Journal of Big Data

METHODOLOGY Open Access

Random forest implementation and optimization for Big Data analytics on LexisNexis's high performance computing cluster platform

Victor M. Herrera[1*], Taghi M. Khoshgoftaar[1], Flavio Villanustre[2] and Borko Furht[1]

*Correspondence: vherrer1@fau.edu
[1] Department of Computer and Electrical Engineering and Computer Science, Florida Atlantic University, Boca Raton, FL 33431, USA
Full list of author information is available at the end of the article

Abstract
In this paper, we comprehensively explain how we built a novel implementation of the Random Forest algorithm on the High Performance Computing Cluster (HPCC) Systems Platform from LexisNexis. The algorithm was previously unavailable on that platform. Random Forest's learning process is based on the principle of recursive partitioning and although recursion per se is not allowed in ECL (HPCC's programming language), we were able to implement the recursive partition algorithm as an iterative split/partition process. In addition, we analyze the flaws found in our initial implementation and we thoroughly describe all the modifications required to overcome the bottleneck within the iterative split/partition process, i.e., the optimization of the data gathering of selected independent variables which are used for the node's best-split analysis. Essentially, we describe how our initial Random Forest implementation has been optimized and has become an efficient distributed machine learning implementation for Big Data. By taking full advantage of the HPCC Systems Platform's Big Data processing and analytics capabilities, we succeed in enhancing the data gathering method from an inefficient *Pass them All and Filter* approach into an effective and completely parallelized *Fetching on Demand* approach. Finally, based upon the results of our learning process runtime comparison between these two approaches, we confirm the speed up of our optimized Random Forest implementation.

Keywords: Random forest, LexisNexis's high performance computing cluster (HPCC) systems platform, Optimization for Big Data, Distributed machine learning, Turning recursion into iteration

Introduction

The HPCC Systems Platform [1, 2] from LexisNexis Risk Solutions is an open source, parallel-processing computing platform designed for Big Data processing and analytics. HPCC is a scalable system based on hardware clusters of commodity servers. HPCC's clusters are coupled with system software to provide numerous data-intensive computing capabilities, such as a distributed file storage system, job execution environment, online query capability, parallel application processing, and parallel programming development tools [2]. The Enterprise Control Language (ECL) [1, 3], part of the HPCC platform, is a data-centric, declarative, and non-procedural programming language designed specifically for Big Data projects using the LexisNexis HPCC platform.

HPCC's machine learning abilities are implemented within the ECL-ML plug-in module, also known as the ECL-ML Library [4], which extends the capabilities of the base HPCC platform. The ECL-ML Library is an open source project [4] created to manage supervised and unsupervised learning, document and text analysis, statistics and probabilities, and general inductive inference-related problems in HPCC.

In 2012, as part of the NSF-sponsored Industry/University Cooperative Research Center for Advanced Knowledge Enablement (CAKE), we initiated the project "Developing Learning Algorithms on HPCC/ECL Platform" [5]. The project's main contributions have been the creation and implementation of machine learning algorithms which were previously unavailable on the HPCC platform, as well as the improvement of various existing algorithms and tools in the ECL-ML Library. One of the supervised learning algorithms added to the HPCC's ECL-ML Library was the Random Forest (RF) classifier [6]. Prior to our contribution, RF was not implemented in the ECL-ML Library. RF is a Decision Tree based ensemble supervised learning algorithm that combines Bootstrap Aggregating [7] (*Bagging*), which improves accuracy by incorporating more diversity and reducing the variance, with Random Feature Subspace Selection [8, 9], which improves efficiency by working faster only on subsets of the data features.

Our goal was to deliver the Random Forest classifier implementation on HPCC Systems Platform as a distributed machine learning algorithm for Big Data, i.e., an ECL-ML module ready to handle large volumes of data, within a reasonable time, by leveraging the HPCC Systems Platform's capabilities, such as data distribution and parallel computing. In order to fulfill our goal, we needed to overcome two main challenges:

- In general, traditional RF implementations do not scale well with Big Data because the computational complexity increases as the data size increases.
- RF's learning process is based on the recursive partitioning [10, 11] of Decision Trees; however, recursive function calls are not allowed in ECL.

Our final RF classifier implementation was completed in two main stages:

- First, the implementation on HPCC's Platform of a fully operative RF classifier for discrete and continuous variables, called initial RF implementation, which includes learning/model building, class probability calculation, and classification functionality.
- Second, the optimization of our initial RF classifier implementation on HPCC's Platform in order to handle the challenges associated with Big Data.

The rest of this paper is organized as follows. In "Related works" section, we present a brief summary of previous work related to the implementation of the RF algorithm. In "Methods" section, we introduce some basics about the HPCC Platform and its programming language (ECL) and present the background of the supervised learning implementations on HPCC's ECL-ML Library, in "Enterprise Control Language (ECL)" and "Supervised learning in HPCC platform" sections. Later, in "Implementing recursive based algorithms through iterations" section, we discuss the implementation of the recursive partitioning of Decision Trees as an iterative split/partition process using ECL. To finish with "Methods", in "Random Forest learning process in ECL-ML" and "Random Forest learning process optimization" sections, we explain our initial RF classifier implementation, analyze its flaws, and describe the modifications used to optimize its learning process. In "Results" and "Discussion" sections, we present and discuss our experimental results. We conclude with the discussion of our conclusions in "Conclusions" section. Additionally, in Appendix, we provide descriptions and examples of the main data structures employed in our implementation showing the operation of the Decision Tree model growth in the iterative process.

Related works

Random Forest [6] is a Decision Tree based ensemble supervised learning algorithm that combines Bootstrap Aggregating, also called Bagging [7], with Random Feature Subspace Selection [8, 9] at the node level (hereinafter referred to as Random Feature Selection). Bagging reduces the variance of single Decision Tree predictions by building an ensemble of independent Decision Trees, using a bootstrap sample of the training data, and making predictions of new instances by aggregating the predictions of the ensemble (majority vote for classification or averaging for regression). Bootstrapped samples of the training data contain instances sampled from the original dataset, with replacement, and are identical in size to the original dataset. The Random Feature Selection decreases the correlations among trees in the Random Forest, which reduces the chance of overfitting, improves the predictive performance, and accelerates the learning process by searching for the best split per node faster on smaller random subsets of the training data features. RF is considered one of the best available machine learning algorithms [12] for classification and regression: it is robust to outliers and noise, it overcomes overfitting problems, it can handle moderately imbalanced class data, and it is user-friendly, with just a couple of RF parameters to set. Furthermore, due to its algorithmic characteristics, such as intensive resampling and independent growing (parallel) of numerous Decision Tree models, RF is a good candidate for parallel implementation.

Since Breiman proposed the Random Forest algorithm in 2001 [6], RF implementations have evolved along with computation power, programming paradigms, the types of the data to be processed and the purpose of the processing. Starting with the original implementation by Breiman and Cutler [13] written in Fortran 77, RF has been implemented in many machine learning collections, such as Weka [14] using Java for its implementation [15], Scikit-Learn [16] developed in Python, and R System for Statistical Computing through the package RandomForest-R [17] implemented in C++/S, to name just a few non-distributed implementations. Looking at the representative examples of RF implementations used in bioinformatics circa 2012 in Boulesteix et al. [18], we noted that distributed RF implementations were not commonly used or did not exist at all by the time (January 2013) that we planned our RF implementation in the ECL-ML Library. Later in 2014, Fernandez-Delgado et al. [12] performed an exhaustive evaluation of 179 classifiers over the UCI Machine Learning Classification database (discarding the large-scale data sets) and found out that best results were achieved by a parallel Random Forest implementation in R.

Because of the fact that RF can learn from high-dimensional data, which is data with a large number of attributes and a large amount of instances, it is used to accurately classify big sets of data in different field domains. Related to RF applications in bioinformatics, Boulesteix et al. [18] stated in their review that "the Random Forest algorithm has evolved to a standard data analysis tool in bioinformatics." The authors summarized ten years of RF development with emphasis on bioinformatics and computational biology, with a focus on practical aspects of the RF algorithm to provide helpful guidelines for applications. In addition, they discussed essential pitfalls and shortcomings of RF and its variable importance measures, as well as alternative approaches to overcome these problems.

The application of machine learning algorithms for anomaly/behavior discovery on mobile devices is currently an emerging area of interest. Alam et al. [19] concluded that RF can be used for detecting Android malware providing an exceptionally high accuracy of over 99.9% of the samples correctly classified in their results. Another relatively new domain is Physical Activity (PA) Recognition using wearables devices; Ellis et al. [20] evaluated the use of the RF algorithm for predicting both PA type (household, stairs, walking, running) and Energy Expenditure (EE) from accelerometer and Heart Rate data. They found that the RF algorithm outperformed published methods for EE estimation and achieved relatively high average accuracy, 92.3% and 87.5% for hip and wrist accelerometers, predicting activity types. They ascribe their results to the RF's ability to model highly nonlinear patterns in the data.

On another subject related to our paper, Big Data analytics [21] is currently a hot topic. It takes special algorithms and professionals who can work with these algorithms to transform Big Data into meaningful information using a reasonable amount of time and resources. Moreover, the need to efficiently handle Big Data is growing extremely fast because of the proliferation of the social use of various features across the web and digital devices (smartphones, sensors, gadgets) producing lots of digital data that was previously unavailable. Random Forest is no stranger to Big Data's new challenges [21] and it is particularly sensitive to Volume, one of the Big Data characteristics defined in 2001 by Laney in his Meta Group (now Gartner) research report [22]. Consequently, RF optimization for large amounts of data became mandatory, with the parallelization of its learning process being key. This was mainly due to the fact that other ML algorithms, such as Convolutional Neural Networks, Support Vector Machines, and Latent Dirichlet Allocation, were extremely accelerated using highly-parallel Graphics Processing Units (GPU). Liao et al. [23] introduced CudaTree, a hybrid parallel GPU Random Forest implementation which adaptively switches between data and task parallelism, and compared its performance with two commonly used multi-core RF libraries: Scikit-learn and wiseRF.

Another approach to deal with Big Data analytics challenges is the use of distributed storage and processing. In a recent paper, Landset et al. [24] presented a comprehensive review of the current state-of-the-art open source scalable tools for ML in the Hadoop ecosystem; taking an in-depth look at the strengths and weaknesses of the various ML tools for Hadoop, they stated that Mahout [25] and MLlib [26] are the most well-rounded Big Data libraries in terms of ML algorithm coverage and both work with Spark [27] and H2O [28]. Even though the Random Forest algorithm has been already implemented as distributed machine learning algorithms in Spark and H20 (both use distributed data frames), it is not the focus of this paper to carry out a comparison between our RF implementation in the ECL-ML Library and those implementations.

Methods

Enterprise Control Language (ECL)

The HPCC Systems Platform also includes the Enterprise Control Language (ECL) [1, 3] which is a powerful high-level, heavily-optimized, data-centric declarative language used for parallel data processing. The ECL code will run, without any change, on different HPCC System Platform configurations regardless of the size of the cluster being used. ECL includes extensive capabilities for data definition, filtering, data management, and data transformation, and provides an extensive set of built-in functions to operate on records in datasets which can include user-defined transformation functions [29].

Functionally, there are two types of ECL code: *attribute* definitions and executable *actions*. Most ECL code is composed of *attribute* definitions and follows a dictionary approach: each ECL definition defines an *attribute* expression and any previously defined *attribute* can then be used in succeeding ECL definitions. *Attribute* definitions only define what is to be done and cannot have side effects; they do not actually execute. The order that *attributes* are defined in the source code does not define their execution order. When an *action* is executed, only the *attributes* needed (drilling down to the lowest level *attributes* upon which others are built) are compiled and optimized [3]. Even though there is no inherent execution order implicit in the order that *attribute* definitions appear in the source code, there is a necessary order for compilation to occur without error. ECL does not have any support for forward references (cannot call undefined *attributes*).

In ECL, almost all data is represented through datasets, which are collections of records of the same previously defined type. This way, the datasets could be automatically stored in/fetched from memory or written to/read from disk over all the slave-nodes in the cluster as required (see Appendix—data structures used in DT learning process).

Supervised learning in HPCC platform

Supervised learning is the machine learning task of inferring a function from labeled training data [30]; it can be divided into two subcategories: classification and regression. The main difference between them comes from the dependent variable (label) to be inferred. Classification works with discrete dependent variables, while regression works with continuous dependent variables. The HPCC Systems Platform provides support for classification and regression tasks through the ECL-ML Library.

In the ECL-ML Library, the classification process is controlled through a combination of common and particular modules/functions implemented in the *Classify* module (ML/Classify.ecl):

- Default module. This *VIRTUAL* module works similar to an abstract class in Java: it outlines the default behavior that modules based on it (specific Classifier modules) will inherit. It declares the default functions *LearnD*, *ClassifyD*, *LearnC*, and *ClassifyC*. These functions should be overridden later for convenience at every specific classification algorithm implementation.
- Compare module. This module wraps many functions that compare one testing dataset's dependent variable (actual class) against the classification results (predicted class) obtained from the same dataset's independent variables. This comparison returns classification performance metrics calculated from the confusion matrix such as Recall, Precision, False Positive Rate and Accuracy.
- Classifier modules: These are the specific classifier implementations where default functions should be overridden by functions based upon the corresponding algorithm. In addition, it is the only place where algorithm-based parameters, necessary for some algorithms, could be passed inside the implementation and used without conflicting the default functions declaration.

