Scalable Machine Learning on Popular Analytic Languages with Parallel Data Summarization

Sikder Tahsin Al-Amin, Carlos Ordonez

Department of Computer Science
University of Houston, Houston TX 77204, USA

Abstract. Machine learning requires scalable processing. An important acceleration mechanism is data summarization, which is accurate for many models and whose summary requires a small amount of RAM. In this paper, we generalize a data summarization matrix to produce one or multiple summaries, which benefits a broader class of models, compared to previous work. Our solution works well in popular languages, like R and Python, on a shared-nothing architecture, the standard in big data analytics. We introduce an algorithm which computes machine learning models in three phases: Phase 0 pre-processes and transfers the data set to the parallel processing nodes; Phase 1 computes one or multiple data summaries in parallel and Phase 2 computes a model in one machine based on such data set summaries. A key innovation is evaluating a demanding vector-vector outer product in C++ code, in a simple function call from a high-level programming language. We show Phase 1 is fully parallel, requiring a simple barrier synchronization at the end. Phase 2 is a sequential bottleneck, but contributes very little to overall time. We present an experimental evaluation with a prototype in the R language, with our summarization algorithm programmed in C++. We first show R is faster and simpler than competing big data analytic systems computing the same models, including Spark (using MLlib, calling Scala functions) and a parallel DBMS (computing data summaries with SQL queries calling UDFs). We then show our parallel solution becomes better than single-node processing as data set size grows.

1 Introduction

Machine learning is essential to big data analytics [1], [10], [20]. With higher data volume and varied data types, many new machine learning models and algorithms aiming at scalable applications [5], [6]. Popular big data systems like parallel DBMS (e.g. Vertica, Teradata) and Hadoop systems (e.g. Spark, HadoopDB, Cassandra) offer ample storage and parallel processing of popular machine learning algorithms but the processing time can be slower. Besides, they do not have efficient native support for matrix-form data and out-of-box sophisticated mathematical computations. Nowadays, with the advancement of cloud technology (e.g. AWS, Azure, Google Cloud), data can be stored in a single machine or a large cluster. Analysts can distribute the data set in the cloud and analyze it instead of avoiding the complex set up process of parallel

systems. However, cloud systems are costly (costs vary depending on services), and using the cloud for simple analysis may not be beneficial (i.e., may not be cost-effective). On the other hand, mathematical systems like Python, R provide comprehensive libraries for machine learning and statistical computation. Analysts can install them easily and analyze small data sets locally in their machine.But those systems are not designed to scale to large data sets and the single machine is challenging to analyze the large data sets [10].

Within big data, data summarization has received much attention [15], [5], [12] among the machine learning practitioners. Summarization in parallel DBMS is losing ground as SQL queries are not a good choice for analytics and UDFs are not portable. Hadoop systems like Spark [21], is a better choice but they are slow for a few processing nodes, has scalability limitations, and even slower than DBMS in case of summarization [15]. With these motivations in mind, here, we present a data summarization algorithm that works in a parallel cluster, does not require a complex set up of parallel systems (e.g. DBMS, Hadoop), is solvable with popular analytic languages (e.g. Python, R) and is faster than the existing popular parallel systems. Exploiting the summarization, we can compute a wide variety of machine learning models and statistics on the data set either in a single machine or in parallel [15] [5].

Our contributions include the following: (1) We present a new three-phase generalized summarization algorithm that works in a parallel cluster (or a remote cluster in the cloud). (2) We improve and optimize the summarization algorithm for classification/clustering problems initially proposed in [5]. (3) We improve and optimize the technique to read the data set from disks in blocks. (4) We study the trade-offs to compute data summarization in a parallel cluster and a single machine. Analysts can have a better understanding which is a common problem nowadays. In our work, we used R as our choice of analytic language combined with C++ to develop our algorithms, but it can be applied to other analytic platforms like Python. With the dedicated physical memory, R or Python itself cannot scale to deal with data sets larger than the proportion of memory allocated and is forced to crash. Also, we used a local parallel cluster to perform the experiments but our research applies to both local parallel cluster and a remote cluster in the cloud. Experimental evaluation shows our generalized summarization algorithm works efficiently in a parallel cluster, scalable and much faster than Spark and a parallel DBMS. This article is a significant extension and deeper study of [15], where the Gamma summarization matrix was initially proposed.

This is the outline for the rest of the article. Section 2 introduces the definitions used throughout the paper. Section 3 presents our theoretical research contributions where we present our new algorithm to compute summarization in a parallel cluster. Section 4 presents an extensive experimental evaluation. We discuss closely related work in Section 5. Conclusions and directions for future work are discussed in Section 6.

Table 1: Basic symbols and their description

Symbol	Description	Symbol	Description
X	Data set	d	Number of attributes/columns in X
X_I	Partitioned data set	Γ	Gamma Summarization Matrix
$\mathbf{X}$	Augmented X with Y	Γ^k	k-Gamma Summarization Matrices
Y	Dependent Variable	Θ	Machine learning model
Z	Augmented X with 1s and Y	p	Number of processoring nodes
n	Number of records/rows in X	b	Blocks to read data

2 Definitions

2.1 Mathematical Definitions

We start by defining the input matrix X which is a set of n *column* vectors. All the models take a $d \times n$ matrix X as input. Let the input data set be defined as $X = \{x_1, ..., x_n\}$ with n points, where each point x_i is a vector in $\mathbf{R}^d$. Intuitively, X is a wide rectangular matrix. X is augmented with a $(d+1)$th dimension containing an output variable Y, making X a $(d+1) \times n$ matrix and we call it $\mathbf{X}$. We use $i = 1 \ldots n$ and $j = 1 \ldots d$ as matrix subscripts. We augment $\mathbf{X}$ with an extra row of n 1s and call that as matrix Z with a $(d+2) \times n$ dimension. Table 1 shows the basic symbols and their description used throughout the paper.

