

IN THE UNITED STATES PATENT AND TRADEMARK OFFICE

S P E C I F I C A T I O N

TITLE OF THE INVENTION

Method of Implementing Scalable, Memory-Efficient Machine Learning and Prediction to Learning Ensembles of Decision Trees.

INVENTOR(S)

Damian Ryan Eads.

FIELD OF THE INVENTION

The present invention relates to machine intelligence. Specifically, the present invention related to a method of scaling machine learning and prediction to very large data sets 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

There are many existing implementations of machine intelligence. Learning ensembles of decision trees are a popular method of machine learning, and one such implementation, known as Random Forests^{TM1}, are a combination of several decision trees to permit the use of multiple types of data that do not need to be normalized.

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 and rates become increase exponentially. There are many Random Forest permutations in the existing art, but they suffer from an inability to take full advantage of intricate computer architecture

¹ Random ForestsTM are the subject of a January 2001 article by Leo Breiman popularizing the term and defining it specifically as a combination of tree predictors such that each tree depends on the values of a random vector sampled independently and with the same distribution for all trees in the forest.

environments in which they are tasked with making sense out of these immense data populations.

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. It is therefore one objective of the present invention to provide a machine intelligence framework that is both efficient and accurate. 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.

BRIEF SUMMARY OF THE INVENTION

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 (hereinafter EDTs) by applying 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 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 reduce memory consumption, 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 robotic 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.

The present invention applies these computer architecture manipulation techniques across both training and testing modes of operation in Random Forest implementations. Training involves learning a forest of decision trees from labeled feature vectors, and testing 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.

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.

DETAILED DESCRIPTION OF THE INVENTION

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.

The present invention provides an approach 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 a significant enhancement of processing speed and memory usage in analyzing large data sets in a time-effective and cost-effective manner.

In machine intelligence, there are two general modes of operation: training (or learning) and testing (or prediction). Training involves learning a forest of decision

trees from labeled feature vectors, and testing takes a learned model based on the training set and generates predictions on a new set of data. Within these two modes, there are generally two functions: 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_i 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 called an Instance. The x_{ij} value is called a feature and the integer j that identifies a feature dimension is called a feature index. The length of x_i is called 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.

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 categorial (e.g. the make of the car). When the output variable is boolean or categorial, 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.

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 categorial features, a categorial feature value is checked for membership to a subset of category indices. If it belongs, the test passes. Otherwise, it fails.

A Decision Tree Node encodes the following information:

- Node Left Child: a handle to the left node for a non-leaf node or a sentinel value for a leaf node.
- Node Right Child: a handle to the right node for a non-leaf node or a sentinel value for a leaf node.
- Node Feature Index: the feature index or feature indices (represented by a collection of feature indices) that is/are needed to apply the test.
- Node Test Parameters: the parameterization of the test, if any.

- 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.
- Node Label: the label to predict for a leaf node or a sentinel value for a non-leaf node.
- Node Uncertainty Parameters: optional uncertainty parameters for a leaf node or a sentinel value for a non-leaf node.
- 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:

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.
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.
3. if the test passes, repeat step 1 where X is now the left node encoded in X (i.e. $X := X.left$)
4. if the test fails, repeat step 1 where X is now the right node encoded in X (i.e. $X := X.right$)

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.

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.

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. This results from wasted processing power from accessing information that is necessary. For example, if only a small portion of each page in a process's memory is needed, then the rest of each page is wasted. Contiguity is hindered because these pages are found at disparate locations. A lack of contiguity hinders a program's ability to make effective use of the processor cache, and can lead to cache misses, cache freezes, or cache thrashing.

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. 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, 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, also have in impact on processing speed. The present invention may encode these assumptions in a type and use partial template specialization to exhibit specialized behavior for each subtype.

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. They do not quickly return memory used for initialization back to the operating system (e.g. via a system call).

The approach of the present invention and the various embodiments described herein address 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.

These mechanisms include application of one or more C++ concepts that represent a set of functions (and typedefs) that a class must have. In the present invention, one such concept is a machine learning dataset (MLDataset) that is a representation of machine learning data sets 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.).

