



(19) **United States**

(12) **Patent Application Publication**  
**EADS**

(10) **Pub. No.: US 2014/0337255 A1**

(43) **Pub. Date: Nov. 13, 2014**

(54) **SCALABLE, MEMORY-EFFICIENT  
MACHINE LEARNING AND PREDICTION  
FOR ENSEMBLES OF DECISION TREES FOR  
HOMOGENEOUS AND HETEROGENEOUS  
DATASETS**

**Publication Classification**

(51) **Int. Cl.**  
**G06N 99/00** (2006.01)  
(52) **U.S. Cl.**  
CPC ..... **G06N 99/005** (2013.01)  
USPC ..... **706/12**

(71) Applicant: **WISE IO, INC.**, Berkeley, CA (US)

(72) Inventor: **DAMIAN RYAN EADS**, SAN FRANCISCO, CA (US)

(57) **ABSTRACT**

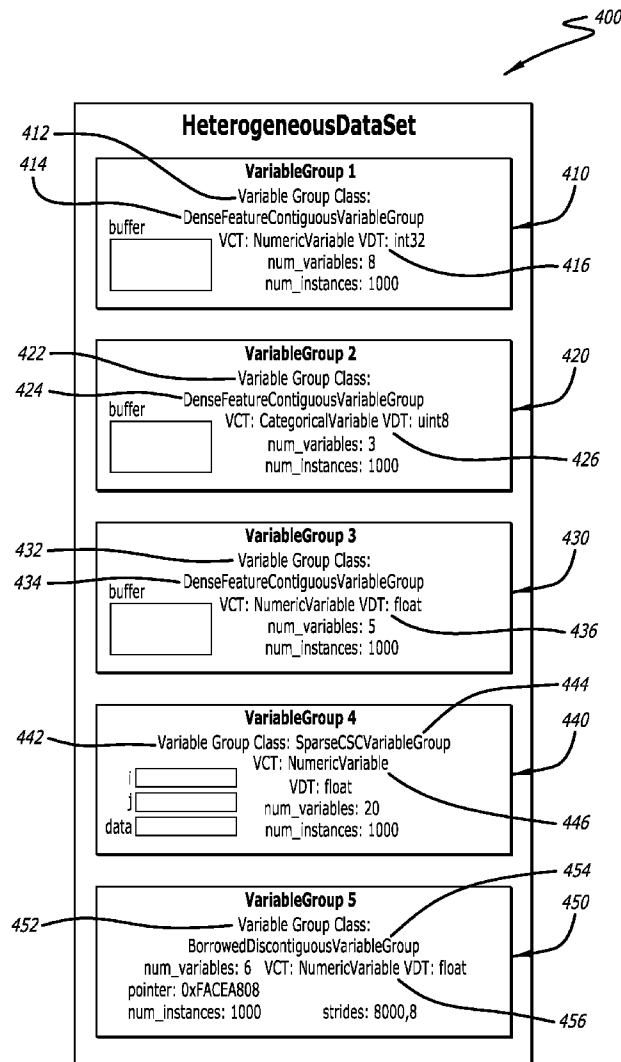
Optimization of machine intelligence utilizes a systemic process through a plurality of computer architecture manipulation techniques that take unique advantage of efficiencies therein to minimize clock cycles and memory usage. The present invention is an application of machine intelligence which overcomes speed and memory issues in learning ensembles of decision trees in a single-machine environment. Such an application of machine intelligence includes inlining relevant statements by integrating function code into a caller's code, ensuring a contiguous buffering arrangement for necessary information to be compiled, and defining and enforcing type constraints on programming interfaces that access and manipulate machine learning data sets.

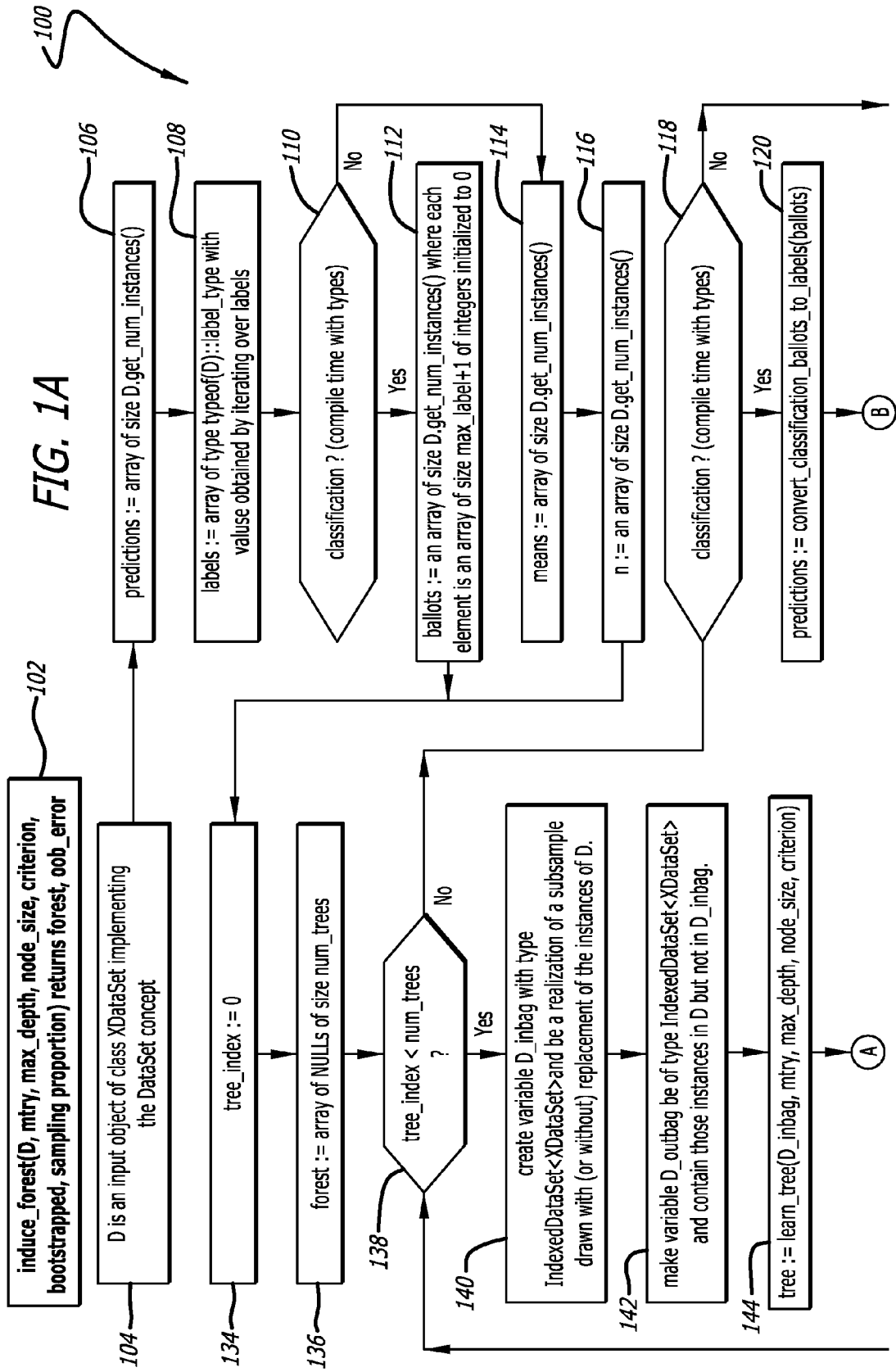
(21) Appl. No.: **14/272,254**

(22) Filed: **May 7, 2014**

**Related U.S. Application Data**

(60) Provisional application No. 61/820,358, filed on May 7, 2013.





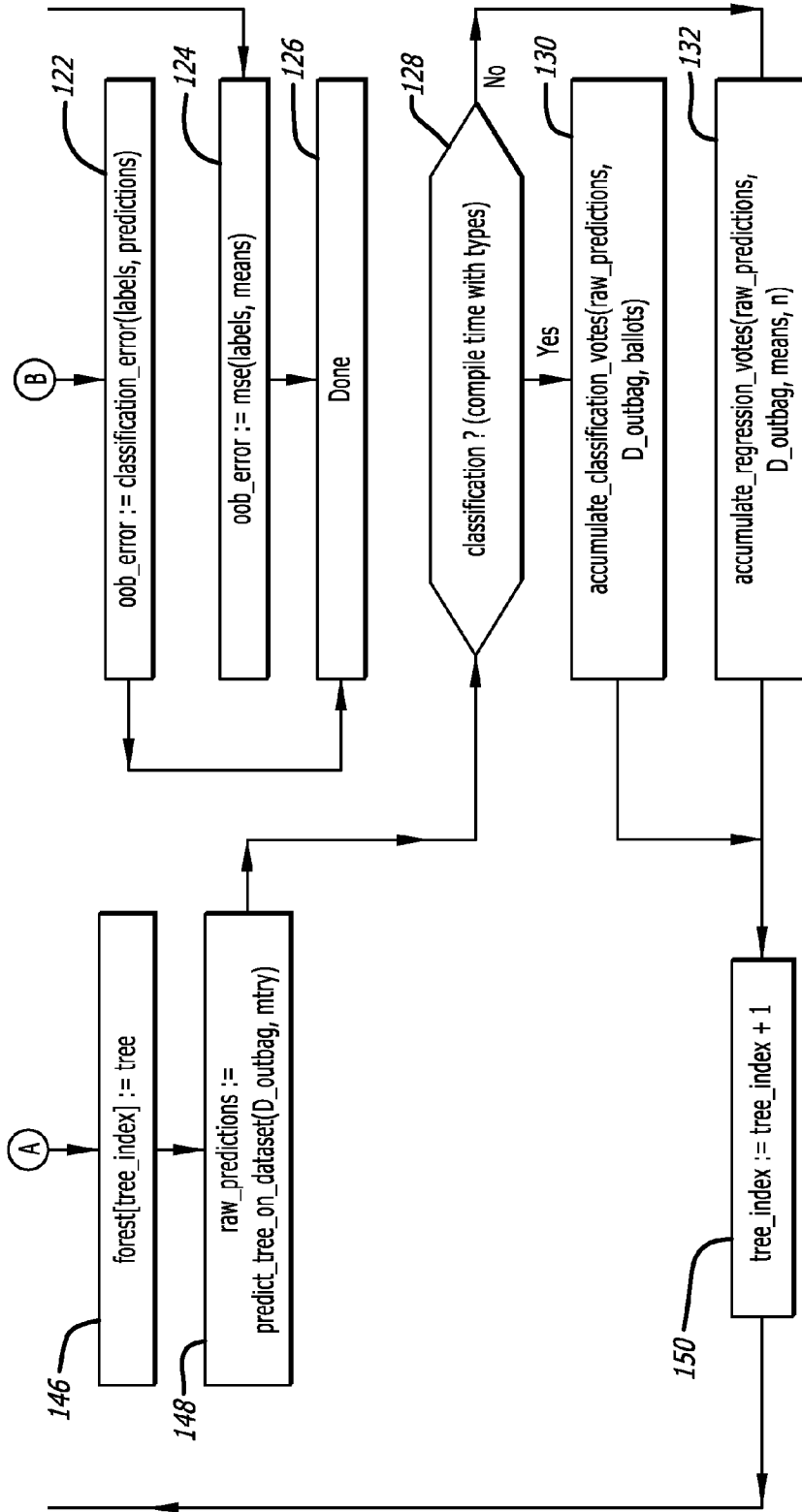
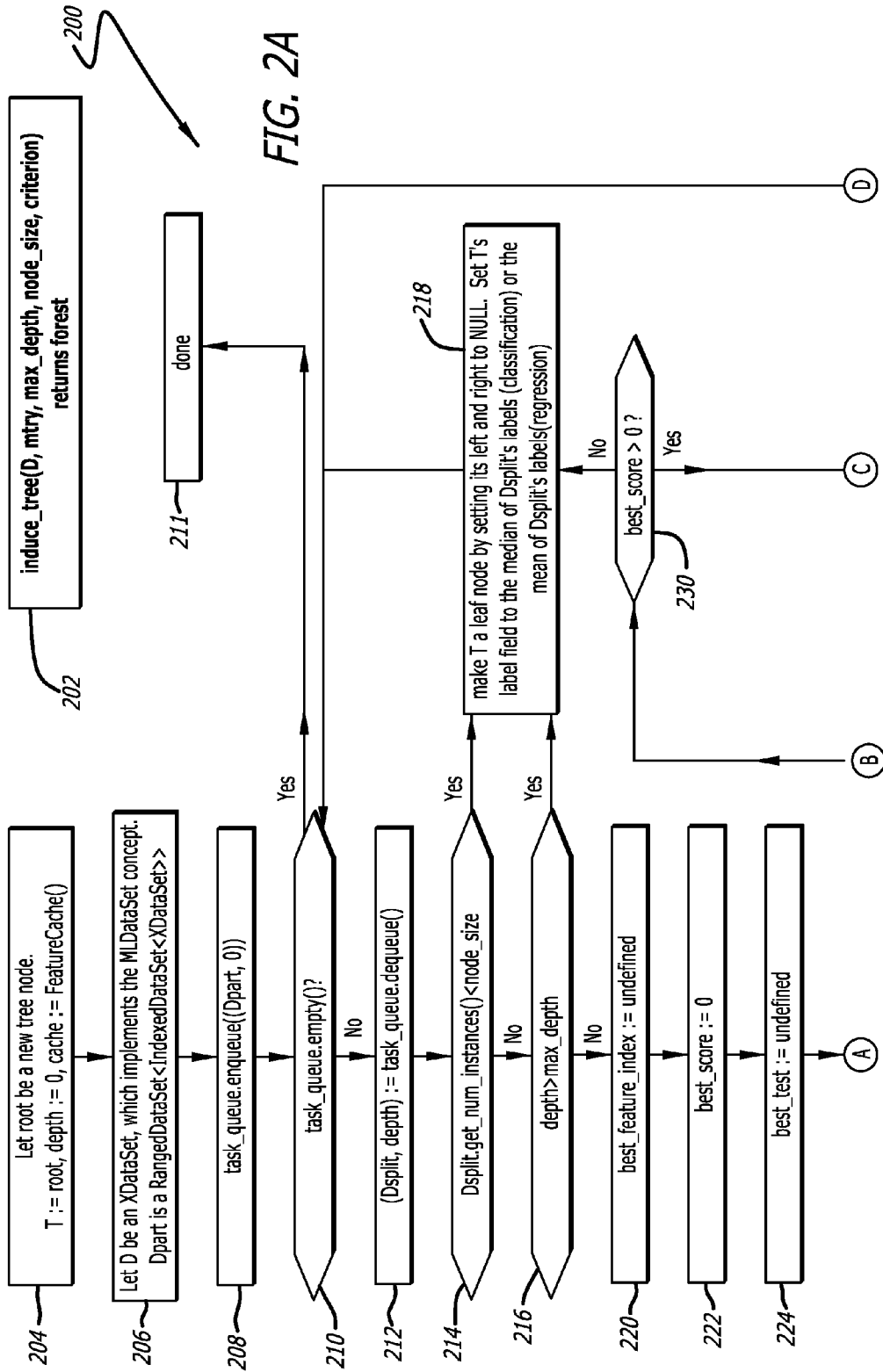


