Distributed Monitoring of the R^2 Statistic for Linear Regression

Kanishka Bhaduri* Kamalika Das† Chris R. Giannella‡

Abstract

The problem of monitoring a multivariate linear regression model is relevant in studying the evolving relationship between a set of input variables (features) and one or more dependent target variables. This problem becomes challenging for large scale data in a distributed computing environment when only a subset of instances is available at individual nodes and the local data changes frequently. Data centralization and periodic model recomputation can add high overhead to tasks like anomaly detection in such dynamic settings. Therefore, the goal is to develop techniques for monitoring and updating the model over the union of all nodes' data in a communication-efficient fashion. Correctness guarantees on such techniques are also often highly desirable, especially in safety-critical application scenarios. In this paper we develop *DReMo* — a distributed algorithm with very low resource overhead, for monitoring the quality of a regression model in terms of its coefficient of determination (R^2 statistic). When the nodes collectively determine that R^2 has dropped below a fixed threshold, the linear regression model is recomputed via a network-wide convergecast and the updated model is broadcast back to all nodes. We show empirically, using both synthetic and real data, that our proposed method is highly communication-efficient and scalable, and also provide theoretical guarantees on correctness.

1 Introduction

Multi-variate linear regression is an important and widely used technique for modeling the behavior of a target variable based on a set of input variables (features). In scenarios where the data changes or evolves over time, monitoring the model for identifying such changes may be essential. This problem becomes more challenging if the data is distributed at a number of different nodes, and the model needs to be recomputed periodically to avoid inaccuracy. If the data is piecewise stationary, periodic model recomputation often wastes a lot of resources.

For example, for large networked distributed systems such as the Cloud and the Internet, anytimeness is extremely important, and monitoring the health of tens of thousands of data centers supporting numerous services requires extremely fast and correct detection of every performance crisis. The target variable for the identification of such crises can be response latency or request throughput [3], and an on-time and accurate alert suggesting a change in the input-target relationship can get the operators' immediate attention towards fault diagnosis and recovery. Similarly, we can monitor the carbon footprint of a community (city/state/country) in the next generation Smart Grids by modeling the carbon emission as a function of power consumption and natural energy production. Any change from the standard model can indicate change in consumption pattern, fault in power generators, etc. and an on-time detection can enable human intervention and guarantee uninterrupted service.

Most existing solutions for monitoring models in such setups usually trade off model fidelity for lower communication cost. Some of the approaches for monitoring models in distributed systems include the sampling-based meta-learning strategy [11] and randomized techniques such as gossip [9]. The first group of algorithms suffer from the drawback that accuracy drops with increasing number of nodes in the network whereas gossip-based algorithms rely on sufficient statistic computation on a random selection of nodes and are extremely communication-intensive for changing data scenarios. In the ideal case, the monitoring algorithm should be able to raise an alert every time an event occurs in the network and should do so with as little communication overhead as possible. Monitoring algorithms that satisfy these properties have been proposed earlier [16], where instead of periodically rebuilding a model, a thresholding criterion is developed to efficiently detect changes in the global model by only monitoring changes in the local data. The provable correctness of this class of algorithms ensures that the distributed algorithm can raise all the alerts that a centralized algorithm (seeing all the data at once) can detect and this is of utmost importance in many critical applications such as the ones mentioned above. However, the problem with the existing technique is that the usefulness of these algorithms is limited by the design parameters chosen by the user. If the user does not have prior knowledge about

*MCT Inc., NASA Ames, Moffett Field CA 94035, Kanishka.Bhaduri-1@nasa.gov. This work was done on a US Govt contract NNA08CG83C.

†SGT Inc., NASA Ames Research Center, Moffett Field CA 94035, Kamalika.Das@nasa.gov

‡The MITRE Corp, 7525 Colshire Drive, McLean, VA 22102-7539, cgiannel@acm.org. This work has been approved for public release, case 10-4075, distribution unlimited. The author's affiliation with The MITRE Corporation is provided for identification purposes only, and is not intended to convey or imply MITRE's concurrence with, or support for, the positions, opinions or viewpoints expressed by the author.

the data, it is possible that a wrong choice of the thresholding criterion can render these algorithms useless.

In this paper we overcome this problem for multivariate linear regression by formulating the monitoring problem in terms of the coefficient of determination R^2, a statistical metric for checking the quality of linear regression models. Since the R^2 statistic lies between 0 and 1, it is a scale free measure for the quality of fit for any data set. The algorithm developed in this paper, *DReMo*, works for horizontally partitioned data (defined later) and offers provable correctness guarantees with minimal communication. It is a reactive algorithm since communication for model recomputation does not happen periodically. Whenever the nodes jointly discover that the model no longer fits the data, an alert is raised and a convergcast/broadcast scheme is then deployed for model recomputation.

2 Related work

Regression being a powerful modeling tool, extensive research has been done for both distributed and centralized modeling and monitoring. In this paper we briefly review existing literature on distributed regression and its monitoring. Hershberger and Kargupta [7] have proposed one of the earliest wavelet transformation-based distributed regression algorithm for vertically partitioned data where each node observes all possible instances of a subset of features. The wavelet transform on the data optimizes the communication overhead by reducing the effect of the cross terms and the local regression models are centralized for building the global model. Another popular distributed regression algorithms has been proposed by Guestrin *et al.* [6] for learning kernel linear regression models in sensor networks. Once the model converges, instead of sending the raw data, the central node can only collect the coefficients of the regression model as a compact representation of the data, thereby reducing communication. The algorithm requires two passes through the entire network per data change to ensure global convergence, in the worst case and, therefore, may require huge number of messages and a long time to converge in dynamic data scenarios. It should be noted here that both these methods, as well as many other distributed regression techniques solve an approximate version of the centralized regression problem and therefore, cannot guarantee provable correctness when adapted for monitoring.

Meta-learning is an interesting class of algorithms which can be adopted for distributed model learning. Proposed by Stolfo *et al.* [15], the basic idea is to learn a model at each site locally (no communication at all) and then, when a new sample comes, predict the output by simply taking the average output of the local model outputs. The underlying assumption is that the data distribution is homogeneous across the nodes *i.e.* they have all been generated from the same distribution. Significant research has been done in the area of distributed computing of complex models. Gossip based computations have been proposed by Kempe *et al.* [9] and Boyd *et al.* [4] for computing simple primitives such as average, min, max etc. of a set of numbers distributed across the network. In gossip protocols, a node exchanges statistics with a random node and this process continues until convergence. Deterministic techniques such as the ones proposed by Scherber and Papadopoulos [13] and Mehyar *et al.* [12] solve a differential equation using messages exchanged between neighboring nodes such that the optimal solution to the equation gives the global average. However, both these classes of techniques require hundreds of messages per node for the computation of just one statistic and are not suitable for dynamic data. A related line of research concerns the monitoring of various kinds of data models over large numbers of data streams. Sharfman *et al.* [14] have developed an algorithm for monitoring arbitrary threshold functions over distributed data streams using both a broadcast based and a central coordinator based communication topology. Broadcast can be expensive for large networks while the coordinator topology assumes one root and all the other nodes as leaves. While, *DReMo* allows an arbitrary tree topology, comparison of *DReMo* under the "all leaf" tree topology versus their central coordinator algorithm is left to future work.

All of the above mentioned techniques can be adopted for monitoring evolving data streams in distributed computing environments, but they suffer from several drawbacks starting from very slow convergence resulting in extremely high communication overhead to lack of performance guarantees in detecting events or significant changes in the model. Recently, Bhaduri *et al.* [1] have proposed an algorithm for doing regression in large P2P networks which checks the squared error between the predicted and the target variables based on a generic monitoring algorithm proposed by Wolff *et al.* in [16]. If the error exceeds a predefined threshold (ϵ), the nodes raise an alert and the regression model is rebuilt. This method is communication-efficient and provably correct, but suffers from the serious disadvantage that the communication as well as model quality is dependent on a parameter ϵ that is input to the algorithm. This choice of ϵ is dependent on the data and can vary from 0 to ∞. If the user has no or limited knowledge about the data distribution in the computing network (which is often the case for all practical purposes), then a wrong choice of ϵ can render the algorithm useless. To overcome this problem, we propose a new regression monitoring algorithm *DReMo* which monitors the coefficient of determination (R^2) which is a tried-and-tested, well-accepted, and widely-used regression diagnostic measurement with $0 \leq R^2 \leq 1$. Closer the value of R^2 is to 1, the better is the model quality and vice versa. However, since the R^2 statistic is no longer the L2 norm of the data, none of the theories developed for monitoring the L2 norm of data in a large network [1] are applicable

here. In the next two sections we define this new monitoring problem and derive the R^2 statistic for distributed change detection of a linear regression model.