We use Θ to represent a machine learning model or a statistical property in a general manner. Thus Θ can be any model like: LR, PCA, NB, KM or any statistical property like: Covariance or Correlation matrix. For each ML model Θ can be defined as, $\Theta = \{list\ of\ matrices/vectors\}$. For LR: $\Theta = \beta$, the vector or regression coefficients; for PCA: $\Theta = U, D$, where U are the eigen vectors and D contains the squared eigenvalues obtained from SVD; for NB: $\Theta = \{W, C, R\}$, where W is the vector of k class priors, C is a set of k mean vectors and R are k diagonal matrices with standard deviations; and for KM: $\Theta = \{W, C, R\}$, where W is a vector of k (number of clusters) weights, C is a set of k centroid vectors and R is a set of k variance matrices.

2.2 Parallel Cluster Architecture

We are using p processing nodes in parallel. Each node has its CPU and memory (shared-nothing architecture) and it cannot directly access another node storage. Therefore, all processing nodes communicate with each other transferring data. And, data is stored on disk, not in virtual memory,.

3 Theory and Algorithm

First, we give an overview of the original summarization matrix introduced in [15] for DBMS. Here, we make several improvements. We propose a new three-phased generalized algorithm that computes summarization in a parallel cluster.

Also, we improve and optimize the k-summarization matrices algorithm which was introduced in [5] (as Diagonal Gamma Matrix). Next, we discuss how we can integrate the parallel algorithm into an analytic language. Finally, we analyze the time and space complexity of our algorithm.

3.1 Gamma Summarization Matrix and ML Model Computation

Here, we review the Gamma summarization matrix (Γ) [5], [15] and computation of several ML models (Θ) exploiting Γ. The main algorithm had two steps:

1. Phase 1: Compute summarization matrix: one matrix Γ or k matrices Γ^k.
2. Phase 2: Compute model Θ based on Gamma matrix (matrices).

Phase 1: Matrix Γ (Gamma), is a fundamental matrix that contains a complete, accurate, and sufficient summary. If we consider X as the input data set, n counts the total number of points in the dataset, L is the linear sum of x_i, and Q is the sum of vector outer products of x_i, then from [15], the Gamma (Γ) is defined below in Eq. 1. We first define n, L, Q as: $n = |X|$, $L = \sum_{i=1}^{n} x_i$, and $Q = XX^T = \sum_{i=1}^{n} x_i \cdot x_i^T$. Now, the Gamma ($\Gamma$) matrix:

$$\Gamma = \begin{bmatrix} n & L^T & \mathbf{1}^T \cdot Y^T \\ L & Q & XY^T \\ Y \cdot \mathbf{1} & YX^T & YY^T \end{bmatrix} = \begin{bmatrix} n & \sum x_i^T & \sum y_i \\ \sum x_i & \sum x_i x_i^T & \sum x_i y_i \\ \sum y_i & \sum y_i x_i^T & \sum y_i^2 \end{bmatrix} \quad (1)$$

X is defined as a $d \times n$ matrix, and Z is defined as a $(d+2) \times n$ matrix as mentioned in Section 2. From [15], we can easily understand that Γ matrix can be computed in the two ways: (1) matrix-matrix multiplication i.e., ZZ^T (2) sum of vector outer products i.e., $\sum_i z_i \cdot z_i^T$ So, in short, the Gamma computation can be defined as: $\Gamma = ZZ^T = \sum_{i=1}^{n} z_i \cdot z_i^T$

Now, from [5], k-Gamma (Γ^k) is given in Eq. 2. The major difference between the two forms of Gamma is, we do not require parameters off the diagonal in Γ^k as in Γ. So, we need only a few parameters out of the whole Γ, namely, n, L, L^T, Q. That is, we require only a few sub-matrices from Γ. Also, in Γ, the Q is computed completely whereas in Γ^k, the Q is diagonal. So, we can also call this a Diagonal-Gamma matrix.

$$\Gamma^k = \begin{bmatrix} n & L^T & 0 \\ L & Q & 0 \\ 0 & 0 & 0 \end{bmatrix}, where\ Q = \begin{bmatrix} Q_{11} & 0 & 0....... & 0 \\ 0 & Q_{22} & 0....... & 0 \\ 0 & 0 & Q_{33}..... & 0 \\ 0 & 0 & 0........ & Q_{dd} \end{bmatrix} \quad (2)$$

Phase 2: Both Γ and Γ^k provide summarization for a different set of machine learning models (Θ). For Linear Regression (LR) and Principal Component Analysis (PCA), we need one full Γ assuming element off-diagonal is not zero. And for Naïve Bayes (NB) and k-means (KM), k-Gamma matrices are needed where k is the number of classes/clusters. We briefly discuss how to compute each model (Θ) below. The details of the model computation are discussed in [5].