The ECL-ML's classification process norm is based upon function and data structure declarations. The Default module regulates the classification function declarations. However, common data types shared by different ML processes, used in classification as well, are declared in the *Types* module (ML/Types.ecl).

Regardless of the algorithm used, the classifier model building is done through the *LearnD* or *LearnC* function, for discrete or continuous independent variables, respectively. The learning function receives the training examples through two unique dataset inputs: independent and dependent (see Appendix—discreteField and numericField data structure). In addition, any algorithm parameter needed is passed at classifier instantiation. Even though different classifiers have different model structures, the resulting model is converted into a common mapped numeric-matrix representation before being returned.

In a similar way, the class prediction is done through *ClassifyD* or *ClassifyC*, based upon the model used to classify. The classifying function needs a model and an independent dataset to perform the classification. It returns one predicted class and a class probability distribution score for each instance input.

Implementing recursive based Algorithms through iterations

It is worth pointing out that we developed the first RF classifier implementation in the ECL-ML Library and before our contribution, there was no working RF implementation on the HPCC Platform. We have already implemented both discrete and continuous RF classifiers in the ECL-ML Library; however, the RF's learning process optimization discussed in this paper has been completely applied only to the discrete version of the classifier. The continuous version of the RF classifier also takes advantage of this learning process optimization, but it requires additional changes to overcome certain unique issues. The optimization of the continuous version of RF is currently in progress. The RF classifier is part of the *Classify* module in the ECL-ML Library and was built under the umbrella of a Default module that defines *Learn*, *Classify*, and *Compare* main functions in order to deal with the supervised learning and classification process.

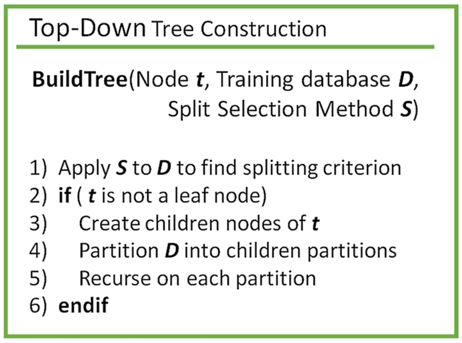
In the RF's learning process, each tree in the ensemble is built based on the principle of recursive partitioning. Although recursion per se is not allowed in ECL, we were able to implement the recursive partition algorithm by transforming recursion into iteration. Basically, our implementation combines the use of an iterative ECL built-in function called LOOP and ad hoc data structures that handle the split/partition process, similar to the Decision Tree's learning process that will be explained in "Recursive partitioning" section.

We can briefly recapitulate the evolution of the RF classifier implementation in the ECL-ML Library as follows. We implemented the first Decision Tree (DT) classifier in the ECL-ML Library based upon an Iterative split/partition approach previously used in a module called *Decision* (ML/Trees.ecl). The *Decision* module is capable of partitioning the training data given in a tree structure, but it cannot build a model suitable for classification purposes. Thus, the *Split* function inside *Decision* module was cloned and improved with the purpose of creating new *Split* functions used to build Decision Tree models. Later, we cloned these new functions and upgraded them to work as *Split* functions used in the first RF classifier implementation in the ECL-ML Library. Finally, we identified the flaws in our initial implementation, and we optimized the code to deliver our final RF classifier implementation designed to work with Big Data.

Recursive partitioning

In this section, we are going to describe how the Decision Tree's recursive partitioning [10, 11] was implemented in the ECL-ML Library. We describe the process by which we transform recursion into iteration. The data-centric feature of ECL obliges us to give details about a few data structures in order to clearly explain the process. For simplicity, we are going to explain only the Gini Impurity/Info Gain Based implementation of the DT Discrete Learning process (*LearnD*); however, we also implemented the C4.5 version. Furthermore, we implemented both discrete and continuous versions of the two approaches.

Breiman et al. recursive partition [10] is based on Hunt's Concept Learning System framework [31], one of the earliest algorithms proposed to build a Decision Tree. Hunt's algorithm forms the mainstay for all DT algorithms, essentially differing only in the choice of the split/partition function. It copes with all necessary elements in Decision Tree building and we will refer to it later in "Decision Tree learning process in ECL-ML" section. Let us take a look at Fig. 1 [32] depicting a high-level description of Hunt's algorithm.



```
Top-Down Tree Construction

BuildTree(Node t, Training database D,
                    Split Selection Method S)

1) Apply S to D to find splitting criterion
2) if ( t is not a leaf node)
3)     Create children nodes of t
4)     Partition D into children partitions
5)     Recurse on each partition
6) endif
```

Fig. 1 Hunt's Concept Learning System framework

We can elaborate from the BuildTree function that the Split Selection Method, ***S***, evaluates based upon the Training database, ***D***, and a splitting criterion if node ***t*** is not a leaf node. The function continuously partitions the database ***D*** into children nodes of ***t***, recursively applying *BuildTree* to each partition until the stop criterion is met. The stop criterion is met when node ***t*** is a leaf node. The DT is composed at the end by the collection of all returned leaf and split nodes (children nodes of ***t***).

Decision Tree learning process in ECL-ML

The Decision Tree classifier follows the learning process norm in the ECL-ML library. It builds one model through the Learn function. *LearnD* receives one training dataset, composed of one independent dataset (*Indep*) and one dependent dataset (*Dep*), as INPUT; and ultimately, it returns the final DT model as OUTPUT. The *Indep* dataset contains the training data's independent variables (attributes) values. Since the number of attributes could be different among training datasets, each instance (training example) is represented by a set of *DiscreteField* records (one for each attribute). The *Dep* dataset contains the dependent variable (class attribute) and it is also represented by a set of *DiscreteField* records; however, in this case, there is only one record per instance.

Comparing the LearnD function to the BuildTree function in Fig. 1, there are still two basic elements missing: node **t** and Split Selection Method **S**. The ML/Classify.DecisionTree module is organized in a way that all DT classifiers are implemented inside it, having a specific classifier-module implemented for each Split Selection Method **S**. From now on, we are going to explain the DT learning process of the GiniImpurityBased learner which uses Gini Impurity/Information Gain as the Split Selection Method, similar to the ID3 algorithm [33]. We also notice that node **t** is not required as input in the LearnD function because LearnD builds the complete DT model from training data.

```
EXPORT GiniImpurityBased(INTEGER1 Depth=10, REAL Purity=1.0):= MODULE(DEFAULT)
  EXPORT LearnD(DATASET(Types.DiscreteField) Indep, DATASET(Types.DiscreteField) Dep) := FUNCTION
    nodes := ML.Trees.SplitsGiniImpurBased(Indep, Dep, Depth, Purity);
    RETURN ML.Trees.ToDiscreteTree(nodes);
  END;
  ...
END
```

ECL Code Block 1 – Gini Impurity based Decision Tree classifier.

Starting from a training dataset composed of Instances, the *GiniImpurityBased* learner builds a DT Model composed of split and leaf nodes. It is controlled by Depth and Purity parameters (control parameters) which are defined upon instantiating the *DecisionTree* module. Depth defines the maximum level that the tree can reach (Decision Tree Growth Stopping Criterion) and Purity establishes the degree of purity required in a node in order to be considered as a Leaf node (Node's Splitting Criterion). Let us take a look at the *DecisionTree.GiniImpurityBased.LearnD* definition shown in ECL Code Block 1.

LearnD gets the DT nodes through the *SplitsGiniImpurBased* function, implemented in ML/Trees.ecl module, from the training dataset using the Depth and Purity parameters. These resultant nodes compound the DT model built, and they are represented by a set of SplitD records. We will explain the model data structure later. At the end, in order to follow the *Classify* VIRTUAL module format, the resultant DT model is transformed into a NumericField record format. This last transformation, applying the *ML.Trees.ToDiscreteTree* function, allows us to generalize all classifier model representations using a common format (learning process norm).

Recursion as iteration Inside the *SplitsGiniImpurBased* function, the training dataset is transformed into a DT model, based upon the Learning Control Parameters, and it is there where the recursive partitioning is implemented iteratively. The key to the process is how to combine all the components involved: iteration, split criterion, data partition, stop criteria, and parameters.

Essentially, instead of applying the Split Selection Method ***S*** individually and recursively at each node (as well as the instances located there), the iterative split/partition process builds a complete *Decision Tree Level* at each iteration (breadth-first approach [34]), as shown at the Pseudocode Block, by applying ***S*** to the entire DT layout (nodes plus instances locations). Then it repeats the process as many times as the desired maximum tree level (Growth Stopping Criterion)—aka Depth. A detailed explanation can be found at Appendix, "Example 1—Tree Level Building". Note that the database ***D*** values do not change at all (data integrity); what changes is the DT's structure and the location of the training instances in the DT. See the data flow in Fig. 2.

```
Learn (Split Selection Method S, Training Data D, Depth, Purity)
DT Layout = Assign D instances to Root Node
For i = 1 to Maximum Tree Level (Depth):
    Apply S to DT Layout's Nodes (level = i )
        Turn Pure Nodes (Purity reached) into Leaf Nodes
        Split Non-Pure Enough Nodes according S criterion
            Create Split Nodes pointing to New Nodes (level = i +1)
            Assign Training Instances into New Nodes created
Return DT Layout as Dec Tree Model
```
Pseudocode Block 1 – Dec. Tree Learning as Iterative Split/Partition process

From the Pseudocode Block 1 we can highlight the need for:

- A data structure that can handle the location of training examples (instances) into the tree's nodes.
- A split/partition function that receives training data located at the tree's nodes and returns the data located in a new node layout.
- A way to call the split/partition function iteratively according to the control parameters (Max Tree Level, Purity).
- A Decision Tree's Growth Stopping Criterion.

The natural way to handle iteration in ECL is through the built-in LOOP function. Mainly, these requirements match the core definition of the LOOP function. There are many ways to call LOOP in ECL. Let's review the documentation for the one used in our earliest *DecisionTree* implementation:

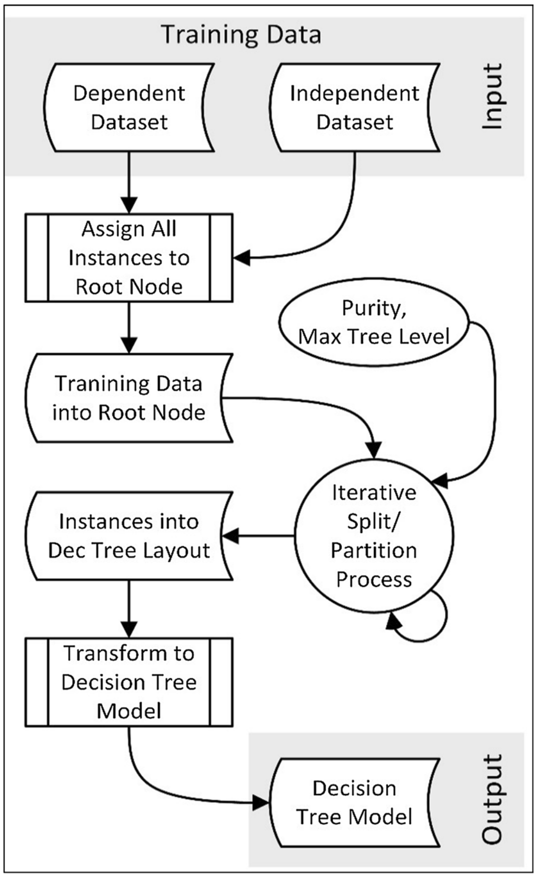

Training Data
Input
Dependent Dataset
Independent Dataset
Assign All Instances to Root Node
Purity, Max Tree Level
Tranining Data into Root Node
Iterative Split/ Partition Process
Instances into Dec Tree Layout
Transform to Decision Tree Model
Output
Decision Tree Model

Fig. 2 Decision Tree learning process

LOOP (dataset, loopcount, loopbody)

- *dataset* *The recordset to process.*
- *loopcount* *An integer expression specifying the number of times to iterate.*
- *loopbody* *The operation to iteratively perform. This may be a PROJECT, JOIN, or other such operation. ROWS(LEFT) is always used as the operation's first parameter, indicating the specified dataset is the input parameter.*

The LOOP function iteratively performs the loopbody operation. There is an implicit COUNTER and it is available for use to return the current iteration.

