCpp"            59.26     99.97      0.00     0.00
"as.character"          0.02      0.03      0.02     0.03
"factor"                0.02      0.03      0.00     0.00
"integer.to.factor"     0.02      0.03      0.00     0.00

$sample.interval
[1] 0.02

$sampling.time
[1] 59.28
```

Figure 7: Ranger Profiling Result

Figure 7 shows the profiling of Ranger. It shows that Ranger has fewer functions as compared to Random Forest. Also, it is observed that ".Call" function is taking the majority of the time which is about 99.97% of the total time for execution. The calls made by ".Call" function are also less as compared to Random Forest's ".C" function. Therefore, from the above result, it makes clear that Ranger is an optimized version of Random Forest.

Chapter 4: Conclusion

Random Forest and Ranger has been successfully compared and analyzed for Letter Recognition dataset. To analyze the Random Forest and Ranger 5000 and 1000 decision trees were built. Based on the performed analysis it became clear that Ranger is the optimized and fast implementation of Random Forest.

Random Forest is giving better speedup and efficiency up to 10 cores and after that, both are decreasing. In the case of Random Forest, slightly more speedup and efficiency are observed for 5000 decision trees as compared to 1000 decision trees. For Ranger speedup and efficiency are better till 8 cores and after that, both are decreasing. Also, it is observed that speedup and efficiency are better for 5000 decision trees then 1000 decision trees. So, we may say that Random Forest is ideal to use till 10 cores and Ranger is ideal to use till 8 cores.

In terms of cost, the cost of both the algorithms is increasing as the number of cores are increasing. If we compare Random Forest and Ranger in terms of cost, then Ranger is cost efficient as compared to Random Forest. Also, profiling of both the functions made it clear that Ranger has fewer functions. The time taken for execution is also very less for Ranger. Thus, it explains that Ranger is a better algorithm and cost efficient implementation as compared to Random Forest.

References

1. Lichman, M. (2013). UCI Machine Learning Repository [http://archive.ics.uci.edu/ml]. Irvine, CA: University of California, School of Information and Computer Science.

2. P. W. Frey and D. J. Slate. "Letter Recognition Using Holland-style Adaptive Classifiers". (Machine Learning Vol 6 #2 March 91)

3. http://www.listendata.com/2014/11/random-forest-with-r.html

4. ranger: A Fast Implementation of Random Forests for High Dimensional Data in C++ and R by Marvin N. Wright and Andreas Ziegler

5. https://users.info.uvt.ro/~petcu/calcul/PC-2.pdf

6. http://parallelcomp.uw.hu/ch05lev1sec2.html

7. http://www.glennklockwood.com/data-intensive/r/on-hpc.html

8. http://www.glennklockwood.com/data-intensive/r/parallel-options.html

9. http://www.bu.edu/tech/support/research/training-consulting/online-tutorials/matlab-pct/scalability/

10. https://cran.r-project.org/web/packages/tictoc/tictoc.pdf

11. https://cran.r-project.org/web/packages/doMC/vignettes/gettingstartedMC.pdf

12. https://www.r-bloggers.com/profiling-r-code/

Appendix

1. slurm script

```
#!/bin/bash

#SBATCH -J Rjob
#SBATCH -o out.stdout
#SBATCH -N 1
#SBATCH -p normal
#SBATCH -t 00:10:00
#SBATCH -n 16

export MC_CORES = 2

module load Rstats
R CMD BATCH foreach_rf_2.r
```

Script1: slurm script

The above script is used for submitting the job to Stampede. For parallel execution, only one node is used and code is executed using multiple cores. So the above example is for 2 cores. If you want to execute the code on multiple cores, then you need to initialize the number of cores to "MC_CORES". Also, the R script which you want to execute will get replaced by "foreach_rf_2.r".

2. Random Forest and Ranger scripts

```
library(randomForest)
library(foreach)
library(doMC)
library(tictoc)

newdata <- read.csv('letter-recognition.data')

registerDoMC(2)

tic()
rf <- foreach(ntree=rep(500,2), .combine = combine, .packages = 'randomForest') %dopar%
randomForest(Letter~X1+X2+X3+X4+X5+X6+X7+X8+X9+X10+X11+X12+X13+X14+X15+X16,da
ta = newdata,ntree=ntree,confusion=TRUE,importance=TRUE)
toc()
```

Script 2 : Random Forest Script

The script 2 is used for executing Random Forest on two cores. In the above script tic() and toc() functions are used for measuring the time taken for execution of the function. Also, in the above script the 500 trees are divided into two cores so, “ntree = rep(500,2)” is there. So if you want to use more than two cores you need to change the “ntree = rep(500,2)” instruction.

```
library(ranger)
library(foreach)
library(doMC)
library(tictoc)

newdata <- read.csv('letter-recognition.data')

registerDoMC(2)

tic()
rf <- foreach(ntree=rep(500,2), .multicombine = TRUE, .packages = 'ranger') %dopar%
ranger(Letter~X1+X2+X3+X4+X5+X6+X7+X8+X9+X10+X11+X12+X13+X14+X15+X16,data =
newdata,num.tree=ntree)
toc()
```

Script 3: Ranger Script

The script 3 is used for executing Ranger on two cores. Same changes as mentioned above needs to be done if you want to execute this script for more than two cores.