LR: We can get the column vector of regression coefficients ($\hat{\beta}$), from the above mentioned Γ, with: $\hat{\beta} = Q^{-1}(\mathbf{X}Y^T)$

PCA: There are two parameters, namely the set of orthogonal vectors U, and the diagonal matrix (D^2) which contains the squared eigen values. We compute ρ, the correlation matrix as $\rho = UD^2U^T = (UD^2U^T)^T$. Then we compute PCA from the ρ by solving Singular Value Decomposition (SVD) on it. Also, we express ρ in terms of sufficient statistics as: $\rho_{ab} = \frac{(nQ_{ab} - L_a L_b)}{(\sqrt{nQ_{aa} - L_a^2}\sqrt{nQ_{bb} - L_b^2})}$

NB: Here, we need the k-Gamma matrix. We focus on $k = 2$ classes for NB. We compute N_g, L_g, Q_g as discussed in Phase 1 for each class. The output is three model parameters: mean (C), variance (R), and the prior probabilities (W). We can compute these parameters from the Γ^k matrix for each class label with the following statistical relations. Here, $N_g = |X_g|$ and we take the diagonal of $L \cdot L^T$ and Q, which can be manipulated as a 1-dimensional array instead of a 2D array.

$$W_g = \frac{N_g}{n};\; C_g = \frac{L_g}{N_g};\; R_g = \frac{Q_g}{N_g} - diag\frac{[L_g L_g^T]}{N_g^2} \tag{3}$$

KM: Similar to NB, we introduce similar model parameters N_j, L_j, Q_j (where $j = 1,..,k$) as the subset of X which belong to cluster k, the total number of points per cluster ($|X_j|$), the sum of points in a cluster ($\sum_{\forall x_i \in X_j} x_i$) and the sum of squared points in each cluster ($\sum_{\forall x_i \in X_j} x_i x_i^t$) respectively. From these statistics, we compute C_j, R_j, W_j as similar to NB presented in Eq. 3. Then, the algorithm iterates executing two steps starting from random initialization until cluster centroids become stable. Step 1 determines the closest cluster for each point (using Euclidean distance) and adds the point to it. And Step 2 updates all the centroids C_k by computing the mean vector of points belonging to cluster k. The cluster weights W_k and diagonal covariance matrices R_k are also updated based on the new centroids.

3.2 Parallel Algorithm to Compute Multiple Data Summaries

Here, we present our main contributions. Our generalized computation of summarization matrix in the parallel cluster using p processing nodes is shown in Figure 1. We propose a new 3 phase algorithm to compute Γ (or Γ^k) in the parallel cluster and how ML models (Θ) can be computed exploiting it.

1. Phase 0: Pre-process the data set. Transfer data to the processing nodes (p nodes).
2. Phase 1: Compute summarization matrix in parallel across p nodes: Γ or Γ^k. This phase will return p partial (local) summarization matrices (Γ_I or Γ_I^k, $I = 1, 2, ..., p$)
3. Phase 2: Add partial summarization matrices to get final Γ or Γ^k on the master node. Compute model Θ based on Γ or Γ^k.

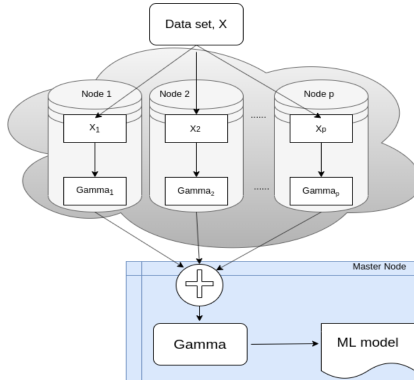


Fig. 1: Computation of Gamma matrix in a parallel cluster.

Phase 0: Data set X is moved to the p processing nodes. Nowadays, data can reside either in the parallel cluster, cloud (remote cluster), or in a large local machine. Also, the number of processing nodes may vary. In any case, data must be transferred into the processing nodes. We split the data set X into p processing nodes. There are several partitioning strategies available but we used the row-based partitioning (horizontal partitioning). As Γ is a $d \times d$ matrix, we need all the d columns in each node. If we choose column-based (vertical partitioning) or block-based partitioning, it is possible that the Γ in different nodes may end up having different sizes. A good way to select the data set size in each partition (row-based) is n/p. So, each node in the parallel cluster has the same number of rows except for the p-th node.

Phase 1: We compute Γ on each node locally. We optimize the technique to read data in blocks so that it can handle very large files. For each node, the partitioned data set (X_I) is read into $b = 1...b$ blocks of same size (m) where $m < |X_I|$. The block size depends on the number of records (n_I) in X_I. As discussed in [14], we define the block size as $\log n_I$. As $\log n_I \ll n_I$, even if n_I is very large, each block will easily fit in the main memory. Processing data one block at a time has many benefits. It is the key to being able to scale the computations without increasing memory requirements. External memory (or out-of-core) algorithms do not require that all of the data be in RAM at one time. Data is processed one block at a time, with intermediate results updated for each block. When all the data is processed, we get the final result. Here, we read each block (b) into the main memory and compute Gamma for that block ($\Gamma(b)$). This partial Gamma is added to the Gamma computed up to the previous block ($b - 1$). We iterate this process until no blocks are left and get the Gamma (Γ_I) for that node. As each node has all the d columns, the size of each Γ_I will be $d \times d$.

