Accelerating Random Forest on Memory-Constrained Devices through Data Storage Reorganization

Camélia Slimani, Stéphane Rubini, Chun-Feng Wu, Yuan-Hao Chang, Jalil Boukhobza

Compas'2021 : Parallélisme/ Architecture / Système/ Temps Réel
MILC - Lyon, France, 5-9 juillet 2021

Accelerating Random Forest on Memory-Constrained Devices through Data Storage Reorganization

Camélia Slimani *, Stéphane Rubini *, Chun-Feng Wu †, Yuan-Hao Chang †, Jalil Boukhobza ‡
* : Univ Brest, Lab-STICC, CNRS, UMR 6285, F-29200 Brest, France
† : Institute of Information Science, Academia Sinica, Taipei, Taiwan
‡ : ENSTA Bretagne, Lab-STICC, CNRS, UMR 6285, F-29200 Brest, France

Abstract
Random forests is a widely used classification algorithm. It consists of a set of decision trees each of which is a classifier built on the basis of a random subset of the training data-set. In an environment where the memory work-space is low in comparison to the data-set size, when training a decision tree, a large proportion of the execution time is related to I/O operations. These are caused by data blocks swap-in from the storage device to the memory work-space. In traditional methods, data blocks contain elements that will be accessed and processed at different moments, thus, the spatial locality is low. In this paper, we seek to reduce the number of data-blocks swap-in operations by enhancing spatial locality. The idea is to re-order data-set blocks such as each block contains elements that are likely to be accessed together. Our experiments show that this method reduces random forest build time by 55% to 89%.

Mots-clés : Forêts Aléatoires, Stockage, Systèmes Embarqués, Hiérarchie Mémoire

1. Introduction

According to the International Data Corporation, the volume of data created between 2020 and 2024 will surpass the one created over the last 30 years [9]. This is mainly due to the billions of end-point devices used in transportation, medicine or entertainment [1]. These devices collect and analyze huge amounts of data to extract meaningful information [6]. Analysis are traditionally performed on the cloud to exploit high computation power. However, this requires to send collected data on the cloud, that is exposing them to security threats and increasing the network traffic and energy consumption [5]. While this is not considered as an issue for several applications, it can cause serious problems for critical data applications.
Processing data directly on embedded devices can be a solution for such an issue. Nevertheless, this solution struggles from the limitation in size of the main memory. In fact, memory capacity can hardly scale as fast as the volume of data to process [7]. In addition, embedded devices may be battery-backed, which makes energy consumption a major challenge to consider when executing such resource-hungry applications.
One of the wide spread data analysis algorithms is classification. Their objective is to assign a category to a recorded information according to observed features [6]. A classification model is trained on a learning set (data-set) of elements characterized by features and labeled with their real classes [6]. Among the classification algorithms, *Random Forests* (RF) [2] is a powerful and widely used one. It individually trains a set of decision trees. Each of them is a set of conditions based on the feature values which group elements that are in the same classes together. Thus,

the decision tree building process consists of identifying the best features that make it possible to obtain this grouping [13]. This process is iterative and requires to try a set of features.
Most of random forest implementations assume that the data-set can entirely fit into the main memory. As a consequence, when it is larger than the available memory work-space, a significant volume of data is swapped in and out of memory (from and to the storage device) according to the sequence of data requested during the tree building process.
Previous experiments measured that building a decision tree with a memory work-space 8 times smaller than the size of the data-set is 25 times slower than the case where the data-set can fit in memory [11]. This is due to two main reasons : **(1)** The same memory page can move several times since the elements that it contains are needed on different decision tree nodes, the temporal locality is poorly exploited in the traditional algorithms. We tackled this issue in our previous work [11]. **(2)** A weak spatial locality of the memory accesses ; the same memory page contains elements that are needed on different nodes, thus, it moves several times between main memory and the storage device as it contains data requested for processing different nodes.
To address this issue, our contribution consists in reorganizing the data-set on the storage device in a way that data that are likely to be accessed together during the tree building process (on the same decision tree nodes) are stored in neighbor blocks (thus enhancing spatial locality of data). This idea is motivated by a random forest decision trees property that we observed experimentally on some data-sets : if a pair of data elements are accessed in a small time frame during one decision tree building, they have a high probability to be accessed in a small time frame for the building of the other decision trees. Based on this property, we have used the first random forest tree building process to infer elements that will probably be accessed together. Then, we reorganized the data-set such as to group these elements on the same blocks. Once the new data-set written, it is used to build the remaining decision trees.
We evaluated our method on three real data-sets while varying the memory pressure (data-set size over the available memory work-space) and observed an enhancement on execution times of 55 to 89% in comparison to state-of-the-art method Ranger [12].
The remainder of this paper is as follows : Section 2 gives a brief background about random forests. The proposed method is detailed in Section 3. In Section 4, we evaluate the proposed method. Finally, Section 5 concludes this work and gives some perspectives.

