RC24882 (W0910-132) October 22, 2009
Mathematics

IBM Research Report

The Parallel Machine Learning (PML) Framework and the Transform Regression Algorithm

Sitaram Asur, Amol Ghoting, Ramesh Natarajan, Edwin Pednault
IBM Research Division
Thomas J. Watson Research Center
P.O. Box 218
Yorktown Heights, NY 10598

The Parallel Machine Learning (PML) Framework and the Transform Regression Algorithm

Sitaram Asur[*], Amol Ghoting, Ramesh Natarajan, Edwin Pednault
IBM T. J. Watson Research Center
P. O. Box 218
Yorktown Heights, NY, 10598.

October 14, 2009

Abstract

Machine learning techniques are increasingly being used with massive training data sets in application areas such as internet, retail, insurance, finance, manufacturing and life sciences. The Parallel Machine Learning (PML) toolkit is a software framework for machine learning algorithms on high-performance computer (HPC) platforms (such as the IBM Blue Gene/P supercomputer). Several well-known algorithms have been implemented using the PML framework to date, and we specifically describe the detailed implementation of the transform regression (TREG) algorithm, in view of its novelty, parallel scalability and wide applicability.

1 Introduction

Massive training data sets, that range in size from tens of gigabytes to several terabytes, and which, therefore, cannot be stored in entirety within the main memory of typical general-purpose computing platforms, are increasingly becoming important in machine learning research. The relevant applications arise in diverse disciplines, including for example, text mining of web corpora, multimedia analysis of image and video data, retail modeling of consumer transaction data, bio-informatic analysis of genomic and micro-array data, medical analysis of clinical diagnostic data such as fMRI images, and environmental modeling using sensor and streaming data.

An excellent summary of scalable techniques for machine learning has been provided by Provost and Kolluri [4], from which we glean three broad directions of relevant research. The first approach is concerned with sub-sampling of the large data set along the case and feature dimensions, and the focus is on the accuracy of the resulting model or model ensembles, as a function of the sub-sampling rate. The second approach is concerned with the use of pre-processing techniques, such as clustering, principal components and feature selection, in order to reduce the data set size and dimensionality for subsequent analysis. The third approach, which is the primary focus of the present paper, is concerned with the development of scalable

[*]work performed during summer internship at IBM Research, currently at HP Labs

machine learning algorithms, particularly in the case when acceptable model accuracy cannot be obtained by sub-sampling or data reduction, but only through algorithmic reformulation and computational parallelization.

The Parallel Machine Learning (PML) toolkit [16] is a software framework, which provides a consistent and scalable implementation of the common functionality and services for machine learning algorithms. This framework is specifically designed for massive training data sets on parallel computers, and is based on a master-worker execution model, where in general, the worker processes execute independently in parallel on separate row partitions of the training data set, with occasional inter-process synchronization and data exchange. In particular, as described below, PML has many specific performance optimizations for distributed-memory HPC platforms, which take maximum advantage of the I/O bandwidth, memory capacity and processing capability of these platforms.

The PML framework provides a specific applications programming interface (API) for incorporating individual machine learning algorithms, and most well-known algorithms can be adapted to this API with few or no modifications. Although the resulting adaptation of this algorithm may not always be equivalent to a customized "one-off" parallel implementation, the primary advantage of using the PML API is that all the details of the parallel task control, inter-processor communication and data I/O are abstracted away from the algorithm developer, and this common functionality is uniformly provided to all the enabled algorithms in the PML framework itself.

The PML control layer is implemented using the MPICH2 message-passing library for parallel programming [11], which is widely supported on several parallel platforms, ranging from multicore SMP microprocessors to LAN-based workstation clusters to distributed-memory HPC platforms. In particular, on distributed-memory HPC platforms such as the IBM Blue Gene/P, the preferred programming model is message-passing which has been widely adopted for scientific computing applications, and most HPC vendors provide highly-optimized, custom versions of the MPICH2 library [3]) for their computing platforms.

The basic parallelization ideas in PML, however, do not depend on either the MPICH2 library, or the message-passing model. In fact, PML was initially developed as an extensible analytical engine for embedded predictive modeling algorithms in a parallel relational database [2] based on stored procedures and user-defined functions, but with essentially the same parallelization model. Similarly, the PML control layer can also be easily adapted to distributed computing infrastructures such as Hadoop [7], which provide an alternative and complementary platform to HPC and parallel databases for machine learning algorithms with massive data sets.

Several well-known machine learning algorithms have been implemented in PML, including K-Means for clustering, PCA (principal components analysis) for feature transformation, and SVM (support vector machines) for binary classification. The description of these implementations can be found in Appendix A of the PML User Guide [17].

In this paper, we have focused on the Transform Regression (TREG) algorithm [13] in view of its specific novelty, scalability and wide applicability, and since in particular, the implementation of this algorithm illustrates and exemplifies many of the capabilities in the PML framework. The TREG algorithm has a wide variety of potential applications, which include for example, least squares and logistic regression, counts regression, quantile regression and support vector machines. As a consequence, the TREG implementation in PML provides a unique capability for fast, flexible and accurate regression modeling for massive, high-dimensional data sets.

The outline of this paper is as follows. Section 2 is an overview of the PML programming API and computational model. Section 3 is an overview of the IBM Blue Gene/P architectural issues that are relevant to the important performance issues for PML on massively parallel platforms. Section 4 describes the TREG algorithm, particularly in the context of parallelism within the PML framework. Section 5 presents numerical results for TREG model quality, along with performance and scalability results. Section 6 provides the summary discussion.

2 Description of PML Framework

2.1 PML Programming Interface for Algorithms

A schematic of the PML computational model is shown in Figure 2.1, which is based on the abstract PML *Model* object with the virtual methods *BeginDataScan*, *ParallelDataScan* and *EndDataScan* (another important virtual method *MergeDataScan*, is not explicitly shown Figure 2.1, but is discussed further below). The PML control layer continuously executes the loop in Figure 2.1 until termination, and depending on the specified state for the *Model* object in the control layer (e.g., *Training*, *Calibration*, *Evaluation* etc.), the appropriate concrete methods are dispatched during each loop execution.

At the beginning of each execution loop , the *BeginDataScan* method is invoked by master process, in order to initialize the model sufficient statistics in the *Model* object, which is then propagated to each of the worker processes using the *ParallelDistribute* function. Each worker process then scans a pre-defined, independent row-partition of the input data, and accumulates its partial contribution to the required model sufficient statistics. Since in practice, each *Model* object will in turn contain other embedded *Model* objects, and since the *DataPoint* object corresponding to each concrete *Model* object may contain derived or computed fields that are not part of the input data set, therefore, in the PML design, the appropriate *DataPoint* objects are "pushed" to each individual *Model* object, as shown in Figure 2.1. At the end of the parallel data scan, the distributed *Model* objects are aggregated, using the abstract method *MergeDataScan* mentioned earlier, within the *ParallelMerge* function. The master processor then executes the *EndDataScan* method, in which the accumulated sufficient statistics are processed. Further data scans are then carried out as required, until the final model convergence is obtained, and the execution loop is exited.

The concrete PML *Model* objects for the different algorithms that use this API do not require any explicit parallel programming, and all the required parallelization details are encapsulated within the PML control layer itself. Similarly, no explicit programming is required for the object serialization and materialization which is required for the inter-processor object communication, and this functionality is also obtained in a straighforward way by using and extending the PML object hierarchy, as described below.

2.2 PML Class Library Overview

The PML class library is implemented in C++, and all PML objects are thread-safe and exception-safe. The class library provides smart pointers and container classes for prevent-

```
Model model;
DataPoint datpt;
while(true) {
    if (model.BeginDataScan()) {
        ParallelDistribute(model);
        while (datpt) {
            model.ParallelDataScan(datpt);
        }
        ParallelMerge(model);
        if (model.EndDataScan()) break;
    }
}
```

Figure 1: Overview of PML Control Layer

ing memory leaks, detecting memory clobbering, and for tracking memory usage. The tracking of memory usage is used internally in PML for footprint-aware memory management, so that applications execute within the processor memory limits, and avoid any potential virtual-memory paging that can degrade application performance. In addition, many distributed-memory HPC platforms do not support virtual memory on the individual node processors. For most PML applications, these memory usage restrictions can be easily traded off by keeping the working set size small, and instead performing additional input data scans, which is far more efficient from a computational performance standpoint.

The PML container classes also provide support for object serialization and materialization, in binary as well as text formats (the latter being used for debugging purposes). The object serialization/materialization interface is extensible and fully integrated with the memory management functions, and is used for the coarse-grained, communication of objects between individual processor nodes using the ParallelDistribute and ParallelCollectandMerge functions, and therefore abstracts the details of the communication layer on which these functions are implemented in the parallel platform (which is typically a message-passing layer for distributed-memory HPC).

2.3 Feature discovery, Discretization and Statistics

In many large datasets, the range and distribution of the values of one or more categorical data fields in the input data may not be explicitly specified in the metadata, and these characteristics must therefore be discovered by examining the data itself. In addition, in some cases, it is often unclear whether certain variables are intrinsically numerical, or possibly number-valued categorical fields of high cardinality (e.g., address zipcodes). Finally, many machine learning algorithms require certain distributional information for the discretization, sampling or approximation of the individual data fields in the input data.