FIG. 1B



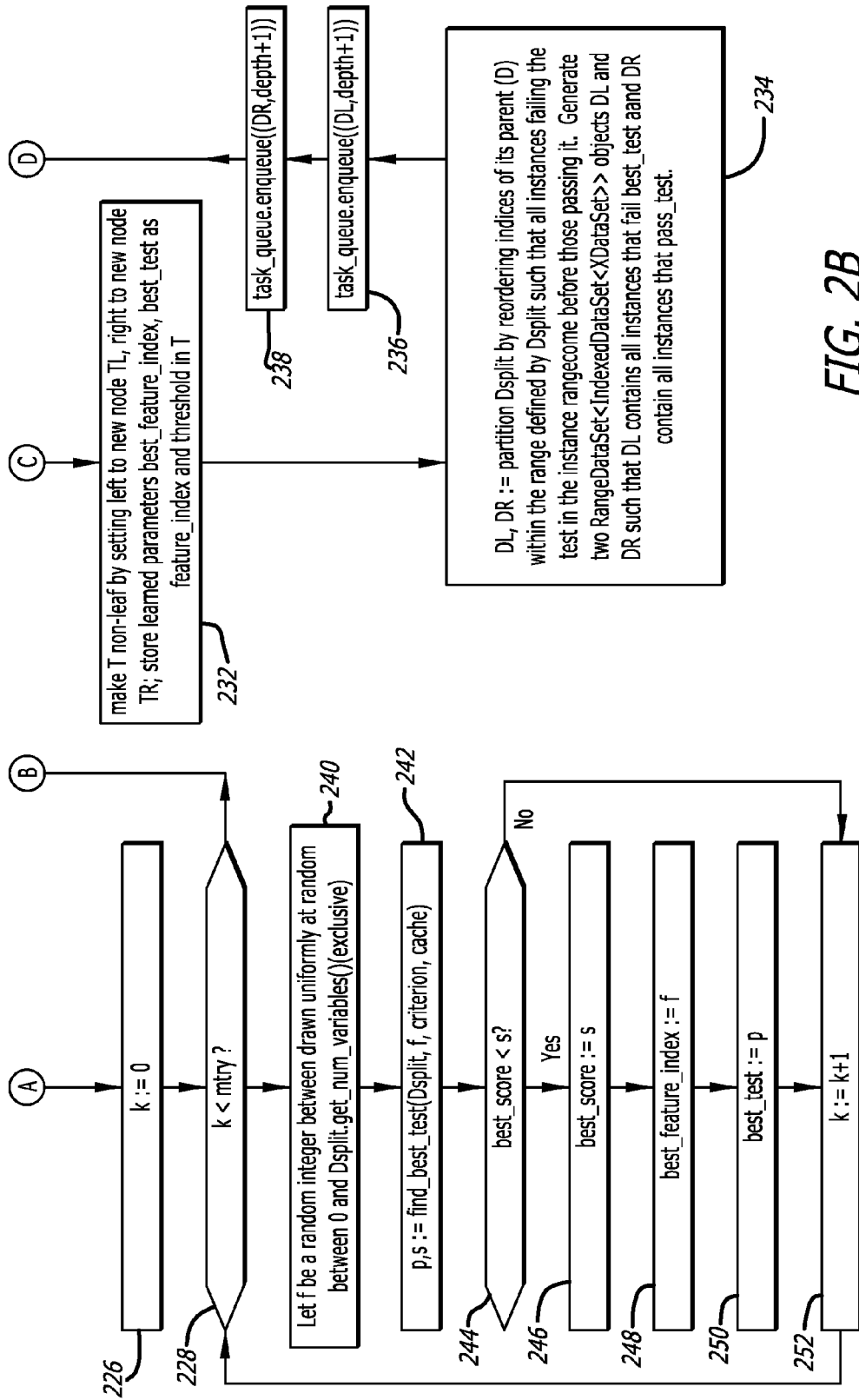


FIG. 2B

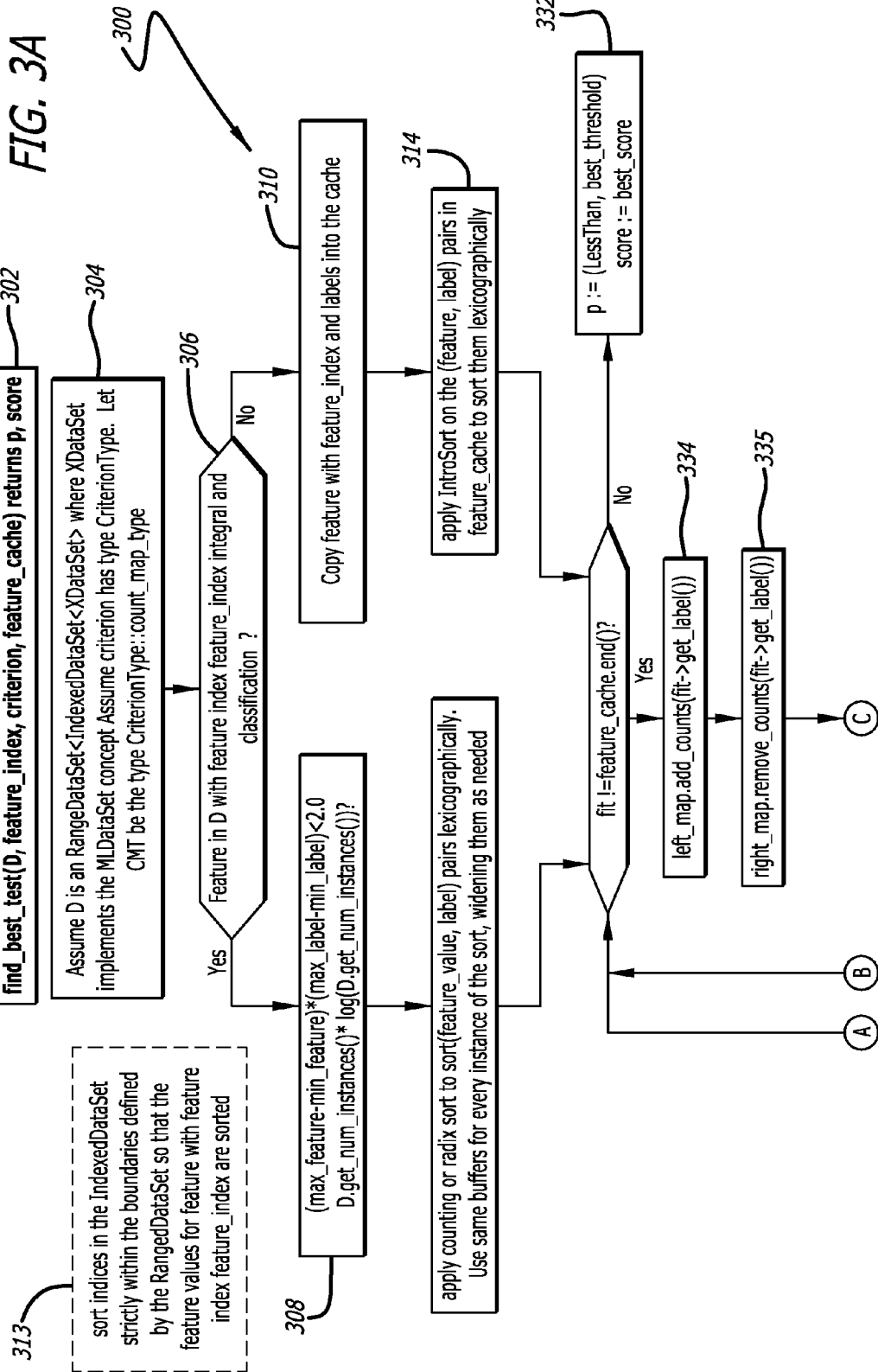


FIG. 3B

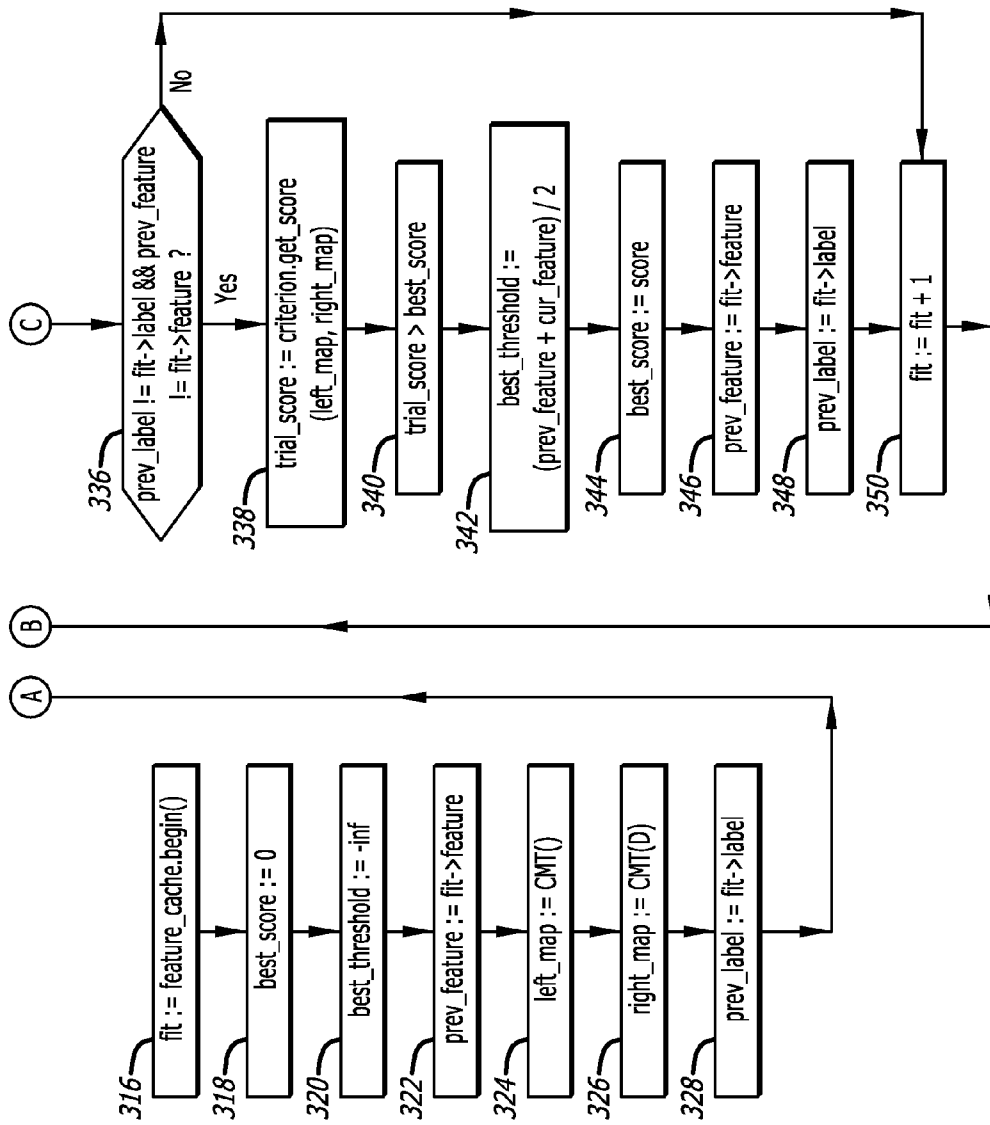
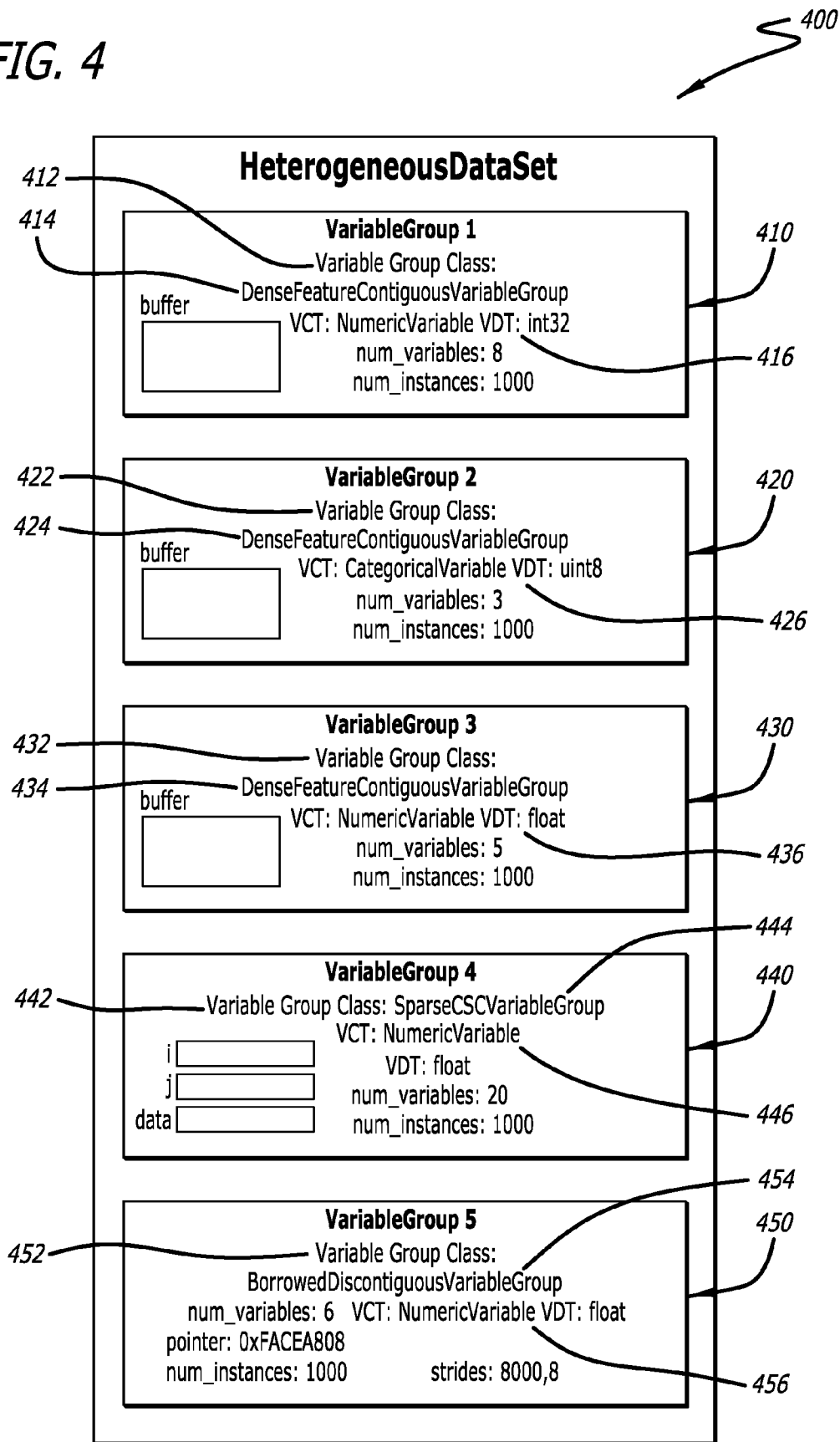


FIG. 4



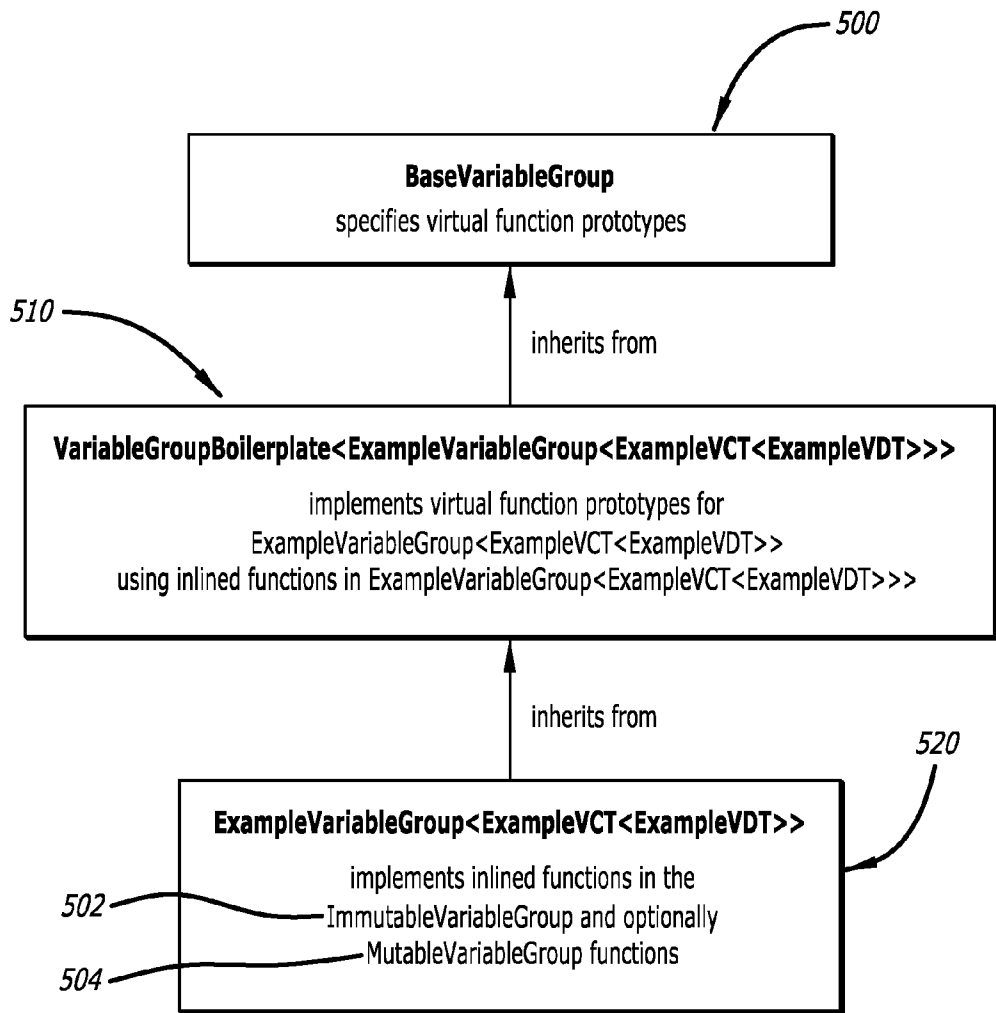


FIG. 5

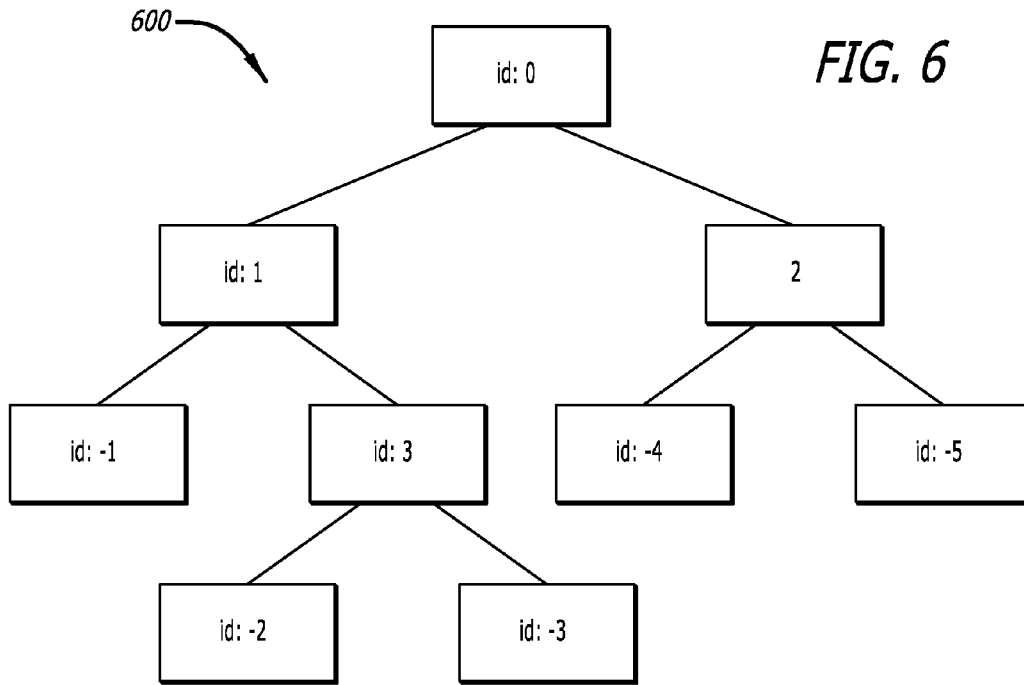
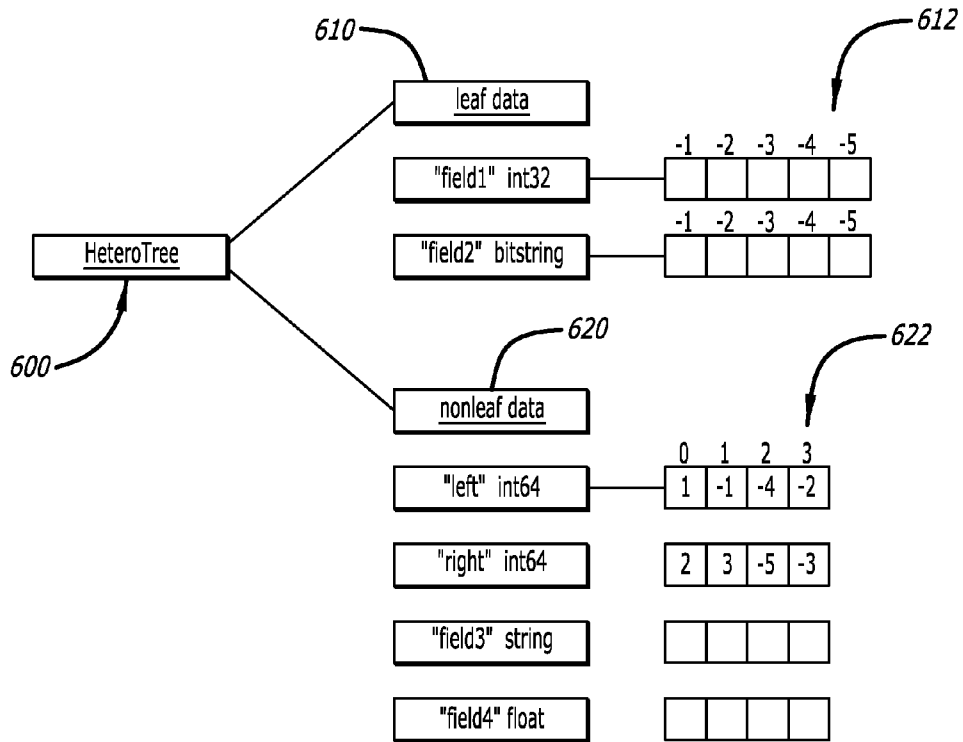


FIG. 6



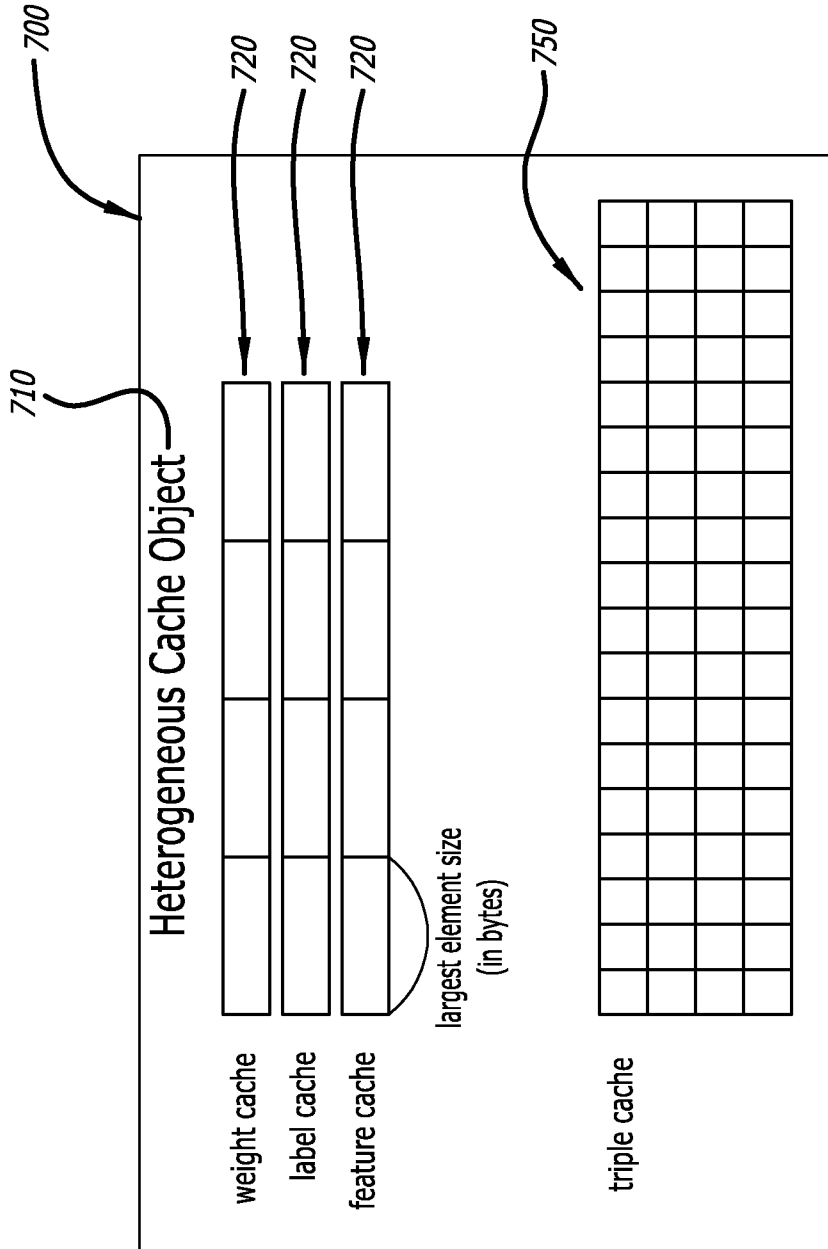
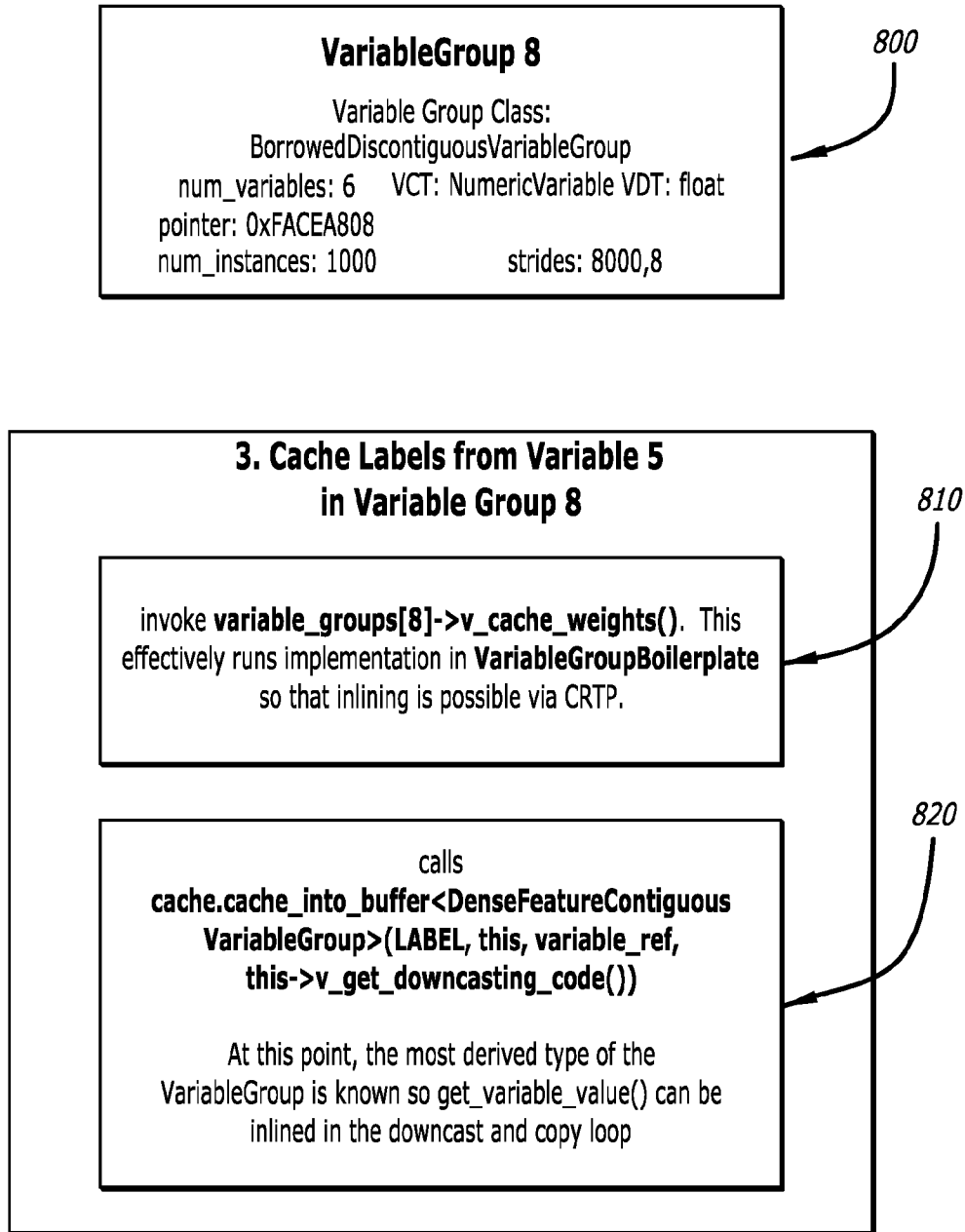