Now, let us take a look at our LOOP call (shown in ECL Code Block 2) inside SplitsGiniImpurBased function and explain how it works in more detail using the default values of the Control Parameters: Depth = 10 and Purity = 1.

```
results := LOOP(root_inst, Depth, PartitionGiniImpurityBased(ROWS(LEFT), COUNTER, Purity));
```

ECL Code Block 2 – Decision Tree LOOP statement

The *LOOP* statement diagram, shown in Fig. 3, represents the data flow corresponding to the ECL Code Block 2 above. All the training instances are initially located at the root node (node_id = 1, level = 1), the dataset that contains these locations is defined as root_inst. The code shown above will iteratively perform the loopbody split/partition function *PartitionGiniImpurityBased* over Node Instance Assign

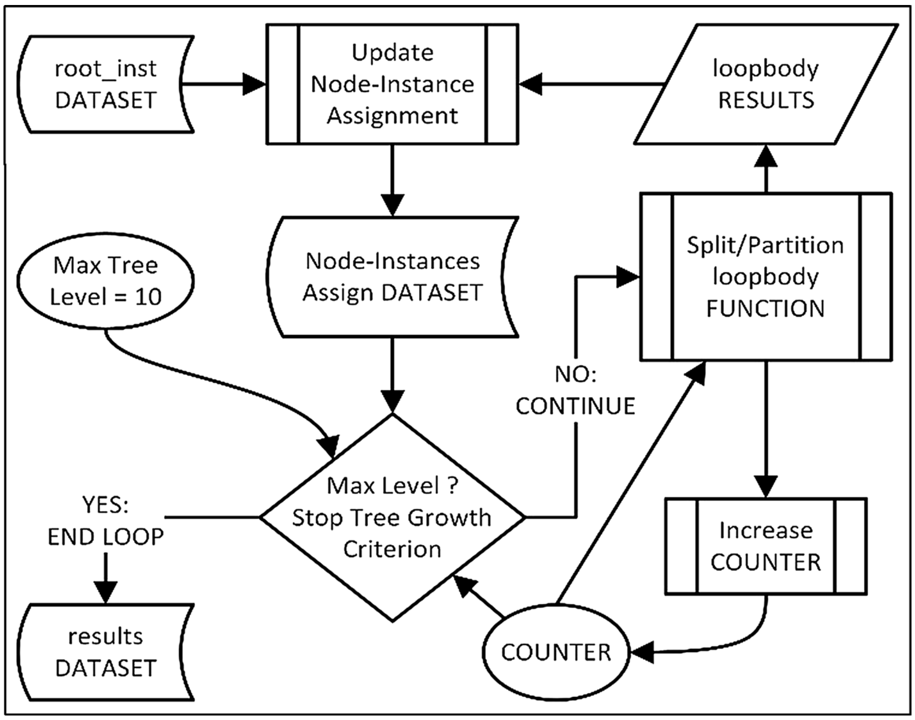

root_inst DATASET
Update Node-Instance Assignment
loopbody RESULTS
Max Tree Level = 10
Node-Instances Assign DATASET
Split/Partition loopbody FUNCTION
NO: CONTINUE
YES: END LOOP
Max Level ? Stop Tree Growth Criterion
Increase COUNTER
results DATASET
COUNTER

Fig. 3 LOOP statement diagram

dataset (initially root_inst) until the maximum tree level is processed (10 times) and will return all instances' new locations throughout the recently built tree's layout. Essentially, this means that the *loopbody* function will process the root_inst dataset only in the first iteration. Then, for the remaining iterations, the *loopbody* function will process the previous iteration *loopbody*'s output. In addition, the implicit *COUNTER* starts at value one and is increased by one after each iteration completion. The *COUNTER* not only controls the CONTINUE/END *LOOP* execution (Stop Tree Growth Criterion). It is also used to inform the *loopbody* function about the level of the DT that must be processed at any current iteration.

Split/partition loopbody function In order to fully understand Decision Tree and Random Forest implementations, as well as their optimizations, we need to explain the main data transformations inside the DT split/partition loopbody function.

First, we need to observe that the loopbody function's INPUT and OUTPUT must have the same data type because one iteration's OUTPUT is the next iteration's INPUT. During the learning process the Decision Tree layout is composed of split and leaf nodes (tree structure) plus the location of the training data in those nodes. In order to handle both the tree node structure and the location of the training instances with the same data type, our loopbody function uses *NodeInstDiscrete* RECORDs with multiple meanings. Basically, *NodeInstDiscrete* combines node information with instance data to articulate instance-location and identify Split and Leaf nodes. See Appendix, "Example 1—Tree Level Building".

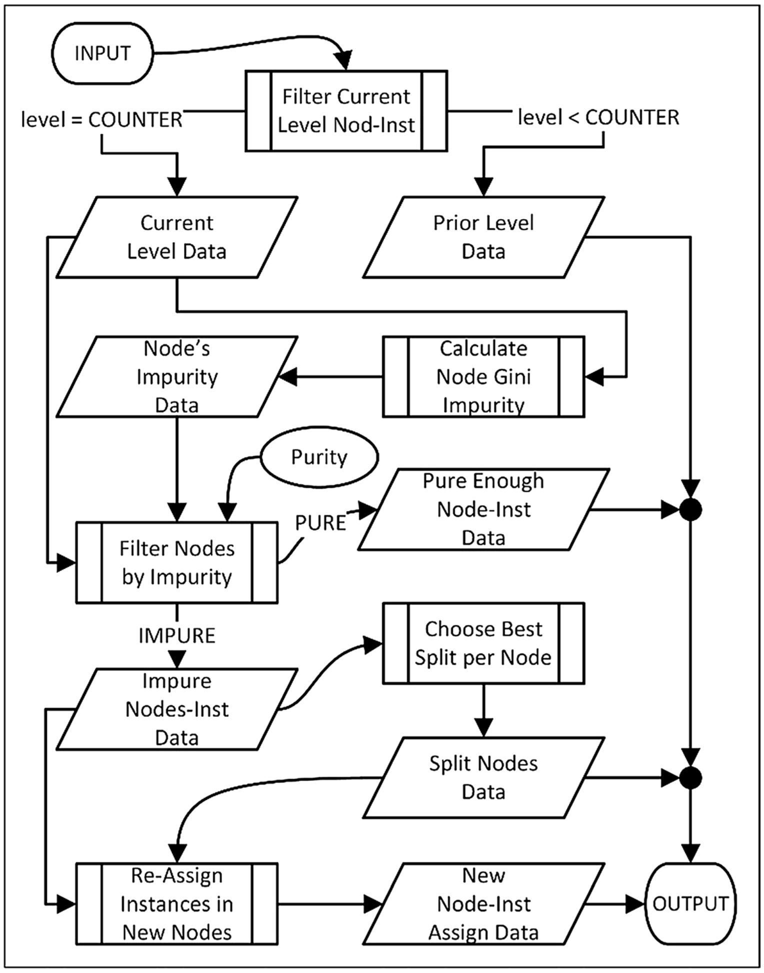


Fig. 4 Decision Tree split/partition function diagram

In Fig. 4, Decision Tree Split/Partition Function diagram, we can observe the data metamorphosis. Starting from the previous location of the instances in a tree structure as INPUT, the function processes only the current level data, identifies pure nodes, chooses the best splits for impure nodes to create branches to higher level nodes, and relocates the training instances in the new tree structure, culminating in four separate datasets which are collected and returned via OUTPUT: Prior Level, Pure Enough Node-Instances, Split Nodes and New Node-Instance.

Following the data flow in Fig. 4, the first action is to separate the data that has already been processed (Prior Level) from unprocessed data (Current Level). The function uses the implicit LOOP COUNTER to identify which tree's level is being processed. All INPUT records with level field values less than COUNTER do not need to be processed because they have already reached their final states in a previous iteration; therefore, they are temporarily stored into the Prior Level dataset. On the contrary, records with level field values equal to COUNTER are temporarily stored into the Current Level dataset for further processing. Prior and Current level data are separated and stored without any transformation from the INPUT.

Similar to recursive partitioning, the loopbody function needs to identify the nodes that are Pure from the Current Level data, and somehow proceed to mark them so they are not subsequently processed. In order to do this, the Gini Impurity is calculated in every node and temporarily stored into the Node's Impurity dataset. It is important to note that only the dependent variable (class) of each instance is needed for the calculation. Since every training instance is represented by one or more *NodeInstDiscrete* RECORDs (one for each independent variable) and the dependent variable was already copied into all of them, the Current Level data needs to be filtered out to only one record per instance (usually filtering the first attribute records) before the Node Gini Impurity calculation. Afterward, the Current Level data is separated into two disjoint datasets (without any transformation) using the Node Impurity results and the Purity parameter. All records belonging to Pure Nodes (Leaf) are temporarily stored into Pure Enough Node-Inst dataset, otherwise, they are stored into the Impure Node-Inst dataset for further processing.

Subsequently, Impure Node's instances must be relocated into new child nodes created based upon best split selection. Only Impure Node data is used to choose the Best Split per Node and generate branches. Branches are links between parent nodes and their children. Two or more Branches form a Split node (one for each possible value of the selected split attribute). They share the same parent node but have distinct child destinations—always new nodes of immediate higher level (child node level = parent node level + 1). The Branches data is temporarily stored into the Split Nodes dataset. After that, all instances located at Impure Nodes are relocated into the recently created New Child Nodes (Split Nodes) based upon their attribute values, and they are temporarily stored into the New Node-Inst dataset. Finally, the four temporary results, Prior Level, Pure Enough Node-Inst, Split Nodes, and New Node-Inst, are appended to form the new tree layout, plus instances locations, and sent to the OUTPUT.

Recap To summarize, the Decision Tree's learning process, based on the recursive partitioning algorithm, is implemented in ECL by splitting nodes and relocating instances' data iteratively using an ECL built-in *LOOP* function. Starting with all training instances located in the root node, the split/partition function is applied repeatedly until the Stop Criterion is achieved, where all the instances reached a Leaf node or the DT grew up to the maximum level desired. Both the recursive partitioning algorithm and the iterative split/partition process deliver the same DT model given the same training data and learning parameters; however, the methods of operations of their split/partition functions are different. The iterative approach uses the split/partition function to process a whole DT level (breadth-first approach [34]), instead of using the function to independently process each node as in the recursive approach.

Random Forest learning process in ECL-ML

The RF algorithm [6] builds one DT model for each sample with repetition from the original training data, where the DT's nodes splitting is based only upon a randomly selected subset of the independent variables (Random Feature Selection).

In the same way as with the DT, the RF classifier follows the learning process norm in ECL-ML: it builds a model from one training dataset using the *LearnD* or *LearnC*

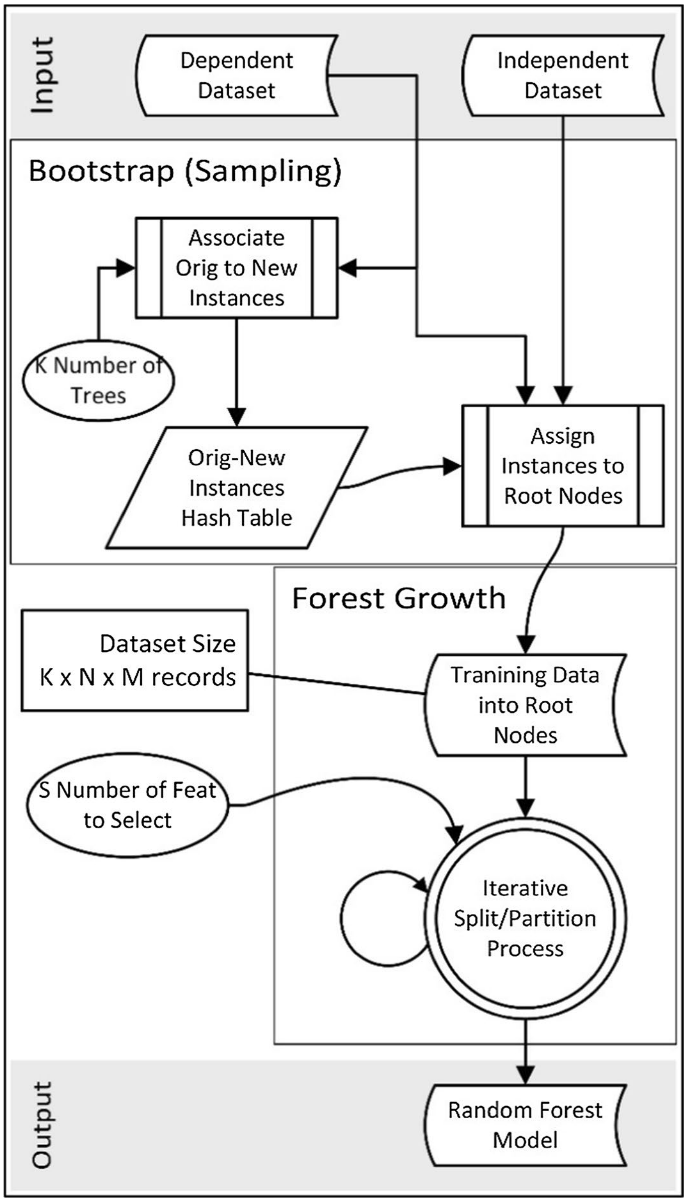

Input
Dependent Dataset
Independent Dataset
Bootstrap (Sampling)
Associate Orig to New Instances
K Number of Trees
Orig-New Instances Hash Table
Assign Instances to Root Nodes
Forest Growth
Dataset Size K x N x M records
Tranining Data into Root Nodes
S Number of Feat to Select
Iterative Split/Partition Process
Output
Random Forest Model

Fig. 5 Random Forest initial implementation

function and it returns the final RF model as output. Moreover, the final RF model is an ensemble of DTs, and thus the RF learning process is also based upon the iterative split/partition process (as mentioned in "Recursion as iteration" section) with a couple of modifications in order to handle both Bagging and Random Feature Selection (as shown in Fig. 5).