In PML, the processing of the input data fields to extract this range and distributional information is carried out in a preliminary parallel input data scan, in which, for each input data field, the empirical cumulant distribution is estimated from the sampled subset of the values, and the equispaced quantiles of this cumulant distribution is then used to discretize the fields values into bins. For each bin in this discretization, the min, max and average values of the field values are also stored, which can be used subsequently to generate random samples from these input fields, based on an approximation to their empirical marginal distribution. These random samples generated from these approximations are used for example in the variable-importance ranking computation described in Section (4.5).

2.4 Predictive Model Composition and Feed-forward Cascades

An important capability of PML, is the ability to embed one PML *Model* object inside another, which enables arbitrary heirarchies or cascades of predictive models to be created. In particular, the outputs of a predictive model based on a certain algorithm can be chained to the inputs of another predictive model which may be based on a possibly different algorithm. An example of this capability is provided by the TREG algorithm described in Section (4).

2.5 Cross-validation, Validation and Holdout data sets

PML supports model selection via cross-validation and model accuracy evaluation using holdout data, which are common requirements for many machine learning algorithms. In PML, each individual data record, which consists of a collection of input data-field values, is consistently and uniquely mapped by the PML I/O layer to one of $n_{cv} + 2$ data channels, comprising of the n_{cv} “cross-validation” channels, the “validation” channel, and the “hold-out” channel. The data in the hold-out channel, which is not used for model training, provides the unbiased “test data” estimate of the final model accuracy. The data in the validation channel is used in the model training to estimate the accuracy and determine convergence, while the data in the n_{cv} cross-validation channels are used for the model selection.

The assignment of input data records to data channels is based on the data-field values in the input data record, which, in the absence of duplicate data records, ensures a unique and consistent mapping assignment, that does not depend on the order in which data records are accessed, or even on the partitioning of the input data set among different processors. The mapping assignment is based on a composite uniform hash code into the unit interval, evaluated from the set of data-field values in the input data record; this hash-code output interval is partitioned into a set of $n_{cv} + 2$ non-overlapping bins, and the bin sizes can be adjusted to change the proportion of the data records to be assigned to each input data channel.

An example of how individual algorithms can use this channel identifier for model selection, estimation, and accuracy evaluation, is also described for the TREG algorithm in Section (4.1).

3 PML Performance Optimizations

The PML framework incorporates certain performance optimizations for distributed-memory HPC platforms, including in particular, for the IBM Blue Gene/P system, whose details are briefly described below.

3.1 Blue Gene/P overview

The IBM Blue Gene/P (BG/P) system is a distributed memory supercomputer [1], in which each individual processing node is a symmetric multiprocessor (SMP) comprising of 4 individual 850 MHz PowerPC processors, each of which has private write-through L1 caches (32 KB I-cache and 32KB D-cache), private L2 caches (seven-stream prefetching), along with a shared L3 cache (8 MB) and 2-4 GB of shared main memory on the node. Each PowerPC 450 processor has a dual-pipeline FPU unit that can simultaneously execute 2 fused multiply-add instructions per machine cycle per processor, so that the peak node performance when these dual FPU units are fully utilized is 13.6 Gflops per BG/P node. A single rack of BG/P comprising of 1024 nodes has a collective peak performance of 13.9 Tflops and aggregate memory of 2-4 TeraBytes. The largest BG/P configuration can comprise of upto 256 individual racks, with a collective peak performance of roughly 3.56 Petaflops.

There are two main communication networks on BG/P, a 3-D Torus network which is used for point-to-point and multicast communication, and a Tree network that is used for collective communication operations such as MPI broadcast/reduce.

The Torus network has a hardware latency of less than 1 microsecond per link hop, and a bidirectional link bandwidth of 425 MB/s (and since each node can be simultaneously sending or receiving to its nearest neighbors on 6 channels on this torus network, the aggregated bandwidth per node is 5.1 GB/s). On the individual nodes, the Torus network communication can be off-loaded to a cache-coherent DMA engine, so that there is effective support for overlapping of communication with computation on the nodes. Multi-hop message transmissions such as broadcasts incur little or no extra overhead on the torus network due to specialized hardware-support for dynamic cut-through and message copy-forwarding.

The Tree network has a worst-case round trip message latency of about 5 microseconds and 850 MB/s of bidirectional bandwidth per link (since each BG/P node can be sending and receiving simultaneously on all its 3 links on this network, the peak aggregate bandwidth per node is 5.1 GB/s). There is no DMA engine support for the Tree network, particularly since the communication on this network is usually global and blocking in any case.

The BG/P system has a set of dedicated I/O nodes which are used to communicate with an external file system and host computers through a 10-Gigabit Ethernet interface. These I/O nodes are independent and functionally different from the compute nodes, and their number is configurable upto a maximum of 1 I/O node per 16 compute nodes, which is the configuration with the best I/O performance. The use of the high-capacity GPFS file system on BG/P is of special interest for I/O-intensive machine learning applications, since this file system supports high-throughput concurrent, interleaved access to application files, and supports POSIX as well as MPI-IO library calls.

Although BG/P applications can use a variety of programming models, the MPI message-passing model is the preferred approach, particularly when application performance is the critical issue. In particular, the generic MPI communication primitives are customized to the BG/P hardware by writing them in terms of the the Deep Computing Messaging Framework (DCMF) layer for the point-to-point messaging, and the Component Collective Messaging Interface (CCMI) for collective communications, and the resulting hardware-optimized versions of these generic primitives provide the best communication performance for MPI-based user applications.

Application programs for BG/P are compiled using the gcc compiler, or alternatively using the XL compilers for FORTRAN, C and C++ (however, the instruction generation for the dual FPU units on the node processors was only supported by the XL compiler). The XL compilers also support OpenMP directive for compiler parallelization, so that a hybrid programming model is possible, with multithread parallelism being used on the SMP node processors, and MPI being used for inter-node parallel communication and synchronization.

3.2 PML Model Distribution and Collection

The *ParallelDistribute* and *ParallelCollectAndMerge* functions are customized for distributed-memory HPC platforms with a large number of processors p, so that these operations are performed in $\log_2 p$ phases, with significant parallelism in each phase. The implementation of these collective operations is similar to that of the broadcast/reduce primitives in MPI, but with PML-specific methods being used for matching and merging objects, and for object serialization and materialization.

From a performance viewpoint, for large p, such an implementation of the required collective communications, is far superior to the equivalent naive implementation in which each worker processor directly communicates with the master processor, since the latter is essentially serialized on the master processor, and is therefore clearly unsuitable for large p. Furthermore, the individual communication steps in the former approach take place on a virtual binary-tree of processors rooted at the master processor, and therefore the individual communication steps in this collective operation can be scheduled to ensure a conflict and contention-free utilization of the maximum interprocessor bandwidth and communication ports in the torus interconnection network of BG/P.

3.3 PML Data Conversion and Data I/O

PML requires the input data files to be in the binary format, and a separate data conversion utility is provided for other common text formats, such as comma-separated values (CSV) or the attribute-relation file format (ARFF) used in WEKA (`http://www.cs.waikato.ac.nz/~ml/weka/arff.html`). This data conversion utility is also a parallel progam, as this otherwise this step would be the computational bottleneck for massive data sets.

The input data file is partitioned row-wise among the processor nodes, and during each data scan, the records in each partition are individually scanned, and the required data fields are pushed to the corresponding model objects for updating the relevant model sufficient statistics, as described in Section 2.1. In practice, for I/O efficiency, the input data is transferred from the attached file system in large blocks comprising of several records, to in-memory file buffers on the processor nodes. Furthermore, for large p, the individual partitions of the input data assigned to each node processor are often small enough to be stored entirely within these in-memory buffers, which obviates the need for any disk accesses after the first input data scan. The PML Data I/O layer supports this block transfer and memory buffering of disk files for greater I/O efficiency on distributed-memory HPC platforms.

3.4 PML Data Redistribution

Most machine learning algorithms implemented in PML only use the basic API described in Figure 2.1, in which worker processes access only their independent partitions of the input data in each data scan. However, a few algorithms (e.g., certain versions of SVM's) require each worker process to access the entire data set in each data scan.

In the latter case, the I/O costs can be dramatically reduced whenever the partitioned data set fits entirely in the individual node processor memory, as described in 3.3. Starting with the initial partitioning stored in the in-memory buffers, each data partition can be shuffled across the set of p processors, so that at the end of the p shuffle steps, each processor has accessed the entire data set, after the initial disk access I/O operation, only the faster internal network of the HPC platform is used. In this way, each loop in Figure 2.1 is executed p times to effect a data scan over the entire data set, and the only distinction is the the state in the worker processes is retained between these p data scans, unlike in Figure 2.1, where the state of the worker processes was initialized at the beginning, and the model objects were merged and destructed at the end of each iteration.

3.5 Parallel Performance and Scalability

The parallel performance of PML applications on a distributed-memory HPC platform can be characterized by two measures, viz., strong and weak scalability, whose definition and relevance for machine learning applications is considered in this section.

