Efficient Machine Learning on Data Science Languages with Parallel Data Summarization

Sikder Tahsin Al Amin, Carlos Ordonez

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

Abstract

Nowadays, data science analysts prefer "easy" high-level languages for machine learning computation like R and Python, but they present memory and speed limitations. Also, scalability is another issue when the data set size grows. On the other hand, acceleration of machine learning algorithms can be achieved with data summarization which has been a fundamental technique in data mining. With these motivations in mind, we present an efficient way to compute the statistical and machine learning models with parallel data summarization that can work with popular data science languages. Our summarization produces one or multiple summaries, accelerates a broader class of statistical and machine learning models, and requires a small amount of RAM. We present an algorithm that works in three phases and is capable to handle data sets bigger than the main memory. Our solution evaluates a vector-vector outer product with C++ code to escape the bottleneck of the high-level programming languages. We present an experimental evaluation section with a prototype in the R language where the summarization is programmed in C++. Our experiments prove that our solution can work on both data subsets and full data set without any performance penalty. Also, we compare our solution (R combined with C++) with other parallel big data systems, Spark (Spark-MLlib library), and a parallel DBMS (similar approach implemented with UDFs and SQL queries). We show our solution is simpler and mostly faster than Spark based on the storage of the data set, and it is much faster than a parallel DBMS regardless of the storage of the data set.

Keywords: Machine Learning, Parallel Computation, Statistics, Data Summarization.

1. Introduction

The computation of most machine learning algorithms is heavily impacted by the volume of data being processed. Data volumes rise at a much higher rate than the processing speeds. Hence, scalability has become an issue as the data set size grows. Mathematical systems like Python, R provide comprehensive libraries for machine learning and statistical computation. However, they are not designed to scale to large data sets and the single machine is challenging to analyze the large data sets [1]. Accelerating machine learning algorithms does not always mean adding new hardware and memory. Therefore, processing and analyzing large volumes of data becomes non-feasible using a traditional serial approach and parallel processing has emerged to solve the problem. Among parallel systems, Hadoop systems like Spark, Hive, Cassandra, have gained popularity for storing and processing big data. Other parallel systems like parallel DBMSs (e.g. Vertica, Teradata) have also been used for processing big data in some cases [2], [3], [4]. Though the parallel systems offer ample storage and processing power, the execution time can be slower. These systems also suffer from the absence of efficient native support for matrix-form data and out-of-box sophisticated mathematical computations.

Data summarization is fairly popular among data scientists to accelerate machine learning algorithms [2], [5], [4]. The aforementioned parallel systems are not a good choice for data summarization as they have scalability limitations. Also, summarization in parallel DBMSs is hard due to the portability of UDFs (user defined functions) and Spark is even slower than parallel DBMSs for a few processing nodes [2]. Moreover, the complex set up of these parallel systems makes it even harder for average data analysts to begin with. On the other hand, data science languages like Python and R are popular among analysts but they do not work in parallel by default. Although they provide parallel libraries, the analysts are limited by the functionality of these libraries. With these motivations in mind, we present a technique to accelerate statistical and machine learning models exploiting a summarization matrix that can work in parallel and perform as fast as popular existing parallel big data systems.

Here, our contributions are the following: (1) We present a technique to accelerate statistical and machine learning models exploiting summarization matrix/matrices. (2) Our summarization is a three-phase generalized algorithm that can work in a parallel cluster (or a remote cluster in the cloud). (3) We present how to compute descriptive statistics and perform

statistical tests in addition to computing machine learning models utilizing our summarization matrix. (4) For each machine learning model, we discuss how to compute it for new data points. In our work, we used R as our choice of data science language combined with C++ to develop our algorithms, but it can be applied to other analytic platforms like Python. Here, we used a local parallel cluster to perform the experiments but our research applies to both a 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 faster than Spark and parallel DBMS. This article is a significant extension and deeper study of [5], where parallel data summarization with analytic languages (R) was initially proposed.

This is the outline for the rest of the article. Section 2 introduces the definitions used throughout the paper and our parallel cluster architecture. Section 3 presents our theoretical research contributions where we present our technique to accelerate statistical and ML models. 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.

2. Definitions

This section presents the mathematical definitions and symbols used throughout this paper. Also, parallel cluster architecture is discussed.

2.1. Mathematical Definition

We define the input matrix as X, a set of n-column vectors. X can 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. We augment X with an output variable Y, making X a $(d+1) \times n$ matrix and we call it $\mathbf{X}$. In short, X has the independent variables (input dimensions) and Y is the dependent variable. We define Z as $\mathbf{X}$ with an extra row of n 1s, a $(d+2) \times n$ dimensional matrix.