Initial implementation

The code that implements the ensemble DTs growth in ECL was cloned from the DT learning process and upgraded to fulfill particular RF learning requirements:

Bootstrapping and Random Feature Selection. In order to identify which tree of the ensemble an elemebuild a RF discrete model betweennt belongs to, a new field called *group_id* was added to all the key data structures used in the RF learning process.

Bootstrap (sampling) In general, sampling is not an issue in ECL. The simplest way to implement sampling is to create a hash table (association list) between the new instance IDs of the sampled dataset and the corresponding instance IDs randomly selected from the original dataset; then it is possible to use the hash table to retrieve the dataset values whenever they are needed.

The bootstrap of the training data within the RF learning process is performed at the beginning of the process just before growing the ensemble of DTs, as shown in Fig. 5. For each tree in the ensemble, one new dataset is generated through sampling with replacement from the original dataset. Each new sampled dataset must have the same size as the original, be properly identified by *group_id*, be assigned to a root node identified by *node_id=group_id*, and finally be stored in the Training Data into Root Nodes dataset. See Pseudocode Block 2. It is important to note that the resulting Roots dataset will have $K \times N \times M$ records, where K is the number of Trees to grow, N the number of instances and M the number of features in the original training dataset.

This way, every single root node with its associated new Sampled Training Data is ready to go through the iterative split/partition process and build its own independent tree. It is very important to note that the forest growth is done in a breadth-first [34] way; i.e., at each iteration, all current-level nodes of the tree ensemble are processed simultaneously to build a whole RF level, similarly to what was previously explained in "Decision Tree learning process in ECL-ML" section.

```
Bootstrap (Dep = dependent Training Data, N = size of Dep,
    Indep = independent Training Data, K = Number of Trees)
For i = 1 to K              // Generate Hash Table (i) from Dep
   For j = 1 to N           // Sampling with repetition
      Hash record = (group_id = i, orig_instance_ID = j,
                  new_instance_ID = M*(i-1) + RndNumber(M))
For each HashTable record as HT    // Training Data to Roots
   For each Indep (id = HT.orig_instance_ID) record as ind
      Append to Roots (node_id = HT.group_id,
          inst_id = HT.new_instance_ID, indep_data = ind,
         dep_data = Dep (id = orig_instance_ID))
```
Pseudocode Block 2- Bootstrap (Sampling)

Random Feature Selection The Random Selection of Features task within the split/partition function is what differentiates the learning process of RF ensemble Trees (see Fig. 6) from regular Decision Trees. Right after any node is considered not pure enough to become a Leaf node (IMPURE), only a random subset of features is selected to accomplish the best split analysis.

The Random Feature Selection (RFS) task is implemented within the *RndFeatSelPartitionGIBased* loopbody function. Initially, we were focused on overcoming the overhead that Bagging added to the process and we were willing to accept the tradeoff between accuracy and simplicity. Therefore, instead of performing RFS for every single node at

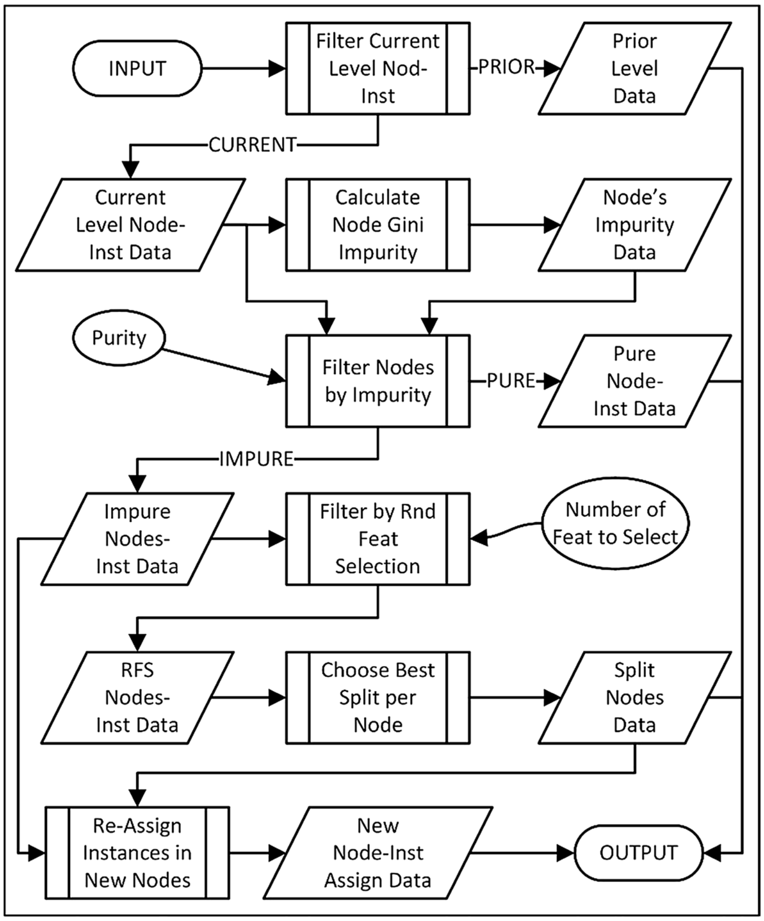


Fig. 6 Random Forest split/partition function diagram

each iteration as the RF algorithm stated, only one RFS per Tree at each iteration was implemented.

Basically, within the loopbody function, one randomly selected list of features is generated for each tree in the ensemble (controlled through the parameter Number of Features to Select), then all these lists are appended in one hash table. The hash table is used to filter the Impure Node-Instances dataset, selecting only the Node-Instance feature values listed per tree. The resulting filtered records are stored in the RndFeat Select Node-Instance temporary dataset, ready to continue with the Choose Best Split per Node task.

Completing the cycle After RFS, the function chooses the best split attribute for each node. The selection is accomplished in the same way as in the DT learning: it chooses the best attribute to split a node based on Information Gain and it creates the Branches that represent the Split. In addition, in order to complete the split/partition process, the instances located at Impure Nodes (Impure Node-Inst dataset) need to be relocated into the new nodes to which the recently created branches are pointing. Finally, the four temporary results, Prior Level, Pure Enough Node-Inst, Split Nodes, and New

Node-Inst, are appended to form the new tree layout (including the new locations of instances) and sent to the OUTPUT.

Notice that all Node-Instances datasets used in the split/partition function, except the RndFeat Sel Nodes-Inst, contain one record for each attribute value per instance.

Final notes Our initial implementation was able to build a Random Forest model from a training dataset; however, it was far from being efficient. The rise in the learning processing time, due to increasing the training data cardinality-dimensionality and/or the number of trees to build the model, limited the usefulness of our initial RF implementation. It was beyond the scope of our research to find out the escalation break point since the cluster characteristics (number of slaves, HW configuration) were part of the equation. It was more important to find out the flaws in our initial approach and fix them.

Random Forest learning process optimization

In order to optimize our RF module, it was imperative to re-examine our initial implementation and identify any flaws throughout the process, data flow, or programming [35]. The first issue that drew our attention within the implemented RF Learning Process, was the fact that for each iteration of the iterative split/partition *LOOP*, which builds the RF model, every single Node-Instance RECORD is sent into the loopbody function to be processed, regardless of whether its processing was completed. Sending records that do not require additional processing to the next iteration of the learning process causes the algorithm to take longer to finalize, due to the unnecessary use of large amounts of computational resources (memory, disk and CPU cycles).

The second issue that drew our attention was the fact that the whole data per Instance, consisting of a dependent value and many independent values, are sent to the loopbody function for each iteration, even though not all of the independent values per instance are required to complete the loopbody internal tasks that build a DT level (see Fig. 6 to identify the internal tasks).

Node Impurities are calculated based upon Instance dependent values only, which need only one RECORD per instance to be represented. Choosing the Best Split per Node based upon Information Gain (IG) needs both instances' dependent and independent values; however, only a randomly selected subset of the independent data is used to calculate IG per Node, which can be represented with just some RECORDs per instance (but not all of them at any particular iteration). Finally, in order to arrange the new partition of the data, just one RECORD per instance, containing node info and basic instance info (ID and dependent value), would be enough to indicate the location of the instances in the new nodes. Therefore, to include the Independent data as part of the loopbody function, INPUT is a waste of resources.

The revision of the initial implementation helped us to re-organize the process and data flows. We have improved our initial approach so that the loopbody function receives, at each iteration, only one RECORD per instance (with its dependent value). Thereby, only the necessary independent data are fetched, using the IDs of the received instances, from within the function. See Fig. 7. The new approach also allowed us to

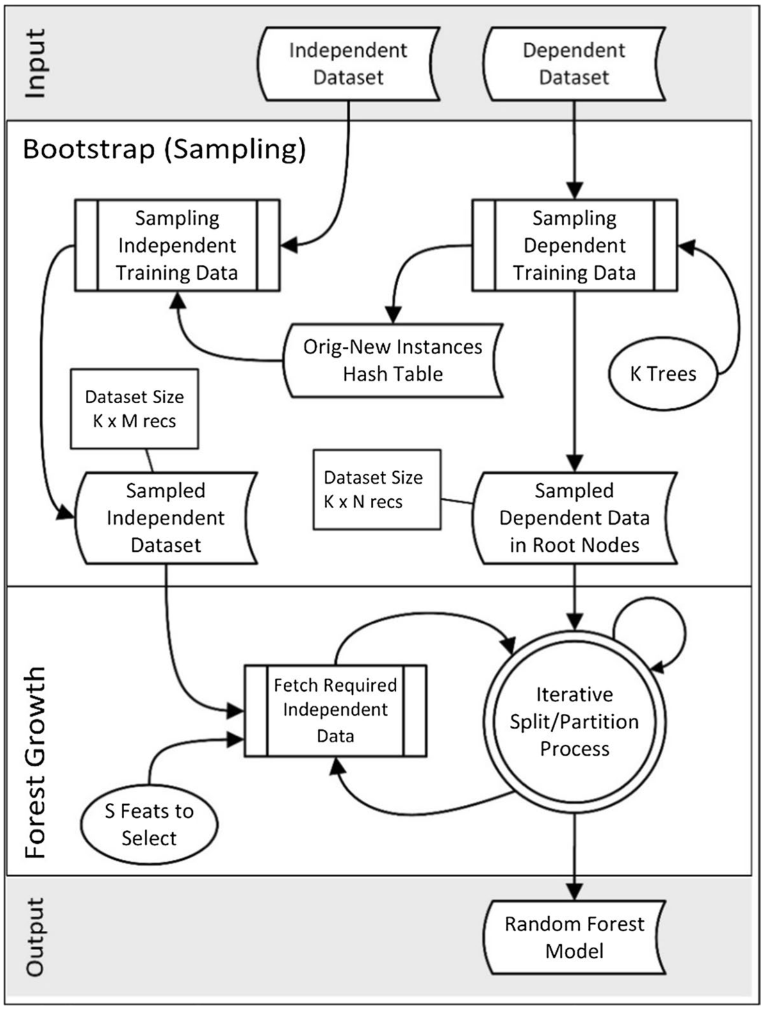

Input
Independent Dataset
Dependent Dataset
Bootstrap (Sampling)
Sampling Independent Training Data
Sampling Dependent Training Data
Orig-New Instances Hash Table
K Trees
Dataset Size K x M recs
Dataset Size K x N recs
Sampled Independent Dataset
Sampled Dependent Data in Root Nodes
Forest Growth
Fetch Required Independent Data
Iterative Split/Partition Process
S Feats to Select
Output
Random Forest Model

Fig. 7 Random Forest optimized implementation

take full advantage of distributed data storage and parallel processing capabilities of the HPCC Platform, speeding up the RF learning process. In the following sections, we will explain our approach enhancements in detail.

Current level filtering

Remember that at any iteration if a node is pure enough—Leaf node—then it is not split, and its instances are not relocated; otherwise, the node is split into new nodes (children nodes) and its instance data is partitioned among them (New Nodes-Instances Relocation). The loopbody function should only process the current level Node-Instances data (identified by the level parameter). Therefore in our initial RF implementation, for each iteration all records from previous levels needed to be filtered out, saved in a temporary dataset and returned without any change at the end of the function. Moreover, the number of RECORDs that still require processing decreases as the LOOP progresses, because many of them have already reached their final state (Leaf nodes). Consequently, too many pointless read/write operations were performed during the iterative process.