If T_p denotes the measured execution time for a parallel application on p nodes, and T_b denotes the equivalent baseline time, then the parallel speedup is defined by $S_p = bT_p/T_b$, and the parallel efficiency by S_p/p.

An objective measure for the baseline time T_b is the equivalent, optimized serial program time on a single node ($b = 1$). However, this equivalent serial program is often not available for performance benchmarking, and even when available, the execution time may be too long for the massive data sets of interest. For these reasons, in practice, the baseline performance is often obtained using the parallel program itself, with the smallest possible number of nodes that yields a reasonable execution time. On Blue Gene/P, for many of the data sets that we have examined, this baseline T_b is typically obtained with 16 or 32 nodes. The definitions of the parallel speedup and parallel efficiency are then based on extrapolating the baseline performance from b nodes to a single node.

For the strong scalability metric, the data size and various algorithm parameters are held fixed as p is increased, so that the memory requirement per node decreases. In contrast, for the weak scalability metric, the data size is also increased as p is increased, in such a way that the memory requirement per node remains fixed.

The strong scalability analysis is relevant when p is increased in order to reduce the execution time for a given problem of fixed size. Here, in the ideal case, the speedup S_p will be linearly proportional to p, but the relative increase in the cost of the serial fraction of the problem, as well as the increased overhead of communication, will lead to S_p eventually leveling off, and even decreasing, as p is increased. In certain cases, there may be an apparently anomalous increase in S_p, which is usually attributable to the dramatic improvement in the node computation peformance as the data partitions on each node become small enough to fit entirely into a higher level of the processor storage heirarchy (e.g., main memory for disk based data, or the processor cache for main-memory based data, as the case may be).

On the other hand, the weak scalability analysis is relevant when p is increased, so that larger problems can be modeled in a fixed execution time. For weak scalability, in the ideal case, the speedup S_p stays close to 1 as p is increased, and although the ratio of communication to computation remains fixed in a weak scalability analysis, the increased latency of global synchronization, and the decrease in the bisectional bandwidth of the communication network, as p is increased, will invariably lead to S_p dropping off below unity.

4 Transform Regression

The TREG algorithm (Pednault [13]) is described in this section, with an emphasis on its applicability to a broad class of regression problems for massive data sets using the PML framework.

4.1 Overview

The input data record for regression modeling is denoted by $\{\boldsymbol{x}, y\}$, where $\boldsymbol{x} = [x_1, x_2, \ldots, x_M]$, and the covariates x_j are such that $x_j \in \mathbb{R}$ for continuous numeric features, or $x_j \in \tau_j$ of cardinality $|\tau_j|$ for categorical features. Similarly, y denotes the response field, which can take values in $\mathbb{R}$ for ordinary, median, robust and quantile regression, in the set $\{-1, 1\}$ for binary classification and logistic regression, or in the set of non-negative integers $\{0, \mathbb{N}_+\}$ for Poisson regression.

The collection of N input data records in the $n_{cv} + 1$ folds of the cross-validation and validation channels, denoted by $\{y_i, \boldsymbol{x}_i\}_{i=1}^N$, is the training data for obtaining the regression model. The regression function $\hat{f}(\boldsymbol{x})$ for the desired response mean $E(y|\boldsymbol{x})$ is then obtained by minimizing the appropriate loss function over this training data,

$$\hat{f}(\boldsymbol{x}) = \arg\min{}_f \mathcal{L}(f), \tag{1}$$

where

$$\mathcal{L}(f) = N^{-1} \sum_{i=1}^{N} L(y_i, f(\boldsymbol{x}_i)). \tag{2}$$

Since the regression function $\hat{f}(\boldsymbol{x})$ from (1) and (2) may overfit the data, an unbiased estimate of the resulting model accuracy is obtained by re-evaluating (2) on the N_H data records in the hold-out data channel.

Table (4.1) shows the loss functions $\mathcal{L}(f)$ for several applications, which include least-squares, median and robust regression, binary logistic regression, SVM-based binary class estimation, poisson regression and quantile regression. The loss functions in Table (4.1) are in all cases, convex functions of the argument f over which the optimization in (2) is performed.

4.2 Motivation

In order to motivate the TREG algorithm, we consider Friedman's Gradient Boosting (GB) algorithm [6, 9], in which the regression function $\hat{f}(\boldsymbol{x})$ in (2) is obtained as a stagewise expansion

Response Type and Range	Loss Function, $L(y, f)$
Least Squares regression, $y \in \mathbb{R}$	$(y - f)^2$
Median regression, $y \in \mathbb{R}$	$\lvert y - f\rvert$
Robust regression, $y \in \mathbb{R}$	$\begin{cases} (y-f)^2, & \lvert y-f\rvert < \delta, \\ \delta(2\lvert y-f\rvert - \delta), & \lvert y-f\rvert \geq \delta \end{cases}$
Logistic two-class probability, $y \in [-1, 1]$	$\log(1 + \exp(-2yf))$
Poisson regression, $y \in \{0, \mathbb{N}_+\}$	$-yf + \exp(yf)$
SVM (two-class), $y = \{-1, 1\}$	$\max(0, 1 - yf)$
τ-Quantile regression, $y \in (-\infty, +\infty)$	$\begin{cases} \tau(y-f), & y-f > 0, \\ (\tau-1)(y-f), & y-f \geq 0 \end{cases}$

Table 1: Loss functions used in Transform Regression

of the form

$$f_T(\boldsymbol{x}) = \sum_{t=1}^{T} \alpha_t g_t(\boldsymbol{x}; \theta_t). \tag{3}$$

The functions $g_t(\boldsymbol{x}; \theta_t)$ in (3), which are often termed "weak learners" in gradient boosting terminology, are typically selected from a family of low-order adaptive basis functions parameterized by θ_t.

The stagewise computation of (3), as shown in Figure 4.2, consists of a two-step procedure in each stage. In the first step, the parameters θ_t are estimated from a least squares regression fit (4) to the pseudo-residuals of the loss function (2) evaluated at the previous stage. Then, in the second step, the coefficient α_t in (3), is obtained from a univariate line-search optimization (5). The case of the squared-error loss function in Table (4.1) is special, since at each stage, the pseudo-residuals $r_{t,i}$ are exactly the usual stage residuals $y_i - f_t(\boldsymbol{x}_i)$, and the optimal coefficient α_t is exactly 1.

The following four aspects of the GB algorithm ([6]) are relevant for motivating the TREG algorithm.

First, the stage basis functions $g(\boldsymbol{x}, \theta_t)$ in (4) are themselves regression models obtained from the stagewise squared-error loss function (irrespective of the overall loss function (2), which is typically chosen from Table (4.1 based on the specific application). The use of the stagewise squared-error loss function is computationally advantageous since the required $g(\boldsymbol{x}, \theta_t)$ can be computed in a rapid, modular fashion. Typically, the computed stage basis functions take a very simple form, e.g., regression stumps of small depth, in which the parameters θ correspond to the split conditions and response mean estimates in the leaf nodes of the regression stumps. The stagewise squared-error loss function in (4) implies that $g(\boldsymbol{x}, \theta_t)$ is the specific member of

INITIALIZATION: Let $f_0(\boldsymbol{x}) = 0, t = 0$.

ITERATION: For $t = 1$ to T, let $r_t = (\partial\mathcal{L}(f)/\partial f)_{f_{t-1}(\boldsymbol{x})}$ denote the stage pseudo-residuals.

1. Compute the stage basis function $g_t(\boldsymbol{x};\theta_t)$ from the least squares problem

$$\theta_t = \arg\min_\theta \sum_{i=1}^{N}(r_{t,i} - g_t(\boldsymbol{x}_i;\theta))^2. \tag{4}$$

2. Compute α_t by solving the univariate optimization problem

$$\alpha_t = \arg\min_\alpha \sum_{i=1}^{N} L(y_i, f_{t-1}(\boldsymbol{x}_i) + \alpha g_t(\boldsymbol{x}_i;\theta_t)). \tag{5}$$

3. Set $f_t(\boldsymbol{x}) = f_{t-1}(\boldsymbol{x}_i) + \alpha_t g_t(\boldsymbol{x};\theta_t), t = t+1$ and repeat.

TERMINATION: Final model $f(\boldsymbol{x}) = f_T(\boldsymbol{x})$.

Figure 2: Overview of Gradient Boosting algorithm

the stage basis-function family that is maximally correlated with pseudo-residual evaluated at the previous stage $t-1$. However, one disadvantage of using regression stumps as the stage basis functions is that even the simple response surface which just has a linear dependency on the covariates, requires a large number of regression stump basis functions in (3) for a satisfactory model fit.

Second, the univariate line optimization step in (5) does not have to be solved to optimality, as the subsequent stages in (3) can always reintroduce the same basis function, so that in this sense, the GB algorithm is ‘self-correcting.” In addition, the use of sub-optimal values for α_t may even be desirable, particularly in the earlier stages, in order to avoid early overfitting in (3). This is very similar to the use of explicit shrinkage factors, which has been mooted in [6] to avoid early-stage basis function overfitting.