3 Problem setup

3.1 Notation: Let $V = \{P_1, \ldots, P_n\}$ be a set of computing nodes connected to one another via an underlying communication infrastructure, such that the set of P_i's immediate neighbors, N_i, is known to P_i (and P_i is unaware of the existence of any other nodes). At any time instance, the local data of P_i is a stream of tuples in $\mathbb{R}^d$ and is denoted by $S_i = \left[\left(\overrightarrow{x_1^i}, y_1^i\right), \left(\overrightarrow{x_2^i}, y_2^i\right), \ldots, \left(\overrightarrow{x_{m(i)}^i}, y_{m(i)}^i\right)\right]^T$, where $\overrightarrow{x_j^i} = [x_{j.1}^i \ldots x_{j.(d-1)}^i] \in \mathbb{R}^{d-1}$ and $y_j^i \in \mathbb{R}$. Every local data tuple can be viewed as an input and output pair. Note that S_i is time-varying, but for notational simplicity, we suppress an implicit t subscript. Let $G = \bigcup_{i=1}^n S_i$ denote the global data over all the nodes.

Nodes communicate with one another by sending sufficient statistics of a set of input vectors. We denote the sufficient statistics sent by node P_i to P_j as $|X_{i,j}|$ and $\overrightarrow{X_{i,j}}$, where $|X_{i,j}|$ is the size of a set of vectors and $\overrightarrow{X_{i,j}}$ is the average vector of that set. Computation of these quantities is discussed in the next section. We assume that reliable message passing is ensured by the underlying network and, therefore, if P_i sends a message to P_j, then P_j will receive it. Thus, both nodes know $X_{i,j}$ and $X_{j,i}$. We also assume that an overlay tree topology is maintained and it forms the network seen by the algorithm, *e.g.* the neighbors N_i of node P_i are the node's children and parent in the overlay. Note that, as shown in [2], such an overlay tree can be efficiently constructed and maintained using variations of Bellman-Ford algorithms [5][8]. Intuitively, the assumption of a tree overlay topology is needed to avoid 'double-counting' when communicating aggregate statistics. This will be discussed later when describing the specifics of the *DReMo* algorithm.

3.2 Problem definition: At any given time, each node holds f, a linear regression model (the same for each node). When the algorithm is initialized, a convergecast and broadcast mechanism is used to compute the f over G and distribute it to each node. After this, the goal is for the nodes, through ongoing distributed computation, to monitor the quality of f (in terms of how well it fits the global data) and, when the quality becomes sufficiently low, raise an alert and initiate another convergecast and broadcast to recompute f and distribute it to each node. Quality is measured using the *coefficient of determination*, R^2. Letting $\widehat{y_j^i}$ and $f(\overrightarrow{x_j^i})$ denote the true and estimated values of the target variable,

$$R^2 = 1 - \frac{\sum_{i=1}^n \sum_{j=1}^{m(i)} (y_j^i - \hat{y}_j^i)^2}{\sum_{i=1}^n \sum_{j=1}^{m(i)} (y_j^i - \frac{\sum_{i=1}^n \sum_{j=1}^{m(i)} y_j^i}{M})^2}$$

where $M = \sum_{i=1}^n m(i)$. This coefficient is between 0 and 1, equalling 1 when the data perfectly fits f. The ratio compares the variance of f's predictions (captured by the numerator) with the total variance of the data (captured by the denominator). Intuitively, this ratio captures the quality of f with respect to a baseline predictor which always returns the average value over all the observed data (the denominator). When R^2 is close to one, f provides a much better prediction of the observed values than the baseline. It is standard statistical practise when computing a regression model to compute R^2 as a measure of model quality.

The value of R^2 is time-varying. The goal for the nodes is to determine whether the accuracy of f is unacceptable. Specifically, for a fixed, user-defined threshold ϵ, the nodes monitor whether R^2 is below ϵ. If yes, then an alert is raised and computation is carried out to evaluate a new f over the most up-to-date global data G. Therefore, the crux of the problem is for the nodes to carry out this quality monitoring in a communication-efficient manner. Since R^2 is a nonlinear function of the local data held by all nodes, solving such a problem is challenging. Before addressing this problem, a caveat is in order. The setup described so far has the network fixed. However, adjusting the algorithm to accommodate nodes arriving and leaving or communication links going up and down is straightforward and omitted for descriptive simplicity.

Next we will define data vectors $\overrightarrow{v^i} \in \mathbb{R}^2$ (based on S_i and f), one for each node P_i, and a monitoring function $g : \mathbb{R}^2 \to \mathbb{R}$. We will show that the problem of monitoring whether R^2 is below ϵ can be reformulated as monitoring whether g is below zero when applied to a convex combination of $\overrightarrow{v^i}$'s. This result forms the basis of the *DReMo* algorithm proposed in this paper.

3.3 Monitoring R^2: Let $\tilde{\epsilon} = 1 - \epsilon$, $w_i = \tilde{\epsilon} \sum_{j=1}^{m(i)} (y_j^i)^2 - \sum_{j=1}^{m(i)} (y_j^i - \widehat{y_j^i})^2$, and $\overline{y^i} = \frac{\sum_{j=1}^{m(i)} y_j^i}{m(i)}$. We have $R^2 > \epsilon$

$$\Leftrightarrow \quad 1 - \frac{\sum_{i=1}^n \sum_{j=1}^{m(i)} (y_j^i - \hat{y}_j^i)^2}{\sum_{i=1}^n \sum_{j=1}^{m(i)} \left(y_j^i - \frac{\sum_{i=1}^n \sum_{j=1}^{m(i)} y_j^i}{M}\right)^2} > \epsilon$$

$$\Leftrightarrow \quad \tilde{\epsilon} \sum_{i=1}^n \sum_{j=1}^{m(i)} \left(y_j^i - \frac{\sum_{i=1}^n \sum_{j=1}^{m(i)} y_j^i}{M}\right)^2 - \sum_{i=1}^n \sum_{j=1}^{m(i)} \left(y_j^i - \widehat{y_j^i}\right)^2 > 0$$

$$\Leftrightarrow \quad \sum_{i=1}^n \tilde{\epsilon} \sum_{j=1}^{m(i)} (y_j^i)^2 - 2\tilde{\epsilon} \left(\sum_{i=1}^n \sum_{j=1}^{m(i)} y_j^i\right) \left(\frac{\sum_{i=1}^n \sum_{j=1}^{m(i)} y_j^i}{M}\right)$$

$$+ \quad \tilde{\epsilon} M \left(\frac{\sum_{i=1}^n \sum_{j=1}^{m(i)} y_j^i}{M}\right)^2 - \sum_{i=1}^n \sum_{j=1}^{m(i)} \left(y_j^i - \widehat{y_j^i}\right)^2 > 0$$

$$\Leftrightarrow \quad \sum_{i=1}^{n}\left(\tilde{\epsilon}\sum_{j=1}^{m(i)}(y_j^i)^2 - \sum_{j=1}^{m(i)}(y_j^i-\widehat{y_j^i})^2\right)$$

$$- \quad \tilde{\epsilon}M\left(\frac{\sum_{i=1}^{n}\sum_{j=1}^{m(i)} y_j^i}{M}\right)^2 > 0$$

$$\Leftrightarrow \sum_{i=1}^{n}(w_i) - \tilde{\epsilon}M\left(\frac{\sum_{i=1}^{n}\sum_{j=1}^{m(i)} y_j^i}{M}\right)^2 > 0$$

$$\Leftrightarrow \quad \sum_{i=1}^{n}\left(\frac{w_i}{M}\right) - \tilde{\epsilon}\left(\frac{\sum_{i=1}^{n}\sum_{j=1}^{m(i)} y_j^i}{M}\right)^2 > 0$$

$$\Leftrightarrow \quad \sum_{i=1}^{n}\left(\frac{m(i)}{M}\right)\left(\frac{w_i}{m(i)}\right) - \tilde{\epsilon}\left(\sum_{i=1}^{n}\left(\frac{m(i)}{M}\right)\overline{y^i}\right)^2 > 0$$

Let $\overrightarrow{v^i} = \left[\frac{w_i}{m(i)}, \overline{y^i}\right]$ and $g : \overrightarrow{a} \in \mathbb{R}^2 \mapsto a_1 - \hat{\epsilon}a_2^2$. Thus, we have that R^2 being below ϵ is equivalent to g being below zero when applied to a convex combination of $\overrightarrow{v^i}$'s:

$$R^2 > \epsilon \Leftrightarrow g\left(\sum_{i=1}^{n}\left(\frac{m(i)}{M}\right)\overrightarrow{v^i}\right) > 0. \tag{3.1}$$

g is a parabola and $\{\overrightarrow{a} \in \mathbb{R}^2 : g(\overrightarrow{a}) = 0\}$ splits $\mathbb{R}^2$ into two regions: the area inside the parabola (which is convex), and the area outside (which is not convex). We denote $\overrightarrow{v^G} = \sum_{i=1}^{n}\left(\frac{m(i)}{M}\right)\overrightarrow{v^i}$ as the global statistic computed over all the local $\overrightarrow{v^i}$-s. Since $\overrightarrow{v^G}$ is a convex combination of $\overrightarrow{v^i}$'s, then if each node P_i determines that $\overrightarrow{v^i}$ is in the area inside the parabola, then $\overrightarrow{v^G}$ must be too. This forms the basis for a very nice distributed algorithm. However, the same reasoning does not hold for the area outside the parabola. To get around this problem, the area outside the parabola is approximated as a union of overlapping half-planes (defined by tangent lines of g). If each node determines that $\overrightarrow{v^i}$ is in the same half-space for all nodes, then $\overrightarrow{v^G}$ must be in the half-plane as well (therefore, outside the parabola). In this case no communication is necessary since all nodes are in agreement. On the other hand, if different nodes have their $\overrightarrow{v^i}$-s in different convex regions (either inside the parabola or one of the half-spaces), or in none of these convex regions, then communication is required to come to a consensus. The goal now boils down to monitoring $g(\overrightarrow{v^G})$ using the quantity $g(\overrightarrow{v^i})$. The next section develops this idea into a concrete algorithm.