FIG. 7



**FIG. 8**

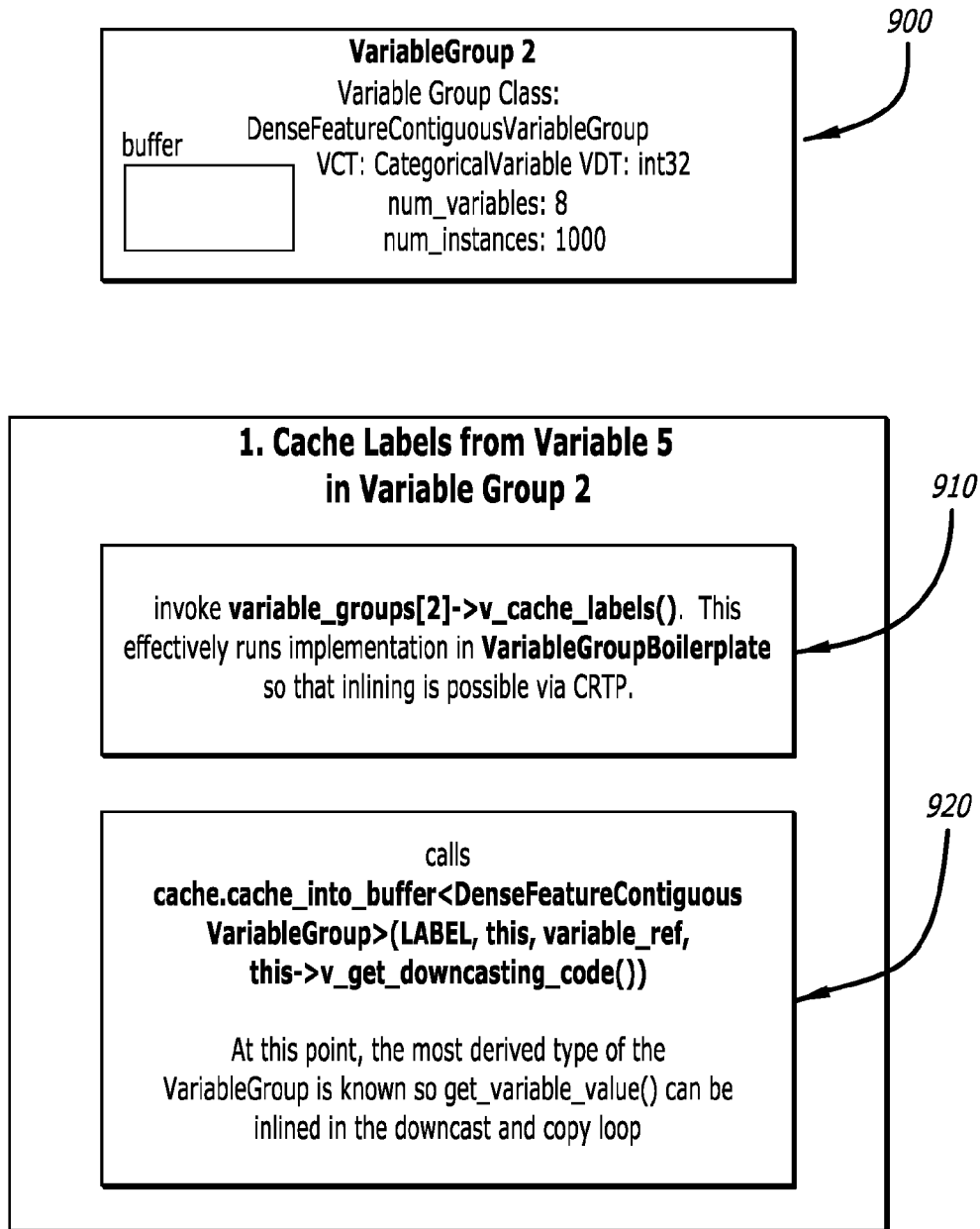


FIG. 9

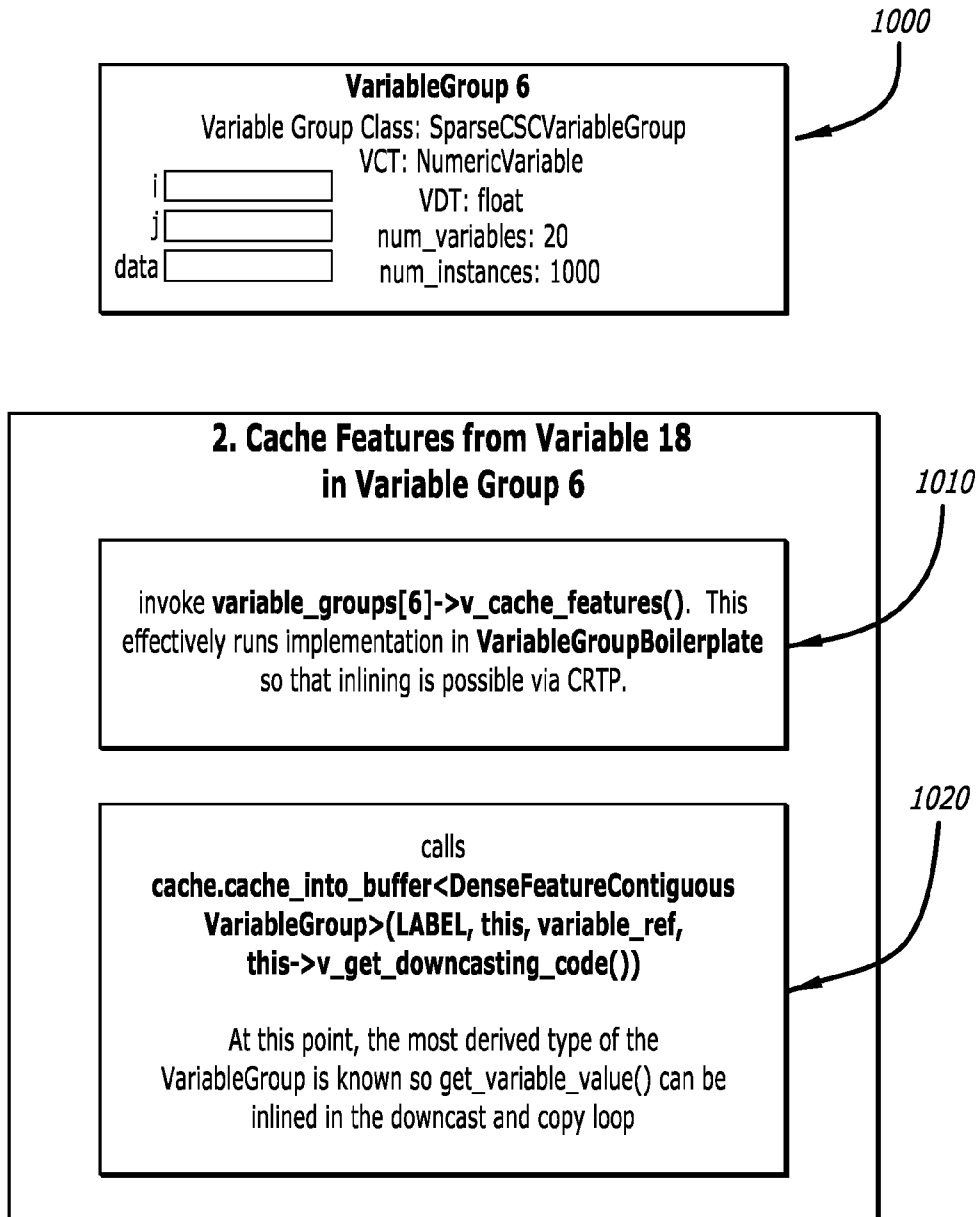
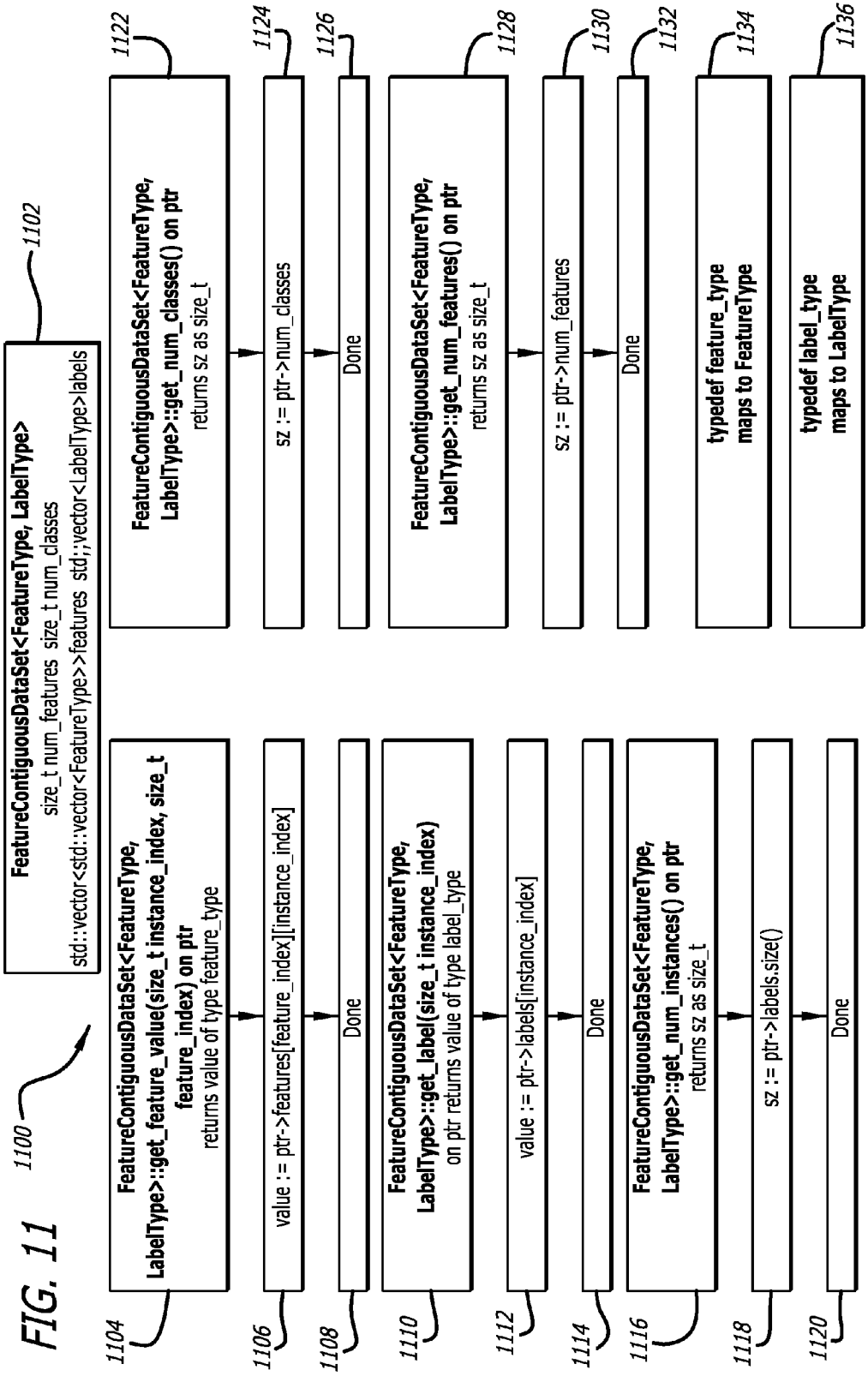
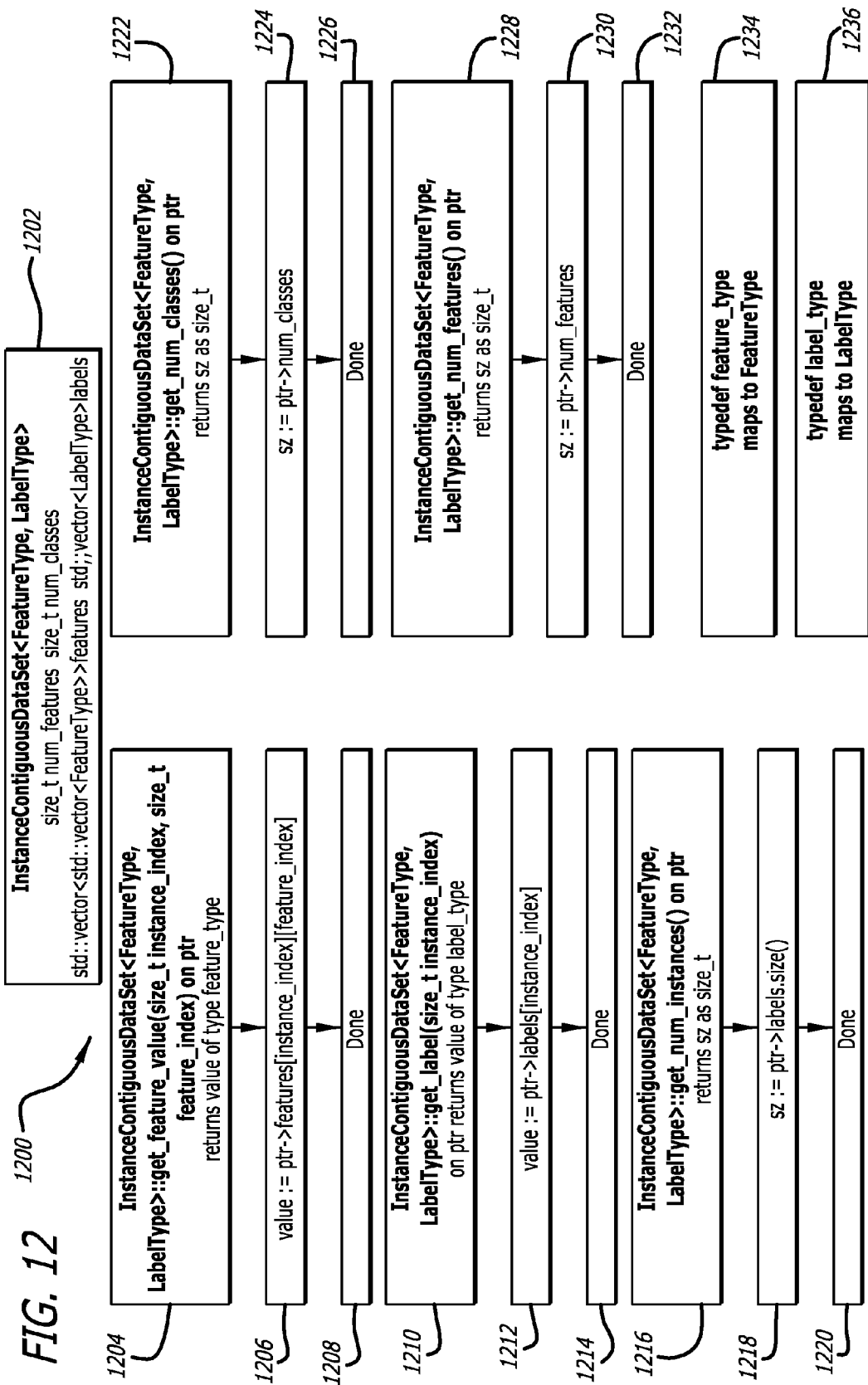
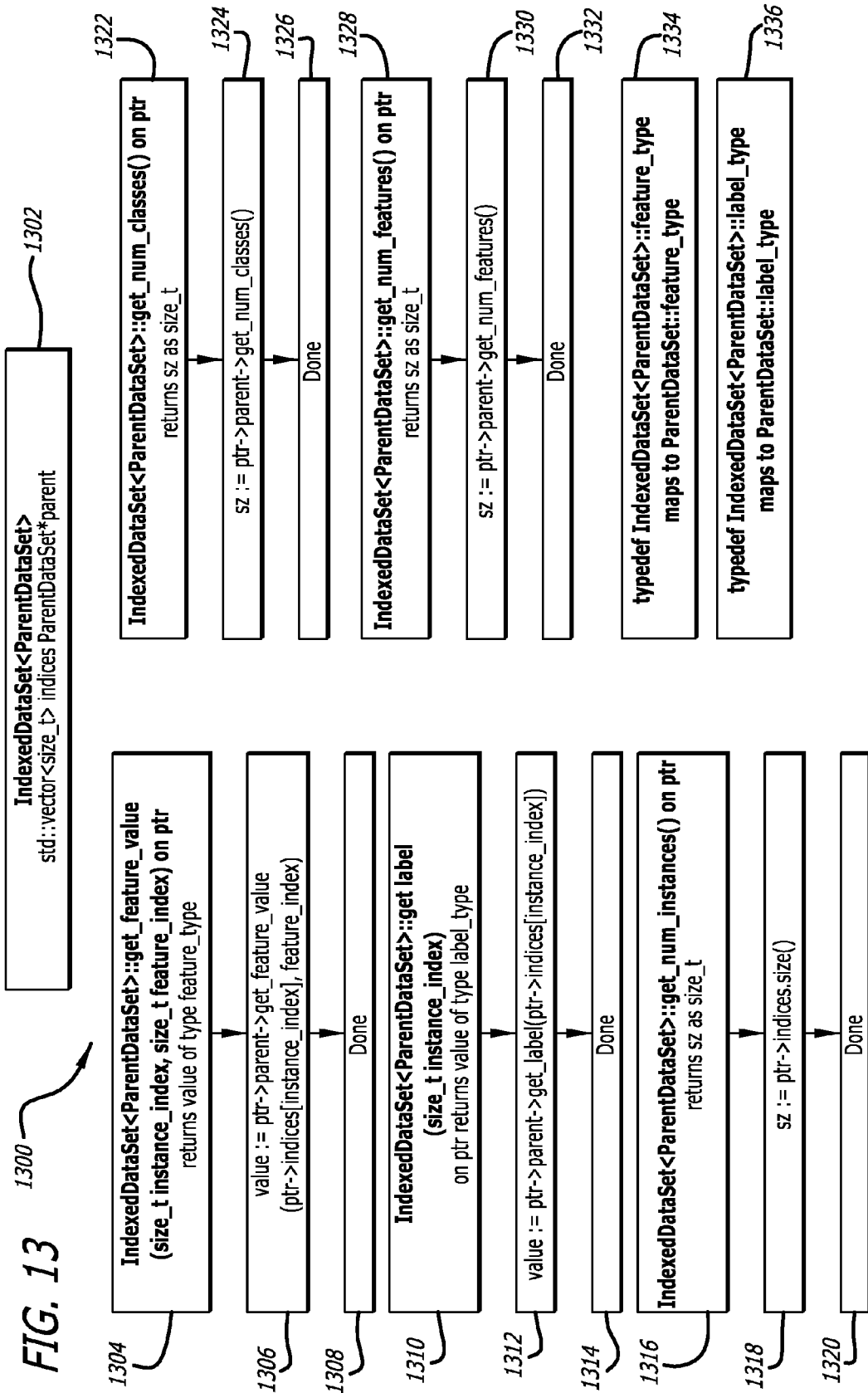
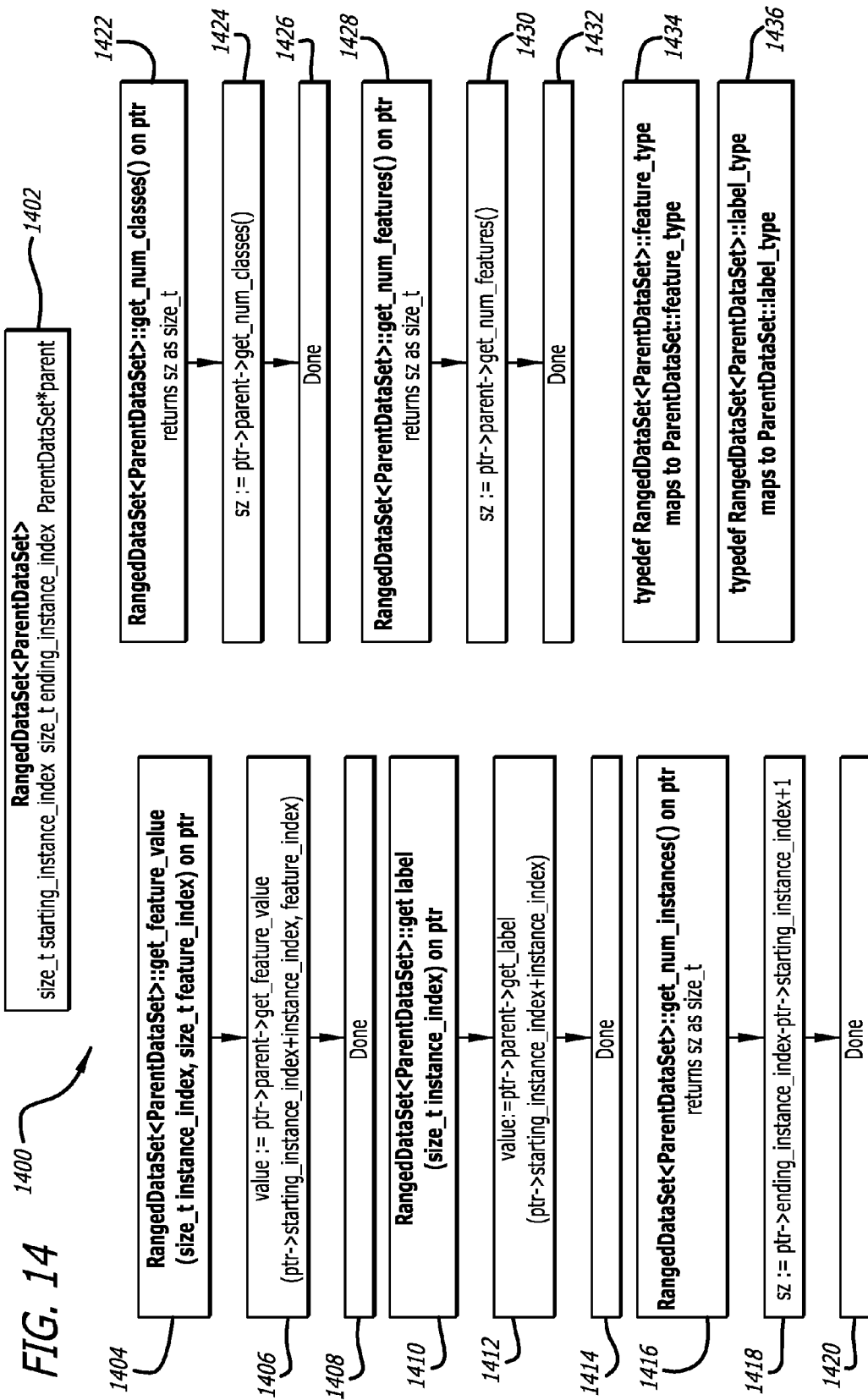