Third, in Figure (4.2), the coefficients α_1 through α_{t-1} in (3) are not readjusted, when the new basis function $g(\boldsymbol{x}, \theta_t)$ is added to the model in stage t of Figure (4.2). This is in contrast with other statistical stagewise regression procedures, where the coefficients of the existing regressors are always readjusted to reflect the addition or removal of regressors from the model.

Fourth, the choice of the basis function family $g(\boldsymbol{x}, \theta)$ potentially leads to systematic errors in the model fit, particularly in the case when the regression stumps are used, and where the response surface has significant covariate interaction effects. For example, in (3), regression stumps can only model additive effects in the covariates, and trees of depth 2 are required for modeling first-order covariate interactions. For a given regression problem, the nature of the response surface is invariably unknown, so that some experimentation is required in order to find the right basis function set for gradient boosting. Similarly, if the “correct” basis function family $g(\boldsymbol{x}, \theta)$ for the required regression surface is not used, then the final gradient boosting model will have systematic errors that cannot be removed by simply adding more stages to the GB expansion (3).

The TREG algorithm is similar to the GB algorithm Figure (4.2), but with differences in these four aspects mentioned above.

4.3 TREG Algorithm Description

There are many variations and subtleties in the implementation of the GB algorithm [5, 6, 15], such as the choice of expansion basis functions, the use of learning rate parameters and data sub-sampling for the stage computations, and the tuning of the finer details of the line-search optimization procedures. In general, however, the overall GB procedure is quite unsuitable for parallel computing with massive data sets.

The two main reasons for this are the fine-grained nature of the computations performed in each individual stage, and the inherently sequential and I/O-inefficient nature of the stagewise expansion procedure. Therefore, the TREG algorithm may be viewed as a modification which provides a regression model of equivalent accuracy in most cases, while addressing the parallel computing limitations in two ways; by increasing the computational granularity of each individual stage, and by reducing the number of overall sequential stages in the stagewise expansion.

Therefore, in TREG, the "weak learner" basis functions of the GB algorithm are replaced by more complex "multivariate smoother" basis functions, which reduces the overall number of stages in (3), since each TREG stage is potentially comparable to several stages of the equivalent GB formulation. The specific multivariate smoothers used in TREG are carefully chosen to avoid any possibility of overfitting in the computation of these more-complex stage basis functions, and to have sufficient computational granularity so as to permit an efficient parallel implementation of the stage calculations. In summary, in the TREG approach, the choice of the stage basis function in the form of a multivariate smoother, makes it is possible to obtain an accurate regression model with far fewer training data scans, and with far greater overall parallelism, than would be possible in the fine-grained and sequential GB algorithm.

The specific stage basis function used in TREG relies crucially on the ability of PML framework to set the outputs of predictive models as "computed fields" in an "extended" input data record, which can then be used as dynamically-generated features for other predictive models, which is used in the TREG algorithm, as described below.

The GB algorithm itself is an example of a feed-forward cascade, albeit with a very simple structure, in which the output of the predictive model at the previous stage $t-1$ (or more correctly, the pseudo-residual based on this output), is taken as the response variable for generating the basis functions at stage t. In TREG, in addition to a similar usage of the previous-stage output, all the previous-stage model outputs are also used in stage t as additional explanatory variables, so that the final TREG model involves a more complex feed-forward cascade of composite model transformations compared to the GB algorithm.

The stagewise basis functions used in TREG are explicitly given by

$$g_t(\boldsymbol{x}, \boldsymbol{\lambda}) = \sum_{j=1}^{M} \lambda_j h_{tj}(x_j; f_1, \ldots, f_{t-1}) + \sum_{j=1}^{t-1} \lambda_{M+j} \hat{h}_{tj}(f_j; f_1, \ldots, f_{j-1}), \tag{6}$$

which is a linear combination of certain $M + t - 1$ sub-basis functions with coefficients $\boldsymbol{\lambda} = \{\lambda_j\}_{j=1}^{M+t-1}$. Each sub-basis function in (6) is conceptually similar to the regression stumps used as stage basis functions in the GB algorithm. However, piecewise-linear regression stumps are used in (6), which subsumes the piecewise-constant regression stumps used in the GB algorithm as a special case, and is especially useful for succintly modeling linear covariate dependencies that are common in regression problems, for which a large number of basis functions would be required in the GB algorithm for equivalent accuracy.

The notation $h_{tj}(x_j; f_1, \ldots, f_{t-1})$ in (6) denotes a piecewise-linear regression stump, which is computed using the pseudo-residual of the loss function computed at the previous stage $t-1$ as the response, and in which x_j is the only explanatory variable used for splitting, while $\{x_j, f_1, \ldots, f_{t-1}\}$ are the only explanatory variables allowed in the segment regression models (however in the case of categorical x_j, this variable is used only as a splitting variable, and is not included in the segment linear regression models). Therefore, for continuous x_j, the relevant sub-basis function in (6) can be written in the form

$$h_{tj}(x_j; f_1, \ldots, f_{t-1}) = \sum_{k=1}^{K_t} (a_{k0j} + a_{k1j} f_1 + \ldots + a_{k(t-1)j} f_{t-1} + a_{ktj} x_j) I([x_j \in \Gamma_{t,k}]), \qquad (7)$$

where K_t is the number of leaf-node segments, $\Gamma_{t,k}$ denotes the k'th leaf-node segment, and $I([x_j \in \Gamma_{t,k}])$ denotes the indicator function for leaf-node membership (the set of segments $\{\Gamma_{t,k}\}$ is a mutually-exclusive, collectively-exhaustive univariate partition of x_j at stage t). For categorical x_j, these sub-basis functions (7) have the same form but will exclude the term with coefficient a_{ktj}. The required piecewise-linear sub-basis functions (7) in (6) can be computed independently in parallel, as described further below.

The sub-basis functions (7) are chosen so as to reduce the number of stages in the stagewise expansion (3). First, the use of the outputs of the previous $(t-1)$ stages in (3) as explanatory variables in the segment regression models $\{h_{tj}(x_j; f_1, \ldots, f_{t-1})\}_{j=1}^{M}$ in (7), implicitly "orthogonalizes" against all the previous-stage, sub-basis functions, and in this way eliminates redundancy from the expansion (3). Second, as a consequence of this, these previous-stage model outputs are included as splitting variables for $\{\hat{h}_{tj}(f_j; f_1, \ldots, f_{j-1})\}_{j=1}^{t-1}$ in (7), so that while the previous-stage coefficients of these sub-basis functions in (3) are unmodified, there is an implicit readjustment in the current stage that reflects the addition of each new sub-basis function $g_t(\boldsymbol{x})$ at stage t to the overall model expansion.

These two refinements mentioned above, can be motivated by analogy with stepwise linear regression procedures, in which new regressors are implicitly or expliicitly orthogonalized against the existing regressors to avoid introducing redundant or collinear features into the regression model. Furthermore, in these stepwise regression procedures, the coefficients of existing regressors are also re-adjusted to reflect the addition (or removal) of regressors from the model. Thus, (6) is a nonlinear generalization of these ideas, which can lead to a substantial reduction in the overall number of stages in (3), with concomitant savings in the I/O and computational costs, since in fact, the required stage computations for these refinements can be implemented in parallel, without any extra data I/O overhead.

In particular, the second refinement above is of particular interest in reducing the systematic errors for response surfaces $f(\boldsymbol{x})$ with potential non-additive, covariate interaction effects that may not be directly captured by the untransformed set of piecewise-linear basis functions used in (3). In TREG, the resulting systematic errors can be reduced or even eliminated by incorporating the entire set of previous-stage basis functions as splitting variables in the sub-basis functions $\{\hat{h}_{tj}(f_j; f_1, \ldots, f_{j-1})\}_{j=1}^{t-1}$ at stage t, which not only provides a mechanism for implicit readjustment of the coefficients of the previous stage basis function $\{f_1, \ldots, f_{j-1}\}$, but in addition, the piecewise-linear regression tree transformations of the previous stage basis functions implicitly introduces cross-product terms into the regression function corresponding to the non-linear transformation of the sums of previous stage basis functions in (3). The resulting ability of (3) to model non-additive covariate effects that may not have been explicitly included in the the stage basis functions has been illustrated in (Pednault [13] using a synthetic example, and a beuristic explanation for this property was given by analogy between (3) and the form of the universal approximating functions in the Kolmogorov Superposition Theorem (a similar analogy has also been used to motivate the form of multi-layer neural networks in Hecht-Nielsen [10]). The use of input features that incorporate nonlinear transformations of the previous stage outputs in each stage in (3), and the resulting ability of this expansion to model regression functions involving complex variable interactions, is the rationale for the name of the TREG algorithm.

INITIALIZATION: Let $f_0(\boldsymbol{x}) = 0, t = 0$.

ITERATION: For $t = 1$ to T, let $r_t = (\partial\mathcal{L}(f)/\partial f)_{f_{t-1}(\boldsymbol{x})}$ denote the stage pseudo-residuals.