4 *DReMo*: algorithm description

Based on the formulation of R^2 monitoring, we can develop a distributed algorithm which requires far less communication than centralizing all the information from all the nodes to one location. The intuition is to develop a set of conditions based on the data at each node. If these conditions are satisfied at all nodes independently, then we can guarantee some globally correct condition. This allows any node to cease communication and output the correct result. In the remainder of this section we first develop one such local stopping criterion and then describe how it can be effectively used for developing a distributed regression monitoring algorithm.

4.1 Thresholding criterion: Recall that $\overrightarrow{v^i}$-s are the local vectors of any node P_i. Now, we know that, if all $\overrightarrow{v^i}$-s lie in a convex region, then their convex combination is also in the same convex region. In order to check in a distributed fashion whether this condition is satisfied, each node needs to maintain the information about its neighbors' $\overrightarrow{v^i}$-s. The following sufficient statistics defined exclusively on local inputs allow a node to do this computation:

1. *knowledge*: $\overrightarrow{\mathcal{K}_i}$ is defined as the convex combination of the local monitoring input $\overrightarrow{v^i}$ and all the information that P_i has received from all its neighbors *i.e.* $\overrightarrow{X_{j,i}}$
2. *agreement*: $\overrightarrow{\mathcal{A}_{i,j}}$ is the information that both P_i and P_j share
3. *withheld*: $\mathcal{H}_{i,j}$, is the information that P_i has and has not yet shared with P_j

We also define the sizes of these statistics as the number of elements over which the statistic is computed. Thus, the sizes of these statistics are defined as,
1. $|\mathcal{K}_i| = m(i) + \sum_{P_j \in N_i} |X_{j,i}|$
2. $|\mathcal{A}_{i,j}| = |X_{i,j}| + |X_{j,i}|$
3. $|\mathcal{H}_{i,j}| = |\mathcal{K}_i| - |\mathcal{A}_{i,j}|$.
Using these, the vectors themselves are defined as:
1. $\overrightarrow{\mathcal{K}_i} = \frac{m(i)}{|\mathcal{K}_i|}\overrightarrow{v^i} + \sum_{P_j \in N_i} \frac{|X_{j,i}|}{|\mathcal{K}_i|}\overrightarrow{X_{j,i}}$
2. $\overrightarrow{\mathcal{A}_{i,j}} = \frac{|X_{i,j}|}{|\mathcal{A}_{i,j}|}\overrightarrow{X_{i,j}} + \frac{|X_{j,i}|}{|\mathcal{A}_{i,j}|}\overrightarrow{X_{j,i}}$
3. $\overrightarrow{\mathcal{H}_{i,j}} = \frac{|\mathcal{K}_i|}{|\mathcal{H}_{i,j}|}\overrightarrow{\mathcal{K}_i} - \frac{|\mathcal{A}_{i,j}|}{|\mathcal{H}_{i,j}|}\overrightarrow{\mathcal{A}_{i,j}}$

Also, we can define the exchanged information (messages) between P_i and P_j to be $\overrightarrow{X_{i,j}} = \frac{|\mathcal{K}_i|\overrightarrow{\mathcal{K}_i} - |X_{j,i}|\overrightarrow{X_{j,i}}}{|\mathcal{K}_i| - |X_{j,i}|}$ and $|X_{i,j}| = |\mathcal{K}_i| - |X_{j,i}|$. First note that, when the algorithm initiates, no messages have been sent, and so, both $\overrightarrow{\mathcal{K}_i}$ and $\overrightarrow{X_{i,j}}$ equals $\overrightarrow{v^i}$. As the algorithm proceeds, the messages sent by any node consists of knowledge at that node minus the information received by that node, to avoid duplicate counting.

Now, in order to check if the convex combination of $\overrightarrow{\mathcal{K}_i}$'s (and hence $\overrightarrow{v^i}$'s) are all in the same convex region, we need to split the domain of monitoring function g into non-overlapping convex regions. Figure 1 shows the regions. First of all, note that inside of the parabola is con-

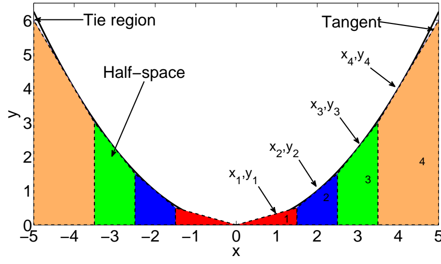


Figure 1: The parabola and the tangent lines that define the half spaces as shown in different colors. The tangents are drawn at the points shown in the figure.

vex by definition. The outside of the parabola is not convex; however it can be covered by hyper-planes defined by tangents to the parabola, thereby splitting it into convex regions. Therefore, the convex regions for this monitoring are: (1) $C_{in} = \{\overrightarrow{e} \in \mathbb{R}^2 : g(\overrightarrow{e}) > 0\}$, and (2) $C_\ell = \{\overrightarrow{e} \in \mathbb{R}^2 : \overrightarrow{u_\ell} \cdot \overrightarrow{e} > 0\}$, where $\overrightarrow{u_\ell}$ is the ℓ-th unit normal of the tangent to the parabola. These convex regions are collectively defined as $C = \{C_{in}, C_1, \ldots, C_t\}$. Also shown in the figure are the *tie* regions — those regions which lie outside the parabola, and also not inside any half-space. Given the convex regions, we state the following theorem which gives us a condition by which any node P_i can decide if the global vector $\overrightarrow{v^G}$ is inside any convex region based on only $\overrightarrow{\mathcal{K}_i}, \overrightarrow{\mathcal{A}_{i,j}}$, and $\overrightarrow{\mathcal{H}_{i,j}}$.

THEOREM 4.1. **[Thresholding Rule]***[16] Given any region $R \in C$, if no messages traverse the network, and for each P_i, $\overrightarrow{\mathcal{K}_i} \in R$ and for every $P_j \in N_i$, $\overrightarrow{\mathcal{A}_{i,j}} \in R$ and either $\overrightarrow{\mathcal{H}_{i,j}} \in R$ or $\mathcal{H}_{i,j} = \emptyset$, then $\overrightarrow{v^G} \in R$.*

Proof (SKETCH): We omit the formal proof here due to shortage of space. The intuition behind the proof is to take one node, say P_i, and combine its data with any of its neighbor's data. Let P_k be a neighbor of P_i who sends all of its $\overrightarrow{\mathcal{H}_{k,i}}$ to P_i. P_i on receiving this will set its new $\overrightarrow{\mathcal{K}_i}$ to be the convex combination of the old $\overrightarrow{\mathcal{K}_i}$ and $\overrightarrow{\mathcal{H}_{k,i}}$. Since both are in the same convex region by assumption, their convex combination will also be in the same convex region. It is also easy to verify that the agreements and withheld knowledges of P_i with any other neighbor P_j will also lie in the same convex region after this step. Thus, we can eliminate node P_k since its information has already been incorporated into that of P_i. Continuing this process of elimination we will have the knowledge of a single node equal to $\overrightarrow{v^G}$ (since the convex combination of all $\overrightarrow{\mathcal{K}_i}$-s is $\overrightarrow{v^G}$). Now since in each of these elimination steps, $\overrightarrow{\mathcal{K}_i}$ always remain inside the convex region, so will $\overrightarrow{v^G}$. ∎

Each node can apply this stopping condition to its local vectors and if the condition is satisfied, then it need not communicate any messages even if its local data changes or it receives any message from its neighbors. Unfortunately, when $\overrightarrow{\mathcal{K}_i}$ lies in the tie region, the stopping condition cannot be applied. In this case, the only way a node can guarantee correctness is by sending all of the local information $\overrightarrow{\mathcal{K}_i}$ to all its neighbors. The goal is therefore, to place the tangent lines such that the area of this region is minimized. The following lemma shows us how to achieve this.

LEMMA 4.1. *Given a parabola defined by $y = \tilde{\epsilon}x^2$, let $T = \{(x_1, \tilde{\epsilon}x_1^2), (x_2, \tilde{\epsilon}x_2^2), \ldots, (x_t, \tilde{\epsilon}x_t^2)\}$ be points on the parabola at which the tangent lines are drawn. Minimizing the area of the tie region leads to the following values of the x-coordinates of points in T: $x_\ell = \frac{2\ell x_{max}}{(2t+1)}$, $\forall \ell = 1 : t$, where x_{max} is the maximum value of the x-coordinate.*

Proof. Proof in provided in Appendix A.