2. Background on RF and Decision Tree Building

Here, we introduce RF and the decision tree building process. Table 1 gives the notations used. RF is a supervised machine learning algorithm used for classification and regression [13]. It is composed of (T) decision trees. The input of the learning phase of a RF is a *data-set*, that is a set of N observed data elements characterized by d features and labeled with their real class. Each decision tree is trained on a subset of this data-set according to the method explained in the next section. The objective of a decision tree is to predict the class of an element knowing its observed features. RF builds a set of decision trees to limit the prediction error. The final predicted class of the elements is the predominant class among the predicted classes of the T trees of the forest. In what follows, we will focus on binary decision trees.
A binary decision tree is a tree-like graph where each node represents a boolean condition that depends on a feature of the learning set. This boolean condition allows to split the elements of the learning set into two subsets : a subset of elements for which the condition is 'true' ; and another for which the condition is 'false'. The topmost node of the tree is called the *root node*. All the elements of the learning set are assigned to this node at the beginning of the decision

tree building process. The bottom nodes of a tree, called *leaf nodes*, are *pure* meaning that all elements assigned to them are from the same class. Thus, the objective of the training process is to find the sequences of conditions that allow to obtain pure nodes.

Notation	Description
D	Data-set
N	Number of elements of the data-set
T	Number of decision trees
M	Memory work-space size
d	Number of features of an element
F	Set of features to be tested

TABLE 1 – Notation table

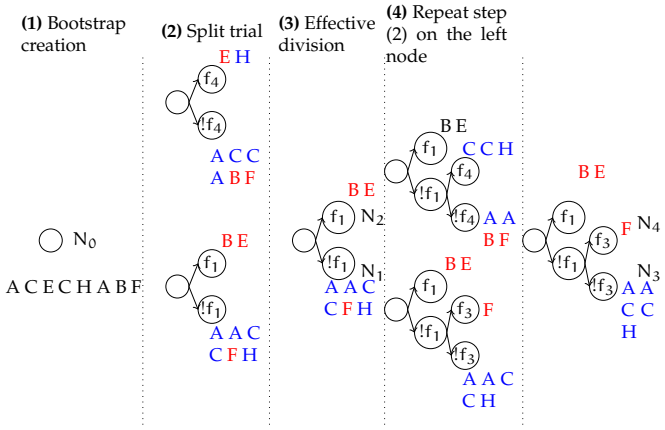

FIGURE 1 – Bagging Algorithm Illustration

A decision tree is built according to the bagging Algorithm [2]. To illustrate this process, we use the following example. We assume a data-set, given in Table 2, composed of 8 elements characterized by 4 features $\{f_1, f_2, f_3, f_4\}$. Figure 1 shows the steps of building a decision tree on the basis of this data-set :

1. Creation of the bootstrap (step 1) : A subset of the data-set, called *bootstrap*, is formed by a random sampling with replacement (i.e., each element can be sampled multiple times). Elements of the bootstrap are assigned to the root node.

2. Split trial (step 2) : The second step consists of splitting the elements of the bootstrap according to boolean conditions based on a random subset F of features. Once F is formed, the elements of the bootstrap are distributed according to their value for each feature, resulting in |F| potential trees. In Figure 1, the sampled features are f_1 and f_4. Elements of the bootstrap are distributed according to each of these features resulting in two possible trees.

3. Effective split (step 3) : This step consists of choosing the best splitting feature among the previous subset F. That is the one that allows to group the most elements of the same class together. Then, the node is effectively split according to the best feature. In Figure 1, f_1 is chosen since it already gives a pure leaf node as all the elements assigned to it are from the same class.