We use Θ to represent a machine learning model or a statistical property in a general manner. Thus Θ can be any ML model like Linear Regression (LR), Principal Component Analysis (PCA), Naïve Bayes (NB), K-means (KM), or any statistical property like mean, variance, or correlation. Table 1 shows the basic symbols and their description used throughout the paper.

Table 1: Basic symbols and their description

Sym.	Description	Sym.	Description
X	Data set	d	Number of columns in X
X_I	Partitioned data set	Γ	Gamma Summarization Matrix
$\mathbf{X}$	X with Y	Γ^k	k-Gamma Summarization Matrix
Y	Dependent Variable	Θ	ML/Statistical model
Z	X with 1s and Y	N	Number of processing nodes
n	Number of rows in X	b	Blocks to read data
L	Linear sum	μ	mean
Q	Quadratic sum	σ	variance

71 *2.2. Parallel Cluster Architecture*

72 We define the number of machines (processing nodes) as N. Each node
73 has its CPU and memory (shared-nothing architecture) and it cannot directly
74 access another node's storage. Therefore, all processing nodes communicate
75 with each other transferring data. Also, data is stored on a disk, not in virtual
76 memory. All the necessary programming languages and libraries are installed
77 in each machine.

78 3. Theory and Algorithm

79 This section presents our main technical contributions. First, we give an
80 overview of the original data summarization technique proposed previously.
81 Then we present how to compute the statistical and ML models utilizing the
82 summarization matrix. Finally, we discuss how we can integrate our proposed
83 solution to popular data science programming languages.

84 *3.1. Data Summarization*

85 Here, we review our summarization matrix, named Gamma (Γ) introduced
86 in [6], [2]. The matrix Γ (Gamma), contains an accurate, yet complete,
87 summary of the data set, and therefore it represents a fundamental matrix in
88 our research. As given in Section 2, if we consider X as the input data set, Y
89 is the dependent variable, n counts the total number of points in the data
90 set, L is the linear sum of x_i, and Q is the sum of vector outer products of
91 x_i (quadratic sum), then from [2], the Gamma summarization matrix (Γ) is
92 defined below in Eq. 1. We first define n, L, Q as: $n = |X|$, $L = \sum_{i=1}^{n} x_i$, and
93 $Q = XX^T = \sum_{i=1}^{n} x_i \cdot x_i^T$. Here, L and Q are complementary, they cannot

be added or multiplied with each other. We will use these n, L, and Q to compute the ML and statistical models later in this Section.

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

As mentioned Section 2, X is defined as a $d \times n$ matrix, and Z is defined as a $(d+2) \times n$ matrix. From [2], we can easily understand that Γ matrix can be computed in the two alternative ways: (1) matrix-matrix multiplication i.e., ZZ^T, or (2) sum of vector outer products i.e., $\sum_i z_i \cdot z_i^T$ So, the Gamma computation can be done as ZZ^T or the sum of outer products presented in Eq 2. Here, in this paper, we evaluate the later one.

$$\Gamma = ZZ^T = \sum_{i=1}^{n} z_i \cdot z_i^T \quad (2)$$

The Γ matrix presented above does not work for classification/clustering models. To solve these kinds of models, we introduced k-Gamma matrix (Γ^k) in [6] named Diagonal-Gamma matrix, given in Eq. 3. The major difference between the two forms of summarization matrix is, we do not require parameters off the diagonal in Γ^k as in Γ. Also, the complexity of Gamma is higher than k-Gamma which is explained in Section 3.5.

$$\Gamma^k = \begin{bmatrix} n & L^T & 0 \\ L & Q & 0 \\ 0 & 0 & 0 \end{bmatrix}, where\ Q = \begin{bmatrix} Q_{11} & 0 & 0....... & 0 \\ 0 & Q_{22} & 0....... & 0 \\ 0 & 0 & Q_{33}..... & 0 \\ 0 & 0 & 0........ & Q_{dd} \end{bmatrix} \quad (3)$$

Here we can see that 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. 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, according to [5], Γ^k can be represented as a single matrix with L and $diag(Q)$ of size $d \times 2k$ instead of multiple k matrices. We still need to store the value of n in a row, which makes the Γ^k as $(d+1) \times 2k$. So, we are using minimal memory to store Γ^k even if the value of k is large.