4.2 Distributed regression monitoring algorithm (*DReMo*): *DReMo* utilizes the condition of Theorem 4.1 to decide when to stop sending messages to its neighbors. The pseudo code is shown in Alg. 4.1, 4.2 and 4.3. The algorithm is entirely event driven. Events can be any one of the following: (1) change in local data S_i, (2) a message received, or (3) change in N_i. If any one of these events occur, P_i first checks the received message buffer and updates its local vectors. It then checks the conditions for sending messages. First it finds the region in which $\overrightarrow{\mathcal{K}_i}$ lies and sets a variable k_{loc} accordingly: (1) $k_{loc} = 1$ if $g(\overrightarrow{\mathcal{K}_i}) > 0$ (inside parabola), or (2) $k_{loc} = 2$ if there exists one $\overrightarrow{u_\ell}$ such that $\overrightarrow{u_\ell} \cdot \overrightarrow{\mathcal{K}_i} > 0$ (inside any half space), or (3) $k_{loc} = 3$ otherwise (tie region). Now based on the outcome of this test, a node needs to send a message to its neighbor P_j if any of the following occurs:

1. $(k_{loc} == 3) \bigwedge \left(\overrightarrow{\mathcal{K}_i} \neq \overrightarrow{\mathcal{A}_{i,j}}\right) \bigwedge (|\mathcal{K}_i| \neq |\mathcal{A}_{i,j}|)$
2. $(k_{loc} == 1 \bigvee k_{loc} == 2) \bigwedge |\mathcal{H}_{i,j}| == 0 \bigwedge \left(\overrightarrow{\mathcal{K}_i} \neq \overrightarrow{\mathcal{A}_{i,j}}\right)$
3. $(k_{loc} == 1 \bigvee k_{loc} == 2) \bigwedge |\mathcal{H}_{i,j}| \neq 0 \bigwedge \textbf{NotInside}\left(\overrightarrow{\mathcal{A}_{i,j}}, k_{loc}\right) \bigwedge \textbf{NotInside}\left(\overrightarrow{\mathcal{H}_{i,j}}, k_{loc}\right)$

Case 1 occurs when $\overrightarrow{\mathcal{K}_i}$ lies in the tie region and a node needs to send its local information unless it has already sent everything in a previous communication. Cases 2 and 3 are for directly checking the conditions of Theorem 4.1. Note that we need to take special care when $|\mathcal{H}_{i,j}| = 0$, in which case, $\overrightarrow{\mathcal{H}_{i,j}}$ is undefined. This means that a node has communicated everything to its neighbor and does not need to send a message again, unless another event occurs. Case 2 can occur, for example, when the node has already sent

everything and then the local data changes, thereby making $\overrightarrow{\mathcal{K}_i} \neq \overrightarrow{\mathcal{A}_{i,j}}$. Case 3 directly checks the condition of Theorem 4.1 with an added exception built-in to prevent this checking in case of $|\mathcal{H}_{i,j}| = 0$.

If any one of these conditions occur, a node can set $\overrightarrow{X_{i,j}} \leftarrow \frac{|\mathcal{K}_i|\overrightarrow{\mathcal{K}_i} - |X_{j,i}|\overrightarrow{X_{j,i}}}{|\mathcal{K}_i| - |X_{j,i}|}$ and $|X_{i,j}| \leftarrow |\mathcal{K}_i| - |X_{j,i}|$ and send it to P_j. However, as it turns out, if we set $|\mathcal{H}_{i,j}| = 0$, and the data changes again, a node might need to communicate because $\overrightarrow{\mathcal{K}_i} \neq \overrightarrow{\mathcal{A}_{i,j}}$ may be violated. To avoid this, we set $X_{i,j}$ to be equal to the smallest value for which either both $\overrightarrow{\mathcal{K}_i}$ and $\overrightarrow{\mathcal{A}_{i,j}}$ goes inside the convex region or $|\mathcal{H}_{i,j}| = 0$. The pseudo code for this step is shown in Alg. 4.3.

ALGORITHM 4.1. ***DReMo***
Input: ϵ, C, S_i, N_i and L.
Output: 0 if $g(\overrightarrow{\mathcal{K}_i}) > 0$, 1 otherwise
Initialization: Initialize $v^i, \overrightarrow{\mathcal{K}_i}, \overrightarrow{\mathcal{A}_{i,j}}, \overrightarrow{\mathcal{H}_{i,j}}$

On an event:
if $\textbf{MessageRecvd}\left(P_j, \overrightarrow{X}, |X|\right)$ **then**
 $\overrightarrow{X_{j,i}} \leftarrow \overrightarrow{X}$ and $|X_{j,i}| \leftarrow |X|$;
end if
Update $v^i, \overrightarrow{\mathcal{K}_i}, \overrightarrow{\mathcal{A}_{i,j}}, \overrightarrow{\mathcal{H}_{i,j}}$
for all Neighbors P_j **do**
 Call $\textbf{CheckMsg}(\overrightarrow{\mathcal{K}_i}, \overrightarrow{\mathcal{A}_{i,j}}, \overrightarrow{\mathcal{H}_{i,j}}, P_j)$;
end for

ALGORITHM 4.2. Procedure **CheckMsg**
Input: $\overrightarrow{\mathcal{K}_i}, \overrightarrow{\mathcal{A}_{i,j}}, \overrightarrow{\mathcal{H}_{i,j}}, P_j$

k_{loc} = **CheckKiLocation**($\overrightarrow{\mathcal{K}_i}$);
for all Neighbors P_j **do**
 if $(k_{loc} == 3) \bigwedge \left(\overrightarrow{\mathcal{K}_i} \neq \overrightarrow{\mathcal{A}_{i,j}}\right) \bigwedge (|\mathcal{K}_i| \neq |\mathcal{A}_{i,j}|)$ **then**
  SendMsg=true; {/*Tie Region*/}
 end if
 if ($k_{loc} ==$ 1 or 2) $\bigwedge |\mathcal{H}_{i,j}| == 0 \bigwedge \left(\overrightarrow{\mathcal{K}_i} \neq \overrightarrow{\mathcal{A}_{i,j}}\right)$ **then**
  SendMsg=true {/*Theorem Condition*/}
 end if
 if ($k_{loc} ==$ 1 or 2) $\bigwedge |\mathcal{H}_{i,j}| \neq 0 \bigwedge \textbf{NotInside}\left(\overrightarrow{\mathcal{A}_{i,j}}, k_{loc}\right)$ $\bigwedge \textbf{NotInside}\left(\overrightarrow{\mathcal{H}_{i,j}}, k_{loc}\right)$ **then**
  SendMsg=true {/*Theorem Condition*/}
 end if
 if (SendMsg==true) then
  Call $\textbf{SendMessage}\left(\overrightarrow{X_{i,j}}, \overrightarrow{\mathcal{K}_i}, \overrightarrow{\mathcal{A}_{i,j}}, \overrightarrow{\mathcal{H}_{i,j}}, P_j\right)$
 end if
end for

ALGORITHM 4.3. Procedure **SendMessage**
Input: $\overrightarrow{X_{i,j}}, \overrightarrow{\mathcal{K}_i}, \overrightarrow{\mathcal{A}_{i,j}}, \overrightarrow{\mathcal{H}_{i,j}}, P_j$

$\overrightarrow{X_{i,j}} \leftarrow \frac{|\mathcal{K}_i|\overrightarrow{\mathcal{K}_i} - |X_{j,i}|\overrightarrow{X_{j,i}}}{|\mathcal{K}_i| - |X_{j,i}|}$;
$s = 1/2$;
$|X_{i,j}| \leftarrow (1-s) * (|\mathcal{K}_i| - |X_{j,i}|)$;
Update all vectors;
While ($\textbf{NotInside}\left(\overrightarrow{\mathcal{A}_{i,j}}, k_{loc}\right)$ or $\textbf{NotInside}\left(\overrightarrow{\mathcal{H}_{i,j}}, k_{loc}\right)$) and $|\mathcal{H}_{i,j}| \neq 0$
 $|X_{i,j}| \leftarrow (1-s) * (|\mathcal{K}_i| - |X_{j,i}|)$;
 Update all vectors;
 $s = \lfloor s/2 \rfloor$;
end while
Send $(P_j, \overrightarrow{X_{i,j}}, |X_{i,j}|)$ to P_j;

4.3 Re-computing model using convergecast/broadcast:

Whenever the model at any node does not fit the data, it sets the output of *DReMo* to 1 based on $g(\overrightarrow{\mathcal{K}_i}) < 0$. Once the nodes jointly discover that the current model is out-of-date, an alert is raised at each node and model recomputation becomes necessary.