1. Compute the $M + t - 1$ stage sub-basis functions (7), $\{h_{tj}(x_j; x_j, f_1, \ldots, f_{t-1})\}_{j=1}^{M}$ and $\{\hat{h}_{tj}(f_j; f_1, \ldots, f_{j-1})\}_{j=1}^{t-1}$ with r_t as response, using Section (4.4.1)
2. Compute the stage basis function (6), $g_t(\boldsymbol{x}, \boldsymbol{\lambda}_t)$ by solving the least-squares problem
$$\boldsymbol{\lambda}_t = \arg\min_{\boldsymbol{\lambda}} \sum_{i=1}^{N} (r_{t,i} - g_t(\boldsymbol{x}_i, \boldsymbol{\lambda}))^2, \tag{8}$$
using Section (4.4.2).
3. Compute α_t by solving the univariate optimization problem
$$\alpha_t = \arg\min_{\alpha} \sum_{i=1}^{N} L(y_i, f_{t-1}(\boldsymbol{x}_i) + \alpha g_t(\boldsymbol{x}, \boldsymbol{\lambda}_t)), \tag{9}$$
using Section (4.4.3).
4. Set $f_t(\boldsymbol{x}) = f_{t-1}(\boldsymbol{x}) + \alpha_t g_t(\boldsymbol{x}, \boldsymbol{\lambda}_t), t = t + 1$ and repeat.

TERMINATION: Final model $f(\boldsymbol{x}) = f_T(\boldsymbol{x})$.

Figure 3: Overview of Transform Regression algorithm

The initialization and termination of the TREG algorithm Figure (3), along with the three main steps in the iteration, comprising of the computation of the sub-basis functions (1), the computation of the stage basis function (8), and the line optimization step (9), are all individually described below.

4.4 TREG Expansion: Initialization and Termination

The expansions in Figure (3) can be initialized in alternative ways besides $f_0(\boldsymbol{x}) = 0$. For example, in the least squares regression application, $f_0(\boldsymbol{x})$ can be set to the unconditional response mean in the training data. In our implementation of TREG, $f_0(\boldsymbol{x})$ is taken as the linear regression model based on the continuous explanatory variables, which can be computed as a "side-effect" of the initial scan over input data which primarily for estimating the distributional properties of the individual input data fields (this initial data scan is required for category discovery, and for discretizing the continuous fields for the segmented tree algorithms used in subsequent stages as described in Section (4.4.1)).

Similarly, several alternative criteria can be used for the terminating the stagewise expansion in Figure (3), including user-specified criteria such as the maximum number of stages, or the overall computational time. The preferred termination criterion, however, is based on the estimates of the cross-validation loss (2) and the variance of this loss, which are obtained at each stage during the expansion, and statistical significance tests can be used to determine whether a given sequence of successive stages has lead to no significant decrease in this loss. The expansion is then terminated and the stagewise expansion is pruned back to the number of stages with the minimum cross-validation loss estimate.

4.4.1 Details of Sub-basis Function Computations

In PML, continuous-valued features x_j are pre-discretized into intervals after estimating certain "knot" points in the range of x_j (by default, these knot points are based on the equi-spaced quantiles of the empirical cumulative distribution of x_j), so that each segment $\Gamma_{t,k}$ in (7) is then a collection of contiguous intervals in this discretization. For categorical-valued variables x_j, the feature values are unordered, and so each segment $\Gamma_{t,k}$ is an arbitrary collection of one or more category values in x_j.

The computation of the stage sub-basis functions $h_{tj}(x_j; f_1, \ldots, f_{t-1})$ (7) is shown in Figure 4, where the final segmentation $\Gamma_{t,k}$ along with the corresponding segment linear models is generated in a single pass over the training data using a parallel version of the Linear Regression Trees (LRT) algorithm described in [14]. This parallelization of the LRT algorithm using the PML API requires two extra steps, as shown Figure (4), and provides an illustrative example of how existing serial machine learning algorithm can be incorporated into PML with some simple modifications.

In Figure (4), the range of x_j is assumed to be pre-discretized into the intervals $\Delta x_{jl} = (x_{j(l)}, x_{j(l+1)}\})$. The data fields required for computing the LRT model corresponding to $h_{tj}(x_j; f_1, \ldots, f_{t-1})$ are denoted by $\boldsymbol{\xi}_{t,j} = \{x_j, f_1, \ldots, f_{t-1}, r_t\}$ which consists of an input data feature x_j and the computed features $f_1 \ldots f_{t-1}$. The response for each of these individual LRT models is the pseudo-residual from the previous stage r_t.

The LRT algorithm heuristically minimizes the negative log-likelihood for the linear Gaussian model in the final segmentation $\Gamma_{t,k}$ using the following procedure. For a given interval Δx_{jl} in the pre-discretization of x_j, the required sufficient statistics for the linear Gaussian model are the sample counts N_{tjl}, and the multivariate sample means $\boldsymbol{\mu}_{tjl}$ and covariance $\boldsymbol{S}_{tjl}$, computed over the relevant data fields $(r_t, \boldsymbol{\xi}_{t,j})$ for all the data records whose values x_j are in the interval Δx_{jl}). These sufficient statistics, which are computed separately for each of the $n_{cv} + 2$ data channels, can be obtained in a single input data scan. The only modifications required in the data-parallel case, are that each node updates a local copy of the sufficient statistics over its partition of the input data records, and these partial values are then combined to obtain the global sufficient statistics for the entire data set. The updating and the merging formulas that are required for the parallel computation of the sufficient statistics for $\boldsymbol{\mu}_{tjl}$ and $\boldsymbol{S}_{tjl}$ are given in [14].

The final step in the evaluation of the sub-basis functions, which is typically very fast, is performed serially on the master node. The initial segmentation is taken to be each interval Δx_{jl} in the pre-discretization of x_j, and the incremental cholesky-based algorithm for solving the normal equations as described in Section (4.4.2) (see also [14]), is used to obtain the linear models within each segment. The sufficient statistics in the n_{cv} cross-validation channels is used to estimate the degrees of freedom for each segment linear regression models. The model coefficients $\{a_{lsj}\}_{s=0}^{t+1}$ with required degrees of freedom, as the model variance σ_{lj} for each segment Δx_{jl}, is then obtained from the aggregated data in the n_{cv} channels. The sufficient statistics in the validation-fold channel then provides an unbiased estimate for the squared residual error $\tilde{\rho}_{jl}$, so that the segment contribution of negative log-likelihood for the linear Gaussian model is given by

$$\mathcal{G}_{tjl} = \frac{1}{2}\left[\log 2\pi\sigma_{jl}^2 + \frac{\tilde{\rho}_{jl}}{\sigma_l}\right], \tag{10}$$

using the estimates σ_{jl}^2 and $\tilde{\rho}_{jl}$ for the model variance squared residual error for the segment linear regression model.

Starting with this initial segmentation, a series of pairwise bottom-up combine steps is then performed to obtain the final segmentation in (7). For continuous x_j, the sufficient statistics of adjacent segment pairs are combined if the resulting segment leads to a reduction in the negative log-likelihood of the collection of segment linear Gaussian models. Specifically, if two adjacent segments are denoted by subscripts L and R, and their combined segment by $L+R$ respectively, and if the corresponding negative log-likelihoods for the linear Gaussian model are denoted by $\mathcal{G}_L$, $\mathcal{G}_L$ and $\mathcal{G}_{L+R}$ respectively, then

$$\Delta\mathcal{G} = N_{(L+R)}\mathcal{G}_{tj(L+R)} - (N_L\mathcal{G}_{tjL} - N_R\mathcal{G}_{tjR}), \tag{11}$$

is evaluated and the segment pairs for which $\Delta\mathcal{G}$ is maximally negative are combined. This bottom-up combine step typically does not increase the overall degrees of freedom in the collection of segment models, e.g., if n_L, n_R and n_{L+R} denote the degrees of freedom in the linear models in the respective segments, then typically $n_{L+R} \leq n_L + n_R + 1$. However, occasionally the number of degrees of freedom increases with $n_{L+R} > n_L + n_R + 1$, and in that case, the segment pair is not considered for combining. This bottom-up combine process is repeated successively until there is no further decrease in $\Delta\mathcal{G}$, which then yields the required final segmentation $\Gamma_{t,j}$ for the feature x_j in (7).

A similar bottom-up combine procedure is used if x_j is a categorical feature, where starting with the initial segmentation corresponding to each category level in τ_j, any two segments can be succesively combined in the bottom-up combine procedure (unlike the continuous feature case discussed above, where only adjacent segment groups in the discretization of x_j could be considered for combining).

Although the number of initial segment models in the LRT algorithm for obtaining all the sub-basis functions in (7) can be very large, the dimensionality of the individual segment linear regression models is quite small. Specifically, the number of features for each initial segment model is the dimension of $\boldsymbol{\xi}_{t,j}$ or $t+2$, where t is the number of previous stages in the TREG algorithm. Typically, the maximum number of stages in TREG is 5 to 10.

B̆eginDataScan: Master node - For each initial segment $\{\Delta x_{jl}\}_{l=1}^{L_j}$ of each feature x_j, initialize the sufficient statistics $\{N_{tjl}, \boldsymbol{\mu}_{tjl}, \boldsymbol{S}_{tjl}\}$ required for the LRT model for the response r_t.

P̆arallelDistribute: Distribute the model $f_{t-1}(\boldsymbol{x})$, and the initialized segment sufficient statistics.