This issue was fixed by changing the way the built-in LOOP function was used. Fortunately, there is a way to call LOOP that filters the records passed to the loopbody function:

```
LOOP ( dataset, loopfilter, loopcondition, loopbody )
-   dataset         The recordset to process.
-   loopfilter      A logical expression that specifies the set of records whose processing is not yet
    complete. The set of records not meeting the condition are no longer iteratively processed and are
    placed into the final result set. This evaluation occurs before each iteration of the loopbody
-   loopcondition   A logical expression specifying continuing loopbody iteration while TRUE
-   loopbody        The operation to iteratively perform. …
```

The syntax is slightly different, although the way it operates is well suited for our improvement purpose. The loopfilter logical expression will let pass only the current level records to the split/partition loopbody function. In addition, the *loopcondition* logical expression keeps the LOOP going until the maximum tree level was processed. The new LOOP call statement inside *SplitsGiniImpurBased* function looks like:

```
res := LOOP (root_inst, LEFT.level=COUNTER, COUNTER < Depth , RndFeatSelPartitionGIBased(ROWS(LEFT), COUNTER));
```

ECL Code Block 3 – Optimized Random Forest LOOP statement

Note that the optimized loopbody function (explained in the next section) will process all records received. The RECORDs level filtering is now the responsibility of the LOOP statement (shown in ECL Code Block 3), yet, it is still necessary to pass the level value in order to properly generate new Nodes.

Reducing R/W operations within iterative split/partition process

The second issue required an innovative approach in order to avoid unnecessary read/write (R/W) operations throughout the Iterative split/partition process. The intention of the new approach is to complete the RF learning process faster, by considerably reducing the amount of Node-Instance RECORDs supplied into the split/partition loopbody function.

The first and last tasks within the *RndFeatSelPartitionGIBased* loopbody function, the Node Impurity Calculation and the Relocating Instances in New Nodes sub-processes, could be accomplished using just one RECORD per instance. The former is based upon Node-Instance dependent values only. It would be sufficient to use just one RECORD containing the dependent value per Instance. Likewise, the latter could use just one RECORD to relocate an instance into a node (containing the dependent value as well). In addition, for the subsequent iteration, only the previous iteration's New Node-Instances will be sent to the loopbody function to be processed (as per Current Level Filtering). Therefore, the loopbody function could be simplified to receive and return just one RECORD per instance. This way it would deal with only K (number of trees) times N (number of original instances) RECORDs instead of K × N times M (number of dependent variables) RECORDs per iteration as the initial implementation did.

Conversely, the Best Split per Node analysis is based upon subsets of Node-Instance independent values randomly selected per node, requiring several RECORDs per instance but not all of them (RFS). This cannot be changed. However, the RndFeat Select Node-Instance dataset (Choose Best Split per Node's INPUT) could be manufactured using a more efficient method. Rather than filtering the required dependent variables, it would be better to fetch them. The former method extracts the RFS chosen features from all the independent values per instance, implying pass them all through the loop-body function (K × N × M records). The latter method fetches (from an external source) only the instances' dependent values selected by RFS. It needs just one RECORD per instance passing through the loopbody function (K × N records) and the complete independent variables dataset outside the function to fetch from. Therefore, the Bootstrap and the RFS processes needed to be restructured in order to fit this new approach with reduced R/W operations.

Parallel processing through data distribution and locally performed operations One way to make the most of the HPCC platform is by distributing the data evenly among the Cluster Slave Nodes (CLUSTER-NODE), in a manner that later computations based on this data could be completed independently on each CLUSTER-NODE (LOCAL-ly) or with minimal interaction between them. Most of the data transformations used in our implementations are performed through some of the built-in functions available in ECL: *JOIN, PROJECT, TABLE, COUNT, LOOP, DEDUP*, etc. These functions process the data distributed among the HPCC CLUSTER-NODEs according to ECL code and the ECL compiler's automatic optimization. However, the automatic optimizations do not always get the best performance results; a deep understanding of how the data flows inside a process will help to tune the relevant ECL code and turn it into a more efficient process.

Along the RF learning process, more precisely at each iteration of the split/partition loopbody function, most of the calculations, mainly aggregations, performed are based upon Node-Instance data. Moreover, these calculations are completely independent from one node to another, regardless which tree of the ensemble the node belongs to. We optimized the loopbody function code to DISTRIBUTE the Node-Instance data by *{group_id, node_id}*, meaning that all the Node-Instance RECORDs with the same *node_id* value will be stored in the same CLUSTER-NODE and the process could execute calculations and store their results on each CLUSTER-NODE independently, thus reducing runtime.

There is another high resource consumption task within the loopbody function, dealing with the Training Independent data. As we mentioned earlier, we changed our approach from filtering to fetching. The new approach implementation is supported by the Bootstrap Optimization task, where the entire Sampled Training Independent (K × N × M records) dataset is generated and DISTRIBUTED by *instance ID* only once. Thus, the sampled data is ready to be used as many times as needed for independent data retrieving, and these fetch operations are performed on each HPCC CLUSTER-NODE independently, reducing runtime as well. More detailed explanations are discussed in further sections.

Bootstrap Optimization The optimized Bootstrap process operates as shown in Pseudocode Block 3. Since only dependent values will pass through iterative split/partition's iterations, the independent values do not need to be present in the Training Data-Root Nodes dataset (see Fig. 7). The resulting Sampled Training Independent dataset contains only one record per new instance, having K × N records regardless of the number of features in the original Training dataset. At the same time, the Sampled Training Independent dataset is generated and DISTRIBUTED, using the new *instance ID* as KEY, all over the cluster in order to speed up its retrieval. The retrieval task will be explained later in the *Fetching instances' independent data at RFS* section. Notice that the *group_id* field is no longer necessary since the retrieval of independent data (FETCH) is now accomplished using only the new *instance ID*s and the feature numbers.

```
Bootstrap (Dep = dependent Training Data,
N = size of Dep, K = Number of Trees)
For i = 1 to K          // Generate Hash Table (i) from Dep
  For j = 1 to N        // Sampling with repetition
    Hash record = (group_id = i, orig_instance_ID = j,
          new_instance_ID = M*(i-1) + RndNumber(M))
For each HashTable record as HT
  Append to Roots (node_id = HT.group_id,
                   inst_id = HT.new_instance_ID,
                   dep_data = Dep (id = orig_instance_ID))
  Generate Sampled Training Independent dataset // ST-Indep
    For each Indep (id = HT.orig_instance_ID) record as ind
      ST-Indep record = (inst_id = HT.new_instance_ID,
                         indep_data = ind)
Distribute ST-Indep dataset over Cluster using inst_id as KEY
```
Pseudocode Block 3- Bootstrap Optimization

RF split/partition loopbody function optimizations As a result of implementing the Bootstrap Optimization and moving the Current Level Filtering outside the RF split/partition loopbody function, the function's RFS task was optimized to Fetch Instances' independent data, instead of filtering them, in order to allow the loopbody function to complete its task by receiving and delivering a Node-Instance dataset containing only one Node-Instance RECORD per instance.

No filtering current level data: Filtering the Current Level Node-Instances is no longer necessary within the loopbody function as it was in the previous version (compare Figs. 6, 8). The optimized function now assumes that all records received belong to the Current Level and need to be processed. In addition, the Node-Instance INPUT dataset contains only one RECORD per instance and these records contain the dependent values. Thus, each node's Gini Impurity calculations are performed directly and simultaneously using the input dataset just as it is, without filtering out the data to a record per instance as with the previous version.

Furthermore, the Node's Impurity data is used to separate[1] (without any transformation) the INPUT dataset into Pure and Impure Node-Instances datasets. Pure Enough Nodes data is sent to the OUTPUT (just one record per instance) and Impure Nodes data is sent to the RFS task to continue the split process.

Fetching instances' independent data at RFS: The final version of the optimized loopbody function implements RFS at the node level properly as per the RF algorithm [6].

[1] These operations were done *LOCAL*-ly as per "Parallel processing through data distribution and locally performed operations" section.

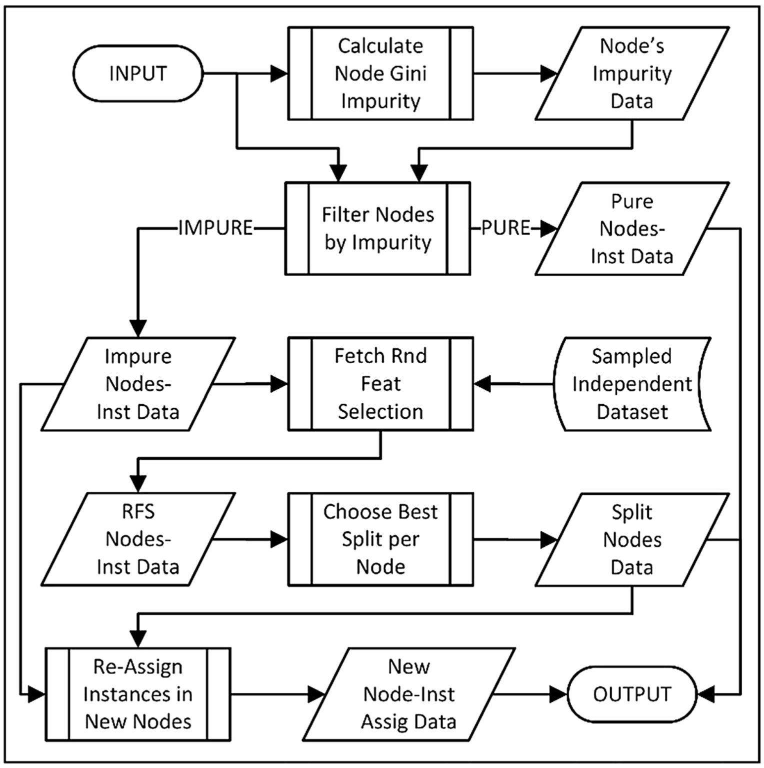


Fig. 8 Optimized Random Forest split/partition loopbody function

Randomly selected feature data subsets from Impure Nodes' instances are gathered in a faster and more efficient approach by fetching the RFS instances' independent values, i.e., a hash table of Selected Features per Node (SFN) is randomly generated at each iteration, then the data required to choose the Best Split per Node is fetched from the Sampled Training Independent dataset created at the Bootstrap phase according to the SFN hash table.

The Pseudocode Block 4 shows how the RFS-Fetching implementation works. It takes full advantage of the data-centric, distributive and parallel HPCC Platform capabilities to speed up the data retrieval:

- The Sampled Training Independent dataset generated at Bootstrap phase is evenly DISTRIBUTED all over the CLUSTER NODES by instance ID. It is used as the SOURCE of the independent values retrieval at every iteration.
- The Node-Inst-Feat dataset is generated by JOINing the Impure Nodes and SFN Hash Table datasets. The JOIN operation is performed on each CLUSTER NODE independently because both datasets are DISTRIBUTED by *{group_id, node_id}*.
- The resultant Node-Instance-Feature dataset is DISTRIBUTED using ID as KEY in order to generate the RETRIEVER dataset for the independent values retrieval.
- Both SOURCE and RETRIEVER datasets were DISTRIBUTED by *instance ID*, and hence, the independent values can be quickly retrieved using a JOIN-LOCAL operation executed on each CLUSTER NODE independently.

Initially, the RFS in the optimized function version was implemented at tree level like the initial version. However, further tests found some issues with the resultant trees. The generalization of the model was somehow affected negatively due to the fact that at each iteration, during the construction of the model, nodes from the same tree chose their best splits from the same feature subsets.

```
Random Feature Selection – Fetch Independent data (Dec.Tree Layout,
K = Number of Features to select, M = Number of independent variables)
Source = Sampled Training Independent data (distributed by instance_id)
For each Dec.Tree Layout Impure node as Node          // NodexKoutofM
  // Generate Selected Features per Node (SFN) Hash Table
  Feats_selected = Sampling K numbers from {1 to M} without repetition
  For i = 1 to K
    SFN record = (group_id = Node.group_id, node_id = Node.node_id,
                  number = Feats_selected(i) )
  Node-Inst-Feat record = JOIN (Node.instances, SFN) by inst_id, number
Retriever = Distribute Node-Instance-Features using instance_ID as KEY
RFSData = JOIN (Source, Retriever) by instance_id
```

Pseudocode Block 4- RFS Fetching Independent data

Completing the optimized cycle: With the intention of accelerating the tasks of Choose the Best Split per Node and generating the Split Nodes dataset, the toSplit dataset is DISTRIBUTED by *{group_id, node_id}* and all data transformations are performed on each CLUSTER NODE independently according to "Parallel processing through data distribution and locally performed operations" section. In addition, in order to complete the split/partition iteration, the loopbody function relocates all Impure Nodes into the new nodes to which the *Split Nodes* are pointing. However, keeping in mind our attempt to reduce read/write operations within iterative split/partition process initial explanation, the relocation in the optimized version is completed in a more efficient way. It uses only one RECORD to relocate an instance into a node (New Node-Inst dataset); moreover, this operation is achieved quickly in just one step by combining the toSplit dataset with the Split Nodes dataset using the locally performed operation *JOIN-LOOKUP*. Finally, the three temporary results, Pure Enough Node-Inst, Split Nodes and New Node-Inst, are appended to form the new Forest Layout dataset and sent to the OUTPUT.