Optimization of k-Gamma Matrix: Similarly, for k-Gamma matrices (Γ^k) we perform the same procedure as mentioned above. First, we partition

the data set, then compute partial k-Gamma (Γ_I^k) in each node locally. We study the algorithm further and made an improvement to compute Γ^k from [5]. As discussed previously, for k-Gamma, we only need n, L and $diag(Q)$. Here both L and $diag(Q)$ can be represented as a single vector and we do not need to store Q as a matrix. Hence, Γ^k can be represented as a single matrix of size $d \times 2k$ where each Gamma is represented in two columns (L and Q). We still need to store the value of n in a row, which makes the Γ^k as $(d+1) \times 2k$. Hence, we are using minimal memory to store Γ^k even if the value of k is very large. This is a major improvement from the previous version where one Gamma Matrix was stored per class/cluster in the main memory. Also, we have to access only one matrix which is faster than accessing from a list of matrices in any programming language. Computing Γ_I on each node can be shown in Algorithm 1. Computing Γ_I^k will be similar to Algorithm 1.

Data: Partitioned Data Set ($X_I, I = 1, 2..p$) from Phase 0
Result: Γ_I
Read X_I into $b = 1, 2, ..., b$ blocks;
while *next(b)* **do**
  read(b) ;
  Γ_b = Gamma(b) ;
  $\Gamma_I = \Gamma_b + \Gamma_I$;
end
return Γ_I

Algorithm 1: Sequential Gamma computation on each node (Phase 1)

Phase 2: After all the Γ_Is ($\Gamma_1, \Gamma_2, ..., \Gamma_p$) are computed in each node locally, we need to combine them and get the final Γ. In Phase 2, at first, all the partial Γ_Is are sent to a master node to perform the addition (sequential) or we can perform it in a hierarchical binary tree manner. Hierarchical processing performs the addition in multiple levels (bottom-up) until we get the final addition at the top level. On the other hand, in sequential processing, all the partial Γ_I are transferred to the main memory of the master node. The partial Γ_Is are sent in a compressed format. So, we decompress it on the master node. Now, to get the final Gamma matrix (Γ), we just need to perform a simple matrix-addition operation of all the partial Γ_Is. That is, we compute $\Gamma = \Gamma_1 + \Gamma_2 + ... + \Gamma_p$. Similarly, the final Γ^k will be $\Gamma^k = \Gamma_1^k + \Gamma_2^k + ... + \Gamma_p^k$. Now, using this Γ or Γ^k, we compute the machine learning models (Θ) at the end of Phase 2. Model (Θ) computations are discussed in Section 3.1.

3.3 Integrating the Parallel Algorithm into an Analytic Language

We will discuss how we integrated our algorithm, into the R language, using the Rcpp [7] library. R, a dynamic language, provides a wide variety of statistical and graphical techniques, and is highly extensible. However, our solution is applicable to any other programming language which provides an API to call C++ code. Specifically, our solution can easily work in Python, launching k Python Gamma

processes in parallel. On the other hand, SQL queries are slow, UDFs are not portable, Spark not easy to debug and Java is slower than C++. So, analytic languages like Python and R are more popular among analysts nowadays.

Key insight: Phase 1 must work in C++ (or C). The sum of vector outer products must be computed block by block in C++, not in the host language. Computing $z_i * z_i^T$ in a loop in R or any other analytic language is slow: usually one-row-at-a-time. Computing $Z * Z^T$ with traditional matrix multiplication is slow due to Z^T materialization, even in RAM. We used Rcpp, an R add-on package that facilitates extending R with C++ functions to compute Phase 1. Rcpp can be used to accelerate computation by replacing an R function with its C++ equivalent function. In Rcpp, only the reference gets passed to the other side but not the actual value when we pass the values. So, memory consumption is very efficient and the run time is the same. In addition to Rcpp, we used the RCurl [11] package to communicate over the network.

Model computation in Phase 2 can be efficiently done calling existing R (or other analytic languages) functions. While Phase 1 is basically exploiting C++, Phase 2 uses the analytic language "as is". It would be too difficult and error-prone to reprogram all the ML models. Instead, our solution requires just changing certain steps in each numerical method, rewriting their equations based on the data summaries (1 or k). Our experiments will show model computation takes less than one second in every case, even for high d models.

As an extra benefit, our solution gives flexibility to the analyst to compute data summaries in a parallel cluster (local or cloud), but explore many statistics matrices and models locally. That is, the analysts can enjoy analytics "for free" without the overhead and price to use the cloud. Moreover, our parallel solution is simple, more general and we did not need any complicated library like "Revolution R" [17] that requires Windows operating system or "pbdR" [16] that provides high-level interfaces to MPI requires a complex set up process.