3.2. Statistical and Machine Learning Models Computation

Here, we explain how we can exploit our summarization matrix to compute various data science computations. We present our contributions in three levels going an increasing level of mathematical complexity: (1) Descriptive statistics, (2) Statistical tests, and (3) Machine Learning models.

3.2.1. Descriptive Statistics

Descriptive statistics are common computations and they can tell a lot about the data. We can compute the descriptive statistics like mean, variance, and correlation from our summarization matrix based on n, L, and Q. Variance is the average of the squared differences from the mean which tells the degree of spread in the data set. And correlation is the measure of the strength of a linear relationship between two quantitative variables (e.g. height, weight). These statistics can also be exploited to compute other models as they appear frequently in many statistical and ML models. We compute mean (μ), variance (σ), and correlation (ρ) directly from our summarization matrix (Γ) exploiting n, L, and Q in Equation 4, 5, and 6 respectively. Later we will generalize and use these to compute machine learning models.

$$\mu = \frac{L}{n} \tag{4}$$

$$\sigma = \frac{Q}{n} - \frac{LL^T}{n^2} \tag{5}$$

$$\rho_{ab} = \frac{nQ_{ab} - L_a L_b}{\sqrt{nQ_{aa} - L_a^2}\sqrt{nQ_{bb} - L_b^2}} \tag{6}$$

Our summarization matrix can also work on the subsets of data. That is, we can separate the data set into multiple data subsets based on gender or age (more discussion on Section 3.4). Then we can compute the descriptive statistics on each subset exploiting our summarization matrix. These computations help to understand the data set better before applying any machine learning or statistical models. We can rewrite Eq 4, 5, 6 based on

data subsets (j) as:

$$\mu_j = \frac{L_j}{n_j}$$
$$\sigma_j = \frac{Q_j}{n_j} - \frac{L_j L_j^T}{n_j^2} \tag{7}$$
$$\rho_{ab_j} = \frac{n_j Q_{ab_j} - L_{a_j} L_{b_j}}{\sqrt{n_j Q_{aa_j} - L_{a_j}^2}\sqrt{n_j Q_{bb_j} - L_{b_j}^2}}$$

3.2.2. Statistical Tests

Statistical tests are used in hypothesis testing and are different kinds of computations than descriptive statistics or machine learning models. A statistical test can be used to determine whether there is enough evidence to "reject" the null hypothesis. This is important for medicine, surgery, and clinical trials. For statistical tests, we present a parametric test comparing means μ_1, μ_2 from two disjoint data subsets, where the size of each data subset is n_1, n_2. Our summarization matrix does not apply to all the statistical tests. Here, we are discussing a popular test that is compatible with our summarization matrix. Each data subset can be obtained from the original data set based on some filters as mentioned previously. We assume each data subset is independent and it has small cardinality. Otherwise, statistical tests using our summarization matrix are not possible. The null hypothesis H_0 states that $\mu_1 = \mu_2$ and we need to find the group pairs where H_0 can be rejected with high confidence $1 - p$, where the threshold of p is generally $p \in 0.01, 0.05, 0.10$. The so-called alternative hypothesis H_1 states that $\mu_1 \neq \mu_2$. When H_0 can be rejected the test will return the significance level p; such outcome will allow us to provide strong statistical evidence supporting $H_1 : \mu_1 \neq \mu_2$. We use a two-tailed test which allows finding a significant difference on both tails of the Gaussian distribution to compare means in any order ($\mu_1 < \mu_2$ or $\mu_2 < \mu_1$). The statistical test relies on Eq. 8 to compute a random variable v with probability distribution function (PDF) $N(0, 1)$:

$$v = \frac{|\mu_1 - \mu_2|}{\sqrt{\sigma_1^2/n_1 + \sigma_2^2/n_2}} \tag{8}$$

Here, μ and σ^2 are the estimated mean and standard deviation respectively

165 which can be computed from our summarization matrix as follows:

$$\mu_j = \frac{L_j}{n_j}$$
$$\sigma_j^2 = \sqrt{diag(\frac{Q_j}{n_j} - \frac{L_j L_j^T}{n_j^2})} \tag{9}$$

166 When both groups are large, we can use our summarization matrix. The v
167 value just needs to be compared with $v_{p/2}$ in the cumulative probability table
168 for $N(0,1)$. Generally in big data, both groups are large and we can compute
169 v efficiently using our solution. However, if either group is small, then we can
170 compute v directly without computing the summarization matrix first.