Results

The optimization of the Random Forest implementation in ECL is the outcome of many consecutive upgrades and testing cycles. Some of them come from other module's improvements, such as Decision Trees and sampling, and others are specifically designed to overcome RF issues, such as Fetching the Independent data from a Sampled Training Data located outside the loopbody function (location related to the program scope). Furthermore, the implementation of this new approach (Fetching) involved a deep restructuration of the RF Learning Process; thus, a Preliminary Evaluation (PE) using a not completely optimized version was performed to confirm that Fetching could reduce the learning process runtime. After verifying our Preliminary Evaluation's

PE's positive results, the RF learning process was comprehensively optimized and its improvement evaluated again in the Final Assessment.

Preliminary evaluation

It is important to note that by this time the two implementations to be compared already filtered the Node-Instance records that did not require further processing at the *LOOP* level and also that their RFS were implemented at tree level (not at node level as the original algorithm states, which is discussed in "Random Feature Selection" section). The main objective of the evaluation was to confirm that keeping the Sampled Training Independent data outside the split/partition loopbody function could reduce RF learning runtime.

Preliminary experimental design

We compare the time to build a RF discrete model between the Fetching on Demand Independent data approach (NEW) and the Filtering Independent data approach (OLD) using a discretized version of the UCI Machine Learning Adult test data set [36] (16,281 instances × 13 features + class) where the missing values where replaced by zero and the "*fnlwgt*" attribute was removed. Note that we did not use the models to classify because the classification performance was not the topic of interest in this evaluation.

Using an HPCC Cluster with 50 nodes we build several RF models, keeping the Number of Features Selected constant to 6 (approx. half the total) and varying the Number of Trees and Depth learning parameters. The experiment cases consider four different Number of Trees, NoT = {25, 50, 75, 100} and six levels of Depth, D = {10, 25, 50, 75, 100, 125}. For both NEW and OLD approaches, a total of 10 runs for each case was performed and the runtime measure.

Preliminary evaluation results

The graphs below (Figs. 9, 10) depict the results of the NEW approach, on the left, and the OLD approach, on the right, by averaging the runtime of the 10 runs per case.

In Fig. 9, we can observe that for all Number of Trees configurations, identified by different color lines, the Learning Time increases for both approaches, as expected, when the Depth of the tree increases; however, the slope (Learning Time change over numTrees change) seems to be larger for OLD approach lines. In addition, the slope of both approach lines tends to increase with Number of Trees. However, the increments are significantly larger for the OLD approach lines, meaning that the NEW approach performed faster compared to the OLD approach as the Number of Trees increases.

Furthermore, by changing the grouping of the results by Depth, as shown in Fig. 10, we note that for models with Depth > 25, the Learning Time of the NEW approach is less than the OLD one; in addition, the ratio of the changes for both approach lines tend to increase when the Depth increases. However, the increment is larger for the OLD approach.

Final assessment

It is important to note that the final optimization of the Fetching Independent data approach takes full advantage of the distributive and parallel HPCC Platform capabilities

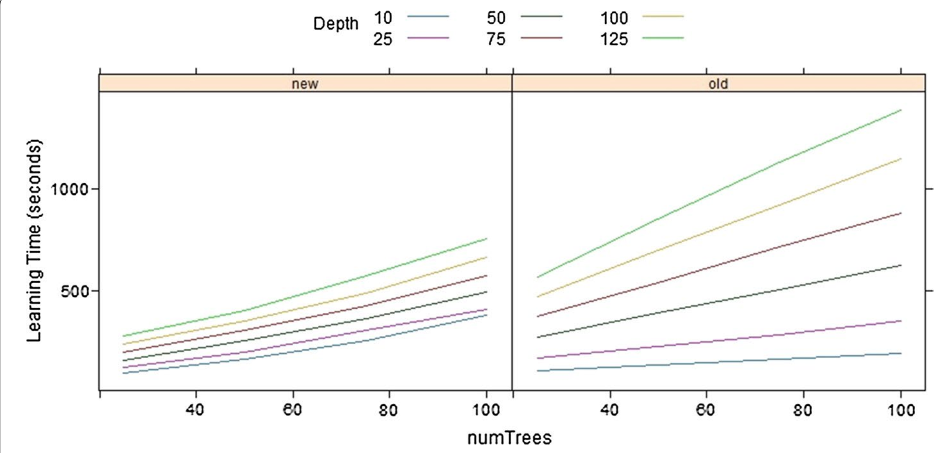


Fig. 9 Preliminary Learning runtime comparison for increasing Number of Trees (numTrees)

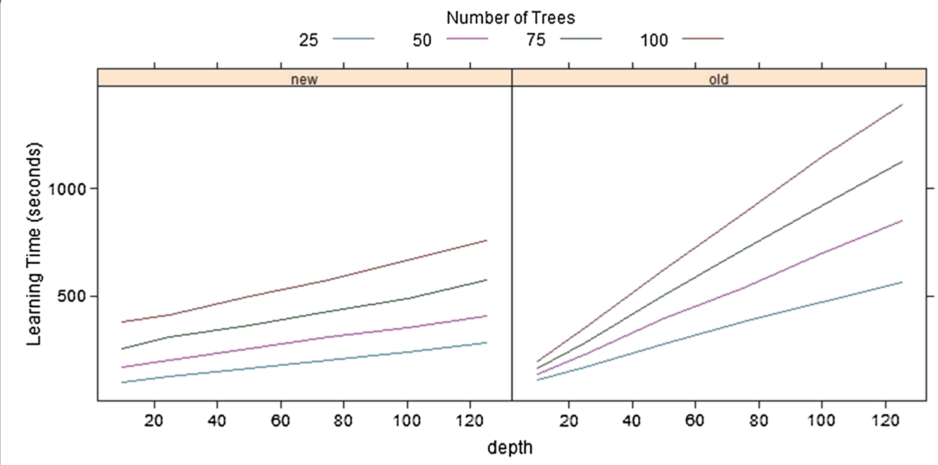


Fig. 10 Preliminary Learning runtime comparison for increasing maximum level of trees (depth)

and implements the Random Feature Selection (RFS) at node level as the algorithm states (as explained in "Reducing R/W operations within Iterative split/partition process" section). The main objective of this final assessment was to evaluate the RF learning runtime with the fully optimized RF implementation.

Final experimental design

We compare the time to build a RF discrete model between the final optimization of the Fetching Independent data approach (NEWFINAL) against the Filtering Independent data approach (OLD) using a bigger dataset. We use subsamples of the UCI Machine Learning Cover Type data set [36, 37] (581,012 instances × 54 features + class), with more attributes and more instances and having 7 different values for the class attribute. It was necessary to preprocess the data, because the first 10 attributes of the dataset had too many values. Thus, we reduced them by BUCKETING to 20 values. It is important to note that we used only a 10% sample size to run the experiments because the OLD

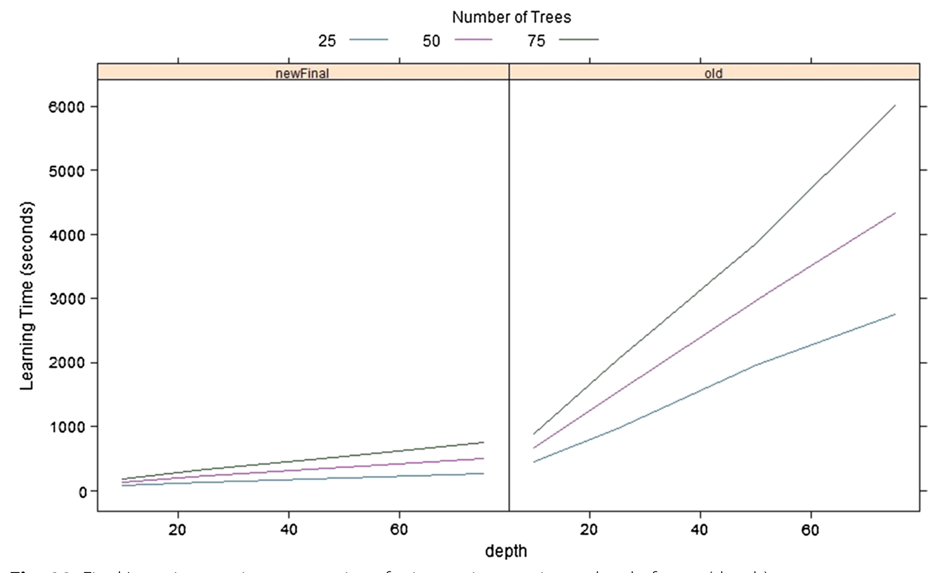


Fig. 11 Final Learning runtime comparison for increasing maximum level of trees (depth)

implementation would take too much time to process the original dataset. Also, we did not use the models to classify because the classification performance was not the topic of interest in this evaluation.

Using an HPCC Cluster with 50 nodes we build several RF models, keeping the Number of Features Selected constant to 7 (approximately the square root of the number of features) and varying the Number of Trees and Depth learning parameters. The experiment cases consider three different Number of Trees, NoT = {25, 50, 75} and four levels of Depth, D = {10, 25, 50, 75}. For both the NEWFINAL and the OLD approach, a total of 7 runs for each experiment case were performed and the runtime measured.

Fetching vs filtering results

The graphs below (Figs. 11, 12) depict the results of the NEWFINAL approach, on the left, and the OLD approach, on the right, by averaging the runtime of the 7 runs per case.

In Fig. 11, we can observe that for all Number of Trees configurations, identified by different color lines, the Learning Time increases for both approaches, as expected, when the Depth of the tree increases; however, the impact of the Depth increase seems to be much larger for the OLD approach.

We can observe in the graph in Fig. 12 the same results grouped by Depth, identified by different color lines, that for all Depth cases the Learning Time of the NEWFINAL approach is smaller than the OLD one; in addition, the slope of both approach lines (Learning Time change over numTrees change) tends to increase with Depth. However, the increments are significantly larger for the OLD approach lines.

Moreover, Tables 1 and 2 present the same speed up ratio results (ratioOldNew) ordered differently. Table 1 results are sorted by numTrees + Depth and Table 2 results are sorted by Depth + numTrees. The ratio value called ratioOldNew is the

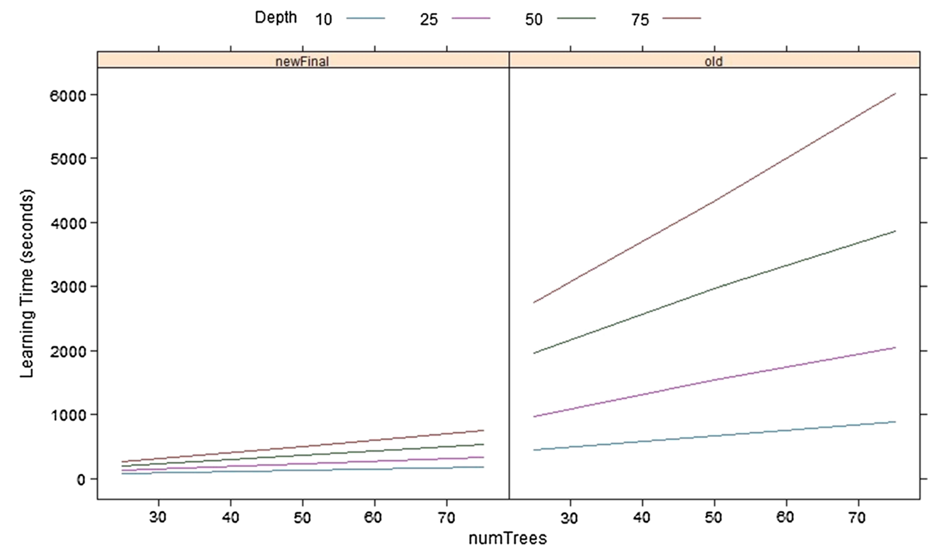


Fig. 12 Final Learning runtime comparison increasing Number of trees (numTrees)

Table 1 SpeedUp ratio sorted by numTrees + depth

numTrees	Depth	ratioOldNew
25	10	5.67519
25	25	7.840796
25	50	9.877268
25	75	10.301324
50	10	5.038769
50	25	6.952996
50	50	8.395452
50	75	8.832542
75	10	4.889199
75	25	6.306244
75	50	7.233024
75	75	8.071576
Average ratio	7.45	

result of dividing the averages of the OLD approach runtimes over the NEWFINAL approach runtimes per experiment case. For all numTrees cases the ratioOldNew increases when the Depth increases; however, the ratio range tends to decrease for models with a larger number of Trees.