D̆ataScan: Worker nodes - Scan the input data records, and update the local copies of the sufficient statistics from the data values x_j of each field in the input data record.

P̆arallelMerge: Merge and aggregate the sufficient statistics from the worker nodes on the master node.

Ĕ ndDataScan: Master node - For each feature x_j, starting from the initial segmentation $\{\Delta x_{jl}\}_{l=1}^{L_j}$, obtain the final segmentation $\Gamma_{t,j}$ using the bottom-up combine process -

1. For each segment Δx_{jl}, estimate the number of degrees in the linear regression model from the data in the cross-validation folds.
2. For each segment, combine the data in the cross-validation folds to estimate the regression coefficients and model variance.
3. For each segment, estimate the squared error from the data in the validation foldand compute the negative log-likelihood for the linear Gaussian model $\mathcal{G}$.
4. Combine any pair of admissible segments that lead to the maximum decrease in the negative log-likelihood for the linear Gaussian model in the combined segment.
5. Stop when there is no further improvement from combining and exit with the final segmentation $\Gamma_{t,j}$.

OUTPUT: For each feature x_j, the final segmentation for the sub-basis function $h_{tj}(\boldsymbol{\xi}_{t,j})$ is given by $\Gamma_{t,j}$.

Figure 4: Parallel one-pass algorithm for the piecewise-linear regression stumps used for the sub-basis function computation (7) in TREG.

4.4.2 Details of Stage Basis Function Computations

Once the sub-basis functions are computed as described above, the coefficients $\{\lambda_j\}_{j=1}^{M+t-1}$ in (6) are obtained from the least-squares model (8).

The data fields required for (8) are given by $\boldsymbol{\eta}_t = \{\{h_{t,j}\}_{j=1}^{M}, \{\hat{h}_{t,j}\}_{j=1}^{t-1}, r_t\}$ (these are all computed fields in terms of the sub-basis functions in Section (4.4.1) above). The required local copies of the sufficient statistics are updated in a parallel data scan, and then merged into the global sufficient statistics on the master node. This sequence is steps is exactly the same used for the independent segment models in the LRT algorithm Figure (4), unlike the numerous but small-dimensional linear regression models being solved in that case, here there is only one linear regression model of potentially very large dimension.

The incremental Choleski factorization procedure described in [14] is used for the solution of least-squares problem, and this procedure uses the sufficient statistics in the cross-validation folds to estimate the number of degrees of freedom to be used in the linear regression model. For each cross-validation fold, starting with a null model, a sequence of nested models is obtained by using a greedy forward-selection procedure, in which features are sequentially added to the model in order to best reduce the regression loss (8). The minimum of the cross-validation estimate of the loss then determines the number of degrees of freedom in the linear model. A variant of this is to select the degrees of freedom corresponding to the smallest model with a cross-validation loss estimate that is within one standard error of the best value (the 1-SE rule), and this incorporates some shrinkage to eliminate the possibility of overfitting while obtaining stage basis function $g_t(\boldsymbol{x}, \boldsymbol{\lambda})$ itself. Finally, the coefficients of the linear regression model with the specified number of degrees of freedom is then estimated from the combined data on the cross-validation folds.

4.4.3 Line Search Optimization Details

For the squared-error loss function in Table (4.1), the line optimization step (9) is not required, since the optimum α_t is always 1. For the other loss functions, (9) is equivalent to determining the coeffient α_t in a regression fit for a linear model with offset $f_{t-1}(\boldsymbol{x})$ with explanatory feature $g_t(\boldsymbol{x}, \boldsymbol{\lambda}_t)$. In the case of the logistic or the poisson loss functions, this amounts to solving a Generalized Linear Model for estimating the parameter α_t. A more general approach, that is applicable to all loss functions is univariate line-search optimization, for which robust algorithms have been extensively developed in the nonlinear optimization literature, and can be readily used with the GB algorithm [6], although for massive data sets, these algorithms will require multiple input data scans.

In TREG, the two approaches that have been used to estimate α_t, which can be implemented with just a single input data scan, although with multiple data scans, these estimates can be further refined if required.

The first approach, termed the "one-step multi-α" method, the objective function (9) is evaluated on a grid of K values $\{\alpha_k\}_{k=1}^{K}$ (typical values of K are 100), and α_t is chosen as the minimizer from this set,

$$\alpha_t = \arg\min_{\alpha_k} \sum_{i=1}^{N} L(y_i, f_{t-1}(\boldsymbol{x}_i) + \alpha_k g_t(\boldsymbol{x}_i, \boldsymbol{\lambda}_t)). \quad (12)$$

B̆eginDataScan: Master node - For the linear regression model with r_t as response, and $\boldsymbol{\eta}_t = \{\{h_{t,j}\}_{j=1}^{M}, \{\hat{h}_{t,j}\}_{j=1}^{t-1}\}$ as explanatory fields, initialize the sufficient statistics $\{N_t, \boldsymbol{\mu}_t, \boldsymbol{S}_t\}$.

P̆arallelDistribute: Distribute the models $f_{t-1}(\boldsymbol{x})$, the models $\boldsymbol{\eta}_t$ for the sub-basis functions, and the initialized sufficient statistics.

D̆ataScan: Worker nodes - For every data record, update the local copy of the sufficient statistics.

P̆arallelMerge: Merge the updated local sufficient statistics to obtain the global sufficient statistics.

Ĕ ndDataScan: Master node -

1. Compute a sequence of nested linear regression models from the data on the cross-validation folds and estimate the optimal number of degrees of freedom from the cross-validation loss.
2. Compute the linear regression model with the optimal degrees of freedom using the aggregated data on the cross-validation channels.

OUTPUT: The final linear regression model for $g_t(\boldsymbol{x}, \boldsymbol{\lambda})$.

Figure 5: Parallel one-pass Linear Regression algorithm used for the basis function computation (8) in TREG.

The second approach, termed the "one-step Newton" method, can be used for twice-differentiable loss functions (2), which require the derivatives of $\mathcal{L}(f)$ (2) evaluated at $\alpha = 0$, given by,

$$\begin{aligned}
\mathcal{L}_\alpha(f_{t-1}) &= \sum_{i=1}^{N} \left(\frac{\partial L(y_i, f)}{\partial f} \right)_{f_{t-1}} g_t(\boldsymbol{x}_i, \boldsymbol{\lambda}_t), \\
\mathcal{L}_{\alpha\alpha}(f_{t-1}) &= \sum_{i=1}^{N} \left(\frac{\partial^2 L(y_i, f)}{\partial f^2} \right)_{f_{t-1}} g_t(\boldsymbol{x}_i, \boldsymbol{\lambda}_t)^2, \qquad (13)
\end{aligned}$$

using which, the one-step Newton estimate is given by

$$\alpha_t = -\frac{\mathcal{L}_\alpha(f_{t-1})}{\mathcal{L}_{\alpha\alpha}(f_{t-1})}. \qquad (14)$$

The "one-step Newton" is generally preferable to the the "one-step multi-α" when applicable, as the selection of the initial set of grid values $\{\alpha_k\}_{k=1}^{K}$ for the function evaluation in (12) can be quite arbitrary.

```
B̆eginDataScan: Master node - Initialize the scores $\mathcal{L}_\alpha(f_{t-1})$, $\mathcal{L}_{\alpha\alpha}(f_{t-1})$ in (13).
P̆arallelDistribute: Distribute the current model state $f_{t-1}(\boldsymbol{x})$, stage basis function $g_t(\boldsymbol{x}, \boldsymbol{\lambda}_t)$
    and initialized scores.
D̆ataScan: Worker nodes - For every record, update the local contribution to the scores.
P̆arallelMerge: Merge the updated local scores to obtain the global scores.
Ĕ ndDataScan: Master node - Compute $\alpha_t$ from the global scores using (14).
OUTPUT: The coefficient $\alpha_t$ for stage $t$.
```

Figure 6: Parallel one-pass "one-step Newton" method used for the stage coefficient computation (14) in TREG.

4.5 TREG Variable Importance Diagnostics

The relative influence of the various input data variables on the regression function $\hat{f}(\boldsymbol{x})$ is an important consideration for many applications. Friedman [6] has discussed certain diagnostics for the relative variable importance and partial variable dependencies in connection with the GB algorithm, which can also be generally used for other complex ensemble models including the TREG algorithm. However, the evaluation of these diagnostics is computationally expensive and difficult to parallelize.

In TREG, therefore, a simple variable importance score is computed for each input data variable x_j, defined by

$$\mathcal{I}(x_j) = \frac{1}{N_V} \sum_{i=1}^{N_V} \left(\hat{f}(\boldsymbol{x}_i) - \hat{f}(\boldsymbol{x}_{(j)i}) \right)^2, \qquad (15)$$

where $\boldsymbol{x}_i$ denotes the i'th validation data record, and N_V is the total number of validation channel data records. In (15), $\boldsymbol{x}_{(j)i}$ represents a modification in which the actual value for the field x_j in the input data is replaced by a value that is sampled from an approximation to the empirical marginal distribution of x_j. The importance ranking of the input variables is then obtained by sorting the values of $\{\mathcal{I}(x_j)\}_{j=1}^{M}$ according to their magnitude.