We leverage the fact that a linear regression model can be easily computed by solving a linear set of equations. The coefficients of this equation can be written as a running sum over all the data points. Let the input-output be related linearly as follows: $\widehat{y_j^i} = f(\overrightarrow{x_j^i}) = w_0 + w_1 x_{j.1}^i + w_2 x_{j.2}^i + \ldots + w_{d-1} x_{j.(d-1)}^i$. For simplicity, we separate the input data matrix at node P_i as $S_i = [X_i \quad y_i]$, by partitioning the input and output into separate matrices. We can do this for $G = [X \quad y]$ in a similar fashion. We then augment the input matrix X_i with a column of 1-s at the beginning, but for notational simplicity refer to it by X_i itself. Using least square technique for model fitting, we need to compute two matrices over the global data: $X^T X$ and $X^T y$. As shown below, both these matrices are decomposable over local inputs:

$$X^T X = \begin{pmatrix} \sum_{i=1}^n m(i) & \sum_{i=1}^n \sum_{j=1}^{m(i)} x_{j.1}^i & \cdots \\ \sum_{i=1}^n \sum_{j=1}^{m(i)} x_{j.1}^i & \sum_{i=1}^n \sum_{j=1}^{m(i)} (x_{j.1}^i)^2 & \cdots \\ \vdots & \vdots & \vdots \\ \sum_{i=1}^n \sum_{j=1}^{m(i)} x_{j.(d-1)}^i & \sum_{i=1}^n \sum_{j=1}^{m(i)} (x_{j.(d-1)}^i)^2 & \cdots \end{pmatrix}$$

$$= \sum_{i=1}^n \begin{pmatrix} m(i) & \sum_{j=1}^{m(i)} x_{j.1}^i & \cdots \\ \sum_{j=1}^{m(i)} x_{j.1}^i & \sum_{j=1}^{m(i)} (x_{j.1}^i)^2 & \cdots \\ \vdots & \vdots & \vdots \\ \sum_{j=1}^{m(i)} x_{j.(d-1)}^i & \sum_{j=1}^{m(i)} (x_{j.(d-1)}^i)^2 & \cdots \end{pmatrix} = \sum_{i=1}^n X_i^T X_i$$

Similarly, it is easy to verify that $X^T y = \sum_{i=1}^n X_i^T y_i$. Once these two matrices are known globally, the set of weights $\overrightarrow{w} = w_0, \ldots, w_{d-1}$ can be computed as

$$\overrightarrow{w} = \left(\sum_{i=1}^n X_i^T X_i\right)^{-1} \left(\sum_{i=1}^n X_i^T y_i\right)$$

The goal is then to coordinate this computation across the nodes over the topology tree that is already maintained for the monitoring phase. A simple strategy is to use an alternating convergecast-broadcast scheme. For convergecast, whenever a node P_i detects that the output of the monitoring algorithm is 1, it sets an alert flag and starts a timer (*alert wait period*) to τ time units. When the timer expires and if the flag is still set, P_i checks to see if it is a leaf. If it is, it sends both $X_i^T X_i$ and $X_i^T y_i$ to its neighbor from which it has not yet received any data and sets its state as convergecast. If, on the other hand, the monitoring algorithm dictates

that the model fits the data, the flag is reset. When any intermediate node gets $X_j^T X_j$ and $X_j^T y_j$ from one of its neighbors P_j, it first adds the received data to its received buffer B. It then checks if its alert flag is set, the timer has expired and if it has received data from all but one neighbor. If all these conditions are valid, it adds its own data to the received buffer B and sends it to its neighbor from whom it has not received any data and sets its state to convergecast. When a node gets data from all neighbors, it becomes the root. It then solves the regression equation to find a new $\overrightarrow{w}$ and broadcasts this $\overrightarrow{w}$ to all its neighbors. Any node on receiving this new model, changes it state to 'broadcast' and resets its alert flag and timer. It then forwards the new model to all its children. We use the alert wait period to minimize the number of false alarms by making a node wait for τ time units before the alert is acted upon.

Note that the use of linear regression allows us to compute the weights in an exact fashion compared to centralization. Moreover, the dimensionality of the matrices $X_i^T X_i$ and $X_i^T y_i$ are $d.d + d.1 = O(d^2)$. This shows that the communication complexity is only dependent on the degree of the polynomial or the number of attributes and independent of the size of the dataset.

It must be noted that a new convergecast/brodacast round is invoked whenever R^2 goes below ϵ at all the nodes. $R^2 < \epsilon$ implies that the data has changed and the model does not fit the data. To improve model quality, the coefficients of function f are recomputed using the convergecaset/broadcast procedure. However, another scenario where R^2 becomes less than ϵ is when the assumption of linearity is no longer valid. In this case it may still be possible to avoid multiple convergecast rounds by using a sufficiently low (yet significant) value of ϵ in application scenarios where an approximate fit is good enough such as in anomaly detection. Another way of addressing this problem is to directly compare the coefficients of the old and the new model and then to stop the recomputation phase if the two models are close while the R^2 is still low. More details of monitoring nonlinear models is beyond the scope of this work.

4.4 Correctness of *DReMo*: Correctness of *DReMo* is based on Theorem 4.1. Based on the conditions, any node will keep sending messages until one of the following conditions occur: (1) for every node, $\overrightarrow{\mathcal{K}_i} = \overrightarrow{v^G}$ or, (2) for every P_i and every neighbor P_j, $\overrightarrow{\mathcal{K}_i}$, $\overrightarrow{\mathcal{A}_{i,j}}$,and $\overrightarrow{\mathcal{H}_{i,j}} \in R$. In the former case, obviously $g(\overrightarrow{\mathcal{K}_i}) = g(\overrightarrow{v^G})$. In the latter case, Theorem 4.1 dictates that $\overrightarrow{v^G} \in R$. Therefore, in either of the cases $g(\overrightarrow{\mathcal{K}_i}) = g(\overrightarrow{v^G})$, thereby guaranteeing global correctness. Also since we are computing linear regression models, the decomposability of the convergecast matrices $X^T X = \sum_{i=1}^n X_i^T X_i$ and $X^T y = \sum_{i=1}^n X_i^T y_i$ also ensure that the model built is the same as a centralized algorithm having access to all of the data.

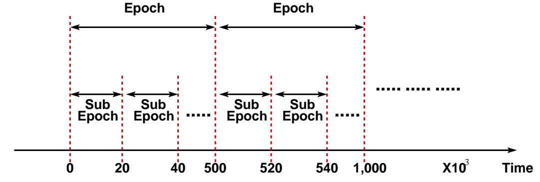


Figure 2: Each *epoch* is of 500,000 ticks and consists of several 20,000 ticks *subepochs*.

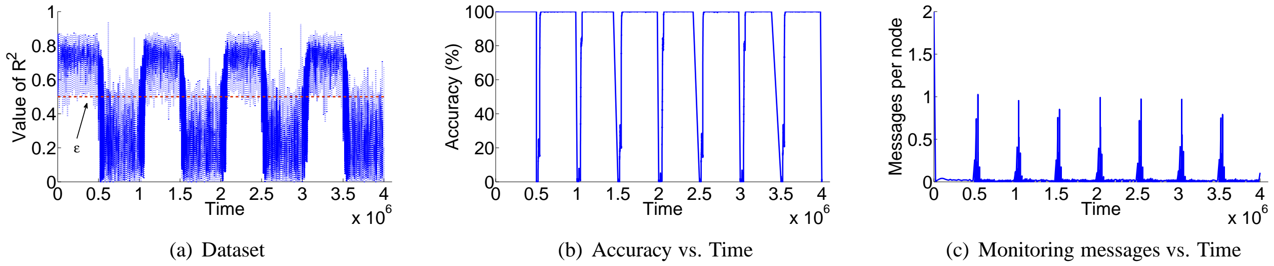
5 Experimental results

In order to analyze the performance of *DReMo*, we have performed a variety of experiments under different conditions. We first describe the simulation environment and the dataset, followed by the performance of the algorithm.

5.1 Experimental setup: We have used a simulated environment for running the experiments. The simulations have been run on a dual processor machine of 3.3 GHz each with 4GB of physical memory running Windows XP. The distributed network has been simulated on this machine using a topology generated by BRITE (`http://www.cs.bu.edu/brite/`). We have experimented with the Barabasi Albert (BA) model. We convert the edge delays to simulator ticks for time measurement since wall time is meaningless when simulating thousands of nodes on a single PC. Each simulator tick in our experiment corresponds to 1 msec in BRITE topology. On top of each network generated by BRITE, we overlay a communication tree. We make the assumption that the time required for local processing is trivial compared to the overall network latency and therefore, convergence time for *DReMo* is reported in terms of the average edge delay.

In our experiments we have used a leaky bucket mechanism which prevents a node from sending two messages within the same leaky bucket period. Whenever a node P_i gets a message, it sets a timer to L simulator time units and down counts. If another event occurs while the timer is still active, P_i does not send another message. Only if an event occurs after the expiration of the timer, P_i is allowed to send another message. Note that this technique does not affect the correctness of the algorithm, since we do not destroy any events. In the worst case, it may only delay convergence. In our experiments we have set the value of L such that any node is able to send between 10 to 20 messages fir each sub-epoch. This rate is enough to allow *DReMo* to converge while offering a very low communication overhead.