3.4 Time and Space Complexity Analysis

For parallel computation in the cluster, let p be the number of processing nodes under a shared-nothing architecture. We assume $d \ll n$ and $p \ll n$. From [15], the time complexity for computing the full Gamma in a single machine is $O(d^2n)$. As we are computing Γ in blocks per node, the time complexity is proportional to block size. Let, m be the number of records in each block and b be the total number of blocks per processing nodes and each block size is fixed. Then, for each block time complexity of computing Γ will be $O(d^2b)$. For a total of m blocks, it will be $O(md^2b)$. When all the blocks are read, $mb = n$. In our case of parallel computation, as each $X_I \epsilon X$ is hashed to p processing nodes, the time complexity will be $O(d^2n/p)$ per processing nodes. Computing Γ in total of b blocks of fixed size m will make the time complexity $O(md^2b/p)$. In case of k-Gamma matrix, we only compute L and diagonal of Q of the whole Gamma matrix. So, for Γ^k, it will be $O(mdb/p)$ in each processing nodes.

In case of transferring all the partial Γ, if we transfer to the master node all at once: $O(d^2)$, for sequential transfer: $O(d^2p)$, for hierarchical binary tree fashion:

Table 2: Base data sets description

Data set	d	n	Description	Models Applied
YearPredictionMSD	90	515K	predict if there is rain or not	LR, PCA
CreditCard	30	285K	predict if there is raise in credit line	NB, KM

$O(d^2p + log_2(p)d^2)$. We take advantage of Gamma to accelerate computing the machine learning models. So, the time complexity of this part does not depend on n and is $\Omega(d^3)$.

In case of space complexity and memory analysis, our algorithm uses very little RAM. In each node, space required by Γ in main memory is $O(d^2)$. And it is $O(kd)$ for Γ^k, where k is the number of classes/clusters. As Γ or Γ^k does not depend on n, the space required by each processing node in the parallel cluster will be same as computing it in a single node ($O(d^2)$ and $O(kd)$ respectively). Also, as we are adding the new Γ with the previous one for each block, the space does not depend on the number of blocks.

4 Experimental Evaluation

We present an experimental evaluation in this section. First, we introduce the systems, input data sets, and our choice of programming languages. We compare our proposed algorithm with Spark running in parallel clusters and a parallel DBMS to make sure our algorithm is competitive with other parallel systems. We also compare processing in parallel cluster vs single machine. All the time measurements were taken five times and we report the average excluding the maximum and minimum value.

4.1 Experimental Setup

Hardware and Software: We performed our experiments using our 8-node parallel cluster each with Pentium(R) Quadcore CPU running at 1.60 GHz, 8 GB RAM, and 1 TB disk space. For single machine, we conducted our experiments on a machine with Intel Pentium(R) Quadcore CPU running at 1.60 GHz, 8 GB RAM, 1 TB disk, and Linux Ubuntu 14.04 operating system. We developed our algorithms using standard R and C++. For parallel comparison, we used Spark-MLlib and programmed the models using Scala. And we used Vertica as a parallel DBMS.

Data sets: Computing machine learning models on raw data is not practical. Also, it has hard to obtain public data sets to compute all the models. Therefore, we had to use common data sets available and replicate them to mimic large data sets. We used two data sets as our base data sets: YearPredictionMSD and CreditCard data set, summarized in Table 2, obtained from the UCI machine learning repository. We include the information about the models which utilize these data sets. We sampled and replicated the data sets to get varying n (data set size)

Table 3: Time (in Seconds) to compute the ML models in our solution ($p = 8$ nodes) with Γ and in Spark ($p = 8$ nodes) (M=Millions)

Θ			Our solution (p=8)				Spark (p=8)		
(Data set)	n	d	Phase 0	Phase 1	Phase 2	Total	Partition	Compute Θ	Total
LR	1M	10	9	3	9	21	7	41	48
(Year-	10M	10	23	20	9	52	17	286	303
Prediction)	100M	10	317	209	9	535	161	1780	1941
PCA	1M	10	9	3	9	21	7	15	22
(Year-	10M	10	23	20	9	52	17	46	63
Prediction)	100M	10	317	209	9	535	161	277	438
NB	1M	10	11	4	9	24	7	Crash	Crash
(credit-	10M	10	28	27	9	64	25	Crash	Crash
card)	100M	10	335	243	9	587	231	Crash	Crash
KM	1M	10	11	4	9	24	7	64	71
(credit-	10M	10	28	27	9	64	25	392	417
card)	100M	10	335	243	9	587	231	Stop	Stop

and d (dimensionality), without altering its statistical properties. We replicate them in random order. The columns of the data sets were replicated to get $d = (5, 10, 20, 40)$ and rows were replicated to get $n = (100K, 1M, 10M, 100M)$. In both cases, we chose d randomly from the original data set.

4.2 Comparison with Hadoop Parallel Big Data Systems: Spark

We compare the ML models in parallel nodes ($p = 8$) with Spark, a data processing engine developed to provide faster and easy-to-use analytics than Hadoop MapReduce. We partition the data set using HDFS and then run the algorithm in Spark-MLlib. We used the available functions in MLlib, Spark's scalable machine learning library to run the ML models. We emphasize that we used the recommended settings and parameters as given in the library documentation. Here, we are taking the data sets with a higher n ($n = 1M, 10M, 100M$) and medium d ($d = 10$) to demonstrate how large data sets perform on both.