FIG. 10









**SCALABLE, MEMORY-EFFICIENT  
MACHINE LEARNING AND PREDICTION  
FOR ENSEMBLES OF DECISION TREES FOR  
HOMOGENEOUS AND HETEROGENEOUS  
DATASETS**

CROSS-REFERENCE TO RELATED PATENT  
APPLICATIONS

**[0001]** This patent application claims priority to U.S. provisional application 61/820,358, filed on May 7, 2013, the contents of which are incorporated in their entirety herein.

FIELD OF THE INVENTION

**[0002]** The present invention relates to machine intelligence. Specifically, the present invention relates to a method and system of scaling machine learning and prediction to very large data sets comprised of homogeneous and heterogeneous datasets in which manipulation of computer architecture is applied in various techniques to efficiently utilize available memory and minimize limitations on accuracy and speed of data processing.

BACKGROUND OF THE INVENTION

**[0003]** There are many existing implementations of machine intelligence. Learning ensembles of decision trees is a popular method of machine learning, and one such implementation, known as Random Forests, are a combination of several decision trees. Decision trees permit the use of multiple types of data that do not need to be normalized. Random Forests have been specifically defined as a combination of tree predictors such that each tree is learned on a random sample of the instances for all trees in the forest.

**[0004]** Despite the popularity of machine intelligence implementations such as Random Forests™, there are a number of inherent limitations which discourage practical application to very large data sets. These types of traditional analytics tools often fall flat, since they cannot function properly with large, high-dimensional, complicated data sets, particular as data sizes become very large, and also because of the frequent presence of difference data formats and types. There are many Random Forest, permutations in the existing art, but they suffer from an inability to take full advantage of intricate computer architecture environments in which they are tasked to make sense out of these immense data populations in an efficient manner.

**[0005]** As the global economy relies more and more on rapid data-driven analytics, there is an immediate need, unrealized by existing implementations, for fast, scalable, and easy-to-use machine intelligence that can perform accurate prediction and extract deep and meaningful insight out of large data sets. There is a further need for fast, scalable, and easy-to-use machine intelligence that can accomplish these tests with both homogeneous data sets as well as with heterogeneous data sets comprised of both numeric and non-numeric data.

**[0006]** It is therefore one objective of the present invention to provide a machine intelligence framework that is efficient, accurate, and fast. It is a further objective of the present invention to provide such a framework in a common, single-computer implementation in which both machine learning and machine prediction are scalable to arbitrarily large data sets. It is yet another objective of the present invention to

provide such a framework for large data sets that are comprised of homogeneous and heterogeneous sets of data.

BRIEF SUMMARY OF THE INVENTION

**[0007]** The present invention is an application of machine intelligence which overcomes speed and accuracy issues in groups of decision trees known as Ensembles of Decision Trees (which may herein be also referred to as “EDTs”) by applying a systematic process through a plurality of computer architecture manipulation techniques. These techniques take unique advantage of computer architecture efficiencies to minimize clock cycles and memory usage that slow down big data analytics. In one aspect, the objectives identified above are achieved by utilizing inlining procedures to efficiently manage contents of registers to reduce function call overhead from excessive clock cycles, configuring information to be compiled in one or more contiguous buffers to improve cache coherency, and applying principles of static polymorphism using one or more dataset concepts to implement functions that minimize inheritance from dynamic dispatch overhead. The present invention therefore implements several techniques used in the practice of computer vision to extract a much faster response from computer architecture than previously contemplated in the existing art. The present invention can also be thought of as applying type constraints on programming interfaces that access and manipulate machine learning data sets to optimize available memory usage hampered by unnecessary data copying and minimize speed limitations resulting from so-called cache misses, function call overhead, and excessive processing time from performing multiple if/then conditional statements.

**[0008]** The present invention applies these computer architecture manipulation techniques across both training and prediction modes of operation in Random Forest implementations. Training involves learning a forest of decision trees from labeled feature vectors, and prediction takes a learned model based on the training set and generates predictions on a new set of data. Within these two modes, the techniques disclosed in the present invention address the two main functions, classification and regression, of processing data that lead to memory efficiency issues and speed constraints as noted above.

**[0009]** Other embodiments, features and advantages of the present invention will become apparent from the following description of the embodiments, taken together with the accompanying drawings, which illustrate, by way of example, the principles of the invention.

BRIEF DESCRIPTION OF THE SEVERAL  
VIEWS OF THE DRAWINGS

**[0010]** The accompanying drawings, which are incorporated in and constitute a part of this specification, illustrate several embodiments of the invention and together with the description, serve to explain the principles of the invention.

**[0011]** FIG. 1 is a flow diagram representing steps in inducing a forest in the processing of learning a decision forest for homogeneous datasets according to one embodiment of the present invention;

**[0012]** FIG. 2 is a flow diagram representing steps in inducing a tree in the process of learning a decision forest for homogeneous datasets according to one embodiment of the present invention;

[0013] FIG. 3 is a flow diagram representing steps in inducing a node in the process of learning a decision forest for homogeneous datasets according to one embodiment of the present invention;

[0014] FIG. 4 is a block diagram of a heterogeneous dataset for processing according to another embodiment of the present invention;

[0015] FIG. 5 is an exemplary block diagram illustrating inlining functions for a heterogeneous dataset as in FIG. 5;

[0016] FIG. 6 is a block diagram illustrating application of a heterogeneous tree using a heterogeneous dataset according to the present invention;

[0017] FIG. 7 shows an example of buffers in a heterogeneous cache object for storing a heterogeneous dataset according to the present invention;

[0018] FIG. 8 is an exemplary block diagram illustrating of caching of weights prior to populating a triple buffer for a heterogeneous cache object according to the present invention;

[0019] FIG. 9 is an exemplary block diagram illustrating of caching of labels prior to populating a triple buffer for a heterogeneous cache object according to the present invention;

[0020] FIG. 10 is an exemplary block diagram illustrating of caching of features prior to populating a triple buffer for a heterogeneous cache object according to the present invention;

[0021] FIG. 11 is a conceptual flow diagram illustrating a class implementation for training of data within a MLDataSet concept according to the present invention;

[0022] FIG. 12 is a conceptual flow diagram illustrating a class implementation for caching an instance's feature for prediction of data within a MLDataSet concept according to the present invention;

[0023] FIG. 13 is a conceptual flow diagram illustrating a class implementation for use of inlining to index instances of another data set object to enable memory-efficient subsampling and computational reductions within a MLDataSet concept according to the present invention; and

[0024] FIG. 14 is a conceptual flow diagram illustrating a class implementation for a contiguous partition of instances within another data set to enable memory-efficient within a MLDataSet concept according to the present invention.

#### DETAILED DESCRIPTION OF THE INVENTION

[0025] In the following description of the present invention reference is made to exemplary embodiments illustrating the principles of the present invention and how it is practiced. Other embodiments will be utilized to practice the present invention and structural and functional changes will be made thereto without departing from the scope of the present invention.

[0026] The present invention provides approaches to implementing a learning ensemble of decision trees in a single-machine environment in which inlining to optimize relevant material for compilation by integrating function code into a caller's code at compilation, ensuring a contiguous buffer arrangement for necessary information to be compiled, and configuring one or more mechanisms for defining and enforcing constraints on types, known as C++ concepts, are techniques utilized to maximize memory usage and minimize speed constraints. Each of these applications is discussed in further detail herein. Each of inlining, buffer contiguity, and C++ concepts, when combined and applied to a learning

ensemble of decision trees, represents an enhancement of processing speed and memory usage in analyzing large data sets in a time-effective and cost-effective manner.

[0027] In machine intelligence, there are two general modes of operation: training (or learning) and prediction (or testing). Training involves learning a forest of decision trees from labeled feature vectors, and prediction takes a learned model based on the training set and generates predictions on a new set of data. Within these two modes, there are two kinds of prediction: classification and regression. A training set of data is a group of  $n$  pairs  $(x_1, y_1), (x_2, y_2), \dots, (x_n, y_n)$  where  $x_j$  is a vector (or, feature vector) representing the input variables (or, features) and  $y$  represents the output variable (or, label). A (label, feature vector) pair is referred to as an "instance." The  $x_j$  value is known as a feature and the integer  $j$  that identifies a feature dimension is called a feature index. The length of  $x_j$  is referred to as the feature dimension. An unlabeled data set is group of feature vectors  $x_1, x_2, \dots, x_n$  without any labels associated with them.

[0028] A decision tree is a data structure used to predict an output variable from an input set of variables. The variables can be real-valued (e.g., the price of a used car), Boolean (e.g. whether the car has been in accident or not), or categorical (e.g. the make of the car). When the output variable is Boolean or categorical, we say the decision tree is a classification tree; when this output variable is real-valued, we say the decision tree is a regression tree.

[0029] At a particular tree node, a Splitting Test is applied to instances given to the tree node. In the case of integer/ordinal and real-valued features, a thresholding is applied. If a real-valued or integer feature is below a threshold value, the test passes, otherwise, it fails. For categorical features, a categorical feature value is checked for membership to a subset of category indices. If it belongs, the test passes. Otherwise, it fails.

[0030] A Decision Tree Node encodes the following information:

[0031] Node Left Child: a handle to the left node for a non-leaf node or a sentinel value for a leaf node.

[0032] Node Right Child: a handle to the right node for a non-leaf node or a sentinel value for a leaf node.

[0033] Node Feature Index: the feature index or feature indices (represented by a collection of feature indices) that is/are needed to apply the test.

[0034] Node Test Parameters: the parameterization of the test, if any.

[0035] Node Threshold: the threshold (real-valued and integer features) or container of category indices representing a subset (categorical features) used for the test for non-leaf nodes. This value is a special sentinel value if the tree node is a node.

[0036] Node Label: the label to predict for a leaf node or a sentinel value for a non-leaf node.

[0037] Node Uncertainty Parameters: optional uncertainty parameters for a leaf node or a sentinel value for a non-leaf node.

[0038] Node UID: an optional unique identifier.

The process for predicting on a decision tree begins at the root decision tree node. Then, the following recursive process is performed on a decision tree node  $X$ :

[0039] 1. if the decision tree node  $X$  is a leaf node, output the label encoded in  $X$ , output the uncertainty (if available) encoded in  $X$ , and output the unique identifier

encoded in X (if requested). Stop. Prediction on the target decision tree node is complete.

**[0040]** 2. if the decision tree node X is not a leaf node, apply the test given the parameterization encoded in X on the feature or features corresponding to the feature indices also encoded in X.

**[0041]** 3. if the test passes, repeat step 1 where X is now the left node encoded in X (i.e.  $X:=X.left$ )

**[0042]** 4. if the test fails, repeat step 1 where X is now the right node encoded in X (i.e.  $X:=X.right$ )

**[0043]** An ensemble of decision trees (EDT) is a group of decision trees, as noted above. The EDT is often represented as some collection of handles to root decision tree nodes. Predictions are made from an EDT by having each decision tree in the EDT “cast” a vote, and aggregating the votes. A classification EDT (CEDT) is an ensemble of classification trees, and similarly, a regression EDT (REDT) is an ensemble of regression trees. In this simplest case, predictions are made from a CEDT by taking the majority vote of the predictions made by the classification trees, and the weighted (by uncertainty) or unweighted mean is used to predict from an REDT. Many versions of EDTs are often referred to in relevant literature as a Random Forest.

**[0044]** The approach of the present invention identified above represents a significant advancement over existing implementations of machine intelligence that are either slow or use too much memory, or both. Speed is a significant concern when applying machine learning techniques to large data sets, since it affects the ability to quickly and efficiently capture the deeply embedded insight available from applying analytics to the large volumes of data available. There are several issues affecting speed in machine intelligence.

**[0045]** The primary causes of speed problems in machine intelligence include a lack of contiguity, which can be thought of as a lack of locality of reference. For example, suppose there are 1024 32-bit floats are needed for a computation. If they are located in the same 4096 byte page, then only a single page is needed. If each float instead resides in a different page, then 1024 pages are needed, with 4 bytes being used in each page, and the remaining 4092 bytes are wasted. A page fault results when a page in virtual memory does not reside in physical memory. A cache miss occurs when a block of memory needed for computation is outside the processor cache. Both page faults and cache misses can slow down the run time of a procedure. A lack of contiguity hinders a program’s ability to make effective use of the processor cache and memory hierarchy, and can lead to page faults, cache misses, cache freezes, or cache thrashing. All of these incidents can slow down the run time of a procedure significantly.

**[0046]** Another of the primary causes affecting speed is function, or procedural call overhead. Most existing implementations of learning ensembles of decision trees are decomposed into sub-procedures, some of which are called recursively. The content of the processor (e.g., its registers) are often saved before a procedure call and restored when the sub-procedure returns control back to its caller. Still another of the primary causes affecting speed is repeated evaluation of if/then conditional statements on values that do not change during the execution of a procedure. The assumed value can be encoded as a type and passed as a template argument to a class or function. Partial template specialization can be used to exhibit specialized behavior for specific values. Some algorithms repeatedly compute an expression that evaluates to a Boolean repeatedly. By performing as much of this compu-

tation at compile time, program run time can be reduced. Dynamic dispatch is another issue affecting speed. Dynamic dispatch is a result of using inheritance and principles of dynamic polymorphism, and is a commonly-known way to abstract a sub-procedure’s behavior in virtual methods of a subclass to execute the desired code. Many dynamic dispatch mechanisms (such as virtual functions in C++, abstract methods in Java, and virtual methods in Python) can incur substantial overhead. Additionally, the use of the same general algorithms (e.g. sorting, partitioning, statistics computation, set data structures) that are agnostic to the type of feature or shape of the data set, in terms of number of features or number of instances, may not exploit efficiencies, and this can have a significant impact on processing speed. The present invention may encode these assumptions in a type and use partial template specialization to exhibit specialized behavior that exploits efficiencies for each subtype.

**[0047]** As noted above, excessive memory usage is also a significant problem for existing decision tree implementations of machine intelligence. The primary causes for this high memory usage include unnecessary copying of data to generate subsets before learning a tree or inducing a split. Metadata is another cause of high memory usage, since some data structures incur substantial metadata overhead (e.g., a red-black tree or a linked list). Also, memory initialization requirements are problematic, since some implementations use significant memory resources for initialization. Some implementations do not quickly return memory used for initialization back to the operating system (e.g. via a system call).

**[0048]** The approach of the present invention and the various embodiments described herein addresses these root causes of speed and memory issues affecting machine intelligence. This is achieved in several mechanisms that when combined, result in performance improvements that enable significant applicability of the present invention in a single machine environment.

**[0049]** These mechanisms include application of one or more C++ concepts; each concept represents a set of function signatures, and typedefs that a class must have to “implement the concept”. In the present invention, one such concept is a machine learning dataset (MLDataset) that is a representation of machine learning datasets in C++. Application of such a concept addresses all of the memory and speed bottlenecks outlined above and allows for instance subsetting, feature subsetting, multiple views of the same data set, problem-specific memory layouts, and wrappers for data structures passed from other programming languages (e.g. Java, Python, R, MATLAB, etc.).

**[0050]** Another concept addresses buffer contiguity by implementing solutions to address the problem of a lack of speed associated with avoidable cache misses, page faults, and fragmentation due to data elements needed for a computation are located in disparate virtual memory locations. One solution is a feature buffer cache, which copies dis-contiguous feature values (i.e. all instances with respect to a specific feature dimension) into a compact, contiguous buffer. The mechanism also includes an instance buffer cache which copies a dis-contiguous feature vector for an instance into a contiguous buffer. These solutions ensure that information to be gathered is located in the same buffer, thereby avoiding the delay of large numbers of clock cycles which significantly reduce processing speed.

**[0051]** Another of the concepts employed by the present invention includes encoding assumptions as types. In this aspect, rather than repeatedly evaluating a Boolean expression, the present invention evaluates a Boolean expression only when it could possibly change, and uses the outcome value of the Boolean expression to dispatch a specialization. Still another approach involves automatic assessment of data set assumptions. When a data set is loaded into memory from disk, some initial assumptions are deduced from it and encoded as types. At the compile time, some of these types can be used to dispatch specializations that potentially exploit efficiencies for those assumptions using template meta-programming.

**[0052]** Specialized sorting algorithms may also be employed in an ensemble of decision trees implementation in the present invention. Depending on the characteristics of the data, different variants of sorting algorithms can be used. In template meta-programming, the training process in learning ensembles of decision trees is parameterized with template arguments. Template-based pattern matching is often used to allow for more complex specialization of variants of the learning ensemble of decision trees algorithm.