Storage Block	Label	f_1	f_2	f_3	f_4	Class
(1)	A	0	0	0	0	0
	B	1	0	1	0	1
(2)	C	0	1	0	1	0
	D	0	0	0	0	0
(3)	E	1	1	1	1	1
	F	0	1	1	0	1
(4)	G	1	0	0	1	0
	H	0	1	0	1	0

TABLE 2 – An example Data-set

Node	Elements	Accessed blocks	Percentage of used data in each block
N_0	{A, A, B, C, C, E, F, H}	(1), (2), (3), (4)	100%, 50%, 100%, 50%
N_1	{A, A, C, C, F, H}	(1), (2), (3), (4)	50%, 50%, 50%, 50%
N_2	{B, E}	(2), (3)	50%, 50%
N_3	{A, A, C, C, H}	(1), (2), (4)	50%, 50%, 50%
N_4	{F}	(3)	50%

TABLE 3 – I/O access pattern for splitting each node

Step 1, 2 and 3 are repeated with the resulting impure nodes until obtaining pure nodes.

3. Data Storage Reorganization for Random Forest

In this section, we give a motivational example. Then, we show that for a pair of elements assigned to the same leaf node in a given decision tree, the probability that they belong to the same node on another one is high. Finally, we describe our contribution based on this principle.

3.1. Motivational Example

In our study, we focus on storage device accesses that occur during the RF learning process. We consider the case where the data-set is larger than the available memory work-space.
Let's assume the data-set given in Table 2, and a memory work-space that can hold 4 elements with storage blocks of 2 elements each. The distribution of data-set elements within each blocks is given on the first column of Table 2.
Table 3 shows, for each node in the final tree in Figure 1, the elements that need to be accessed, the blocks that need to be swapped in (memory), and for each block the percentage of effectively used data. We draw two observations from this example : **(1)** When a data block is swapped in, its data are partially used (50% for most blocks) ; **(2)** the elements of each node are distributed on multiple data blocks.
So, the decision tree (we can extrapolate for multiple trees) building process poorly exploits spatial locality in several cases, and a substantial proportion of data are swapped in uselessly. These observations can be generalized to the data-sets where elements are uniformly distributed on the data blocks (that is data-set elements distribution on the blocks does not depend on the class information).

3.2. Measure of Similarity between RF decision trees

In what follows, we try to show that in a RF, if a pair of data elements are classified in the same leaf node in one tree, they are likely to be classified together in another tree. With such a property, the data-set can be reorganized on the storage device so that to reduce the number of swap-in operations. We formalize this property and evaluate its relevance.
Decision Trees Similarity : We assume a bootstrap B that contains N elements. T_1 and T_2 are two decisions trees built on the basis of the bootstrap B. Each decision tree results in a set of leaf nodes that group elements of the bootstrap that share the same features. Thus, each leaf node can be defined as a cluster of the bootstrap elements. Let's assume that tree T_1 (resp. T_2) gives the clustering P_1 (resp. P_2). To know if a pair of elements classified in the same leaf nodes of a decision tree are likely to be classified together in another one, we can compare the similarity of obtained clusterings P_1 and P_2.
In the literature, there exists several metrics to evaluate similarity between clusters [4]. We relied on the Adjusted Rand Index (ARI), which is a widely used one. Its value ranges between -1 and 1 ; a high ARI value means that the clusterings are similar.
Experimental measurement of the ARI : In order to check if this property is relevant, we measured the ARI index of the clusterings obtained using two decision trees, with multiple real data-sets picked from UCI Data-set Repository [3] : Wearable, Adult and Covertype. The obtained ARI values are : 0.27, 0.26 and 0.12, respectively. In order to evaluate the relevance of these results, we compared them to the ARI values considered as satisfactory in state-of-the-art work [4] [10]. The reference values were obtained by comparing clusterings obtained from algorithms (such as K-Means, Birch, GMM, etc), to the real clustering. All of the ARI values obtained in our experiment are included in the range of values considered as satisfactory which

is 0.1 to 0.29 [10] (exposing similarity). Thus, we consider that the similarity property between decision trees is relevant for the tested data-sets. We are working towards the generalization of this property with other data-sets to be generalized.

3.3. Data-set Reorganization Method

Our objective is to reduce swap-in operations when building a RF. As shown in the motivational example, when building a decision tree, elements that need to be accessed to split each node are distributed on multiple data blocks, and the swapped-in blocks are poorly exploited. Thus, our approach is to re-organize data-set blocks such as each of them contains elements that are likely to be accessed during the split of the same node. To do so, we take advantage of the property explained in the previous section. In fact, since the probability of similarity between leaf nodes is high, we extract information about elements clustering from the first tree, and then exploit this information to reorganize the data-set to be used for the remaining decision trees. This induces an additional full data-set write operation on the storage device, but it is profitable in view of the several trees to be built. The steps are explained hereafter.