Table 3 presents the time to compute the ML models in the parallel cluster with R and in Spark. For each entry, we round it up to the nearest integer value. The 'Phase 0' column is the time to split the data set X and transfer X_Is to the processing nodes. We used the standard and fastest UNIX commands available to perform this operation. The 'Phase 1' column shows the maximum time to compute Γ_Is among p machines. We report the maximum time because the parallel execution cannot be faster than the slowest machine. The 'Phase 2' column shows the total time to send the partial Gamma matrices (Γ_I) to the master node in a compressed format, decompress them on the master node, add them to get the final Gamma (Γ) and compute the model (Θ) based on Γ. This time is almost similar regardless of the value of n and d. The reason is that Γ is $d \times d$ which is very small. Moreover, Γ is sent in a compressed format over the network. In the receiving end (master node), we take advantage of R run time.

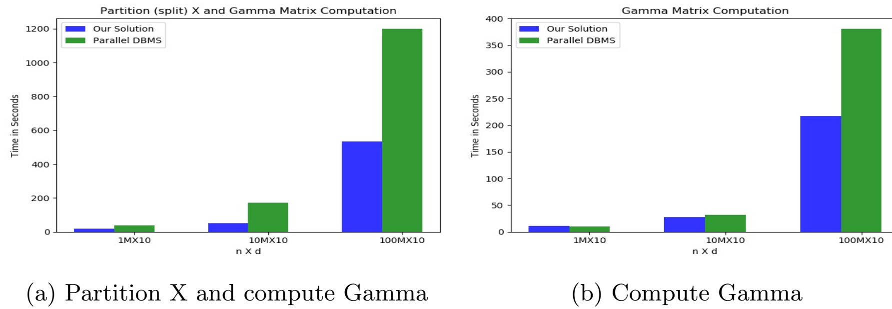
The decompression, addition of the Γ_Is, all these operations are very fast in R (< 1 sec). So, from transferring Γ_Is to get the final Γ, it is almost a constant time (~ 8 seconds). On the other hand, the ML model computation (Θ) also happens in R run time, very fast, and takes a fraction of a second (~ 1 sec). Hence, for 'Phase 2', we put a constant time 9 seconds for each entry. In the Spark part of Table 3, the 'Partition' column is the time to load the data set in HDFS and we report the time to compute the models in Spark-MLlib in 'Compute Θ' column.

Despite HDFS being faster to partition the data sets (Table 3), the total time is much faster in our method for most cases. HDFS is faster in partitioning because we are splitting and transferring the data set sequentially over the network. A parallel partition that sends blocks in parallel is already implemented efficiently in DBMS and is beyond the scope of this paper. In case of Linear Regression (LR), the Spark-MLlib trains and outputs the coefficients and intercept as the model. We load the data set and fit the data set to get the model. Spark's performance is slow when fit is called and we can see that our Γ is almost $10X$ faster. For PCA, the Spark-MLlib uses a similar algorithm as Γ. For a large X, it computes $X^T.X$ by computing the outer product of each row of the matrix by itself, then adding all the results up. This is the Q from our Γ which is manipulated in the main memory by each worker node. Still, Spark is slightly slower for computing the PCA model than our method in all cases. For Naïve Bayes (NB) model, Spark-MLlib implements multinominal Naïve Bayes which takes an RDD labeled point and an optional smoothing parameter as input and outputs the model. The major drawback of this model is, having negative values in the data set crashes the model which has happened for Creditcard data set here. The Spark crashed showing "illegalArgumentException" as the model requires non-negative values. As for k-means, the MLlib implementation includes a parallelized variant of k-means++ [2] which generates a k-means model. The distributed version of the algorithm is roughly $O(k)$, so this suffers a slower start with a large k. It is also expensive when the model is trained. We put "Stop" because Spark could not finish the computation in 30 minutes.

4.3 Comparison with a Parallel DBMS:

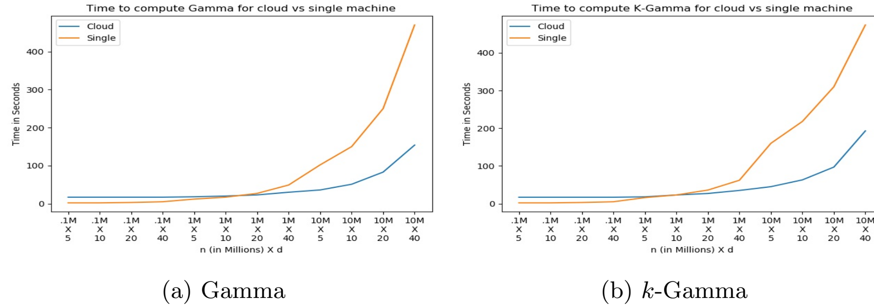
We compare our solution with a parallel columnar DBMS (Vertica) running on p processing nodes. As columnar DBMSs store data by columns and not by rows, it is much faster than a row DBMS [13]. We adapted the solution presented in [15] using UDFs and SQL queries which is the current best solution to compute the Gamma summarization matrix in a parallel array DBMS. As there is no prior solution of k-Gamma matrix in a DBMS, here we only compare our solution with the Gamma matrix. We already know that the ML model (Θ) computation is very fast (~ 1 second) in the main memory exploiting Γ. So, we only report the time to compute the Γ using p processing nodes.