Another concept addresses buffer contiguity by implementing solutions to address the problem of a lack of speed associated with having to continuously access unnecessary information from disparate computer architecture 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 saving the use of large numbers of clock cycles which significantly reduce processing speed.

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 for template meta-programming.

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.

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.

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 heavy use of inlining on procedures to reduce the function call overhead, especially procedures called a large number of times. Histogram approximation of variants 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.

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 time and space usage 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.

Application of C++ concepts as described herein permits the present invention to determine what is needed for compilation at runtime, by inlining code as it required. 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.

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.

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.

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 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. 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.

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 the 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.

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

[10]

enforce constraints on programming interfaces that access and manipulate machine learning data sets.

The `MLDataSet` C++ concept represents a labeled or unlabeled machine learning data set that is either labeled or unlabeled. 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.

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; `label_type`, which represents the type of the labels; `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. 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

[11]

regression); and inline LabelType get_labeling() const, which returns the machine learning problem type using enumerated types (classification or regression).

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.

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_const_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.

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.

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_feature_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.

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.

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.

`WritableMLDataSet` is a separate concept, rather than its constraints and interfaces being included in the `MLDataSet` concept. This allows us to 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).

In the MLDataSet concept, a class that implements facilities for caching features is said to implement the FeatureCache concept as follows:

`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.

`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.

`typedef FeatureCacheIterator` - this is a type for a non-const iterator over (feature, label) pairs in the feature cache.

`typedef FeatureCacheConstIterator` - this is a type for a const iterator over (feature, label) pairs in the feature cache.

`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.

Also,

`inline FeatureCacheIterator begin()`

`inline FeatureCacheIterator end()`

`inline FeatureCacheConstIterator begin() const`

`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.

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

`inline void cache_instance(size_t instance_index)` - caches an instance of a particular instance index into a contiguous buffer.

[14]

typedef InstanceCacheIterator - a type for a non-const iterator over an instance's feature vector.

typedef InstanceCacheConstIterator - a type for a const iterator over an instance's feature vector.

Also,

```
inline InstanceCacheIterator instance_cache_begin()
```

```
inline InstanceCacheIterator instance_cache_end()
```

```
inline InstanceCacheConstIterator instance_cache_begin() const
```

```
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).

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.

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 contiguously 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.

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.

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.

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.

A view implementation of an MLDataSet can (optionally) also implements 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`.

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.

The MLDataSet concept may be configured to allow for one implementation of a DataSet 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 MLConcept. 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:

```
typedef FeatureContiguousMemoryDataSet <float, int> mem_dataset;

FeatureContiguousMemoryDataSet dataset;

Dataset.load_from_csv("file.csv");

IndexDataSet<mem_dataset> indexed_dataset(dataset);

const size_t n(dataset.get_num_instances());

indexed_dataset.sample_indices_iid_uniformly_at_random(n);
```

[17]

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<FeatureContiguousMemoryDataSet> >` 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.

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:

```
indexed_dataset.select_all_from_parent();
```

```
indexed_dataset.random_shuffle(rng);
```

```
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`):

```
PartitionDataSet<IndexDataSet<mem_dataset>, Complement>
```

```
    third_partition_prime(indexed_dataset, 3, 10);
```

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.

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:

- `node_handle` - type to refer to child decision tree nodes.
- `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`.
- `test_parameter_type` - type of the encoding for the Node Test's parameterization
- `threshold_type` - type of thresholds used for threshold tests.
- `uncertainty_parameterization_type` - a type used to parameterize the uncertainty computation when predicting..
- `feature_indices_const_iterator` - a type of random access iterator used to traverse feature indices when a Node Test involves multiple features.

Implementors of the DecisionTreeNode must also implement the following functions:

- `inline handle_type get_left() const` - returns a handle to the Node's Left Child
- `inline handle_type get_right() const` - returns a handle to the Node's Right Child
- `inline int get_feature_index() const` - returns the feature index used to apply a test
- `inline threshold_type get_threshold() const` - returns the threshold used for threshold tests
- `inline uncertainty_type &get_uncertainty_parameters()` - returns the uncertainty parameters used for prediction.
- `inline const uncertainty_type &get_uncertainty_parameters() const` - returns the uncertainty parameters used for prediction.
- `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

- 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.
- 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
- 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.