**[0053]** Also, the approach of the present invention may dispatch instantiations of variants of decision tree learning and prediction. At program initialization, learning and prediction functions are instantiated according to different assumptions. These assumptions are encoded with types, and partial template specializations are instantiated by the compiler. Objects are created from these instantiations and stored in an associative array. These objects are keyed by a string of meta-data.

**[0054]** Other concepts which may be employed include inlining of variants to reduce function call overhead, as discussed in detail herein, and histogram approximation variants. The present invention contemplates use of inlining on procedures to reduce the function call overhead, especially procedures called a large number of times. Building a histogram to approximate a large feature column is another mechanism for achieving the objectives of the present invention. In this approach, the present invention approximates a feature with a specific feature index in a data set by using histograms rather than exact feature values.

**[0055]** The present invention therefore integrates, in one embodiment, multiple approaches to learning ensembles of design trees that when combined, take advantage of existing features of computer architecture and result in increased processing speed and efficient memory usage. One of these approaches, inlining, is a compiler optimization that replaces a function call site with the body of the callee that improves computation time at runtime. Inlining is invoked as a C++ concept and operates to copy the compiled code for a function into the caller's compiled code to avoid function call overhead (e.g. pushing register content and local variables to the stack for each function call). Use of this technique eliminates having to incur this function call overhead (potentially) billions of times. For each push to the stack from a register (or local variable) during each function call, all register data and local variable is popped off the stack to restore previous content, consuming several clock cycles (the exact number of cycles depends on the specific processor family, model, and subtype) for each push and pop. Inlining minimizes these pushes and pops to and from the stack to conserve the usage of clock cycles. This speeds up the process of calling a function, thereby reducing function call overhead.

**[0056]** Application of C++ concepts as described herein permits the present invention to determine what template specializations are needed at runtime, by inlining code as it required during compilation. This avoids a further problem associated with conditional if statements for which processing time must be devoted to perform each instance of. Employing C++ concepts to avoid conditional if statements is called template meta-programming.

**[0057]** Another approach seeks to implement an arrangement of properly aligned buffers to place bytes that are known to be necessary for compilation in a contiguous fashion, so that the present invention pulls significantly fewer pages of data from memory to blocks in the cache. When the necessary bytes are instead placed in disparate locations across the memory landscape in a computer's architecture, more cache misses result. Application of C++ concepts permits the implementation of contiguous buffers to overcome cache miss or cache freeze by preventing the modification of modification when it is not needed.

**[0058]** In one specific embodiment to this approach, the present invention takes advantage of areas in computer architecture where increases in speed may be found. For example, the present invention is able to make more efficient use of the cache by targeting aspects of computer architecture where there is such speed, such as by first operating in the smaller and faster L1 cache in the hierarchy of caches, rather other areas such as the L2 or L3 caches. By doing so, for example, on some processors, computation on words in the cache can be 50x faster than an approach that involves pulling data from RAM to the cache for most repeated accesses and manipulations to the same data.

**[0059]** The third approach seeks to define and enforce type constraints on programming interfaces that access and manipulate machine learning data sets, in one or more specific C++ concepts that address specific kinds of datasets. A C++ concept is generally a mechanism for defining and enforcing constraints on types. Examples of these constraints include requiring specifically-named methods to be defined, sometimes with a specific return type and argument types, which is also known as a "method signature". Another constraint involves overloading of named methods (e.g., const and non-const methods on a class). Still another constraint requires specifically-named typedefs to be defined; these typedefs are useful so implementations of algorithms can make use.

**[0060]** Dataset C++ concepts as applied in the present invention seek to minimize inheritance by utilizing primarily a static, rather than dynamic, polymorphism approach. C++ concepts impose class templates and function templates to impart restrictions on the types that they take, so that any class can be supplied as a template parameter so long as it supports all of the operations that users of actual instantiations upon that type use. In the case of the function, the requirement an argument must meet is clear, but in the case of a template interface an object must meet is implicit in the implementation of that template. Concepts therefore provide a mechanism for codifying the programming interface that a template parameter must meet.

**[0061]** The type constraints in a C++ concept are checked when a class that claims to adhere to the concept is needed by the C++ compiler at the time of compilation. The failure to satisfy any constraints will result in the failure of the program's compilation. The present invention therefore employs, in at least several embodiments, a specific C++

concept referred to further herein as an `MLDataSet` to define a mechanism to enforce constraints on programming interfaces that access and manipulate machine learning data sets.

**[0062]** The `MLDataSet` C++ concept represents either a labeled or unlabeled machine learning data set. As indicated above, this data set has  $m$  instances, which are  $(x,y)$  pairs where  $x$  is a vector of  $n$  data values (called features, observations, or collectively a feature vector) and  $y$  is the label for  $x$ . If  $x$  is unlabeled, it is undefined, and the label  $y$  is stored using a special sentinel value. Instances are indexed by an instance index, and features are indexed by a feature index. The  $j$ 'th feature of the  $i$ 'th instance represents the value  $x_{ij}$ . In the present invention, using a concept-based representation, as opposed to other methods (e.g., inheritance with dynamic dispatch), provides the key advantage in that the same algorithms can be used on different implementations of an `MLDataSet`, but without the dynamic dispatch overhead.

**[0063]** In the present invention, every class implementing the `MLDataSet` concept must define seven typedefs: `label_type`, which represents the type of the labels; `feature_type`, which represents the type of the features; `single_feature_instance_iterator`, which represents the type for a random access iterator over instances in the data set with a fixed feature index  $k$ ; `single_feature_instance_const_iterator`, which is like `single_feature_instance_iterator` but obeys the const random access iterator concept; `single_instance_feature_iterator`, which represents the type for a random access iterator over instances in the data set with a fixed instance index  $k$ ; `single_instance_feature_const_iterator`, which is like `single_instance_feature_iterator`, but obeys the const iterator concept; and a `sparse_feature_pair`, which is defined as a `std::pair<size_t, feature_type>`, which encodes a feature index and a feature value at that feature index for the purposes of enabling compatibility of sparse data files and providing for the manipulation of sparse data.

**[0064]** The `MLDataSet` must also provide functions for accessing size and problem type information: inline `size_t get_num_instances() const`, which returns the total number of instances or feature vectors in the data set; inline `size_t get_num_features() const`, which returns the total number of features in a feature vector; inline `size_t get_num_classes() const`, which returns the total number of classes (equal to 1 for regression); and inline `LabelType get_labeling() const`, which returns the machine learning problem type using enumerated types (classification or regression).

**[0065]** The `MLDataSet` concept must also provide functions for accessing individual feature values and labels: inline `const label_type &get_label(size_t instance_index) const`, which returns the label of a specific feature index; and inline `const feature_type &operator() (size_t instance_index, size_t feature_index) const`. Another function, the `get_num_classes()` function returns the number of classes for classification and 1 for regression; it may be used for other purposes for other problem domains. The `get_labeling()` function returns either real, integer, or nominal.

**[0066]** The `MLDataSet` concept must also provide functions for generating iterators to feature values over a fixed feature index or a fixed instance index. The function inline `single_feature_instance_iterator single_feature_instance_begin(size_t feature_index) const` returns a const random access iterator that points to the feature value of the first instance index and inline `single_feature_instance_const_iterator single_feature_instance_end(size_t feature_index)`

`const` returns a const iterator pointing to the feature value of one element past the last instance index and a fixed feature index.

**[0067]** The `MLDataSet` concept also must provide functions for generating iterators to feature values over a fixed instance index. The inline `single_instance_feature_const_iterator single_instance_feature_begin(size_t instance_index) const` returns a const random access iterator that points to the feature value of the first feature index and a fixed instance index. The inline `single_instance_feature_const_iterator single_instance_feature_end(size_t instance_index) const` returns a const iterator pointing to the feature value of one element past the last feature index and a fixed instance index.

**[0068]** FIG. 1, FIG. 2 and FIG. 3 are exemplary flow diagrams representing procedures for inducing a forest, a tree, and a node, respectively, for instantiating a class implementing a `MLDataSet` concept in the process of learning a decision forest for homogeneous datasets. Referring to FIG. 1, inducing a forest for instantiating a class implementing a `MLDataSet` concept for homogeneous datasets begins by calling an `induce_forest` routine 100. In step 102 a plurality of input variables are specified for the routine 100, such as  $D$ ,  $m$ try,  $max\_depth$ ,  $node\_size$ ,  $criterion$ ,  $bootstrapped$ , and  $sampling\_proportion$ . Box 102 also provides that the routine will return the induced forest and the  $oob\_error$  value, which is the error of the decision forest when each tree is applied exclusively to the instances that were not present in the subsample used to induce the tree.

**[0069]** In step 104, it is noted that  $D$  is an input object of a class `XDataSet` for implementing the `MLDataSet` concept. In step 106, predictions are defined as an array of size  $D.get\_num\_instances()$ , and in step 108, labels are defined as an array of type `typeof(D)::label_type` with values obtained by iterating over labels.

**[0070]** The routine 100 progresses by initial determining whether to compile for a classification with types in step 110. If yes, the routine 100 proceeds with step 112 and defines ballots as an array of size  $D.get\_num\_instances()$  where each element is an array of size  $max\_label+1$  of integers initialized to 0. The routine 100 then proceeds with step 134, and also to step 116 by defining  $n$  as an array of size  $D.get\_num\_instances()$ . If the routine 100 initially determines not to compile for a classification with types in step 110, the routine 100 skips ahead to step 114, where a variable  $means$  is defined as an array of size  $D.get\_num\_instances()$ , and from there to step 116 where a variable  $n$  is defined as an array of size  $D.get\_num\_instances()$  as above. Following step 116, the routine 100 proceeds to step 134.

**[0071]** The routine 100 performs Step 118 is performed once the condition in step 138 fails. At this point again determines whether to compile for a classification with types in step 118. If not, the routine skips to step 124 and returns an  $oob\_error$  (out-of-bag mean squared error) value defined as the value returned by `mse(labels, means)`, and the routine 100 terminates at step 126. If at step 118 a compile time type is determined to be for classification, then the routine 100 defines predictions by the value `convert_classification_ballots_to_labels(ballots)` at box 120, which returns a label with the highest vote for each ballot. An  $oob\_error$  value is then defined as the value returned by calling `classification_error(labels, predictions)`, which returns the classification error given the labels and predictions at step 122, and the routine 100 terminates at step 126.

[0072] As noted above, the routine 100 performs steps 128-150 after defining ballots as an array of size `D.get_num_instances()` where each element is an array of size `max_label+1` of integers initialized to 0 in step 112. In box 134 the `tree_index` variable is set to 0, and in step 136 a forest is defined as an array of NULLs of a size defined by the number of trees. The `tree_index` must be less than the number of trees in step 138, else the routine 100 proceeds to step 118 as above. If the `tree_index` value is less than the number of trees in step 138, then the routine 100 proceeds with creating a variable `D_inbag` as in step 140. The `D_inbag` variable is created with type `IndexedDataSet<XDataSet>` and is a realization of a subsample drawn with (or without optionally without) replacement of instances of the value `D` according to the sampling proportion specified as input to routine 100.

[0073] The routine 100 proceeds in step 142 with making the variable `D_outbag` to be of the type `IndexedDataSet<XDataSet>` and to contain those instances of `D` not in `D_inbag`, ie those instances not in the subsample used to induce the tree. In step 144, a tree is by the value `learn_tree(D_inbag, mtry, max_depth, node_size, criterion)`, and in step 146 a tree is further equated to a forest[`tree_index`]. In step 148, `raw_predictions` are equated by the value `predict_tree_on_dataset(D_outbag, mtry)`, and in step 128, the routine 100 again determines whether to compile with types for a classification. If yes, then in step 130 the routine 100 calls `accumulate_classification_votes(raw_predictions, D_outbag, ballots)`, which casts a vote for each instance in `D_outbag` in the ballot with the same index as the parent instance index in the array of ballots. The routine 100 then sets `tree_index` to be `tree_index+1` in step 150 and returns to step 138. If in step 128 is not a classification at compile-time, then the routine 100 calls `accumulate_regression_votes(raw_predictions, D_outbag, means, n)`, which updates the means for a regression in step 132, and proceeds to step 150.

[0074] FIG. 2 is an exemplary diagram for a procedure for inducing a tree for instantiating a class implementing a `MLDataSet` concept in the process of learning a decision forest for homogeneous datasets. In step 202 a plurality of variables are specified for a routine 200, such as `D`, `mtry`, `max_depth`, `node_size`, and `criterion`. The routine 200 is also provided with a return root instruction. In step 204, a root is defined as a new tree node, a pointer `T` is made to this variable root, a `depth` is equated to zero, and a `cache` is defined as `FeatureCache()`. In step 206, `D` is defined as an `XDataSet`, which implements the `MLDataSet` concept (or `DataSet` concept for short). `Dpart` is defined as a `RangedDataSet<IndexedDataSet<MLDataSet>>`.

[0075] In step 208, `task_queue.enqueue((Dpart, 0))` is specified, and in step 210, the routine 200 determines if the queue is empty by calling `task_queue.empty()`. If yes, then the routine 200 is terminated at step 211. If no, the routine 200 proceeds to step 212 and equates `(Dsplit, depth)` with `task_queue.dequeue()`. `Dsplit` represents the partition of the subsample `Dpart` used to induce a decision tree node. At box 214, the routine 200 checks `Dsplit.get_num_instances() < node_size`, and if so, continues to step 218, and if not, continues to step 216. In box 216, routine 200 then determines whether a `depth` is greater than the value for `max_depth`, and if true, continues to step 218. In step 218 the value for `T` is made a leaf node by setting its left and right to NULL. `T`'s label field is also set to the median of `Dsplit`'s labels for classification, and to the mean of `Dsplit`'s labels for regression. Following step 218, routine 200 proceeds to step 210.

[0076] If the `depth` is not greater than the `max_depth` value in box 216, then `best_feature_index` is undefined in box 220, `best_score` is equated to zero in step 222, `best_test` is undefined as in step 224, and `k` is equated to zero in step 226. In step 228, the routine 200 looks to whether `k` is less than `mtry`, and if it is, then the routine 200 lets `f` be a random integer selected, or drawn, uniformly at random between 0 and `Dsplit.get_num_variables()` (exclusive) in step 240. This integer is used to represent the index of the next feature to try inducing a split on `Dpart`.

[0077] The routine 200 proceeds by defining `p,s` as `find_best_test(Dsplit, f, criterion, cache)` in step 242. In step 244, the routine 200 asks whether `best_score` is less than the value `s`, and if it is, then `best_score` is equated to the value `s` in step 246, `best_feature_index` is equated to a value `f` in step 248, `best_test` is equated to a value `p` in step 250, and `k` is equated to `k+1` in step 252. The routine 200 then loops to step 228 to determine whether `k` is less than the value for `mtry`. If `best_score` is not less than `s`, the routine 200 proceeds directly to step 252.

[0078] If `k` is not less than `mtry`, then routine 200 proceeds to step 230 and determines whether `best_score` is greater than 0. If not, then the routine 200 invokes step 218 as noted above, and proceeds to step 210.

[0079] If `best_score` is greater than zero, the routine 200 makes `T` non-leaf by setting left to new node `TL`, and right to new node `TR` in step 232. The routine also stores learned parameters `best_feature_index`, `best_test` as `feature_index` and threshold in `T`. The routine 200 then defines `DL`, `DR` by partitioning `Dsplit` in step 234, by reordering indices of its parent `D` within the range defined by `Dsplit` such that all instances failing the test in the instance range come before those passing it. The routine 200 in this step then generates two `RangedDataSet<IndexedDataSets>` objects `DL` and `DR`, so that `DL` contains all instances that fail `best_test` and `DR` contains all instances that pass `best_test`.

[0080] The routine 200 then proceeds in step 236 by calling `enqueue(DL, depth+1)` to enqueue onto the task queue (`DL, depth+1`) so that a node can later be induced on those instances in `Dsplit` failing the test, represented by `DL`. Then, routine 200 then proceeds to step 248, where the same is done for the instances in `Dsplit` passing the test (represented by `DR`) by calling `task_queue.enqueue((DR, depth+1))`. The routine 200 then returns to step 210.

[0081] FIG. 3 is an exemplary diagram for a procedure for inducing a tree for instantiating a class implementing a `MLDataSet` concept in the process of learning a decision forest for homogeneous datasets. In step 302 a plurality of variables are specified for a routine 300, such as `D` (an object of a class implementing the `MLDataSet` concept), `feature_index` (represents the index of the feature to learn the best test/threshold), `criterion` (the objective function to optimize) and `feature_cache` (a contiguous buffer to store (feature, label) pairs). Additionally, the routine 300 is specified to return values for `p` and `score`.

[0082] In step 304, the routine 300 assumes a value for `D` to be a `RangedDataSet<IndexedDataSet<XDataSet>>` where `XDataSet` implements the `MLDataSet` concept (or `DataSet` concept for short). The routine 300 further assumes that `criterion` in step 302 have the type `CriterionType`, where `CMT` is the type `CriterionType::count_map_type`.

[0083] In step 306, the routine 300 determines whether a feature in `D` has integral features and the labels are for classification. If not, then in step 310 the routine 300 copies a

feature with feature\_index and labels into the cache, and in step 314 applies IntroSort on the (feature, label) pairs in feature\_cache to sort then lexicographically.

**[0084]** If a feature in D has a feature index of feature\_index integral and classification in step 306, the routine 300 proceeds to step 308 where it determines whether  $(\max\_feature - \min\_feature) * (\max\_label - \min\_label)$  is less than  $2.0 * D.get\_num\_instances() * \log(D.get\_num\_instances())$ . If so, then in step 312 the routine 300 applies a counting or radix sort to sort (feature\_value, label) pairs lexicographically. The routine 300 uses the same buffers for every instance of the sort, widening them as needed.

**[0085]** The routine 300 sorts indices in the IndexedDataSet strictly within boundaries defined by the RangedDataSet<IndexedDataSet<XDataSet>>D so that the feature values for feature with feature index feature\_index are sorted, as noted in box 313. The routine returns p and score values by first defining fit to be the value of feature\_cache.begin() in step 316. In step 318, best\_score is set to 0, and in step 320 best\_threshold is set to -inf. The value for prev\_feature is defined as fit->feature in step 322, and in box 324 and box 326 respectively, left\_map and right\_map are equated to new CMT objects where the right\_map is initialized so that the statistics it maintains are initialized so that all instances are initially assumed to pass the test that is being learned. In box 328, prev\_label is defined as fit->label.

**[0086]** The routine 300 then proceeds to determine, in box 330, whether fit is not equal to feature\_cache.end(). If yes, then in box 332, p is set as (LessThan, best\_threshold), and score is defined as best\_score.

**[0087]** If not, then the routine 300 proceeds by calling left\_map.add\_counts(fit->get\_label()) in box 334, which updates some variables in left\_map needed by criterion.get\_score() so that it can compute a score assuming the instance pointed to by fit now fails the test, and then calls right\_map.remove\_counts(fit->get\_label()) in box 335, which updates variables in right\_map needed by criterion.get\_score() so the instance pointed to by fit is no longer is considered to pass the test. Next, the routine 300 determines whether prev\_label is not equated to fit->label and whether prev\_feature is not equated to fit->feature in step 336. If no, the routine 300 proceeds to step 350 as noted below. If yes, then in box 338, trial\_score is defined as criterion.get\_score(count\_map, left\_map, right\_map), and step 340, trial\_score is set to be greater than best\_score. In the next step, in box 342, best\_threshold is defined as  $(\text{prev\_feature} + \text{cur\_feature}) / 2$  and best\_score is equated to score in box 344. A value for prev\_feature is fit->feature in box 346, and prev\_label is fit->label in box 348, and the iterator fit points to the next instance in the feature\_cache by setting it to fit+1 in box 350. The routine 300 then returns a loop to step 330 to proceed with returning p and score values in step 332, or continuing with steps 334-350.

**[0088]** Another dataset C++ concept in the present invention is a WritableMLDataSet concept, which extends the MLDataSet concept and allows the labels and feature values to be changed, and new instances to be added. In this concept, inline void set\_feature\_value(size\_t instance\_index, size\_t feature\_index, const feature\_type &val) sets a specific feature value (indexed by feature\_index) of an instance's feature vector (indexed by instance\_index). Also, the inline void set\_label(size\_t instance\_index, const label\_type &label) sets the label of the instance indexed by the instance\_index passed to the label passes. Also, template <class FeatureIterator>inline void add\_instance\_with\_dense\_fea-

ture\_vector(label\_type label, FeatureIterator fbegin, FeatureIterator fend) adds an instance to the data set. The label of the new instance is passed as the first argument. Begin and end iterators to a dense representation of the feature vector of the new instance are also passed. Further, template <class FeatureIterator>inline void add\_instance\_with\_sparse\_feature\_vector(label\_type label, FeatureIterator fbegin, FeatureIterator fend) adds an instance to the data set with label "label" and features starting from iterator fbegin and ending at iterator fend (the iterators must point to an object of type or reference to an object of type sparse\_feature\_index\_feature\_value. All existing iterators to elements in the DataSet are invalidated when an instance is added, an instance is removed, or the ordering of instances is changed in any way.