In order to demonstrate the effects of the different parameters of *DReMo* in a controlled manner, we have used synthetically generated data following a linear model.



(a) Dataset (b) Accuracy vs. Time (c) Monitoring messages vs. Time

Figure 3: Dataset, accuracy and messages for *DReMo* algorithm in monitoring mode.

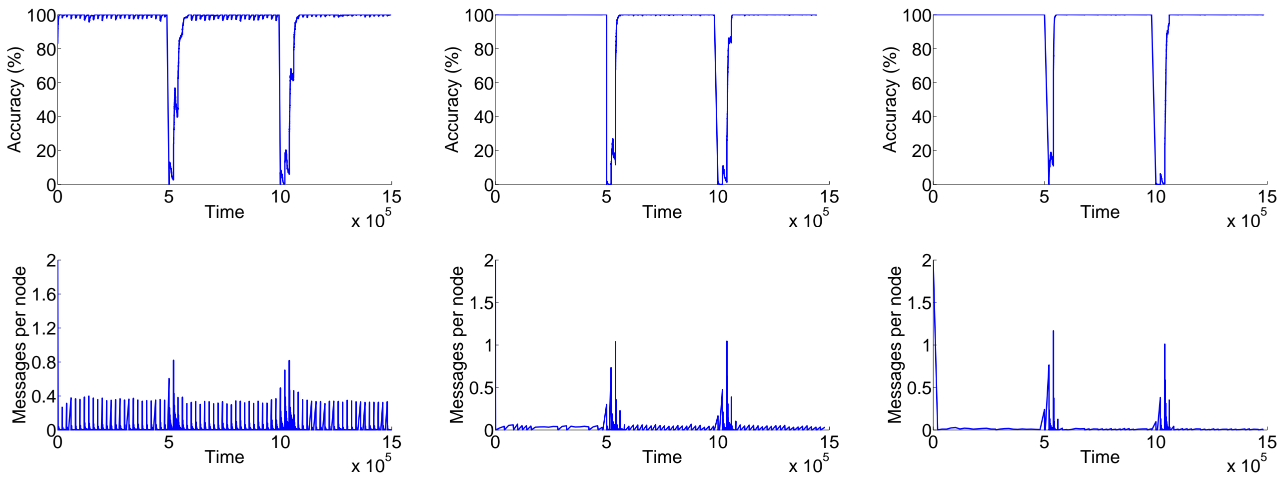


Figure 4: Variation of accuracy (top row) and messages (bottom row) for different $|S_i| = 10$, 50 and 100 from left to right.

(Experiments with real data is given in the next section.) Given an input vector $\overrightarrow{x_j} = x_{j.1} \ldots x_{j.(d-1)}$, the output is generated according to: $\widehat{y_j} = w_0 + \sum_{k=1}^{d-1} w_k x_{j.k} + \theta$, where $\theta \sim \mathcal{N}(0, \sigma^2)$. For any experiment, we have randomly chosen the values of w_i's and σ in the range -5 to +5. Fig. 2 shows the timing diagram for all experiments. At some predefined clock ticks, we have changed the data distribution by randomly changing the weights w_i-s of the data generator. We refer to this time interval as an *epoch*. A epoch consists of several *sub-epochs* — those time points when we replace 20% of the data at each node, generated from the current distribution. Thus each sub-epoch refers to a unit of data change. We choose length of each epoch as 500,000 ticks and sub-epoch as 20,000 ticks. Note that the number of times data is replaced in every epoch is 500,000/20,000=25 and all our experiments are run for many epochs.

We report three quantities for the experiments: *accuracy*, *convergence rate*, and *communication cost*. These quantities are measured differently for the two modes of *DReMo* — (1) when only the monitoring algorithm operates, and (2) when the nodes jointly execute the convergecast-broadcast procedure after the monitoring algorithm raises an alert. For the former mode of operation, accuracy is measured as the number of nodes which correctly identify whether $R^2 \gtrless \epsilon$, while for the other mode, it is measured as the average R^2 value over all the nodes. *Convergence rate* is defined as the number of simulator ticks from the beginning of an epoch to the time the algorithm reaches 99% accuracy. *Communication cost* consists of two types of messages: monitoring messages measured as the number of messages sent by *DReMo* for monitoring and computation messages for rebuilding the model.

For all the experiments, unless otherwise stated, we have used the following default values of the parameters: (1) $|S_i| = 75$, (2) $d = 10$, (3) $L = 2000$, (4) $\epsilon = 0.5$, (5) number of tangent lines = 6, and (6) number of nodes $(n) = 1000$. In the experiments we have not placed the tangents optimally according to Lemma 4.1; rather we have placed the tangents at equidistant points on the x-axis. We ran several experiments and found that this simple technique works quite as well.

5.2 Performance analysis of *DReMo* monitoring phase:

In this mode the nodes are not allowed to deploy the convergecast-broadcast to rebuild the model and only raise alerts when the model is out-of-date. This allows us to demonstrate the convergence properties and message complexity of *DReMo*.

Fig. 3 shows a typical dataset and the performance of the nodes. For this mode of operation of *DReMo*, we have chosen the data such that for the odd epochs, $R^2_{odd} = 0.7429$ while for the even epochs $R^2_{even} = 0.2451$ as shown in Fig. 3(a). This means that, for the odd epochs, the regression coefficients at each node matches with the weights of the data generator, while for the even epochs they do not. The redline is the error threshold ϵ. In all the experiments reported in this mode of operation of *DReMo*, the goal at each node is to check if the model fits the data *i.e.* if $R^2_{odd} > \epsilon$ for the odd epochs and $R^2_{even} < \epsilon$ for the even ones. As we see in Fig. 3(b), accuracy is very high (close to 100%) for each epoch, once the algorithm converges after the initial data change. Fig. 3(c) shows the monitoring messages per node plotted against time. For the default leaky bucket size of 2000, the maximal rate of messages per sub-epoch (*i.e.* data change) is bounded by $2 \times 20,000/2000 = 20$ for *DReMo*, assuming 2 neighbors per node on average. Also, an algorithm which broadcasts the data for each change will have this maximal rate to be 2 per sub-epoch. For *DReMo*, this rate of messages per node per sub-epoch has been calculated to be only 0.025, well below these maximal rates.

For *DReMo*, size of the local dataset plays a vital role in the accuracy and message complexity. Increasing the number of data points per node improves the quality of the local sufficient statistics and hence lowers the messages required by *DReMo* to agree with its neighbors. This hypothesis is verified by Fig. 4, which shows an increase in accuracy (top row) and decrease in messages (bottom row) plotted against time for different $|S_i| = 10$, 50 and 100 (left to right). The rate of messages are 0.67, 0.045, and 0.013 per node per sub-epoch for the three cases respectively.

The actual value of ϵ does not affect the performance of *DReMo*, rather the distance between R^2 and ϵ plays a major role. Closer ϵ is to R^2 for any epoch, the more difficult the problem becomes for that epoch. By varying ϵ between 0 and 1, we demonstrate the performance of *DReMo* with different levels of problem difficulty as shown in Fig. 5. The ϵ values demonstrated here are 0.2, 0.5, and 0.7 from (left to right, all columns). Recall that the value of $R^2_{odd} = 0.7427$ for odd epochs and $R^2_{even} = 0.2451$ for even epochs. For $\epsilon = 0.2$, R^2_{even} is very close to the threshold and hence, the accuracy is close to 60% with high message complexity (leftmost column, both top and bottom figures). For the same ϵ, checking for $R^2_{odd} > 0.2$ is a much simpler problem. On the other hand, for $\epsilon = 0.7$, R^2_{odd} is very close to ϵ. This is reflected in the decrease in accuracy and corresponding increase in messages for the last column (both top and bottom) figures. In this case, checking if $R^2_{even} < 0.7$ becomes simple.

5.2.1 Convergence rate:

Fig. 6(a) demonstrates the convergence rate of *DReMo* for different network sizes. We have plotted the performance from the beginning of one epoch till the time the nodes converge to 99% accuracy. At time 0, the accuracy is 0%. When the data changes at 20,000 ticks, accuracy increases and then again drops because the nodes need more information to agree on the outcome. At 40,000 ticks, when the data changes again, the accuracy increases and it keeps increasing till it reaches close to 100%. Even with data changing at subsequent sub-epochs, we do not see any drop in accuracy. For these network sizes, convergence occurs at the following simulator ticks: 50441 (500 nodes), 49282 (1000 nodes), 47120 (2000 nodes), and 47989 (4000 nodes).

5.2.2 Scalability:

Fig. 6(b) shows the accuracy and Fig. 6(c) shows the messages (separately for the odd and even epochs) as the number of nodes is varied from 500 to 4000. Each point in the accuracy plot is the average accuracy of *DReMo* over the last 80% of time for each epoch. Similarly, each point in the messages plot shows the messages per node per sub-epoch during the later 80% of the epoch. For both these plots, the circles represent the odd epochs, while the squares represent the even epochs and bars represent the standard deviation over 5 runs of the experiment. Since both accuracy and messages do not vary for different network sizes we can conclude that *DReMo* is highly scalable.