1. Build the T_0 Tree : The objective of this first step is to build a decision tree that would allow to get a clustering of data-sets elements, where tree leaf nodes represent the clusters. In order to get a clustering of all data-set elements and not only the bootstrap, the decision tree T_0 (and only this one) is built on the basis of the whole data-set. As a consequence, the decision tree T_0 is deeper (more elements), thus slower to build, than ordinary decision trees. In addition, it is more subject to over-fitting. Thus, in our method, T_0 is a "disposable" tree used to partition data-set elements, it is not saved in the RF decision trees that are used for the inference step.

2. Re-write the data-set on the storage device : Once the decision tree T_0 obtained, we dispose of the clustering of all the data-set elements. The information about the clustering is used to rewrite the whole file in the storage device (we assume there is space on the storage device for such a write operation). A new data-set is written which is organized such as the elements of the same cluster (leaf node) are on the same (sequential) blocks. This way, elements of a block are likely to be accessed altogether during the building of the other trees. Note that, if the size of a block cannot contain all the elements of a cluster, they are stored on sequential blocks.

3. Effective Random Forests Trees Building : Once the data-set blocks are written according to the clustering obtained from the tree T_0, the remaining decision trees of the RF are formed on the basis of the newly stored data-set.

3.4. New I/O Access Pattern

The proposed method allows to form data-set blocks that are likely to contain elements that will be accessed together and thus better exploit spatial locality and reduce I/O accesses. To illustrate this optimization, we apply our method on the small example given in Section 2, considering the same bootstrap. We first form the tree T_0 ; the obtained tree is shown on Figure 2. The new I/O access pattern for the rest of the trees is given in Table 4. When comparing Table 3 to Table 4, we observe that the number of accessed data-set blocks in the proposed method is lower than that of the original bagging algorithm. The average percentage of effectively used data per blocks is 72% compared to 57% in the original method.

4. Evaluation

In this section, we first give our experimental methodology, then, we discuss the results.

Experimental Methodology : We compared the proposed method to the Ranger Framework [12] which is widely used. Ranger Framework proposes two methods of data indexations respecti-

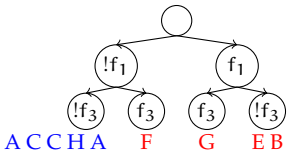

FIGURE 2 – T_0 Tree

Node	Elements	Accessed blocks	Percentage of used data in each block
N_0	{A, A, B, C, C, H, F, B, E}	(1), (2), (3), (4)	100%, 50%, 50%, 100%
N_1	{A, A, C, C, H, F}	(1), (2), (3)	100%, 50%, 50%
N_2	{B, E}	(4)	100%
N_3	{A, A, C, C, H}	(1), (2)	100%, 50%
N_4	{F}	(3)	50%

TABLE 4 – New I/O access pattern

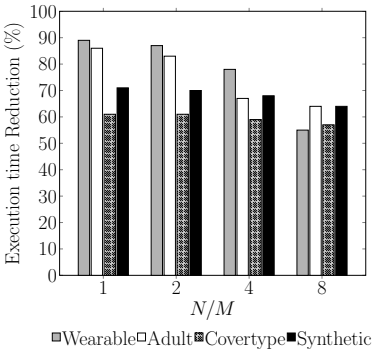

TABLE 5 – Evaluation Results

vely suitable for nodes that contain a lot or a few elements. The objective of ranger is to reduce memory footprint. To do so, we built a random forest with 20 trees, the default number of trees to build in Ranger is 100. If the proposed method is efficient for a limited number of trees (20), it means that it will be even more profitable for a higher number of trees. We ran the experiment under different memory constraints $N/M = \{1, 2, 4, 8\}$, such as N is the size of the data-set and M is the volume of available memory. The experiment is run with multiple data-sets : Real data-sets available on UCI Data-set Repository [3], and a synthetic one consisting of 64 features and 100.000 elements, generated using Scikit-Learn Framework [8].

