**Parallel Stochastic Gradient Descent Method for Multiple Linear Regression**

**1. INTRODUCTION**

Multiple linear regression is one of the most common methods in statistical analysis and machine learning for producing functions that model the effect of a set of independent variables(X) on a single dependent variable(y). The coefficients in the learned function are determined by minimizing the error between the model’s predicted y value for the testing data and the observed y values. Two methods are particularly useful for minimizing the error: gradient descent, which solves the problem in an explicitly iterative manner, and the normal equation, which solves the problem analytically.

**2. Motivation**

Computing the normal equation is a very costly process and as the amount of data gets very large, its performance becomes much worse than gradient descent. However, methods for computing the normal equation rely heavily on matrix multiplications, for which parallel implementations are extremely straightforward. Though the gradient descent process is more efficient for large numbers of inputs, a way to decompose gradient descent is not at all apparent. The inherent iterative nature of the algorithm excludes the possibility of parallelization. Because of this, implementations on contemporary parallel platforms, such as a MapReduce implementation [1] and a CUDA implementation [2], by a number of methods, exploit the normal equation’s inherent parallelizability. However, a feature of the gradient descent algorithm, the learning rate, usually found by manual tuning, lends itself to parallelized speedup.

A current parallel implementation, using CUDA on GPGPUs solves for the minimum of the Sum of Squares Error function, by computing a covariance matrix, the eigenvalues, and the eigenvectors [2]. Common methods for finding the covariance matrix are typically of time complexity O(n3) [3], and computing eigenvalues and vectors isn’t any better. Another implementation, using MapReduce, minimizes the function by solving for the normal equation using a QR decomposition of the feature matrix [1]. Computing the QR decomposition is another O(n3) process. It would be nice if we could find a way to parallelize a more efficient algorithm so that as data volumes get even more massive, processing times don’t explode.

**3. Insights**

Gradient descent uses a value called the learning rate, which dictates the size of step the algorithm takes in the direction it believes the minimum to be in. Choice of a good learning rate is a bit more of an art than a science, but is crucial to the performance of the algorithm. If the learning rate is chosen too small, the algorithm will make little progress and fail to converge in any reasonable stretch of time. If it is chosen too large, the algorithm will shoot off to infinity, continuously increasing the distance to the minimum. There are a number of ways to tune the learning rate, including manual observance, modification, and repetition of the process, implementation of an adaptive learning rate, and processes that attempt to automatically determine a suitable rate ahead of time. One such algorithm is called the Fast Automatic Step Size Estimation (FASSE) [4] and provides an improvement over many learning rate estimations. But this doesn’t verifiably choose the absolute optimal rate; it just provides an empirical improvement. Even if it was optimal, the adaptation of the learning rate while the algorithm runs is another crucial factor; one for which the best solution is usually found heuristically.

**4. Methods**

Using any parallel architecture, within which each processing unit can communicate with any other, potential values for the learning rate and adaptation function can be varied across nodes. The idea is to use the largest value for the learning rate, which can still multiplicatively decrease and converge to the minimum. If a node that is provided optimal values converges on the minimum, it may issue a command to the remaining nodes to cease work. While this sort of parallelization does not serve to reduce the complexity of the algorithm, it does result in generally much faster run times, as the value of the learning rate greatly affects the performance at every iteration. One set of learning rate specifications may result in the algorithm converging orders of magnitude more quickly than the rest.

**5. Implementation and Future Development**

The necessary function of communication between nodes implies that a message-passing paradigm such as MPI is well suited to handle the job. In the future, tricks of this nature may serve to provide the ability of implementation with contemporary standards like MapReduce, CUDA, Spark and so forth. Stochastic gradient descent has the ability to be implemented on parallel architectures. In situations where the size of the problem is large enough, and the amount of processing power available is right, its application may prove to be ideal in a number of data mining scenarios that are typically solved by analytical methods.

**References**

1. Moufida Adjout Rehab and F. Boufares

“Scalable Massively Parallel Learning of Multiple Linear Regression Algorithm with MapReduce”

2. Jyoti B. Kulkarni, A. A. Sawant, Vandana S. Inamdar

“Database Processing by Linear Regression on GPU using CUDA”

3. Christian Igil, Thorsten Suttorp, Nikolaus Hansen

“A Computational Efficient Covariance Matrix Update and a (1+1)-CMA for Evolution Strategies”

4. Yuchuan Qiao, Baldur van Lew, Boudewijn P.F. Lelieveldt and

Marius Staring

“Fast Automatic Step Size Estimation for Gradient Descent Optimization of Image Registration”