Another important thing to note is that the reduction of RF learning runtime in the Final Assessment is much greater than in the Preliminary Evaluation, not just because the optimization improves the performance but also because the proportion of Number of Random Features Selected over Total Number of Features is different. In the Preliminary Evaluation, the proportion RFS/Total was 6/13 and the runtime was reduced around half. In the Final Assessment, the proportion of RFS/Tot was 7/54 and the average ratioOldNew was 7.45, which is close to the inverse of 7/54. The ratio's impact makes sense since, for the OLD implementation, all the independent

Table 2 SpeedUp ratio sorted by depth + numTrees

numTrees	Depth	ratioOldNew
25	10	5.67519
50	10	5.038769
75	10	4.889199
25	25	7.840796
50	25	6.952996
75	25	6.306244
25	50	9.877268
50	50	8.395452
75	50	7.233024
25	75	10.301324
50	75	8.832542
75	75	8.071576
Average ratio	7.45	

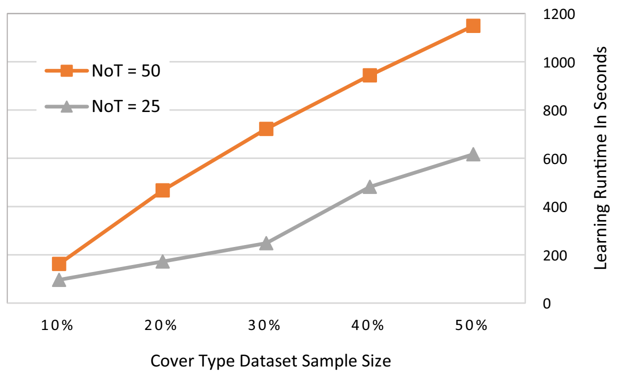


Fig. 13 Scalability results. Learning runtime for increasing training dataset size

values per instance are passed through the split/partition loopbody function, while in the NEW implementation independent values are not passed through and only the required values (Random Features Selected per Node) are fetched from within the loopbody function.

Scalability results

For the Scalability test, we vary the size of the training data as a percentage of the original, we use % Size = {10, 20, 30, 40, 50} and measure the RF Learning runtime for Number of Trees, NoT = {25, 50}.

In Fig. 13, we can observe in the graph that the final optimized version learning runtime increases as the size of the training data increases and that the slope of the line that represents RF learning using 25 trees is slightly less than the one that represents RF learning using 50 trees.

Discussion

The optimization of our RF implementation focused on the speed up of the bottleneck in its learning process, due to an inefficient approach used to gather the data inside the function that iteratively builds the RF model. The initial RF implementation used a *Pass them All and Filter Independent Data* approach. This approach sends the complete Sampled Training Data (independent and dependent datasets) for each iteration through the loopbody function and selects from there the required data for the split/partition calculations. In contrast, the optimized version uses a *Fetch on Demand Independent Data* approach. This enhanced approach sends through the loopbody function only the dependent training data and fetches just the data needed from the independent training dataset located outside the function, thus avoiding many unnecessary read/write operations for each iteration and easing the parallelization of the process.

Based on the results of our Preliminary Evaluation, we confirmed that the Fetching approach helps to reduce the learning process runtime; thus, we proceeded with the complete RF learning restructuration and optimization. Simply put, we added the implementation of Random Feature Selection at the node level and a comprehensive parallelization of the loopbody split/partition function (as explained in "Reducing R/W operations within iterative split/partition process" section).

The results of our Final Assessment demonstrate that there is a significant learning process runtime reduction when using the optimized Random Forest, which implements the *Fetch on Demand Independent Data* approach, compared to the former initial RF implemented with *Pass Them All and Filter Independent Data* approach. The speed up ratio is roughly, on average, the proportion of the Number of Features Selected over Total Number of Features of the Training Data. In addition, the results also indicate that the optimized RF implementation scales well when increasing the number of trees and/or the size of the data.

Conclusions

We successfully implemented the first Random Forest classifier for the LexisNexis HPCC Systems Platform. We were able to implement Random Forest's recursive partition process using the HPCC native programming language called ECL, a data-driven programming language that does not allow recursion per se, by transforming the recursive partition process into an iterative partition process.

Moreover, after many software development cycles, it was possible to identify our initial RF implementation's flaws. Consequently, we replaced its *Pass Them All and Filter Independent Data* approach by a more efficient *Fetch on Demand Independent Data* approach, taking full advantage of the distributive and parallel computing capabilities of the HPCC Platform, to finally deliver our optimized Random Forest implementation as a distributed machine learning algorithm for Big Data.

In the future, the same development methodology described in this paper could be applied to implement and/or optimize other supervised learning algorithms in the ECL-ML Library. First, the proper operation (whether they return the correct results) could be verified and their process runtime evaluated. Second, flaws in their ECL design-implementation could be identified and then corrected through upgrades, taking full advantage of the distributive and parallel capabilities of the HPCC Platform.

Abbreviations
CAKE: Center for Advanced Knowledge Enablement; DT: Decision Tree; ECL: Enterprise Control Language; EE: Energy Expenditure; GPU: Graphics Processing Units; HPCC: high performance computing cluster; IG: Information Gain; ML: Machine Learning; PA: Physical Activity; PE: Preliminary Evaluation; R/W: read/write; RF: Random Forest; RFS: Random Feature Selection; SFN: Selected Features per Node.

Acknowledgements
We are also thankful to LexisNexis Risk Solution for being a loyal member of the Center and for collaborating on this project.

Authors' contributions
VH performed the primary literature review and analysis for this work, and also drafted the manuscript. TMK and FV introduced this topic to VH and provided technical guidance throughout the research and writing phases of the manuscript. TMK, BF, and FV provided a full review of the final manuscript. TMK coordinated with the other authors to complete and finalize this work. All authors read and approved the final manuscript.

Funding
This work has been funded by the NSF Award No. 1464537, Industry/University Cooperative Research Center under NSF 13-542.

Availability of data and materials
The datasets used during the current study are available in the UC Irvine Machine Learning Repository [36].

Competing interests
The authors declare that they have no competing interests.

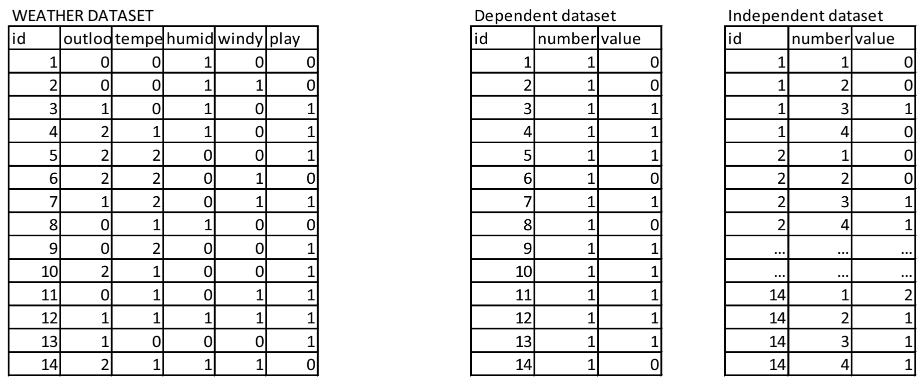
Author details
[1] Department of Computer and Electrical Engineering and Computer Science, Florida Atlantic University, Boca Raton, FL 33431, USA. [2] LexisNexis® Risk Solutions, Alpharetta, GA 30005, USA.

Appendix

This appendix includes descriptions of ECL data structures used in the DT Learning Process and a Tree Level Building example that will help to better understand the material in this paper.

The weather data

The Weather data, shown in Fig. 14, is a tiny dataset for classification created by the University of Waikato's Machine Learning Group [14] and provided as a sample dataset with their WEKA software. The dataset is used in numerous Data Mining related courses and books. It has 14 instances, described by four attributes (related



WEATHER DATASET

id	outloo	tempe	humid	windy	play
1	0	0	1	0	0
2	0	0	1	1	0
3	1	0	1	0	1
4	2	1	1	0	1
5	2	2	0	0	1
6	2	2	0	1	0
7	1	2	0	1	1
8	0	1	1	0	0
9	0	2	0	0	1
10	2	1	0	0	1
11	0	1	0	1	1
12	1	1	1	1	1
13	1	0	0	0	1
14	2	1	1	1	0

Dependent dataset

id	number	value
1	1	0
2	1	0
3	1	1
4	1	1
5	1	1
6	1	0
7	1	1
8	1	0
9	1	1
10	1	1
11	1	1
12	1	1
13	1	1
14	1	0

Independent dataset

id	number	value
1	1	0
1	2	0
1	3	1
1	4	0
2	1	0
2	2	0
2	3	1
2	4	1
…	…	…
…	…	…
14	1	2
14	2	1
14	3	1
14	4	1

Fig. 14 Weather Dataset representation

to weather) and the "class" value—whether or not to play a game. We are going to use the weather dataset to support our explanations and examples. The dataset size and simplicity will help to explicitly show and describe the different data structures and tasks in this appendix. Notice that the data was originally categorical, and we converted it to discrete numerical in order to work with ECL.

Data structures used in DT learning process

DiscreteField and NumericField

These records help to represent any discrete or continuous corresponding dataset in a tabular M *rows by N columns Matrix* format. Any single record is composed of 3 fields:

id: it contains the *instance* identifier number (row). All records with the same id belong to the same training example. ECL-ML Library supports up to eighteen quintillion by default.
number: identifies the *instance*'s variable (column). ECL-ML support up to four billion by default.
value: the value for an instance variable (discrete or continuous feature value).

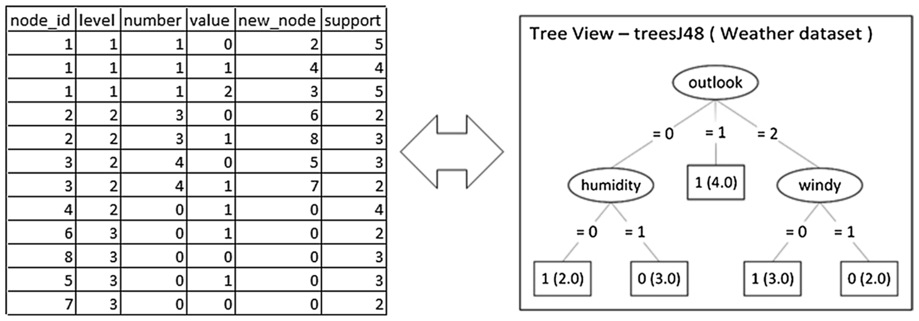
In the ECL-ML Library, a dataset for supervised learning is represented with two datasets: dependent and independent. The dependent dataset is always discrete. One *DiscreteField* record is enough to represent the instance class. The independent dataset needs one record per instance attribute value to represent the independent data of an instance. In this case we are using *DiscreteField* records for the independent dataset because the Weather independent variables are discrete (see Fig. 14), otherwise we were using NumericField records.

NodeID This data structure is used in the definition of tree-related data structures. It wraps the elementary identification of a node. It is composed of 2 fields:

node_id: contains node identifier number.
level: states the Tree level position of the node. Trees start at level one (root).

SplitD The *SplitD* records are used to represent a discrete Decision Tree model composed of a set of Split nodes and Leaf nodes. A Split node is composed of Branches which are links to other nodes. A Leaf node is a kind of "final state" node where the class value can be assigned. The record structure has the following fields:

NodeID: Node Info, all records that compose a Branch or Leaf Node have the same NodeID.
number: identifies Leaf nodes when the value is zero, otherwise it is a Branch node and the value points to the independent variable (attribute) used to split.
value: holds the attribute's value of the Branch, or the class value for Leaf nodes.



node_id	level	number	value	new_node	support
1	1	1	0	2	5
1	1	1	1	4	4
1	1	1	2	3	5
2	2	3	0	6	2
2	2	3	1	8	3
3	2	4	0	5	3
3	2	4	1	7	2
4	2	0	1	0	4
6	3	0	1	0	2
8	3	0	0	0	3
5	3	0	1	0	3
7	3	0	0	0	2

Fig. 15 Decision Tree model representation in HPCC. SplitD dataset—Decision Tree

new_node_id: holds the node_id of the destination node a Branch is linked to.

support: number of instances that got the Branch or Leaf node during the learning process, useful for class probability distribution calculation.

Figure 15 displays, on the left, the SplitD records that represent the Decision Tree built, on the right, learning from the Weather dataset. The top Split node (*root*), composed of many records (*branches*) identified with ***node_id*** = 1 and ***level*** = 1, correspond to the *root* node in the Tree View graph.

The attribute selected to split the *root* node was the attribute number 1, which corresponds to "outlook." Every single record represents one of the branches with info about the attribute value, the destination node "***new_node_id***" and instances count "***support***." Leaf nodes are represented with a record having "***number***" equals to zero and the "***value***" defines the final class, e.g. the node identified with ***node_id*** = 4 represents the leaf node on the second level of the Decision Tree graph, having "1" as final class and four instances ending there (***support***).