**[0089]** The single\_instance\_feature\_iterator single\_instance\_feature\_begin(size\_t instance\_index) is like the function of the same name described in preceding paragraphs but instead returns a non-const random access iterator. The single\_instance\_feature\_iterator single\_instance\_feature\_end(size\_t instance\_index) is like the single\_instance\_feature\_end function just described but returns a non-const random access iterator.

**[0090]** The function inline single\_feature\_instance\_iterator single\_feature\_instance\_begin(size\_t feature\_index) is like the function of the same name described earlier but instead returns a non-const random access iterator. The inline single\_feature\_instance\_iterator single\_feature\_instance\_end(size\_t feature\_index) is like the function of the same name described earlier but instead returns a non-const iterator.

**[0091]** WritableMLDataSet is a separate concept, rather than its constraints and interfaces being included in the MLDataSet concept. This permits separate interfaces that perform read-only manipulations to a data set (MLDataSet) with interfaces that can change feature values, labels, and add instances to a data set (WritableMLDataSet). A read-only contract for MLDataSet ensures that it is safe to let multiple threads use the same copy of the underlying data set (even though the view data sets may be different).

**[0092]** In the MLDataSet concept, a class that implements facilities for caching features implements the FeatureCache concept as follows:

**[0093]** inline void cache\_feature(size\_t feature\_index)—this caches (feature, label) values for a specific feature index over all instances in a contiguous buffer.

**[0094]** inline void cache\_feature(size\_t feature\_index, size\_t begin\_instance\_index, size\_t end\_instance\_index)—this caches (feature, label) values for a specific feature index over a range of instances defined by a beginning instance index and ending instance index.

**[0095]** typedef FeatureCacheIterator—this is a type for a non-const iterator over (feature, label) pairs in the feature cache.

**[0096]** typedef FeatureCacheConstIterator—this is a type for a const iterator over (feature, label) pairs in the feature cache.

**[0097]** typedef FeatureCachePair—this is a type std::pair<label\_type, feature\_type> that represents the underlying value to which objects of the FeatureCacheIterator and FeatureCacheConstIterator classes point.

**[0098]** FIG. 11 is a conceptual flow diagram of Feature-ContiguousDataSet's implementation 1100 of the MLDataSet concept, which is specified in box 1102 as

FeatureContiguousDataSet<FeatureType, Label Type>. This implementation 1100 improves the computational efficiency of learning EDTs on data sets in the present invention by laying out instances within a single feature column in a contiguous fashion. Box 1102 illustrates that FeatureContiguousDataSet has two template arguments, FeatureType and LabelType and has a plurality of member variables num\_features (of type size\_t), num\_classes (of type size\_t), features (an array of arrays of type feature\_type), and labels. Boxes 1134 and 1136 define two typedefs for the class FeatureContiguousDataSet, feature\_type is mapped to type FeatureType and label\_type is mapped to type LabelType. Procedure 1104 illustrates a non-static class method that retrieves individual feature values given an instance index (instance\_index) and a feature index (feature\_index), and returns the feature value in a variable value of type feature\_type. In Step 1106 of Procedure 1104, it assigns value to ptr->features[feature\_index][instance\_index], and terminates in step 1108. Procedure 1110 retrieves an individual label, returning a value of type label\_type. Step 1112 of Procedure 1110 retrieves the label pointed to by instance\_index and stores it in the variable value, and then terminates in step 1114. Procedure 1116 returns the number of instances in the object pointed to by ptr. Step 1118 in Procedure 1116 sets sz to the size of the label array in the object pointed to ptr by calling ptr->labels.size( ) and then terminates in step 1120. Procedure 1122 retrieves the number of classes in the object pointed to by ptr and returns the variable sz. Step 1124 in Procedure 1122 sets sz to ptr->num\_classes and terminates in step 1126. Procedure 1128 returns a variable sz. It sets the sz variable to ptr->num\_features, or the number of features in the data set pointed to by ptr.

Also,

```
[0099] inline FeatureCacheIterator begin()
[0100] inline FeatureCacheIterator end()
[0101] inline FeatureCacheConstIterator begin( ) const
[0102] inline FeatureCacheConstIterator end( ) const
```

return a const or non-const iterator to a pair to the first feature in the feature cache, or a const or non-const iterator to one past the last feature in the feature cache.

[0103] Additionally, a class that implements facilities for caching an instance's feature vector is said to implement the InstanceCache concept:

```
[0104] inline void cache_instance(size_t instance_index)—caches an instance of a particular instance index into a contiguous buffer.
```

```
[0105] typedef InstanceCacheIterator—a type for a non-const iterator over an instance's feature vector.
```

```
[0106] typedef InstanceCacheConstIterator—a type for a const iterator over an instance's feature vector.
```

[0107] FIG. 12 is a conceptual flow diagram of InstanceContiguousDataSet's implementation 1200 of the MLDataSet concept, which is specified in box 1202 as InstanceContiguousDataSet<FeatureType, Label Type>. This implementation 1200 improves the computational efficiency of using learned EDTs for the purpose of prediction on data sets in the present invention by laying out each feature vector within a single instance in a contiguous fashion. Box 1202 illustrates that InstanceContiguousDataSet has two template arguments, FeatureType and LabelType and has a plurality of member variables num\_features (of type size\_t), num\_classes (of type size\_t), features (an array of type array of type feature\_type), and labels (an array of type label\_type).

Boxes 1234 and 1236 define two typedefs for the class InstanceContiguousDataSet, feature\_type is mapped to type FeatureType and label\_type is mapped to type LabelType. Procedure 1204 illustrates a non-static class method that retrieves individual feature values given an instance index (instance\_index) and a feature index (feature\_index), and returns the feature value in a variable value of type feature\_type. In Step 1206 of procedure 1204, it assigns value to ptr->features[instance\_index][feature\_index] where ptr->features[instance\_index] yields a reference to the array that represents the feature vector for instance index, and ptr->features[instance\_index][feature\_index] represents the feature with index feature\_index in that feature vector. Following this, procedure 1204 terminates in step 1208. Procedure 1210 retrieves an individual label, returning a value of type label\_type. Step 1212 of procedure 1210 retrieves the label pointed to by instance\_index and stores it in the variable value (value:=ptr->labels[instance\_index]), and then terminates in step 1214. Procedure 1216 returns the number of instances in the object pointed to by ptr. Step 1218 in procedure 1216 sets sz to the size of the label array in the object pointed to (sz:=ptr->labels.size( ), and then terminates in step 1220. Procedure 1222 retrieves the number of classes in the object pointed to by ptr and returns the variable sz. Step 1224 in procedure 1222 sets sz to ptr->num\_classes and terminates in step 1226. Procedure 1228 returns a variable sz representing the number of features in the data set. Step 1230 sets the sz variable to ptr->num\_features, and terminates in Step 1232.

```
[0108] The class is defined by size_t num_features, size_t num_classes, std::vector<std::vector<FeatureType>>features, and std::vector<LabelType>labels.
```

Also,

```
[0109] inline InstanceCacheIterator instance_cache_begin()
[0110] inline InstanceCacheIterator instance_cache_end()
[0111] inline InstanceCacheConstIterator instance_cache_begin( ) const
[0112] inline InstanceCacheConstIterator instance_cache_end( ) const
```

returns const and non-const iterators to the first feature of the instance cache (first feature in the feature vector) and one past the last feature in the instance cache (one past the last feature in the feature cache).

[0113] The MLDataSet concept is implemented in several classes in which machine learning data is stored in memory. Each uses a memory layout that is optimized for a different workload or purpose. One such class—In Memory Feature-Contiguous Data Set—stores a machine learning data set so that all instances for a specific feature index are stored contiguously. This layout is useful for learning trees because each feature is considered independently during tree induction. Another class, In-Memory Instance-Contiguous Data Set, stores a machine learning data set so that all features for a specific instance index are stored contiguously. This layout is useful for evaluating a tree on a single instance.

[0114] Another class, In Memory Sparse Feature-Contiguous Data Set, stores a machine learning data set in a sparse representation so that the sparse array is first indexed by feature index in an associative array (potentially non-sparse), and then as a list of (instance\_index, feature\_value) pairs. Thus, feature values for a specific feature index are contigu-

ously arranged in memory. Still further, another class, In Memory Sparse Instance-Contiguous Data Set, stores a machine learning data set in a sparse representation so that the sparse array is first indexed by instance index in an associative array (potentially non-sparse), and then as a list of (feature\_index, feature\_value) pairs. Thus, feature values for a specific feature index are contiguously arranged in memory.

**[0115]** As noted above, at least one aspect of the present invention operates by de-correlating decision trees by looking at different subsets of data. The following classes implement the MLDataSet concept and represent subsets of either features or instances by referring to a parent data set. These subsets may be compounded (for example, a ranged subset of an indexed subset of a data set). One such class, Instance-indexed Subset Data Set, controls the view to another data set (called the parent data set) that implements the MLDataSet concept, so that it represents a subset of instances by storing an array of instance indices. This array may contain duplicate instance index values. This class implements a feature-contiguous cache so that feature values for a specific feature index can be cached. This is especially useful when many instances over a single feature index are needed repeatedly, such as when learning a node in a decision tree. It also implements an instance-contiguous cache so that the feature vector for a specific instance index can be cached. This is useful for prediction when all or some of the features for a specific instance are needed to traverse a decision tree or an ensemble of decision trees for the purpose of prediction, computing out-of-bag error, or some measure of feature importance.

**[0116]** In another view, an Instance-indexed Ranged Subset Data Set is a class that controls the view to another data set (called the parent data set) that implements the MLDataSet concept so that it represents a subset of instances by storing a minimum and maximum instance index to its parent. The minimum instance index and maximum instance index are equal if and only if the subset represented is the null set. This class implements a feature-contiguous cache so that feature values for a specific feature index can be cached, or refer to a range over its parent's feature-contiguous cache if it is available. It also implements an instance-contiguous cache so that the feature vector for a specific instance index can be cached, or refer to a range over its parent's instance cache if it is available.

**[0117]** A Feature-indexed Subset Data Set is similar to the Instance-indexed Subset Data Set, but represents a subset of features by storing an array of feature indices instead of an array of instance indices. The cache requirements are the same as the Instance-indexed Ranged Subset Data Set. A Feature-indexed Ranged Subset Data Set is similar to the Instance-indexed Ranged Subset Data Set, but represents a subset of features by storing the minimum feature index and maximum feature index instead of minimum instance index and maximum instance index. The cache requirements are the same as the Instance-indexed Ranged Subset Data Set.

**[0118]** A view implementation of an MLDataSet may also implement a InstanceIndexReconstructable concept so that the instance index in the parent corresponding to an instance index in the view can be reconstructed by calling the function inline size\_t get\_parent\_index(size\_t instance\_index) const.

**[0119]** Other classes implement the MLDataSet concept so that machine learning data sets represented as data structures from other languages can be passed to the present invention and used accordingly. These include C-Contiguous NumPy Array Data Set, Discontiguous NumPy Array Data Set,

Python Sequence Protocol Data Set, Python SciPy Sparse Data Sets (a separate implementation exists for each of the CSC, CSR, BSR, LIL, DOK, COO, and DIA formats supported in SciPy), Python Buffer Protocol Data Set, Java C-Contiguous Primitive Array Data Set, Java Discontiguous Strided Primitive Array, Ruby Primitive Array Data Set, R C-Contiguous Primitive Array Data Set, R Discontiguous Strided Primitive Array Data Set, and Matlab Array Data Set.

**[0120]** The MLDataSet concept may be configured to allow for one implementation of a concept to be a view on another class that implements that same concept. In the following example, a CSV file is loaded into an in-memory, feature-contiguous object of a class that implements the MLDataSet concept. The features are of type float and the labels are of type int. An indexed data set is then created over the in-memory data set, and instances are selected with replacement uniformly at random:

```
[0121] typedef FeatureContiguousMemoryDataSet <float, int> mem_dataset;
```

```
[0122] FeatureContiguousMemoryDataSet dataset;
```

```
[0123] Dataset.load_from_csv("file.csv");
```

```
[0124] IndexedDataSet<mem_dataset>indexed_dataset(dataset);
```

```
[0125] const size_t n(dataset.get_num_instances());
```

```
[0126] indexed_dataset.sample_indices_iid_uniformly_at_random(n);
```

**[0127]** FIG. 13 is a conceptual flow diagram illustrating this class's implementation 1300 for indexing within the MLDataSet concept, which is specified in box 1302 as IndexedDataSet<ParentDataSet>. This implementation 1300 enables memory-efficient subsampling, and computation reductions using the inlining steps of FIG. 13. Box 1302 illustrates that IndexedDataSet has a single template argument ParentDataSet where ParentDataSet is some other class implementing the MLDataSet concept, and two member variables: parent, which is a pointer to a ParentDataSet object containing the instances of the subsample that IndexedDataSet<ParentDataSet> represents and indices, which is an array instance indices that point to instances in parent. IndexedDataSet<ParentDataSet> has two typedefs, feature\_type is mapped to the type ParentDataSet::feature\_type, and label\_type is mapped to the type ParentDataSet::label\_type as illustrated in boxes 1334 and 1336.

**[0128]** Procedure 1304 illustrates a non-static class method that retrieves individual feature values given an instance index (instance\_index) and a feature index (feature\_index), and returns the feature value in a variable value of type feature\_type. In Step 1306, the feature value corresponding to instance with instance index ptr->indices[instance\_index] and feature index feature\_index is assigned to val. It assigns value to ptr->parent->get\_feature\_value(ptr->indices[instance\_index], feature\_index). Following this, Procedure 1304 terminates in step 1308. Procedure 1310 retrieves an individual label, returning a value of type label\_type. Step 1312 of procedure 1310 retrieves the label in the parent indexed by instance\_index from the parent and by setting value to ptr->parent->get\_label(ptr->indices[instance\_index]), and then terminates in step 1314. Procedure 1316 returns the number of instances in the object pointed to by ptr, which is effectively the size of the subsample represented by the object ptr on ptr->parent. Step 1318 in procedure 1316 sets sz to the value returned by ptr->indices.size() and then terminates in step 1320. Procedure 1322 returns the number of classes in the variable sz. Step 1324 in procedure 1322 sets

sz to ptr->parent->get\_num\_classes() and terminates in step 1326. Procedure 1328 returns a variable sz representing the number of features in the data set. Step 1330 in procedure 1328 sets the sz variable to ptr->parent->get\_num\_features() and terminates in step 1332.

[0129] With the IndexDataSet, a subset is formed over any data set with only the overhead to store the instance indices themselves, which is negligible for high-dimensional data. A RangeDataSet is then composed over an IndexDataSet to select only a contiguous range of it. The RangeDataSet has O(1) memory complexity. For example, the first 5 instances only may be selected from the parent data set:

---

```
RangeDataSet<IndexDataSet<mem_dataset>> >
  range_dataset(indexed_dataset, 0, 5);
```

---

The composition of RangeDataSet<IndexDataSet<Feature-ContiguousMemoryDataSet>> is useful when learning each decision tree in an EDT because before an algorithm proceeds to build subtrees of a decision tree node, it partitions (using a Test) the instances so that the left tree is trained with one partition, and the right tree is trained with the other partition.

[0130] A PartitionIndexDataSet also has O(1) memory overhead, and it can be used to represent partitions of a parent data set and their complements. For example, to create the 3rd partition of a data set partitioned into ten partitions:

```
[0131] indexed_dataset.select_all_from_parent();
[0132] indexed_dataset.random_shuffle(rng);
[0133] PartitionDataSet<IndexDataSet<mem_
  dataset>>third_partition(indexed_dataset, 3, 10);
```

Similarly, a view may be created that gives us the complement of the third partition where the universe of discourse is the parent data set (i.e. indexed\_dataset):

```
[0134] PartitionDataSet<IndexDataSet<mem_dataset>,
  Complement>third_partition_prime(indexed_dataset,
  3, 10);
```

[0135] An example where the above would be useful is performing cross-validation in a manner that avoids copying the data when applying a training procedure to each fold.

[0136] FIG. 14 is a conceptual flow diagram illustrating this class's implementation 1400 for partitioning within the MLDataSet concept, which is specified in box 1402 as RangedDataSet<ParentDataSet>. This implementation 1400 enables memory-efficient subsampling of instances defined by a starting index and an ending index. Moreover, function call overhead is reduced by inlining the steps of FIG. 14. Box 1402 illustrates that RangedDataSet has a single template argument ParentDataSet where ParentDataSet is some other class implementing the MLDataSet concept, and a plurality of member variables: parent, which is a pointer to a ParentDataSet object containing the instances of the subsample that IndexedDataSet<ParentDataSet> represents; starting\_instance\_index, which is an index of the instance in the parent that represents the first instance in the subsample defined by the IndexedDataSet<ParentDataSet>; and ending\_instance\_index, which is an index of the instance in the parent that represents the last instance in the subsample defined by the IndexedDataSet<ParentDataSet>.

IndexedDataSet<ParentDataSet> has two typedefs, feature\_type is mapped to the type ParentDataSet::feature\_type, and label\_type is mapped to the type ParentDataSet::label\_type as illustrated in boxes 1434 and 1436.

[0137] Procedure 1404 illustrates a non-static class method that retrieves individual feature values given an instance index (instance\_index) and a feature index (feature\_index), and returns the feature value in a variable value of type feature\_type. In Step 1406, the feature value corresponding to instance with instance index ptr->indices[instance\_index] and feature index feature\_index is assigned to val. It assigns value to ptr->parent->get\_feature\_value(ptr->starting\_instance\_index+instance\_index, feature\_index). Following this, procedure 1404 terminates in step 1408. Procedure 1410 retrieves an individual label, returning a value of type label\_type. Step 1412 of procedure 1410 retrieves the label ptr->parent->get\_label(ptr->indices[instance\_index]) indexed and stores it in value, and then terminates in step 1414. Procedure 1416 returns the number of instances in the object pointed to by ptr, which is effectively the size of the subsample represented by the object ptr on ptr->parent. Step 1418 in procedure 1416 sets sz to the value returned by ptr->indices.size(), and then terminates in step 1420. Procedure 1422 returns the number of classes in the variable sz. Step 1424 in procedure 1422 sets sz to ptr->parent->get\_num\_classes() and terminates in step 1426. Procedure 1428 returns a variable sz representing the number of features in the data set. Step 1430 in procedure 1428 sets the sz variable to ptr->parent->get\_num\_features(), and terminates in step 1432.

[0138] The DecisionTreeNode concept specifies type constraints and interfaces that all implementing classes must obey. Objects of classes implementing the concept represent decision trees. Some typedefs that must be defined include:

```
[0139] node_handle—type to refer to child decision tree
  nodes.
[0140] label_type—type of the labels in a decision tree's
  leaf nodes, and it also represents the type of a prediction
  value when the decision tree is applied to a feature_
  vector.
[0141] test_parameter_type—type of the encoding for
  the Node Test's parameterization
[0142] threshold_type—type of thresholds used for
  threshold tests.
[0143] uncertainty_parameterization_type—a type used
  to parameterize the uncertainty computation when pre-
  dicting.
[0144] feature_indices_const_iterator—a type of ran-
  dom access iterator used to traverse feature indices when
  a Node Test involves multiple features.
```

Implementors of the DecisionTreeNode must also implement the following functions:

```
[0145] inline handle_type get_left() const—returns a
  handle to the Node's Left Child
[0146] inline handle_type get_right() const—returns a
  handle to the Node's Right Child
[0147] inline int get_feature_index() const—returns the
  feature index used to apply a test
[0148] inline threshold_type get_threshold() const—re-
  turns the threshold used for threshold tests
[0149] inline uncertainty_type &get_uncertainty_pa-
  rameters()—returns the uncertainty parameters used for
  prediction.
[0150] inline const uncertainty_type &get_uncertainty_
  parameters() const—returns the uncertainty parameters
  used for prediction as a const reference.
```

[0151] inline feature\_indices\_const\_iterator fi\_begin( )  
const—returns a const random access iterator to the first feature index used in the test for the decision tree node

[0152] inline feature\_indices\_const\_iterator fi\_end( )  
const—returns a const random access iterator to one past the last feature index used in the test for the decision tree node.

[0153] template <class Iterator>inline label\_type predict\_on\_features\_iterator(Iterator features\_begin, Iterator features\_end) const—predicts on a feature vector defined by a random access iterator

[0154] template <class DataSet>inline label\_type predict\_on\_instance\_in\_data\_set(size\_t instance\_index, const DataSet &data\_set)—predicts on a object from a class that implements the MLDataSet concept.

[0155] In decision tree learning, a Classification and Regression Tree (CART) algorithm is often used learn a decision tree from a training set. It is a recursive algorithm that starts from the top of the decision tree, and keeps building the tree downward until stopping criteria are met (e.g. the data set has too few instances, the labels in the data set are homogeneous, or no significant increase in the score is achieved). The steps are outlined as follows (start with a tree node T:=root, D:=data set):

[0156] 1. best\_feature\_index:=undefined; best\_score:=0; best\_test:=undefined;

[0157] 2. k:=0

[0158] 3. if D has exactly the same label for every instance or there are fewer than node\_size instances in D, go to step 6

[0159] 4. If k<mtry

[0160] a. let f<D.get\_num\_instances( ) be an integer drawn uniformly at random

[0161] b. find high scoring test (e.g. threshold) p using node learning procedure and some scoring criteria C, restricting consideration to just feature index f. Let this score be s.