In decision tree learning, a Classification and Regression Tree (CART) algorithm² 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 $T := \text{root}$, $D := \text{data set}$):

1. best_feature_index := undefined; best_score := 0; best_test := undefined;
2. k := 0
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
4. If k < mtry
 - a. let $f < D.\text{get_num_instances}()$ be an integer drawn uniformly at random
 - 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.
 - c. if best_score < s then
 - i. best_score := s
 - ii. best_feature_index := f

² The Classification and Regression Tree (CART) algorithm was first introduced in a book by Leo Breiman, et al in 1981.

- iii. `best_test := p`
 - d. `k := k + 1`
 - e. `goto 4`
 5. If the node learning procedure led to no acceptable increase in score (or decrease in loss, depending on the criteria), goto step 6.
 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.
 7. Otherwise, store the feature index, threshold, and other learned parameters in T.
 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.
 9. Recurse to step 1 with `D := DL` and `T := TL`
 10. Recurse to step 1 with `D := DR` and `T := TR`
 11. Restore the caller's state, and return to it.

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 DataSet before recursing at step 9 or 10.

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.

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).`

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:

- `template <class FeatureCache> inline void increment_count(const FeatureCache &cache)` – adds all instance’s feature values in a feature cache to the statistics recorded by the CountMap
- `template <class LabelType> inline void increment_count(LabelType label)` – updates the statistics curated by the count map by including a single repetition of the label.
- `template <class LabelType> inline void increment_count(LabelType label, int num)` - updates the statistics curated by the count map by including a specified multiple of repetitions of a label.
- `template <class LabelType> inline void decrement_count(LabelType label)` – updates the statistics curated by the count map by excluding a single repetition of the label.
- `template <class LabelType> inline void decrement_count(LabelType label, int num)` - updates the statistics curated by the count map by excluding a specified multiple of repetitions of a label.

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

- regression and (real-valued or integer features)
- regression and categorical features
- classification and (real-valued or integer features)
- classification and (categorical features)

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

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:

- `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.
- `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).
- `inline double get_improvement(double overall_impurity, double test_impurity)` – returns a statistic representing the improvement in impurity induced by the test.

The FlexiAlg concept is used to analyze the characteristics of a 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.

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.

It is to be understood that other embodiments will be utilized and structural and functional changes will be made without departing from the scope of the present

invention. 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 modifications and variations are possible in light of the above teachings. It is therefore intended that the scope of the invention be limited not by this detailed description.

THE CLAIM OR CLAIMS

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

inlining relevant statements by integrating function code into a caller's code to minimize repetitive pushing and popping of register content to and from a stack at each compilation;

ensuring a contiguous arrangement for necessary information to be compiled in a plurality of buffers; and

defining and enforcing type constraints on programming interfaces that access and manipulate machine learning data sets.

2. A method of machine learning and prediction in classification and regression of data, comprising:

minimizing speed constraints associated with excessive data processing times by applying a plurality computer architecture manipulation techniques to effectively utilize a processor cache, reduce function call overhead, and minimize performance of if/then conditional statements; and

optimizing memory usage associated with excessive copying of unnecessary data by applying one or more techniques to instantiate templated functions.

3. A system for optimizing machine intelligence, comprising:

a plurality of machine instructions configured to inline relevant statements by integrating function code into a caller's code to minimize repetitive pushing and popping of register content to and from a stack at each compilation, ensure a contiguous arrangement for necessary information 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, wherein the plurality of machine instructions are embodied in one or more concepts arranged to manipulate computer architecture availability and processing of data sets to be analyzed to minimize speed constraints associated with excessive data processing times and optimize memory usage associated with excessive copying of unnecessary data.

[25]

ABSTRACT OF THE DISCLOSURE

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 by applying 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. This 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.