Fig 2 shows the comparison to compute Γ between our solution and the parallel columnar DBMS. We compute Γ for varying n ($1M, 10M, 100M$) and $d = 10$. Fig 2a shows the comparison when we split the data set into p processing nodes and compute Γ and Fig 2b shows the comparison to just compute Γ using



(a) Partition X and compute Gamma (b) Compute Gamma

Fig. 2: Time (in Sec) comparison for Γ on $p = 8$ nodes: our solution vs parallel DBMS for varying n and d (M=millions)



(a) Gamma (b) k-Gamma

Fig. 3: Time comparison for Γ and Γ^k in parallel cluster ($p = 8$ nodes) and single machine ($p = 1$ node) for varying n and d.

p machines (data set is already partitioned and loaded into DBMS). We show both plots to give the parallel DBMS a fair chance because it is often assumed that data is already stored in the DBMS. We use standard SQL queries to COPY (Partition) the data set in all machines. As partitioning in DBMS is slow, we can see that parallel DBMS performs much slower than our solution for all n when it has to partition the data first. Our solution also performs better for Γ computation as n grows (Fig 2b). Moreover, DBMS solutions using UDF are not portable and they require a lot of memory to scale-up.

4.4 Understanding trade-offs: Parallel cluster and single machine

We compute the Γ and Γ^k on a parallel cluster and single machine to understand the trade-offs between them. For both cases, we used our optimized version of the algorithms. Γ is computed on YearPredictionMSD data set to compute models like LR and PCA, and Γ^k is computed on the CreditCard data set to compute NB and KM models. The total time to compute the Γ and Γ^k for parallel cluster and the single machine can be plotted in Fig 3. We can see that single machine performs better when n and d is low ($\leq 1M \times 10$) in both cases. The reason

is, the parallel cluster is spending much time in partitioning the data set and transferring the partial Γ_I (or Γ_I^k) matrices. Parallel cluster seems to be more faster from $n = 1M$ and $d = 20$. When n is very high ($n = 10M$ or more), the parallel cluster is at least $2X$ to $4X$ faster than the local machine. The reason is, a single machine cannot scale as data size grows due to limited memory. However, the model computation part utilizing Γ or Γ^k is almost same for both. So, the parallel cluster is the obvious choice when it comes to summarizing very large data sets.

5 Related Works

Summarization of scalable machine learning algorithms was done in a parallel manner in [15]. However, this work was developed for a parallel array DBMS and did not work for classification or clustering models. In this paper, we removed the use of DBMS completely which was the main focus on [15]. We adapted the algorithm, generalized, and implemented such a way that it can work in a parallel cluster efficiently. Moreover, we introduced k-Gamma matrics that can compute models like NB and KM which are significantly different from [15]. We also made use of reading data in blocks to read an infinite amount of input data. Similar to our proposed k-Gamma matrix, the summaries of [22] and [4] represent a (constrained) diagonal version of Γ because dimension independence is assumed (i.e. cross-products, covariances, correlations are ignored) and there is a separate vector to capture L. From a computational perspective, our Γ computation boils down to one matrix multiplication, whereas those algorithms work is aggregations. Also, our summarization is more general and it helps to compute more complex models like LR, PCA, NB, and KM that could not be solved with older summaries. Parallel processing for data summarization has received moderate attention. [12] highlights the following techniques: sampling, incremental aggregation, matrix factorization, and similarity joins. Research has developed fast algorithms based mostly on sampling, data summarization, and gradient descent [8], generally working in a sequential manner (data mining). Stochastic (incremental) gradient descent (SGD) [9] is a popular approach, useful when there is a convex function to optimize (like least-squares in LR). As for drawbacks, SGD is naturally sequential (difficult to process in parallel), it obtains an approximate solution and it is difficult to adapt to non-convex functions (e.g. clustering).

From a "systems" angle, R combined with C++ did not exist and nobody thought we could insert efficient C++ code for a very common computation on parallel machines. However, R has been used for parallel computing on computer clusters, on multi-core systems, and in grid computing. There are many available packages in R for parallel computing and they are reviewed and compared in [18] based on development, usability, acceptance, and performance. There is a large body of work on computing machine learning models in Hadoop "Big Data" systems, before with MapReduce [3] and currently with Spark [21]. On the other hand, computing models with parallel DBMSs have received less attention [9],

[19] because they are considered cumbersome and more difficult to program. This article is a significant step forward and is fundamentally different from [5] which worked only on a single machine. We introduced a new generalized algorithm to compute Γ in a parallel cluster. Also, we improved the k-Gamma algorithm where k summarization matrices (each $d \times d$) were needed in [5] to compute NB and KM models. The improved algorithm needs only one matrix $(d + 1 \times 2k)$. Also, [5] cannot scale to big data as it was done in a local machine. Experimental results prove that our new solution does not have any limitation: neither main memory nor CPU power available.

6 Conclusions

We presented an improved 3-phase algorithm to compute ML models. Specifically, we added a pre-processing phase, to partition and distribute the data set. Also, parallel processing is fully automated and we now cover a wide spectrum of unsupervised and supervised ML models. We introduced a general, parallel, summarization algorithm that can work across multiple programming languages and platforms. We then studied how to integrate our parallel algorithm into the R language, a popular language in ML and statistics. We justified why C++ code is required and so we focused on optimizing summarization, with specialized C++ functions for a fundamental vector outer product, returning one or multiple Gamma matrices (Γ^k). We showed the actual model computation, fortunately, can be done with existing R functions, eliminating the need to reprogram them. An experimental evaluation shows our solution is either faster or more scalable than Spark. On the other hand, our solution is remarkably faster than a previous prototype programmed with SQL queries and UDFs, the best previous solution based on the same approach.