This variable importance ranking can be obtained in a single-pass parallel scan over the validation data, although it has the disadvantage that correlation effects among the input variables are not taken into account, and therefore, any input variable that does not appear in the regression function will have a low ranking by this measure, even if it is highly correlated with the predicted response.

5 Transform Regression Performance and Results

In this Section, we describe computational results for the predictive model accuracy of the TREG algorithm, and performance results for the scalability on Blue Gene/P.

5.1 Predictive Modeling Accuracy

TREG is intended to be a fast, flexible, general-purpose modeling procedure for massive data sets, and it makes relatively few assumptions about the form of the model regression function.

The numerical results described in this section are therefore intended to amplify certain aspects of the TREG model prediction accuracy, using a variety of data sets and applications.

The TREG results are informally compared against MART, which is a commercial implementation of the gradient boosting algorithms in Friedman [6], marketed by Salford Systems (`http://www.salford-systems.com`). In these comparisons, consistent default settings are used for all algorithms, which means for the most part, that MART used 6-node trees as basis functions with 10-fold cross-validation, and TREG used 2-fold cross-validation with a stopping criterion based on a maximum of 4 stages without significant improvement in the cross-validation error. The synthetic data sets used here are based on generative models that are very similar to those considered in the paper by Li and Goel [12].

5.1.1 Regression

Synthetic Data: We consider synthetic data sets generated using known regression functions in the form,

$$y = f(\boldsymbol{x}) + \epsilon\mathcal{N}(0,1), \tag{16}$$

where $\boldsymbol{x}$ is a 20-dimensional vector of covariates. The continuous features x_1 through x_{10} are obtained by uniform sampling the interval $(0,1)$, and the 10 nominal features x_{11} through x_{20} by uniform sampling at four levels $1 - -4$. The individual features are all uncorrelated, but many of them are noise features in the generated data sets below.

Two specific regression functions are chosen for evaluation, viz.,

$$f_1(\boldsymbol{x}) = 10(x_1 - 0.5)^2 - x_2 + \sin 2\pi x_2 - 3x_3 + 1.5I([x_{11} \in \{3,4\}]) \tag{17}$$

which is an additive function of a covariate subset, and

$$f_2(\boldsymbol{x}) = \exp\left(-x_1 - x_2 + 2x_3\right) + 3(x_4 - x_5) + 2I([x_{11} \in \{3,4\}] \cup [x_{12} \in \{1,2\}]) \tag{18}$$

which is non-additive with high-order interaction effects between covariates in a subset of the features.

The training and test data sets that are generated consist of of 5000 points each, and choosing ϵ in 16 to be 0.8 yielded data sets with a roughly 2-to-1 signal-to-noise ratio. The squared-error loss function was used for both MART and TREG.

In the additive case, the validation MSE for MART was 1.227 ± 0.021, and for TREG was 0.777 ± 0.015. The coefficient of determination for MART was 0.62, and for TREG was 0.77 respectively. The MART results required 144 stages, while the TREG results required only 8 stages. The MART result appears to have overfitted the additive regression function, which can be confirmed by comparing with the equivalent MART results using 2-node stumps as stage basis functions, for which the validation MSE was 0.671 ± 0.013 with a coefficient of determination of 0.801.

In the non-additive case, the validation MSE for MART was 0.690 ± 0.013 and for TREG was 0.845 ± 0.017. The MART results required 287 stages, and the TREG results required only 7 stages. The coefficient of determination for MART was 0.815 and for TREG was 0.774. A comparison with the MART results for 2-node stumps as stage basis functions, for which the validation MSE was 1.049 ± 0.020) and coefficient of determination was 0.723, clearly indicates that non-additive effects are important in this data set, and that TREG is able incorporate some of these non-additive effects into the regression function, in spite of the fact that it does not explicitly model these effects in the stage basis functions.

In summary, the TREG results are accurate in both the additive and non-additive cases, without any further parameter tuning.

California Housing Data: The California housing data set (`http://lib.stat.cmu.edu/datasets/`) consists of a continuous response variable and 8 continuous covariates. As in [12], the log response was used for the modeling using the least-squares loss function, although ithout trimming of the outlier values for the covariates used there. The original data set with 20640 records was randomly partitioned into training and validation data sets consisting of 13960 and 6680 records respectively. The validation MSE for MART was 0.01247 ± 0.0004 with 200 training stages, and for TREG was 0.01276 ± 0.0004 with 5 training stages. The coefficient of determination for MART was 0.798 and for TREG was 0.793.

5.1.2 Classification

Adult Data: The adult census data set (`http://archive.ics.uci.edu/ml/datasets/Adult`), consists of a binary response and 15 covariates (both continuous and categorical). The training and validation data sets consist of 36632 and 12210 records respectively.

Figure 7 compares the gains charts for MART logistic regression with TREG using various loss functions (viz., squared error, logistic and support vector). Table 5.1.2 shows the model quality results, including the misclassification error, and the Gini coefficient estimated from the cumulative gains charts [8], and the results are quite comparable across the board. We note

	Adult Data Set 36632 (12210)			
	MART	TREG		
	logistic	logistic	least-squares	support vector
Misclassification Error	0.131	0.141	0.140	0.147
Gini	0.836	0.819	0.823	0.785

Table 2: Model Quality Results for Adult Data Set

that the logistic models provide a direct estimate of the class probabilities, which is desirable in certain applications.

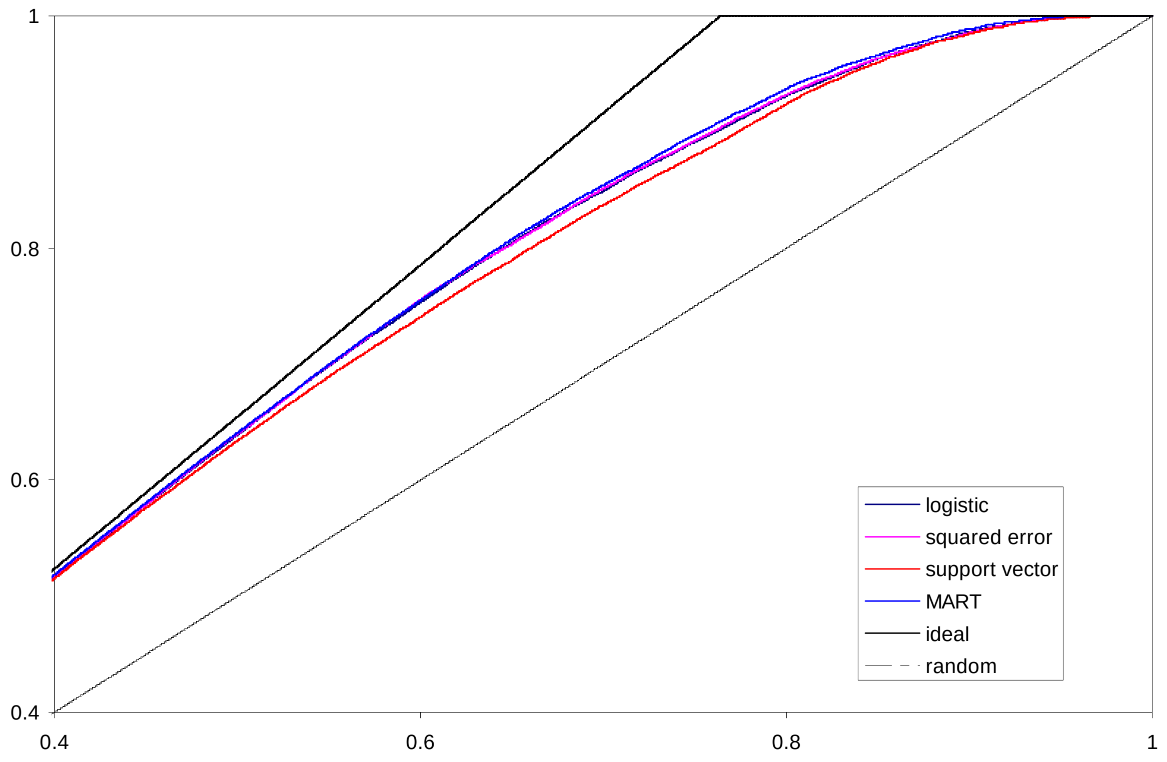

Figure 7: Lift curves for the adult data set. Only the parts with measurable differences in the model lift curves are displayed.

Spambase Data: The Spambase data set (`http://archive.ics.uci.edu/ml/datasets/Spambase`), consists of a binary response and 57 continous covariates. The original data set with 4603 records is randomly partitioned to yield a training set of 3065 records and a validation set of 1536 records.

The resulting model quality results shown in Table 5.1.2 are quite comparable across the board. The MART solution required 300 stages while the TREG solutions required 6 to 8 stages,

	Spambase Data Set 3065 (1536)			
	MART	TREG		
	logistic	logistic	least-squares	support vector
Misclassification Error	0.068	0.050	0.051	0.051
Gini	0.963	0.960	0.946	0.946

Table 3: Model Quality Results for Spambase Data Set

5.2 Parallel Performance Results

In Section 3.5, the theoretical aspects of the parallel performance and scalability of PML applications was considered. The issues discussed there are relevant to the timing results presented here for PML on the IBM Blue Gene/P parallel system.