We have also run several experiments by varying the other parameters — dimension of the data, size of the leaky bucket and number of tangent lines. For all of these parameters, the accuracy and messages do not vary much. We do not present detailed graphs here due to lack of space.

5.3 Performance analysis of *DReMo* with convergecast-broadcast:

We now shift our focus to the other mode of operation of *DReMo* — when the algorithm monitors the model and rebuilds it, if outdated. Fig. 7 shows a typical run of the experiment. Fig. 7(a) shows the R^2 value of all nodes at each time instance. The redline is the default value of $\epsilon = 0.9$. The plot shows that *DReMo* rebuilds the model at every new epoch. Once the model is rebuilt, the value of R^2 drops below ϵ and only the efficient local monitoring algorithm operates for the rest of the epoch. The algorithm has a high true positive rate (alerts raised when necessary) and a very low false positive rate (unnecessary alerts). Fig. 7(b) shows the monitoring messages per node per sub-epoch and Fig. 7(c) shows the cumulative messages exchanged for recomputing the model. As is evident from Fig. 7(b) and 7(c), the algorithm offers a very low overhead of monitoring.

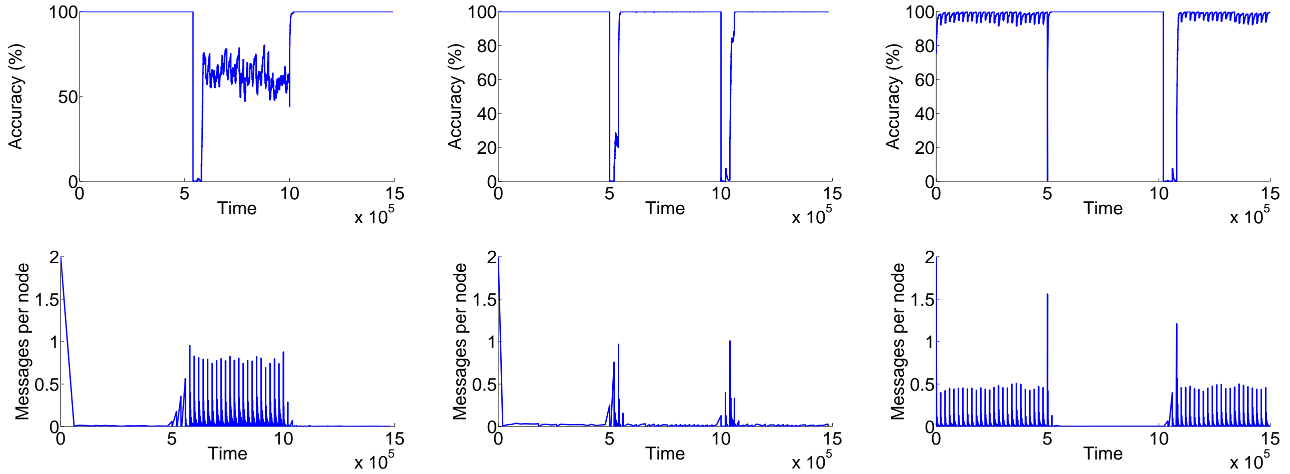
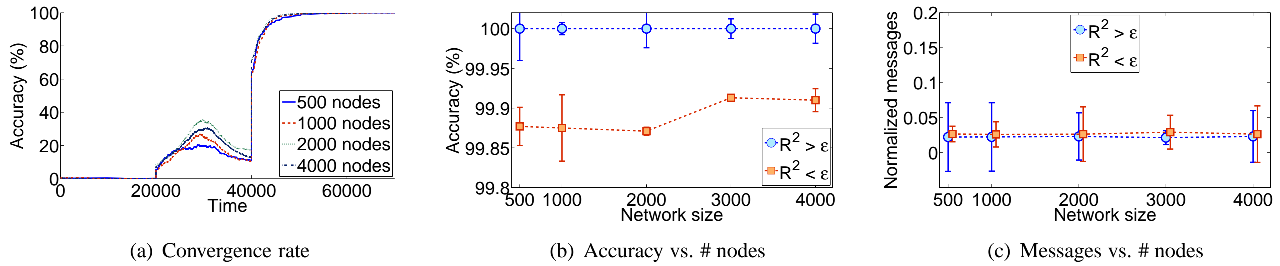
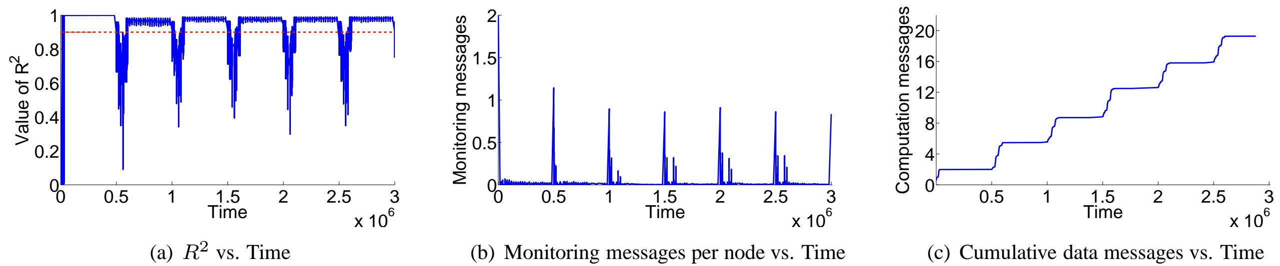


Figure 5: Variation of accuracy (top row) and messages (bottom row) for different $\epsilon = 0.2$, 0.5 and 0.7 from left to right. Recall that $R^2 = 0.2451$ for the even epochs, and hence $\epsilon = 0.2$ makes it very close to the threshold (left column). Similarly, for the odd epochs, $R^2 = 0.7429$ and so $\epsilon = 0.7$ makes this too close to the threshold. In both these cases there is decrease in accuracy and increase in messages.



(a) Convergence rate (b) Accuracy vs. # nodes (c) Messages vs. # nodes

Figure 6: Convergence rate and scalability of *DReMo* for different network sizes. Leftmost plot shows convergence rate. The next two figures demonstrate accuracy and messages with the size of the network.



(a) R^2 vs. Time (b) Monitoring messages per node vs. Time (c) Cumulative data messages vs. Time

Figure 7: Accuracy and messages for entire *DReMo* algorithm while both monitoring and re-building the model.

Each data message consists of the following two matrices: $X_i^T X_i$ $(d \times d)$ and $X_i^T y_i$ $(d \times 1)$. The number of bytes transmitted in each data message is $d^2/2 + d$, d being the number of features.

We study the effect of two parameters *viz.* ϵ and the alert wait period τ on the accuracy and messages of *DReMo*. Fig. 8 shows the variation of average R^2 over all nodes and messages for different values of ϵ. The (red) squares show the average R^2 value over the entire experiment duration, ignoring the epoch transitional periods. The (blue) circles

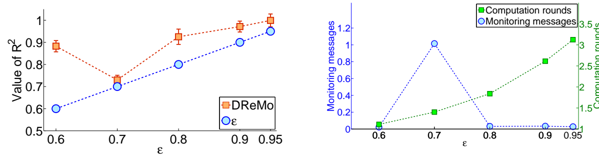


Figure 8: Accuracy and messages of *DReMo* vs. ϵ.

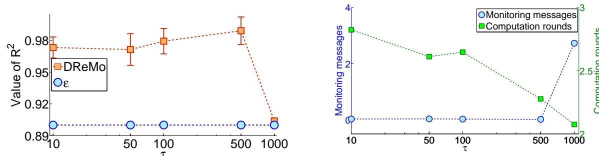


Figure 9: Accuracy and messages of *DReMo* vs. τ.

show the respective ϵ values. This plot shows that (1) the average computed R^2 is always higher than the ϵ value, and (2) the R^2 value increases with increasing ϵ to maintain the required accuracy since higher ϵ implies more rigid model fitting requirement. The second figure shows both the monitoring messages and convergecast broadcast rounds per node per epoch. The monitoring messages vary between 0.028 and 1.0154 – far less than the maximal rate of 20 and 2 messages as discussed earlier. Also the average number of computation rounds vary between 1.1 and 3.12 which means that new models are built between 1 and 3 times per epoch. As can be seen from the graph, this value is small for lower values of ϵ since model fitting requirements are relaxed.

We have also varied the alert wait period τ to take values 10, 50, 100, 500 and 1000. Fig. 9 shows the variation of accuracy and messages for the values of τ. Average R^2 value varies very little, and always stays greater than ϵ. As expected, the convergecast broadcast rounds per node per epoch decrease and the monitoring messages increase since the convergence of *DReMo* is delayed with increasing values of τ. The optimal value of τ is that for which both monitoring messages and computation rounds are minimized.