[0162] c. if best\_score < s then

[0163] i. best\_score:=s

[0164] ii. best\_feature\_index:=f

[0165] iii. best\_test:=p

[0166] d. k:=k+1

[0167] e. goto 4

[0168] 5. If the node learning procedure led to no acceptable increase in score (or decrease in loss, depending on the criteria), goto step 6.

[0169] 6. Let T be a leaf node. Store the label using a summary statistic computed on D (e.g. the median or mean label). Compute the uncertainty parameters if desired. Restore the caller's state and return to it.

[0170] 7. Otherwise, store the feature index, threshold, and other learned parameters in T.

[0171] 8. Partition D into two data sets, with DL representing those instances for which the threshold test passes and DR representing those instances for which the threshold test fails. Create empty trees TL and TR, and attach them as the left node and right node of T, respectively.

[0172] 9. Recurse to step 1 with D:=DL and T:=TL

[0173] 10. Recurse to step 1 with D:=DR and T:=TR

[0174] 11. Restore the caller's state, and return to it.

[0175] There are a large number of ways to implement the steps above. Most approaches use a recursive function, but this incurs function call overhead. A priority queue of task

objects is employed, where each task object represents the state (D, T). Prior to jumping to step 1, the (DL, TL) and (DR, TR) are enqueued. At step 1, if the queue is empty, we stop and return the tree. If it is nonempty, we simply dequeue the next task object and let it be (D, T). Further, the Instance-indexed Data Set class is used to represent the subset of instances used for training the overall decision tree for the forest. The Instance-indexed Range Data Set, which encodes the partition over the instance indices. Together, this prevents a copy of the entire data set before recursing at step 9 or 10.

[0176] A RNG Pool concept is a pool of random number generators where the pool provides a single, dedicated random number generator for each thread of execution.

[0177] A class that implements the DecisionTreeLearner concept must be parameterized by a class that implements an RNG Pool, a class that implements a ReadableDataSet, and a class that implements a Splitting Criteria. It must implement the member function template <class label\_type>inline Node<label\_type>learn(RNGPool &rng\_pool, const DataSet &in, const Criteria &criteria, int mtry, int max\_depth, int node\_size).

[0178] The CountMap concept is used in learning decision tree nodes. It records statistics such as the number of instances for each label value it has encountered (for classification), the number of times each category in a categorical feature for each label (for categorical features), and the mean/variance label (for regression). The following functions must be implemented for this concept:

[0179] template <class FeatureCache>inline void add\_count(const FeatureCache &cache)—adds all instance's feature values in a feature cache to the statistics recorded by the CountMap

[0180] template <class LabelType>inline void add\_count(LabelType label)—updates the statistics curated by the count map by including a single repetition of the label.

[0181] template <class LabelType>inline void add\_count(LabelType label, int num)—updates the statistics curated by the count map by including a specified multiple of repetitions of a label.

[0182] template <class LabelType>inline void remove\_count(LabelType label)—updates the statistics curated by the count map by excluding a single repetition of the label.

[0183] template <class LabelType>inline void remove\_count(LabelType label, int num)—updates the statistics curated by the count map by excluding a specified multiple of repetitions of a label.

[0184] There is a different CountMap implementations depending on the type of learning.

[0185] regression and (real-valued or integer features)

[0186] regression and categorical features

[0187] classification and (real-valued or integer features)

[0188] classification and (categorical features)

[0189] Static dispatch is used to instantiate the appropriate tree node learning implementation and count map.

[0190] The Splitting Criteria concept represents the criteria used to choose the best test among all possible choices or some approximation thereof. It must implement several functions:

[0191] template <class CountMap>inline double get\_score(const CountMap &left, const CountMap &right)—returns a score such that no progress is made if and only if the score returned is 0. Suppose c is an object

from a class C that implements CountMap. Given a threshold test X defined by the CountMap objects leftX and rightX, and a threshold test Y defined by the CountMap object leftY and rightY. The threshold test X is said to be more optimal than threshold test Y if and only if `criteria.get_score(leftX, rightX) > criteria.get_score(leftY, rightY)`.

**[0192]** `template <class CountMap>inline double get_overall_impurity(const CountMap &all)`—returns the overall impurity given the summary statistics recorded in the object ‘all’ of a class implementing the CountMap concept.

**[0193]** `template <class CountMap>inline double get_impurity(const CountMap &left, const CountMap &right)`—returns the impurity given the summary statistics for those instances that passed the test (i.e. the left input) and the summary statistics of those instances that failed the test (i.e. the right input).

**[0194]** `inline double get_improvement(double overall_impurity, double test_impurity)`—returns a statistic representing the improvement in impurity induced by the test.

**[0195]** The FlexiAlg concept is used to analyze the characteristics of an InstanceCache or FeatureCache and dispatches a sort or partitioning algorithm that is deemed to be the most efficient given those characteristics. It uses a mixture of partial template specialization on the type of the labels and the type of the features as well as analyzing the density or histogram of the features given different values of labels in its heuristic.

**[0196]** The TreeNodeLearner concept represents an object that learns a Decision Tree. A subset of its template parameters encode types that implement a Data Set concept, a Splitting Criteria concept, a Count Map concept, and a TreeNode concept. It implements a single function: `inline LearnTreeResult learn_tree_node(DataSet &data, const SplittingCriteria &criteria, const CountMap &all, int feature_index)`. FlexiAlg is used to rearrange or sort the instance cache using the most efficient sort given both the static and dynamic characteristics of the data set. It also must define a child class LearnTreeResult that stores the parameters for the best performing test, the best loss or score, the average loss or score, and the worse loss or score.

**[0197]** There are many different kinds of data that may be trained using an Ensemble of Decision Trees (EDT). For example, an insurance claim may contain the person’s date of birth (date), the number of days they’ve been a customer (integer), and their sex (Male, Female). Each column may consist of real numbers, integers, Booleans, or categories. Though conceptually simple, in practice, supporting such heterogeneity is very difficult. The vast majority of existing implementations of an EDT support training only on numeric data. This limits their applicability for many real world data sets. In another aspect, the present invention also provides mechanisms that enable training an EDT on such data sets without a significant performance penalty.

**[0198]** For ease of explanation, consider the term Variable Characteristic Type (VCT) to refer to the semantic type of the feature values (e.g. dates, real numbers, Booleans, integers, and categories), not the data type (e.g. `int32`, `int64`). For example, a date VCT could be encoded with a 64-bit integer that represents seconds elapsed since Jan. 1, 1970 midnight GMT or a Unicode string of date string in ISO 8601 format. For a real number, one can approximate it with a 32-bit or

64-bit float, but other data types may be used as well. For categories, integers may be used with enough bits to encode every possible category of interest to us (8-bit gives a limit of 256 categories). Some choices of data type offer performance benefits, others allow for the conservation of space. An appropriate choice depends on the use case.

**[0199]** The term Variable Data Type (VDT) refers to the type of data used to store a single feature value. When inducing a decision tree node, a specific training algorithm X must be designed to exploit the domain knowledge and semantics of each VCT but the same generic implementation can be used for multiple VDTs with the same VCT. This applies to other algorithms as well, including caching data into contiguous buffers, statistics computation, and specialized sorting. Variable Group Storage Characteristics (VGSC) define how the VDTs are stored—some variable groups may be sparse, some may be contiguously laid out in memory, others may be dis-contiguous, etc.

**[0200]** AlgRepo is a repository that holds a single implementation of an algorithm intended for one VCT, but holds instantiations for different VDTs. The AlgRepo repository serves the following purposes:

**[0201]** instantiate implementations of algorithms for the same VCT and different VDTs;

**[0202]** assign a unique integral code to each implementation at compile time; and

**[0203]** retrieve an implementation at run-time from its runtime code

**[0204]** The MLDataSet concept above supports only one VCT at a time, and therefore is suitable in the EDT training procedure for problems where all features are numerical or all features are categorical—in other words, where data is homogenous and never mixed, rather than heterogeneous. The present invention therefore includes a heterogeneous data aspect that allows an EDT to be trained on data where the VCT may vary from column to column.

**[0205]** In such an aspect of the present invention, multiple columns with exactly the same VCT and VDT can be grouped together in an object that is an instance of the class VariableGroup. This VariableGroup is represented as a C++ concept, similar to the MLDataSet concept described above. However, no explicit distinction is made between labels and features in such a VariableGroup.

**[0206]** Each column in the VariableGroup represents a single “variable”, and that variable may represent a feature, label, instance weight, or even a cost for incorrect prediction. The context in which a reference to a variable is used indicates its purpose. This allows us much greater flexibility in the kinds of algorithms that can be developed. For example, the column treated as a label may be changed by using a different index to refer to the label. Rather than referring to columns as features as for homogeneous data, for heterogeneous data the columns are generically referred to as “variables” so as to not bias their intended purpose. Every variable within the same VariableGroup has exactly the same VCT.

**[0207]** A Heterogeneous Data Set (or, HeteroDS) is a collection of VariableGroup objects of different VCTs, VDTs and VGSCs, and dispatches the same operation to all VGs in the same set of data. A HeteroDS also computes and maintains data set-wide statistics. Every variable in the HeteroDS is represented by a canonical index, and every VariableGroup has a variable group index. Moreover, a variable within a VariableGroup has a within group index. All three of these

indexes are combined into a single data structure called a VariableRef, which is comprised of:

- [0208] Variable Group Index: the index of a variable group within a HeteroDataSet;
- [0209] Within Group Index: the index of a variable within a variable group;
- [0210] Canonical Variable Index: the index of a variable for the entire HeteroDS.
- [0211] A HeteroDS can retrieve any variable by passing a VariableRef. It maintains an ownership flag so that when true, all VariableGroups contained inside of it are automatically deleted when the containing HeteroDS is deleted, and when false, no deletion is performed when the containing HeteroDS is deleted. The HeteroDS has the following member functions:
  - [0212] DataSet( )—The constructor builds an empty DataSet object (initially no VariableGroups) with the ownership flag set to false.
  - [0213] ~DataSet( )—Calls clear( ).
  - [0214] void set\_ownership(bool k)—Sets the variable group ownership flag to the value of k.
  - [0215] bool get\_ownership( ) const—Returns the variable group ownership flag.
  - [0216] void add\_variable\_group(BaseVariableGroup \*group)—Adds a new VariableGroup to the HeteroDS object. It must contain the same number of instances as all previous variable groups added to the object (if applicable).
  - [0217] size\_t get\_num\_variables( ) const—Returns the total number of variables in the data set object, which is the same as the sum of the v\_get\_num\_variables( ) applied to every variable group object in the HeteroDS object.
  - [0218] size\_t get\_num\_instances( ) const—Returns the total number of instances in the HeteroDS. Every variable group must return exactly the same value for get\_num\_instances( ).
  - [0219] size\_t get\_num\_variable\_groups( ) const—Returns the number of variable groups in the HeteroDS object.
  - [0220] VariableRef get\_variable\_ref(size\_t canonical\_variable\_index) const—Returns a VariableRef object that gives the 3 indices to locate the variable in the DataSet, its the variable group, and the variable within the variable group.
  - [0221] void clear( )—If the ownership flag is set to true, the variable groups in the HeteroDS are deleted.
  - [0222] size\_t get\_num\_variables\_in\_variable\_group(const VariableRef &ref) const—Returns the number of variables in the variable group specified by the VariableRef passed.
  - [0223] void compute\_downcasting( )—Determines the downcasting to perform on variable values in-flight when caching a variable in the heterogeneous cache. This is called exactly once after all variable groups of the HeteroDS are passed to it. This is effectively achieved by calling v\_compute\_downcasting( ) on every variable group in the HeteroDS object.
  - [0224] void assume\_no\_downcasting( )—An alternative to compute\_downcasting, this function ensures that no variables in the HeteroDS are downcasted in-flight when cached into a HeterogeneousCache.
  - [0225] void compute\_summaries( )—Computes summaries on every variable group object in the HeteroDS

object. This is effectively achieved by calling v\_compute\_summary( ) on every variable group in the HeteroDS object.

[0226] FIG. 4 is an exemplary diagram of a heterogeneous dataset 400 (HeteroDS) showing five (5) Variable Groups 410, 420, 430, 440 and 450, containing 42 variables in total and 1000 instances. Each Variable Group includes a Variable Group Class 412, 422, 432, 442, and 452. With regarding to FIG. 4, if an attempt is made to add a VariableGroup that has an inconsistent number of instances, an error results. There are three different implementations of a VariableGroup concept used in the example: DenseFeatureContiguousVariableGroup 414 (indicated in other Variable Groups as 424 and 434), SparseCSCVariableGroup 444, and BorrowedDiscontiguousVariableGroup 454. Two VCTs are represented: NumericVariable 416 (indicated for other VariableGroups as 436, 446, 456) and CategoricalVariable 426 with different VDT encodings.

[0227] FIG. 5 is a block diagram illustrating inlining functions for a class ExampleVariableGroup<ExampleVCT<ExampleVDT> that implements either the ImmutableVariableGroup 502 and potentially MutableVariableGroup 504. All classes implementing ImmutableVariableGroup 502 or MutableVariableGroup 504 must inherit from BaseVariableGroup 500, which is a C++ base class and which specifies virtual function prototypes. A class implementing the ImmutableVariableGroup 502 concept must provide a single typedef: the variable\_type as well as the following functions:

- [0228] size\_t get\_num\_variables( ) const—returns the total number of variables in the variable group
- [0229] size\_t get\_num\_instances( ) const—returns the total number of instances in the variable group
- [0230] const variable\_type &operator ( )(size\_t instance\_index, size\_t variable\_index) const—returns the variable value of the given instance index and variable index
- [0231] int get\_variable\_type\_code( ) const—returns a unique integer for the variable type, which is used, among other things, for dynamic dispatch of algorithms in an AlgRepo.

A MutableVariableGroup 504 defines an additional function in addition to those above:

- [0232] variable\_type &operator ( )(size\_t instance\_index, size\_t variable\_index)
- [0233] As suggested above, the functions in a class X that implements either the ImmutableVariableGroup 502 and MutableVariableGroup 504 may be inlined. However, calling these functions so inlining is possible must be on X rather than X's derived type. Functions called infrequently may not benefit from inlining, and it may hurt performance, so the dynamic dispatch technique is used. The BaseVariableGroup 500 class has the following pure virtual methods:

- [0234] virtual int v\_cache\_feature\_column(const VariableRef &ref, const RangedSubsample<IndexedSubsample>&subsample, FeatureLabelCache &cache) const=0—Caches the variable referenced by ref into the heterogeneous cache as the active feature column in the cache.
- [0235] virtual int v\_cache\_labels(const VariableRef &ref, const AllSubsample &subsample, FeatureLabelCache &cache) const=0—Caches the variable referenced by ref into the heterogeneous cache as the active label column in the cache.

- [0236] virtual void v\_cache\_counts(const VariableRef &ref, const AllSubsample &subsample, FeatureLabelCache &cache) const=0—Caches the variable referenced by ref into the heterogeneous cache as the active multiplicity column in the cache.
- [0237] virtual void v\_cache\_labels(const VariableRef &ref, std::vector<size\_t>&cache) const=0—Caches the variable referenced by ref into a pre-allocated Standard Template Library vector as nonnegative, integral labels.
- [0238] virtual void v\_cache\_labels(const VariableRef &ref, std::vector<float>&cache) const=0—Caches the variable referenced by ref into a preallocated Standard Template Library vector as floating-point labels.
- [0239] virtual size\_t v\_get\_num\_instances() const=0—Returns the number of instances in the variable group. This (less efficient) form is used when the derived type is not available.
- [0240] virtual size\_t v\_get\_num\_variables() const=0—Returns the number of variables in the variable group. This (less efficient) form is used when the derived type is not available.
- [0241] virtual int v\_get\_variable\_type\_id() const=0—Returns a unique integral value that represents the type of the elements stored in the variable group. This value is often used for dispatch.
- [0242] virtual void v\_compute\_summary()=0—Compute summary statistics of the variable group’s columns to assist algorithms
- [0243] virtual void v\_compute\_downcasting()=0—Determines in-flight downcasting to perform during all future cache requests of variables in this variable group.
- [0244] virtual void v\_assume\_no\_downcasting()=0—Avoids in-flight downcasting on all future cache requests of variables in this variable group.
- [0245] virtual size\_t v\_partition(size\_t within\_group\_variable\_index, float threshold, RangedSubsample<IndexedSubsample>&subsample) const=0—Partitions the RangedSubsample given a threshold and the index of the variable on which to apply the threshold within the target variable group
- [0246] virtual size\_t v\_partition(size\_t within\_group\_variable\_index, const std::vector<bool>&subset, RangedSubsample<IndexedSubsample>&subsample) const=0—Partitions the RangedSubsample given a bit-set and the index of the variable on which to apply the subset membership tests within the target variable group.
- [0247] virtual bool v\_is\_categorical() const=0—Returns whether this variable group stores categorical variables.
- [0248] virtual GenericVariableSummary v\_get\_variable\_summary(const VariableRef &) const=0—Returns the summary for the variable referenced.
- [0249] virtual void v\_cache\_feature\_vector(size\_t instance\_index, int32\_t \*cat\_feature\_vector, float \*num\_feature\_vector) const=0—Caches all variable values in a Variable Group of a specific instance index into a contiguous buffer for the purpose of using those values as part of a feature vector. If the variable group stores categorical features, these values are cached in cat\_feature\_vector; otherwise, they’re stored in num\_feature\_vector.
- [0250] virtual void v\_impute()=0—Performs missing value imputation on all variables in the variable group.

[0251] A developer does not need to implement all the above functions for every implementation of the VariableGroup concept, and to do so may add unnecessary development burden, especially when BaseVariableGroup is extended with new virtual function. For example, an implementation

ExampleVariableGroup<ExampleVCT<ExampleVDT>> of the VariableGroup concept, only needs to implement the small list of functions in the MutableVariableGroup 504 and ImmutableVariableGroup 502 concepts. This is achieved by using the Curiously Recurring Template Pattern (CTRP). By making the class ExampleVCT<ExampleVDT> must inherit from the intermediate base class VariableGroup Boilerplate<ExampleVariableGroup<ExampleVCT<ExampleVDT>>>510 (or IBC for short) instead of BaseVariableGroup 500. IBC will then inherit from BaseVariableGroup and implement each virtual function specified therein. These functions in IBC can cast the pointer to the target (this) to a ExampleVariableGroup<ExampleVCT<ExampleVDT>>> and benefit from inlining. The intermediate base class VariableGroupBoilerplate<>510 that uses this list of functions to implement all of the functionality required in the base class BaseVariableGroup’s list of virtual functions.

[0252] FIG. 5 is a block diagram that describes how such a process works. The VariableGroup implementation ExampleVariableGroup 520 implements the four inlined functions specified in ImmutableVariableGroup 502 and the one inlined function in MutableVariableGroup 504.

[0253] The present invention uses a type to represent each VCT. The type takes as one or more template argument a description of the VDT. Then, pattern matching may be performed at the time of compiling on each type to ensure that the appropriate instantiation of an algorithm is dispatched for each combination of VCT and VDT.

[0254] This “type-ification” of the VCT and VDT can also be used to prevent program errors. Most implementations use domain-agnostic types (e.g. int, long, float, double) to represent values. A common source of error is performing an operation on a value that is allowed by the domain-agnostic type, but is ill-defined in the domain context. For example, making a “less than” comparison between two categorical values (e.g. Ford and Chevy). By restricting the operations to the domain, type safety is attained.

[0255] Four types are defined below, each for a different VCT:

[0256] CategoricalVariable<T>: represents a categorical/nominal variable such as gender (Male, Female), car maker (Ford, Honda, Chevy) where only equality comparisons can be performed. No arithmetic is possible with values of this type. Invalid operations will cause the compiler to not compile the program. Native integral types, such as char, int, long, etc. can be used as T.

[0257] NumericVariable<T>: represents a numeric variable such as age or blood pressure. Arithmetic can be performed on values of this type. Any floating point or integral type can be used, e.g. int, float, double, long, char.

[0258] DateVariable<T>: represents a date variable where T is some class used to encode dates at the sufficient granularity. Comparisons are allowed. Subtraction between DateVariables yields a DurationVariable. Subtraction between a DateVariable and a DurationVariable yields a DateVariable.