In machine learning applications, the strong-scalability analysis is typically more relevant, since the training data set sizes are usually fixed in advance. However, in certain cases, the weak-scalability analysis may also be of interest, particularly when the larger training data set is used to to assess the model stability, or to improve the confidence intervals on the model parameter estimates.

In order to emulate both these situations, two synthetic data sets were generated for regression modeling; PERF1 with 50,000 rows and 50 covariate features, and PERF2 with 75,000 rows and 100 covariate features. Since the parallel timing measurements for PML are roughly the same for each iteration in Figure 3, it is sufficient to present the results for a single iteration here (say, iteration 6).

Figure 8 shows the parallel timings for PERF1 and PERF2, and the proportion of time spent in the communication and computation phases of the algorithm are also provided. As expected, with increasing P, the time for the highly-parallel computational phase decreases. Similarly, as expected, the time for the communication phase increases with increasing P, due to the increased network diameter for communication, as well as due to the increase in surface-to-volume effects which controls the relative amount of communication in the parallel implementation. Since neither of these data sets could be run for the $P = 1$ case, using the $P = 4$ case as the base, the speedups with $P = 128$ processors was 2.79 for PERF1, and 3.46 for PERF2.

Figure 9 shows the same results, but indicating the proportion of time spent in the the three parts of the TREG iteration shown in Figure 3. The dominant fraction of the time is in the computation of the stage sub-basis functions, with the computation of the stage basis functions, and the line search optimization, being a much smaller fraction of the overall time. Although it is possible to considerably reduce the time for the computation of the stage sub-basis functions by limiting their number, the effects of such algorithmic modifications on the PML model quality are uncertain, and this direction was not pursued at this point.

6 Summary

The PML software framework, described in this paper, has been used to implement several well-known machine learning algorithms on parallel computer architectures to date. In particular,

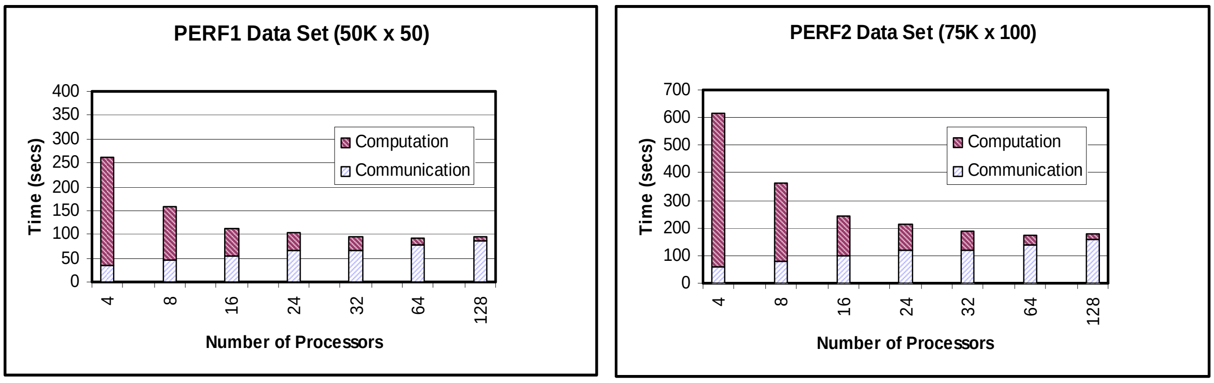

Figure 8: Performance Results.

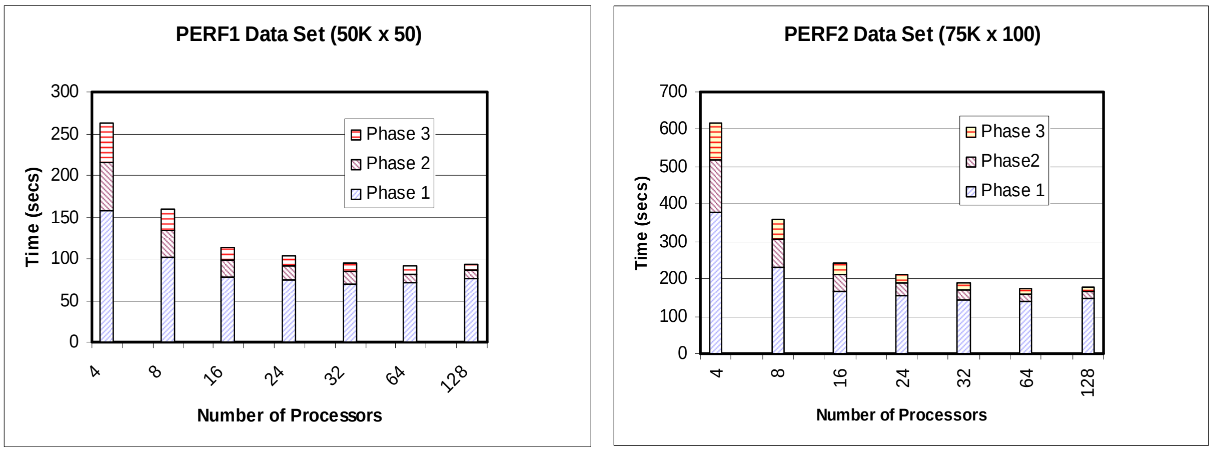

Figure 9: Performance Results.

we have motivated and described the implementation of the TREG (Transform Regression) algorithm in PML. The TREG algorithm is not only of intrinsic interest as a general-purpose regression methodology, but it also exemplifies the use of many of the features and capabilities of PML. The overall TREG methodology and implementation in PML is of interest in many application domains involving massive data sets, for which existing methods in the literature are likely to have limitations in terms of the achievable model accuracy and computational performance.

References

[1] IBM Blue Gene Team, *Overview of the IBM Blue Gene/P Project*, Special Issue on Applications of Massively Parallel Systems, IBM Journal of Research and Development, pp. 199-220, Vol. 52(1/2) (2008).

[2] A. Dorneich, R. Natarajan, E. Pednault and F. Tipu, *Embedded predictive modeling in a parallel relational database*, Proceedings of the 21st ACM Symposium on Applied Computing, Dijon, France (2006).

[3] S. Kumar, G. Dozsa, G. Almasi, P. Heidelberger, D. Chen, M. E. Giampapa, M. Blocksome, A. Faraj, J. Parker, J. Ratterman, B. Smith, C. J. Archer, *The Deep Computing Messaging Framework: Generalized scalable message passing on the Blue Cene/P Supercomputer*, Proc. 22nd Annual International Conference on Supercomputing, pp. 94-103 (2008).

[4] F. Provost and V. Kolluri, *A survey of methods for scaling up inductive algorithms*, Data Mining and Knowledge Discovery, Vol. 3, pp. 131-169, (1999).

[5] J. H. Friedman, *Stochastic gradient boosting*, Computational Statistics and Data Analysis, Vol. 38, pp. 367-378, (1999).

[6] J. H. Friedman, *Greedy function approximation: A gradient boosting machine*, Annals of Statistics, Vol. 29, pp. 1189–1232 (2001).

[7] http://hadoop.apache.org (2005).

[8] D. Hand, *Construction and Assessment of Classification Rules*, John Wiley and Sons, New York (1997).

[9] T. Hastie, R. Tibshirani and J. H. Friedman, *The Elements of Statistical Learning*, Springer-Verlag, New York (2001).

[10] R. Hecht-Nielsen, *Kolmogorov' mapping Neural Network existence theorem*, Proc. IEEE International Conference on Neural Networks, Vol. 3, pp. 11–14 (1987).

[11] `http://www.mcs.anl.gov/research/projects/mpich2/`.

[12] B. Li and P. K. Goel, *Additive regression trees and smoothing splines - predictive modeling and interpretation in data mining*, Contemporary Mathematics, Vol. 443, J. S. Verducci, X. Shen, J. Lafferty (eds.), pp. 83-101 (2007).

[13] E.P.D. Pednault, *Transform Regression and the Kolmogorov Superposition Theorem*, Proceedings of the Sixth SIAM International Conference on Data Mining, Bethesda, Maryland (2006).

[14] R. Natarajan and E.P.D. Pednault, *Segmented Regression Estimators for Massive Data Sets*, Proceedings of the SIAM Second International Conference on Data Mining, Crystal City, Virginia (2002).

[15] G. Ridgeway, *Generalized Boosted Models: A guide to the GBM packagte*, http://cran.r-project.org/web/packages/gbm/vignettes/gbm.pdf (2007).

[16] E. Yom-Tov, U. Aharoni, A. Ghoting, E. Pednault, D. Pelleg, H. Toledano, and R. Natarajan, *An Introduction to the IBM Parallel Mining Toolkit*, http://www.ibm.com/developerworks/grid/library/gr-ipmlt/index.html (2007).

[17] E. Yom-Tov, E. Pednault, R. Natarajan, D. Pelleg, H. Toledano, E. Aharoni, Y. Ben-haim, *Parallel Machine Learning Toolbox: User Guide*, available from http://www.alphaworks.ibm.com/tech/pml.