6 Application to Electrical Smart Grid monitoring

Electrical smart grids (ESG) provide an exciting venue for deploying this algorithm in a realistic setting. In this section we demonstrate how we can monitor the CO_2 emission from energy usage in electric grids. Since data in the electric grid is inherently distributed, this calls for a distributed monitoring algorithm. Unfortunately, a lot of the electric grid performance data is proprietary and not available for experimental purposes. As a result we have used the data available from EIRGRID (`http://www.eirgrid.com/`). It maintains GRID25 — an efficient power generation and transmission infrastructure which seamlessly connects both fossil fuel generation plants and renewable energy sources. All of the major generating plants feed into this grid for power to be transmitted. EIRGRID publishes system performance data every 15 mins. The data consists of the following: electricity demand (in Mega Watts), wind generation (in Mega Watts) and CO_2 emissions (in Tonnes per hour). This dataset has also been used in [10] for forecasting electricity demand using kernel regression.

Our goal in this work is to demonstrate the ability of *DReMo* in assessing the state of the ESG distribution system. We have used wind generation and electricity demand as inputs in order to model CO_2 emission, with the underlying assumption that higher than usual CO_2 level indicates higher fossil fuel burning and hence lesser green energy generation. Detection of such events may ultimately help the grid companies to dynamically switch on or off more renewable energy sources. We have downloaded these three features for a period of 9 months Jan 01, 2010 to Sep 30, 2010 (273 days). Since the data is collected every 15 mins, there are a total of 26,208 samples in our full dataset. In our setup, we take each month's data as an epoch and at every 500,000 simulator ticks replace all of the data of all nodes with the next month's data. We have divided each month's data (approximately 2900 points) into 50 nodes such that each node has approximately 55 data points per epoch. We have taken a small sample of the data from the first epoch and have built a regression model and used it as the reference model throughout the remaining epochs of the distributed experiment. It is worthwhile to mention here that we have used *DReMo* to only detect the changes (no convergecast broadcast). In the absence of real faults in the data, we have altered the CO_2 values of the month of June by $\pm 20\%$ of the actual values. Fig. 10 shows the experimental results. The top figure shows the percentage of nodes agreeing that the model fits the data at each time instance. For the entire period till the end of May we see a good agreement between the model of Jan 2010 and the data, with some intermediate false alarms. The interesting phenomenon occurs during June 2010 and the algorithm correctly detects the event. After June, when the data changes again, the algorithm recovers and shows high accuracy. In order to validate this, we have built regression models for each epoch separately, and found that they are very similar, except the data for June 2010. The bottom plot shows the messages exchanged by *DReMo* for this monitoring.

7 Conclusion

In this paper we have presented a new method for monitoring linear regression models in distributed environments. The proposed algorithm uses R^2 statistic to assess the quality of a linear regression model in a distributed fashion and

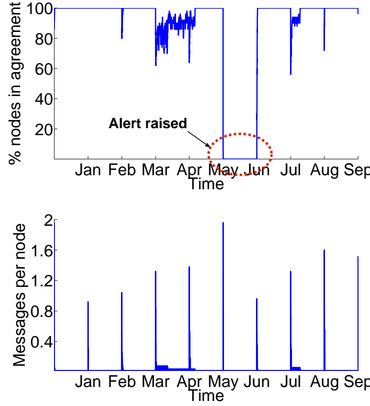


Figure 10: Accuracy and messages of *DReMo* for smart grid data monitoring. The circled region (June) is when the algorithm correctly detects the fault in the system.

raises an alert whenever the value of the statistic drops below a predefined threshold ϵ. The algorithm is provably correct and has very low communication overhead — both ideal for easy deployment on top of existing distributed infrastructures such as data centers and electrical smart grids. Another important characteristic of the proposed algorithm is the choice of R^2 as the monitoring function, which helps one to choose a model fidelity threshold (ϵ) in a data independent manner, making it amenable to diverse applications. Extensive experimental evaluation using both synthetic and real data corroborate the performance claims of *DReMo*. In our future research, we plan to extend this scale-free monitoring algorithm to nonlinear models such as kernel regression.

References

[1] K. Bhaduri and H. Kargupta. A Scalable Local Algorithm for Distributed Multivariate Regression. *Stat. Anal. and Data Mining*, 1(3):177–194, 2008.

[2] Y. Birk, L. Liss, A. Schuster, and R. Wolff. A Local Algorithm for Ad Hoc Majority Voting via Charge Fusion. In *Proc. of DISC'04*, pages 275–289, 2004.

[3] P. Bodik, M. Goldszmidt, A. Foxo, D. Woodard, and H. Andersen. Fingerprinting the Datacenter: Automated Classification of Performance Crises. In *Proc. of ECCV'10*, pages 111–124, 2010.

[4] S. Boyd, A. Ghosh, B. Prabhakar, and D. Shah. Gossip Algorithms: Design, Analysis and Applications. In *Proc. of INFOCOMM'05*, pages 1653–1664, Miami, March 2005.

[5] L.R. Ford and D.R. Fulkerson. *Flows in Networks*. Princton University Press, 1962.

[6] C. Guestrin, P. Bodi, R. Thibau, M. Paski, and S. Madden. Distributed Regression: an Efficient Framework for Modeling Sensor Network Data. In *Proc. of IPSN'04*, pages 1–10, Berkeley, California, 2004.

[7] D. E. Hershberger and H. Kargupta. Distributed Multivariate Regression Using Wavelet-based Collective Data Mining. *JPDC*, 61(3):372–400, 2001.

[8] J.M. Jaffe and F.H. Moss. A Responsive Distributed Routing Algorithm for Computer Networks. *IEEE Trans. on Comm.*, 30(7):1758–1762, 1982.

[9] D. Kempe, A. Dobra, and J. Gehrke. Computing Aggregate Information using Gossip. In *Proc. of FOCS'03*, 2003.

[10] O. Kramer, B Satzger, and J. Lässig. Power Prediction in Smart Grids with Evolutionary Local Kernel Regression. In *Proc. of HAIS'10*, volume 6076, pages 262–269, 2010.

[11] P. Luo, H. Xiong, K. Lü, and Z. Shi. Distributed Classification in Peer-to-Peer Networks. In *Proc. of KDD'07*, pages 968–976, 2007.

[12] M. Mehyar, D. Spanos, J. Pongsajapan, S. H. Low, and R. Murray. Distributed Averaging on Peer-to-Peer Networks. In *Proc. of CDC'05*, Spain, 2005.

[13] D. Scherber and H. Papadopoulos. Distributed Computation of Averages Over ad hoc Networks. *IEEE J. on Selected Areas in Comm.*, 23(4):776–787, 2005.

[14] I. Sharfman, A. Schuster, and D. Keren. A Geometric Approach to Monitoring Threshold Functions Over Distributed Data Streams. *ACM Trans. on Database Sys.*, 32(4):23, 2007.

[15] S. J. Stolfo, A. L. Prodromidis, S. Tselepis, W. Lee, D. W. Fan, and P. K. Chan. JAM: Java Agents for Meta-Learning over Distributed Databases. In *Proc. of KDD'97*, pages 74–81, Newport Beach, California, 1997.

[16] R. Wolff, K. Bhaduri, and H. Kargupta. A Generic Local Algorithm for Mining Data Streams in Large Distributed Systems. *TKDE*, 21(4):465–478, 2009.

A Proof of Lemma 4.1

Proof. Let the equation of the parabola be $y = \tilde{\epsilon}x^2$. Referring to Fig. 1, minimizing the area of the tie region is equivalent to maximizing the area of the colored half-spaces.

Tangent to this parabola at any point (t_x, t_y) is $y = t_y + 2\tilde{\epsilon}t_x(x - t_x)$. Now the tangent at x_1, y_1 meets the x-axis at $(x_1/2, 0)$ and intersects with the tangent at (x_2, y_2) at $(\frac{x_1+x_2}{2}, \tilde{\epsilon}x_1x_2)$, using the fact that $(y_i = \tilde{\epsilon}x_i^2)$. Therefore, area of the red triangle (marked as 1 in the figure): $A_1 = \frac{1}{4}\tilde{\epsilon}x_1x_2^2$. Using tangent equations at (x_2, y_2) and (x_3, y_3), we can express the area of trapezoid 2 as: $A_2 = \frac{1}{4}\tilde{\epsilon}x_2(x_3^2 - x_1^2)$. Similarly the area of trapezoid 3 is $A_3 = \frac{1}{4}\tilde{\epsilon}x_3(x_4^2 - x_4^2)$ and 4 is $A_4 = \frac{1}{4}\tilde{\epsilon}(2x_{max}x_4 - x_4^2 + x_3x_4)(2x_{max} - x_3 - x_4)$. We can extend this for t tangents and maximize the following:

$$\max\left\{\sum_{i=1}^{t} A_i\right\} = \max\{x_1x_2^2 + x_2(x_3^2 - x_1^2) + \cdots + (2x_{max}x_t - x_t^2 + x_{t-1}x_t)(2x_{max} - x_{t-1} - x_t)\}$$

Taking partial derivatives of the above expression with each x_i and setting them to 0 gives us the following values of the x_i's:

$$x_1 = \frac{2x_{max}}{2t+1} \quad x_2 = \frac{4x_{max}}{2t+1} \quad \ldots \quad x_t = \frac{2tx_{max}}{2t+1}. \quad \blacksquare$$