Results : Figure 5 shows the results. The execution time reduction ranges between 55% and 89%. One must note that the execution time measured with our method includes the building of the first tree and the rewriting of the whole data-set on the storage device. The execution time reduction is due to the fact that blocks contain elements that are likely to be accessed together, thus, the number of blocks to be swapped-in to process a given node is reduced. The second observation that can be drawn is that the execution time reduction decreases when the memory constraint is increased. In fact, when the available memory work-space is very low in comparison to the volume of data to process, even if the spatial locality is better exploited, data blocks movements between memory work-space and swap space is substantial which slows down RF building. As N/M increases, both methods would reach an upper bound in term of swap-in operations for processing a given node. Since Ranger is not I/O optimized, it reaches this bound for lower N/M values than our method. Once this bound reached, the execution time of Ranger stabilizes while it continues to increase for our method. Then, the execution time difference between both methods decreases even if it remains good enough.

5. Conclusion

In this paper, we seek to reduce data movements from the storage device to the main memory when building a RF in a memory constrained device. The proposed method reorganizes the data-set in a way to group elements that will probably be accessed together on the same blocks, thus enhances spatial locality. Even though the proposed algorithm needs to rewrite the whole data-set into the storage device, the execution time reduction is substantial as several trees need to be created in RF. The evaluation of this method shows an execution time reduction of up to 89% as compared to a state-of-the-art method. As perspective, we aim to generalize this method to other Ensemble-Learning methods such as Boosting.

Bibliographie

1. Abbas (H.), Saha (I.), Shoukry (Y.), Ehlers (R.), Fainekos (G.), Gupta (R.), Majumdar (R.) et Ulus (D.). – Special session : Embedded software for robotics : Challenges and future directions. – In *2018 International Conference on Embedded Software (EMSOFT)*, pp. 1–10, 2018.
2. Breiman (L.). – Random Forests. *Machine Learning*, vol. 45, n1, 2001.
3. Dua (D.) et Graff (C.). – UCI machine learning repository, 2017.
4. Guyeux (C.), Chrétien (S.), Bou Tayeh (G.), Demerjian (J.) et Bahi (J.). – Introducing and comparing recent clustering methods for massive data management in the internet of things. *Journal of Sensor and Actuator Networks*, vol. 8, n4, 2019.
5. Kukreja (N.), Shilova (A.), Beaumont (O.), Huckelheim (J.), Ferrier (N.), Hovland (P.) et Gorman (G.). – Training on the edge : The why and the how. – In *2019 IEEE International Parallel and Distributed Processing Symposium Workshops*, 2019.
6. Mahdavinejad (M. S.), Rezvan (M.), Barekatain (M.), Adibi (P.), Barnaghi (P.) et Sheth (A. P.). – Machine learning for internet of things data analysis : a survey. *Digital Communications and Networks*, vol. 4, n3, 2018, pp. 161 – 175.
7. Mutlu (O.), Ghose (S.) et Ausavarungnirun (R.). – Recent advances in overcoming bottlenecks in memory systems and managing memory resources in GPU systems. *CoRR*, vol. abs/1805.06407, 2018.
8. Pedregosa (F.), Varoquaux (G.), Gramfort (A.), Michel (V.), Thirion (B.), Grisel (O.), Blondel (M.), Prettenhofer (P.), Weiss (R.), Dubourg (V.), Vanderplas (J.), Passos (A.), Cournapeau (D.), Brucher (M.), Perrot (M.) et Duchesnay (E.). – Scikit-learn : Machine learning in Python. *Journal of Machine Learning Research*, vol. 12, 2011, pp. 2825–2830.
9. Reinsel (D.), Rydning (J.) et Gantz (J. F.). – Worldwide global datasphere forecast, 2020–2024 : The covid-19 data bump and the future of data growth, Apr 2020.
10. Rodriguez (M.), Comin (C.), Casanova (D.), Bruno (O.), Amancio (D.), Rodrigues (F.) et da F. Costa (L.). – Clustering algorithms : A comparative approach. *PLOS ONE*, vol. 14, 12 2016.
11. Slimani (C.), Wu (C.-F.), Chang (Y.-H.), Rubini (S.) et Boukhobza (J.). – Rafio : A random forest i/o-aware algorithm. – In *The 36th ACM Symposium on Applied Computing (SAC 2021)*, 2021. – (To Appear).
12. Wright (M.) et Ziegler (A.). – ranger : A fast implementation of random forests for high dimensional data in C++ and R. *Journal of Statistical Software, Articles*, vol. 77, 2017.
13. Zhang (C.) et Ma (Y.). – *Ensemble machine learning : methods and applications*. – Springer, 2012.