Our research opens many possibilities for future work. We will tackle other ML models, including HMMs, LDA, and SVMs. We plan to compare tradeoffs when integrating our algorithm with Python, another popular language with significantly different syntax and evaluation compared to R. Even though processing in one machine is slower than a parallel cluster, we intend to study how to accelerate computation with multicore CPUs and GPUs in a single box. We would like to encode our result summarization matrix in a general format that can be consumed by any langua ge or system. It should be feasible to detect intermediate computations in analytic source code, where our summarization matrix or matrices may accelerate or simplify processing.

References

1. Al-Jarrah, O.Y., Yoo, P.D., Muhaidat, S., Karagiannidis, G.K., Taha, K.: Efficient machine learning for big data: A review. Big Data Research **2**(3), 87–93 (2015)
2. Arthur, D., Vassilvitskii, S.: k-means++: the advantages of careful seeding. In: Proceedings of the Eighteenth Annual ACM-SIAM Symposium on Discrete Algorithms, SODA. pp. 1027–1035 (2007)

3. Behm, A., Borkar, V., Carey, M., Grover, R., Li, C., Onose, N., Vernica, R., Deutsch, A., Papakonstantinou, Y., Tsotras, V.: ASTERIX: towards a scalable, semistructured data platform for evolving-world models. Distributed and Parallel Databases (DAPD) **29**(3), 185–216 (2011)
4. Bradley, P., Fayyad, U., Reina, C.: Scaling clustering algorithms to large databases. In: Proc. ACM KDD Conference. pp. 9–15 (1998)
5. Chebolu, S.U.S., Ordonez, C., Al-Amin, S.T.: Scalable machine learning in the R language using a summarization matrix. In: Database and Expert Systems Applications - 30th International Conference, DEXA 2019, Linz, Austria, August 26-29, 2019, Proceedings, Part II. pp. 247–262 (2019)
6. Dean, J., Corrado, G., Monga, R., Chen, K., Devin, M., Le, Q.V., Mao, M.Z., Ranzato, M., Senior, A.W., Tucker, P.A., Yang, K., Ng, A.Y.: Large scale distributed deep networks. In: Proc. Advances in Neural Information Processing Systems. pp. 1232–1240 (2012)
7. Eddelbuettel, D.: Seamless R and C++ Integration with Rcpp. Springer, New York (2013)
8. Gemulla, R., Nijkamp, E., Haas, P., Sismanis, Y.: Large-scale matrix factorization with distributed stochastic gradient descent. In: Proc. KDD. pp. 69–77 (2011)
9. Hellerstein, J., Re, C., Schoppmann, F., Wang, D., Fratkin, E., Gorajek, A., Ng, K., Welton, C.: The MADlib analytics library or MAD skills, the SQL. Proc. of VLDB **5**(12), 1700–1711 (2012)
10. Hu, H., Wen, Y., Chua, T., Li, X.: Toward scalable systems for big data analytics: A technology tutorial. IEEE Access **2**, 652–687 (2014)
11. Lang, D.T., Lang, M.D.T.: Package 'rcurl' (2012)
12. Li, F., Nath, S.: Scalable data summarization on big data. Distributed and Parallel Databases **32**(3), 313–314 (2014)
13. Ordonez, C., Cabrera, W., Gurram, A.: Comparing columnar, row and array dbmss to process recursive queries on graphs. Information Systems (2016)
14. Ordonez, C., Omiecinski, E.: Accelerating EM clustering to find high-quality solutions. Knowledge and Information Systems (KAIS) **7**(2), 135–157 (2005)
15. Ordonez, C., Zhang, Y., Cabrera, W.: The Gamma matrix to summarize dense and sparse data sets for big data analytics. IEEE Transactions on Knowledge and Data Engineering (TKDE) **28**(7), 1906–1918 (2016)
16. Ostrouchov, G., Chen, W.C., Schmidt, D., Patel, P.: Programming with big data in r (2012), http://r-pbd.org/
17. Rickert, J.: Big data analysis with revolution r enterprise. Revolution Analytics (2011)
18. Schmidberger, M., Morgan, M., Eddelbuettel, D., Yu, H., Tierney, L., Mansmann, U.: State-of-the-art in parallel computing with r. Journal of Statistical Software **47** (2009)
19. Stonebraker, M., Abadi, D., DeWitt, D., Madden, S., Paulson, E., Pavlo, A., Rasin, A.: MapReduce and parallel DBMSs: friends or foes? Commun. ACM **53**(1), 64–71 (2010)
20. Xing, E.P., Ho, Q., Dai, W., Kim, J.K., Wei, J., Lee, S., Zheng, X., Xie, P., Kumar, A., Yu, Y.: Petuum: A new platform for distributed machine learning on big data. IEEE Trans. Big Data **1**(2), 49–67 (2015)
21. Zaharia, M., Chowdhury, M., Franklin, M., Shenker, S., Stoica, I.: Spark: Cluster computing with working sets. In: HotCloud USENIX Workshop (2010)
22. Zhang, T., Ramakrishnan, R., Livny, M.: BIRCH: An efficient data clustering method for very large databases. In: Proc. ACM SIGMOD Conference. pp. 103–114 (1996)