- [0259] DurationVariable<T,G>: represents a duration where T encodes values and G is a type representing the unit of time. Ordered comparisons are possible.
- [0260] The present invention also includes a subsampling concept. A subsample represents a subset of instances of a HeteroDS, and repetition among instances is permissible. The subsample data structures reside outside the HeteroDS rather than being incorporated within it, to reduce complication from storing variables in different objects depending on the VCT, VDT, and underlying implementation.
- [0261] The subsample concept has the following attributes:
- [0262] size( )—the number of elements (with counting of possible repetition)
- [0263] get\_ancestor\_index(size\_t i)—retrieves the instance index of the i'th instance in the subsample
- [0264] get\_parent\_index(size\_t i)—retrieves the instance index
- [0265] Additionally, there are three classes that implement the subsample concept:
- [0266] AllSubsample—Generates indices 0 . . . N (exclusive) without repetition. Both get\_ancestor\_index and get\_parent\_index are idempotent operations.
- [0267] IndexedSubsample<SubsampleType>—A subsample of some other subsample S of type SubsampleType that implements the subsample type (parent subsample). Stores indices to the parent subsample in an index vector idx. get\_ancestor\_index(i) returns idx[i]. The size of the subsample is the number of elements in idx.
- [0268] RangedSubsample<SubsampleType>—A contiguous subsample of some other subsample S of type SubsampleType that implements the subsample type (parent subsample). Stores the lower and upper bound (both <parent->size( ) of the indices. The size of the subsample is upper-lower+1.
- [0269] With regarding to homogeneous datasets, the present invention includes a traditional data structure to represent decision trees; it looks something like this:

---

```

struct Node {
    Node *left;
    Node *right;
    float label;
    float threshold;
    long feature_index;
};

```

---

[0270] For heterogeneous data sets, the present invention applies a heterogeneous tree. Nodes can have an arbitrary number of fields, which are associative arrays mapping field names (strings) to fields (or HeteroField). A HeteroField<T> is an array of values of type T, where T be any of the following types:

- [0271] 32-bit floating point number
- [0272] 64-bit floating point number
- [0273] 32-bit integer
- [0274] 64-bit integer
- [0275] Boolean
- [0276] variable length bitstring
- [0277] string

This representation is easily extendable to other types.

[0278] A heterogeneous tree alleviates several issues associated with applying the traditional data structure to represent decision trees above to homogeneous data sets. In the tradi-

tional representation, using a fixed-sized struct to represent leaf and nonleaf nodes, is not as efficient as it could be for certain data, as the “label” field is only needed in leaf nodes, whereas the “threshold” and “feature\_index” fields are only needed in nonleaf nodes. In addition, the left and right pointers point to other nodes for nonleaf nodes but point to NULL for leaf nodes.

[0279] If a specialized algorithm needs to store more context or info, it cannot be incorporated in the tree nodes unless the struct is changed. Also, new versions of the struct cannot be read into old versions of the software. This makes it difficult to extend the present invention while maintaining backwards compatibility and conservation of space. Moreover, the nodes can be far from each other in memory leading to less locality of reference, which may impact computational performance.

[0280] FIG. 6 is a block diagram illustrating application of a heterogeneous tree 600. By applying a heterogeneous tree 600, no distinction is made between leaf and nonleaf nodes in the node's type. As FIG. 6 illustrates, leaf nodes 610 are indexed with negative numbers 612, and nonleaf nodes 620 are indexed with nonnegative numbers 622. Each field is an array of values where every value has exactly the same type. Every nonleaf node must store “left” and “right” fields by convention.

[0281] It is to be noted from FIG. 6 that space is saved by not storing pointers to the left and right children for leaf nodes 610. Common fields stored in leaf nodes 610 in Decision Trees for heterogeneous datasets are as follows:

[0282] “label”: 32-bit float for regression, 32-bit int for classification

[0283] “mean” (optional): the mean label value of instances following into the leaf node as a 32-bit float

[0284] “variance” (optional): the variance of the label value of instances following into the leaf node as a 32-bit float

[0285] “nlabel0”, “nlabel1”, . . . (optional): the value of the first, second, etc. histogram bins as a 32-bit or 64-bit integers. This field can be used to store the histogram of labels for training instances that fall into a particular leaf node. This can be used to change the voting process so that each leaf node casts a vote for a label i (for example) using the value of histogram bin i (stored in nlabeli).

[0286] “expXsquared” (optional): the expectation of floating point label values that are squared. This can be used to reconstruct the variance of the labels for the instances that fell into a particular leaf node during training or assist in estimating the uncertainty of a regression.

[0287] A HeteroDS enables variables to have different VCTs and VDTs. Moreover, the (VCT, VDT) combination can vary in implementation, i.e. the class implementing ImmutableVariableGroup and MutableVariableGroup concepts can be different. This poses a challenge in writing algorithms that operate on values from more than one variable—retrieving the values from the two different variables via dynamic dispatch mechanisms is costly. FIG. 7 is a block diagram illustrating a heterogeneous cache 700 (HGCache), which is an intermediate data structure to hold variable values contiguously in memory with simple striding so that the tree node induction is agnostic to the VCT, VDT, and Variable Group implementation details.

[0288] FIG. 7 shows an example of a heterogeneous cache object 710 in a heterogeneous cache 700. It has four buffers 720, 730, 740, and 750, and data from VariableGroup objects

are copied into the first three buffers **720**, **730** and **740** to populate them as follows. Weight cache **720** stores weights for instances involved in a split induction (determined by a bootstrapped or a non-bootstrapped sample). This buffer **720** is allocated so it is aligned to the word boundary of the widest VDT as well as the cache line boundary and contains enough bytes to store enough VDT elements to hold all weights for a sample.

**[0289]** Label cache **730** stores labels for instances involved in a split induction. This buffer **730** is allocated in a similar fashion to the weight cache **720**. Feature cache **740** stores features for instances involved in a split induction. This buffer **740** is allocated in a similar fashion to the weight cache **720**. Triple cache **750** stores enough bytes to hold (feature, label, weight) triples for all instances in a sample on a HeteroDS.

**[0290]** For example, a split is induced on a feature stored in a variable with VariableRef feature ref and the labels stored in a variable with VariableRef label\_ref. First, call `data_set->get_variable(label_ref)->v_cache_labels(label_ref, s, c)` on the cache passing the variable group corresponding to i and call `v_cache_features` on the cache passing the variable group corresponding to j where s is the heterogeneous cache object that will hold the data and c is an object from a class implementing the Subsample concept. If weighted learning is used so that some instances have more contribution to the error than others in proportion to an instance weight, `data_set->get_variable(weight_ref)->v_cache_weights(label_ref, s, c)` is called to use the variable referenced for the instance weights. FIGS. **8**, **9** and **10** show examples of caching weights, labels, and features prior to populating the triple cache.

**[0291]** Returning to FIG. **7**, next the triple cache **750** is populated. `Combiner<LabelVG, FeatureVG, WeightVG>` is a type that inherits from `BaseCombiner` and `v_combine()` is a pure virtual function in `BaseCombiner` and a non-pure virtual function in `Combiner<LabelVG, FeatureVG, WeightVG>`. Build an `AlgRepo` on all valid combinations of `LabelVG`, `FeatureVG`, `WeightVG`. Dispatch the appropriate derived `Combiner< >` based on the integral type codes returned by calling `v_get_variable_type_id()` on the label variable group, feature variable group, and weight variable group. This function copies the elements from the weight, feature, and label buffers **720**, **730** and **740** into the fourth buffer **750** where the elements of the fourth buffer **750** are tuples of the form `std::tuple<LabelVDT, FeatureVDT, WeightVDT>`. Note that none of these buffers (**720**, **730** or **740**) record any info about the VCT or underlying VariableGroup implementation.

**[0292]** An `AlgRepo` of node induction procedures across every combination of `LabelVDT`, `FeatureVDT`, and `WeightVDT` is built beforehand. Given the integral type codes of the variables recently cached in the combine step, the most compatible node induction procedure is dispatched from the `AlgRepo`.

**[0293]** As noted above, FIG. **8**, FIG. **9**, and FIG. **10** are block diagrams illustrating examples of caching weights, labels and features as indicated in FIG. **7** prior to populating the triple cache **750**. FIG. **8** shows a VariableGroup **8**, labeled as box **800**, and steps for caching from Variable 5 in VariableGroup **8**. These steps include invoking a weights call function **810**, which effectively runs an implementation in Variable Group Boilerplate so that inlining can be performed via CRTP. The process then populates the buffer **720** in step **820**.

**[0294]** FIG. **9** shows a VariableGroup **2**, labeled as box **900**, and steps for caching from Variable 5 in Variable Group **2**. These steps include invoking a labels call function **910**, which effectively runs an implementation in Variable Group Boilerplate so that inlining can be performed via CRTP. The process then populates the buffer **730** in step **920**.

**[0295]** FIG. **10** shows a VariableGroup **6**, labeled as box **1000**, and steps for caching from Variable 18 in Variable Group **6**. These steps include invoking a features call function **1010**, which effectively runs an implementation in Variable Group Boilerplate so that inlining can be performed via CRTP. The process then populates the buffer **740** in step **1020**.

**[0296]** As noted above, it is contemplated that the systems and methods of the present invention may be implemented using one or more processors and memory components within a computing environment. However, it is to be understood that the systems and methods of implementing a learning ensemble of decision trees in a single-machine environment for homogeneous and heterogeneous datasets according to the present invention may be further implemented in many different computing environments generally. For example, they may be implemented in conjunction with one or more special purpose computers, programmed microprocessors or microcontrollers and peripheral integrated circuit element(s), an ASIC or other integrated circuits, digital signal processor (s), electronic or logic circuitry such as discrete element circuits, programmable logic devices or gate arrays such as a PLD, PLA, FPGA, PAL, and any comparable means. In general, any means of implementing the systems and methods illustrated herein can be used to implement the various aspects of the present invention. Exemplary hardware that may be utilized in one or more embodiments or aspects of the present invention includes computers, handheld devices, telephones (e.g., cellular, Internet enabled, digital, analog, hybrids, and others), and other such hardware. Some of these devices include processors (e.g., a single or multiple microprocessors), memory, nonvolatile storage, input devices, and output devices. Furthermore, alternative software implementations including, but not limited to, distributed processing, parallel processing, or virtual machine processing can also be configured to perform the systems and methods described herein.

**[0297]** The systems and methods of the present invention may also be partially implemented in software configured to execute one or more routines, functions, or algorithms, and that can be stored on a storage medium, executed on programmed general-purpose computer with the cooperation of a controller and memory, a special purpose computer, a microprocessor, or the like. In these instances, the systems and methods of this invention can be implemented as a program embedded on personal computer such as an applet, JAVA.RTM or CGI script, as a resource residing on a server or computer workstation, as a routine embedded in a dedicated measurement system, system component, or the like. The system can also be implemented by physically incorporating the system and/or method into a software and/or hardware system.

**[0298]** Additionally, the data processing routines, functions and algorithms disclosed herein may be performed by one or more program instructions stored in or executed by such memory, and further may be performed by one or more modules configured to carry out those program instructions. Modules are intended to refer to any known or later developed hardware, software, firmware, artificial intelligence, fuzzy

logic, expert system or combination of hardware and software that is capable of performing the data processing functionality described herein.

**[0299]** The foregoing descriptions of embodiments of the present invention have been presented for the purposes of illustration and description. It is not intended to be exhaustive or to limit the invention to the precise forms disclosed. Accordingly, many alterations, modifications and variations are possible in light of the above teachings, may be made by those having ordinary skill in the art without departing from the spirit and scope of the invention. It is therefore intended that the scope of the invention be limited not by this detailed description. For example, notwithstanding the fact that the elements of a claim are set forth below in a certain combination, it must be expressly understood that the invention includes other combinations of fewer, more or different elements, which are disclosed in above even when not initially claimed in such combinations.

**[0300]** The words used in this specification to describe the invention and its various embodiments are to be understood not only in the sense of their commonly defined meanings, but to include by special definition in this specification structure, material or acts beyond the scope of the commonly defined meanings. Thus if an element can be understood in the context of this specification as including more than one meaning, then its use in a claim must be understood as being generic to all possible meanings supported by the specification and by the word itself.

**[0301]** The definitions of the words or elements of the following claims are, therefore, defined in this specification to include not only the combination of elements which are literally set forth, but all equivalent structure, material or acts for performing substantially the same function in substantially the same way to obtain substantially the same result. In this sense it is therefore contemplated that an equivalent substitution of two or more elements may be made for any one of the elements in the claims below or that a single element may be substituted for two or more elements in a claim. Although elements may be described above as acting in certain combinations and even initially claimed as such, it is to be expressly understood that one or more elements from a claimed combination can in some cases be excised from the combination and that the claimed combination may be directed to a sub-combination or variation of a sub-combination.

**[0302]** Insubstantial changes from the claimed subject matter as viewed by a person with ordinary skill in the art, now known or later devised, are expressly contemplated as being equivalently within the scope of the claims. Therefore, obvious substitutions now or later known to one with ordinary skill in the art are defined to be within the scope of the defined elements.

**[0303]** The claims are thus to be understood to include what is specifically illustrated and described above, what is conceptually equivalent, what can be obviously substituted and also what essentially incorporates the essential idea of the invention.

1. A method of implementing a learning ensemble of decision trees in a single machine computing environment, comprising:

implementing, within a single-machine computing environment comprised of hardware and software components that include at least one processor, the steps of:

inlining relevant statements to integrate function code into a caller's code so that repetitive pushing and popping of register content to and from a stack at each compilation is eliminated;

implementing a contiguous buffer arrangement for register content to be compiled in a plurality of buffers; and

defining and enforcing a plurality of type constraints on programming interfaces that access and manipulate at least one machine learning data set so that a plurality of procedures for inducing a forest induction, a tree induction, and a node induction are instantiated for classes implementing the at least one machine learning data set in learning an ensemble of decision trees.

2. The method of claim 1, further comprising computing a score for the at least one machine learning data set during an instantiation of a procedure for a tree induction for any class implementing the at least one machine learning data set.

3. The method of claim 1, wherein the plurality of buffers in the contiguous buffer arrangement includes an instance buffer cache configured to store a feature vector for an instance to reduce the delay associated with cache misses by improving spatial and temporal locality of reference for data needed during node induction.

4. The method of claim 1, further comprising computing an impurity value for splitting criteria in the at least one machine learning data set during an instantiation of a procedure for a tree induction for classes implementing the at least one machine learning data set.

5. The method of claim 1, further comprising computing an impurity value for splitting criteria in the at least one machine learning data set during an instantiation of a procedure for a node induction for classes implementing the at least one machine learning data set.

6. The method of claim 1, wherein the defining and enforcing a plurality of type constraints on programming interfaces further comprises defining specifically-named typedefs.

7. The method of claim 1, wherein the defining and enforcing a plurality of type constraints on programming interfaces enables checking the plurality of type constraints when a class adhering to a concept is needed at compilation, wherein a failure to satisfy any constraints results in compilation failure.

8. The method of claim 1, wherein the defining and enforcing a plurality of type constraints on programming interfaces that access and manipulate at least one machine learning data set further defines a specific set of homogeneous data within the at least one machine learning data set targeted for a learning ensemble of decision trees.

9. The method of claim 1, further comprising representing a subsample by chaining a plurality of classes together where the plurality of type constraints are defined.

10. The method of claim 1, wherein the at least one machine learning data set provides one or more functions for accessing size and problem type information, the one or more functions configured to return a total number of instances and feature vectors, a total number of features in a feature vector, a total number of classes, and a problem type comprised of one or the other of classification and regression.

11. The method of claim 1, wherein the at least one machine learning data set provides one or more functions for accessing individual feature values and labels configured to return a number of classes for classification and regression, one or more functions configured to generate iterators to feature values over a fixed feature index or a fixed instance

index, and one or more functions configured to generate iterators to feature values over a fixed instance index.

12. The method of claim 1, wherein the at least one machine learning data set is a writable machine learning data set that permits labels and feature values to be changed, and permits new instances to be added.

13. The method of claim 1, wherein the at least one machine learning data set is implemented in multiple classes in which machine learning data is stored in memory, each class using an optimized memory layout for a different purpose.

14. The method of claim 11, wherein one optimized memory layout is configured to store a machine learning data set so that all instances for a specific feature index are stored contiguously, and another optimized memory layout is configured to store a machine learning data set so that all features for a specific instance index are stored contiguously.

15. A system comprising:  
a computer processor; and

at least one computer-readable storage medium operably coupled to the computer processor and having program instructions stored therein, the computer processor being operable to execute the program instructions to optimize machine intelligence for implementing a learning ensemble of decision trees in a single-machine environment in a plurality of data processing modules, the plurality of data processing modules configured to integrate function code into a caller's code by inlining relevant statements so that repetitive pushing and popping of register content to and from a stack at each compilation is eliminated, implement a contiguous arrangement for register content to be compiled in a plurality of buffers, and define and enforce type constraints on programming interfaces that access and manipulate machine learning data sets so that a plurality of procedures for inducing a forest induction, a tree induction, and a node induction are instantiated for classes implementing the machine learning data sets in a process for implement the learning ensemble of decision trees.

16. The system of claim 15, wherein the plurality of data processing functions embody one or more concepts arranged to subsample the machine learning data sets by chaining multiple classes together that implement the machine learning data sets.

17. The system of claim 15, wherein the plurality of buffers includes an instance buffer cache configured to store a feature vector for an instance to reduce the delay associated with cache misses by improving spatial and temporal locality of reference for data needed during node induction.

18. The system of claim 15, wherein the type constraints include specifically-named typedefs.

19. The system of claim 15, wherein the type constraints determine when a class adhering to a concept is needed at compilation, and further wherein a failure to satisfy any constraints results in compilation failure.

20. The system of claim 15, wherein the type constraints enable accessing and manipulating at least one machine learning data set that further defines a specific set of homogeneous data within the at least one machine learning data set targeted for a learning ensemble of decision trees.

21. The system of claim 15, wherein the at least one machine learning data set is a writable machine learning data set that permits labels and feature values to be changed, and permits new instances to be added.

22. The system of claim 15, wherein the at least one machine learning data set is implemented in multiple classes in which machine learning data is stored in memory, each class using an optimized memory layout for a different purpose.

23. The system of claim 15, wherein the plurality of data processing modules are further configured to compute a score for data in the machine learning data sets during an instantiation of a procedure for a tree induction for classes implementing the machine learning data sets.

24. The system of claim 15, wherein the plurality of data processing modules are further configured to compute an impurity value for splitting criteria in the at least one machine learning data set during an instantiation of a procedure for a tree induction for classes implementing the machine learning data sets.

25. The system of claim 15, wherein the plurality of data processing modules are further configured to compute an impurity value for splitting criteria in the machine learning data sets during an instantiation of a procedure for a node induction for classes implementing the machine learning data sets.

26. A method of implementing a learning ensemble of decision trees in a single-machine computing environment, comprising:

implementing, within a single-machine computing environment comprised of hardware and software components that include at least one processor, a compiler optimization routine that determines run time compilation requirements by copying compiled code for a function into a caller's compiled code so that function call overhead from pushing and popping register content and local variables to a stack for each function call is eliminated;

aligning buffers comprising a plurality of caches to place bytes required for compilation in a contiguous arrangement so that fewer pages of data are pulled from memory to blocks in the plurality of caches, the plurality of caches including a feature buffer cache that copies discontiguous feature values comprised of all instances with respect to a specific feature dimension into the contiguous arrangement, and an instance buffer cache that copies a discontiguous feature vector for an instance into the contiguous arrangement; and

applying one or more of class and function templates to target specific datasets by enforcing type constraints on programming interfaces that access and manipulate at least one machine learning data set so that a plurality of procedures for inducing a forest induction, a tree induction, and a node induction are instantiated for classes implementing the at least one machine learning data set in learning an ensemble of decision trees.

27. The method of claim 26, wherein the compiler optimization routine enables inlining of one or more of relevant statements, variants and procedures.

28. The method of claim 26, wherein the one or more class and function templates impart restrictions on types to provide a mechanism for specifying the programming interfaces for template parameters.

29. The method of claim 26, wherein the type constraints include specifically-named typedefs.

30. The method of claim 26, wherein the type constraints determine when a class adhering to a concept is needed at

compilation, and further wherein a failure to satisfy any constraints results in compilation failure.

31. The method of claim 26, wherein the type constraints enable accessing and manipulating at least one machine learning data set that further defines a specific set of homogeneous data within the at least one machine learning data set targeted for a learning ensemble of decision trees.

32. The method of claim 26, wherein the at least one machine learning data set is a writable machine learning data set that permits labels and feature values to be changed, and permits new instances to be added.

33. The method of claim 26, wherein the at least one machine learning data set is implemented in multiple classes in which machine learning data is stored in memory, each class using an optimized memory layout for a different purpose.

34. The method of claim 26, wherein the applying one or more of class and function templates to target specific datasets by enforcing type constraints on programming interfaces further comprises computing a score for the at least one machine learning data set during an instantiation of a procedure

for a tree induction for any class implementing the at least one machine learning data set.

35. The method of claim 26, wherein the applying one or more of class and function templates to target specific datasets by enforcing type constraints on programming interfaces further comprises computing an impurity value for splitting criteria in the at least one machine learning data set during an instantiation of a procedure for a tree induction for classes implementing the at least one machine learning data set.

36. The method of claim 26, wherein the applying one or more of class and function templates to target specific datasets by enforcing type constraints on programming interfaces further comprises computing an impurity value for splitting criteria in the at least one machine learning data set during an instantiation of a procedure for a node induction for classes implementing the at least one machine learning data set.

37. The method of claim 26, further comprising representing a subsample by chaining a plurality of classes together for the one or more of class and function templates.

\* \* \* \* \*