NodeInstDisc The NodeInstDisc records are used in the Decision Tree's learning process. They represent the interim structure of the tree at any moment of the learning process and the current location of the training Instances in this tree. Their data structure is composed of 7 fields:

NodeID (2): ***node_id***, ***level***. Node Info.

DiscreteField (3): ***id***, ***number***, ***value***. Instance Independent Data or Branch's split data.

Depend (1): ***depend***. Instance's dependent value or Branch's child node_id.

Support (1): ***support***. Support during learning.

The *NodeInstDisc* record is the INPUT/OUTPUT datatype of the split/partition loopbody function and has multiple meanings based upon the ***id*** field (a component of *DiscreteField*). The value of ***id*** distinguishes the tree structure related records from records stating Node Instance locations. Records having their ***id*** greater than 0 stand for Node-Instances that need further processing. They carry the Instance data: the dependent value in the ***depend*** field and independent values in the ***DiscreteField***, needing one

Table 3 Node-instance records of iteration 2's input

Node_id	Level	Id	Number	Value	Depend	Support
1	1	0	1	4	1	4
1	1	0	1	3	2	5
1	1	0	1	2	0	5
2	2	1	1	***	0	0
2	2	2	1	***	0	0
2	2	8	1	***	0	0
2	2	9	1	***	1	0
2	2	1	1	***	1	0
3	2	4	1	***	1	0
3	2	5	1	***	1	0
3	2	6	1	***	0	0
3	2	0	1	***	1	0
3	2	4	1	***	0	0
4	2	3	1	***	1	0
4	2	7	1	***	1	0
4	2	2	1..	***	1	0
4	2	3	1	***	1	0

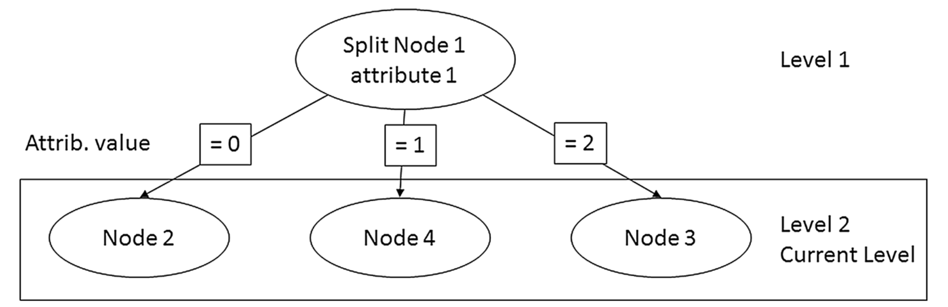


Fig. 16 Decision Tree at the beginning of iteration 2

record for each dependent variable. Moreover, the further processing of these records depends on the ***level*** field.

Records having their ***level*** less than LOOP iteration number (***current level***) belongs to LEAF Nodes. They do not need further processing besides the class support count after the LOOP has finished; whereas, records having their ***level*** equal to current level will be processed by the split/partition function. Records having ***id*** equal to 0 stand for Branches, links between parent nodes and their children, which compose a Split node. Branches carry info about the independent variable used to split the node, the variable's id in the ***number*** field and the variable's value in the ***value*** field.

Example 1—Tree Level Building In this example, we show that the loopbody function of our iterative split/partition process grows the DT using a breadth-first manner, i.e. one complete level at a time. The records in the Table 3 depict the temporary status of the DT at the beginning of iteration number 2 (see Fig. 16).

Table 4 Node-instance records of iteration 2's output

Node_id	Level	Id	Number	Value	Depend	Support
1	1	0	1	4	1	4
1	1	0	1	3	2	5
1	1	0	1	2	0	5
2	2	0	3	8	1	3
2	2	0	3	6	0	2
3	2	0	4	5	0	3
3	2	0	4	7	1	2
4	2	3	1	***	1	0
4	2	7	1	***	1	0
4	2	12	1	***	1	0
4	2	13	1	***	1	0
5	3	4	4	***	1	0
5	3	10	4	***	1	0
5	3	5	4	***	1	0
6	3	9	3	***	1	0
6	3	11	3	***	1	0
7	3	6	4	***	0	0
7	3	14	4	***	0	0
8	3	8	3	***	0	0
8	3	1	3	***	0	0
8	3	2	3	***	0	0

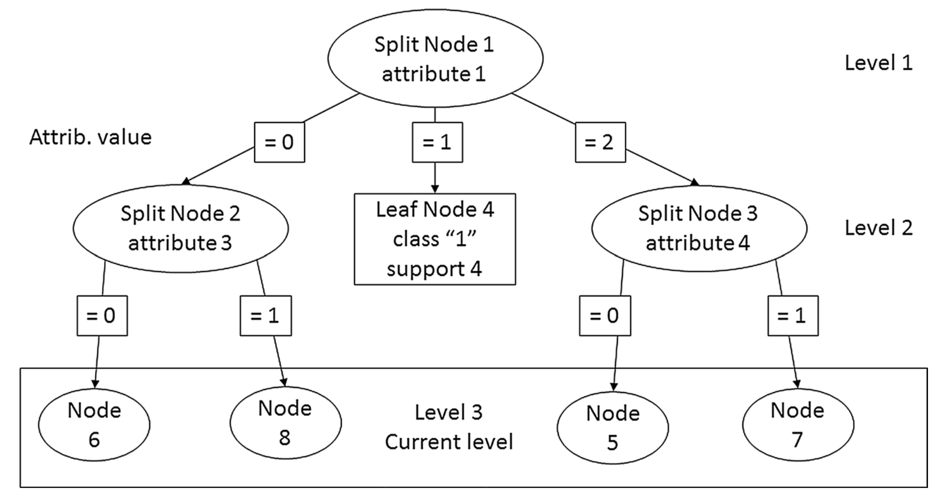


Fig. 17 Decision Tree at the end of iteration 2

The loopbody function receives all *NodeInstDisc* records (shown in Table 3) that represent the DT in construction at the beginning of the 2nd iteration (see Fig. 16). The location of any instance identified by "***id***" is specified by the "***node_id***." Records with "***id***" equal to zero contain split related info, although it is not possible to find such records at the current level processed. The records marked with "***" are repeated once for each attribute value to complete the whole instance data, with the ***number*** identifying the attribute. The function filters out the record of prior levels (***level*** < 2) and performs all the algorithm calculations only using the current level data.

The records in Table 4 depict the temporary status of the DT at the end of iteration number 2 (See Fig. 17).

After the iteration 2, the loopbody function returns all *NodeInstDisc* records (shown in Table 4) that represent the DT in construction (see Fig. 17), just before the iteration 3. Note that the iteration 2 grew the whole DT's level 2. It transformed Node 2 and Node 3 into Split nodes (*branches*) and also, left all the instances located at Node 4 untouched because the node was pure enough to be considered a Leaf node. On a final note, even though the picture shows the Node 4 as a Leaf node of class "1" and support "4," there are really 16 records (4 instances times 4 features as shown in the Table 4) that do not require further loopbody processing.

Received: 14 January 2019 Accepted: 17 July 2019
Published online: 30 July 2019

References

1. Middleton AM, Bayliss DA, Halliday G, Chala A, Furht B. The HPCC/ECL platform for Big Data. In: Big Data technologies and applications. New York: Springer; 2016. p. 159–83.
2. Lexis Nexis Risk Solutions. HPCC Systems: Introduction to HPCC. 2011. http://cdn.hpccsystems.com/whitepapers/wp_introduction_HPCC.pdf.
3. HPCC Systems. ECL Language Reference. 2011. https://hpccsystems.com/training/documentation/ecl-language-reference/html.
4. HPCC Systems. ECL Machine Learning Library. 2010. https://github.com/hpcc-systems/ecl-ml.
5. CAKE: Center for Advanced Knowledge Enablement. CAKE—developing machine learning algorithms on HPCC/ECL Platform. FAU, 2012. http://mlab.fau.edu/cake/developing-machine-learning-algorithms-on-hpcc-ecl-platform/.
6. Breiman L. Random forest. Mach Learn. 2001;45:5–32.
7. Breiman L. Bagging predictors. Mach Learn. 1996;24(2):123–40.
8. Amit Y, Geman D. Shape quantization and recognition with randomized trees. Neural Comput. 1997;9:1545–88.
9. Ho T. The random subspace method for constructing decision forests. In: IEEE Trans. on pattern analysis and machine intelligence. vol. 20, no. 8, 1998. p. 832–44.
10. Breiman L, Friedman J, Olshen R, Stone C. Classification and regression trees. Wadsworth International Group; 1984.
11. Strobl C, Malley J, Tutz G. An introduction to recursive partitioning: rationale, application, and characteristics of classification and regression trees, bagging, and random forests. Psychol Methods. 2009;14(4):323–48.
12. Fernandez-Delgado M, Cernada E, Barro S, Amorim D. Do we need hundreds of classiffiers to solve real world classification problems. J Mach Learn Res. 2014;15:3133–81.
13. Breiman L, Cutler A. Random Forests. Version 5.1. 2004. http://www.stat.berkeley.edu/~breiman/RandomForests/cc_software.htm.
14. Machine Learning Group at the University of Waikato. Weka 3: Data Mining Software in Java. http://www.cs.waikato.ac.nz/ml/weka/index.html.
15. Livingston F. Implementation of breiman's random forest machine learning algorithm. Machine Learning Journal Paper, 2005.
16. Pedregosa F, Varoquaux G, Gramfort A, Michel V, Thirion B, Grisel O, Blondel M, Prettenhofer P, Weiss R, Dubourg V, Vanderplas J. Scikit-learn: machine learning in python. J Mach Learn Res. 2011;12:2825–30.
17. Liaw A, Wiener M. Classification and regression by randomForest. R News. 2002;2(3):18–22.
18. Boulesteix AL, Janitza S, Kruppa J, König IR. Overview of random forest methodology and practical guidance with emphasis on computational biology and bioinformatics. 2012. https://epub.ub.uni-muenchen.de/13766/1/TR.pdf.
19. Alam MS, Vuong ST. Random Forest Classification for Detecting Android Malware. In: Green Computing and Communications (GreenCom), 2013 IEEE and Internet of Things (iThings/CPSCom), IEEE International Conference on and IEEE Cyber, Physical and Social Computing, 2013.
20. Ellis K, Kerr J, Godbole S, Lanckriet G, Wing D, Marshall S. A random forest classifier for the prediction of energy expenditure and type of physical activity from wrist and hip accelerometers. Physiol Measur. 2014;35(11):2191.
21. Hilbert M. Big data for development: a review of promises and challenges. Dev Policy Rev. 2016;34:135–74.
22. Laney D. 3D Data Mangement: controlling data volume, velocity, and variety. Gartner- Meta Group, 2001.
23. Liao Y, Rubinsteyn A, Power R, Li J. Learning Random Forests on the GPU. In: Big Learning 2013: advances in algorithms and data management, Lake Tahoe, 2013.
24. Landset S, Khoshgoftaar T, Richter A, Hasanin T. A survey of open source tools for machine learning with big data in the Hadoop ecosystem. J Big Data. 2015;2:1–36.
25. The Apache Software Foundation, "Apache Mahout," 2014. http://mahout.apache.org.

26. Meng X, Bradley J, Yavuz B, Sparks E, Venkataraman S, Liu D, Freeman J, Tsai DB, Amde M, Owen S, Xin D. Mllib: Machine learning in apache spark. J Mach Learn Res. 2016;17(1):1235–41.
27. The Apache Software Foundation. Apache Spark&trade;—lightning-Fast Cluster Computing. 2014. http://spark.apache.org/.
28. H2O, Inc. Overview—H_2O 3.10.0.9 documentation. 2015. http://docs.h2o.ai/h2o/latest-stable/h2o-docs/index.html.
29. HPCC Systems. Machine Learning Library Reference. 2013. https://hpccsystems.com/download/documentation/machine-learning-ml.
30. Mohri M, Rostamizadeh A, Talwalkar A. Foundations of Machine Learning. New York: MIT Press; 2012.
31. Hunt E, Marin J, Stone P. Experiments in induction. New York: Academic Press; 1966.
32. Verma R. Data Mining Book. 2009. http://www.hypertextbookshop.com/dataminingbook/public_version/.
33. Quinlan JR. Induction of decision trees. Machine Learning. 1986;1:81–106.
34. Li B, Chen X, Li MJ, Huang JZ, Feng S. Scalable Random Forests for Massive Data. In: 16th Pacific-Asia conference on Advances in Knowledge Discovery and Data Mining, Kuala Lumpur, 2012.
35. Bayliss D. HPCC Systems: thinking declaratively. 2011. http://cdn.hpccsystems.com/whitepapers/wp_thinking_declaratively.pdf.
36. Asuncion A. {UCI} Machine Learning Repository, University of California, Irvine, School of Information and Computer Sciences, 2017.
37. Bay S, Kibler D, Pazzani M, Smyth P. The UCI KDD archive of large data sets for data mining research and experimentation. ACM SIGKDD. 2000;2:81–5.

Publisher's Note