171 *3.2.3. Machine Learning Models*

172 Two types of ML models are considered: supervised and unsupervised
173 models. We present how to compute the Linear regression and Naïve Bayes
174 as a representative of the supervised models. And for unsupervised models,
175 we present how we can compute Principal Component Analysis and K-means
176 exploiting our summarization matrix.

177 *Linear Regression:.* We know that the standard definition of LR is given
178 as $Y = \beta^T \mathbf{X} + \epsilon$, where β is the column vector of regression coefficients
179 and ϵ represents the Gaussian error. Also, $\mathbf{X}$ is a $(d+1) \times n$ augmented
180 matrix where we have X with a row of n 1s. And, β can be defined as
181 $\hat{\beta} = (\mathbf{X}\mathbf{X}^T)^{-1}\mathbf{X}Y^T$. Based on the summarization matrix (Γ), we can rewrite
182 this equation as below.

$$\hat{\beta} = Q^{-1}(\mathbf{X}Y^T) \tag{10}$$

183 Using this $\hat{\beta}$, estimated Y value ($\hat{Y}$) can be computed for new points
184 x_i. We do not show this in the experimental section as this computation is
185 straightforward.

186 *Principal Component Analysis:.* PCA can be computed on the covariance
187 matrix (V), or the correlation matrix (ρ). This model requires two parameters.
188 First is U, which is a set of d orthogonal vectors, principal components of the
189 data set, and the diagonal matrix (D^2) which contains the squared eigenvalues.
190 We compute ρ, the correlation matrix as $\rho = UD^2U^T = (UD^2U^T)^T$. It can be
191 computed from our summarization matrix (Γ) as: $\rho_{ab} = \frac{(nQ_{ab} - L_a L_b)}{(\sqrt{nQ_{aa} - L_a^2}\sqrt{nQ_{bb} - L_b^2})}$.

192 Then we compute the model from the ρ by solving Singular Value Decompo-
193 sition on it ($svd(\rho)$). After computing the model, the actual dimensionality
194 reduction of X is straightforward. We just need a matrix multiplication
195 between the matrix derived from SVD and X. For example, if we need to
196 reduce the dimensionality of X to P dimensions, we need to multiply matrices
197 $d \times P$ derived from SVD and X.

198 *Naïve Bayes:.* We need the k-Gamma matrix to compute the NB model as
199 this is a classification/clustering problem. Here, we focus on $k = 2$ classes for
200 NB. We compute N_g, L_g, Q_g as mentioned above for each class. The output is
201 three model parameters: mean (C), variance (R), and the prior probabilities
202 (W) which can be obtained using Eq 11. Here, $N_g = |X_g|$ and we take the
203 diagonal of $L \cdot L^T$ and Q, which can be manipulated as a 1D array instead of
204 a 2D array.

$$W_g = \frac{N_g}{n};\ C_g = \frac{L_g}{N_g};\ R_g = \frac{Q_g}{N_g} - diag\frac{[L_g L_g^T]}{N_g^2} \tag{11}$$

205 After computing the model using the above equation, we can predict the
206 class labels for new data exploiting it. For each point in the new data, we
207 compute a probability value per class using the model parameters in Eq 11,
208 and assign the class with maximum probability. We compute the probability
209 $P(x_i|\theta)$ as, $P(x_i|\theta) = (1/\sqrt{2\pi\sigma_g^2})e^{-0.5(x_i - \mu_{x_i})^2/\sigma_g^2}$

210 *K-means:.* Similar to NB, we compute model parameters N_j, L_j, Q_j (where
211 $j = 1, .., k$) for each cluster. From these statistics, we compute C_j, W_j similar
212 to NB presented in Eq. 11. After computing the model parameters, the
213 algorithm iterates executing two steps starting from random initialization
214 until cluster centroids become stable.

215 1. Determining centroids: This step determines the closest cluster for each
216 point (using Euclidean distance) and adds the point to it. The distance
217 from each point x_i can be determined as $d(x_i, C_j) = (x_i - C_j)^T(x_i - C_j)$.
218 2. Updating centroids: This step updates all the centroids C_j by computing
219 the mean vector of points belonging to cluster j ($j = 1, .., k$). The cluster
220 weights W_j are also updated based on the new centroids.

221 The K-means algorithm stops when centroids change by a marginal fraction
222 in consecutive iterations, measured by the quantization error. With decreasing
223 error at each iteration, K-means is theoretically guaranteed to converge, yet
224 we set a threshold on the number of iterations to avoid excessively long runs.

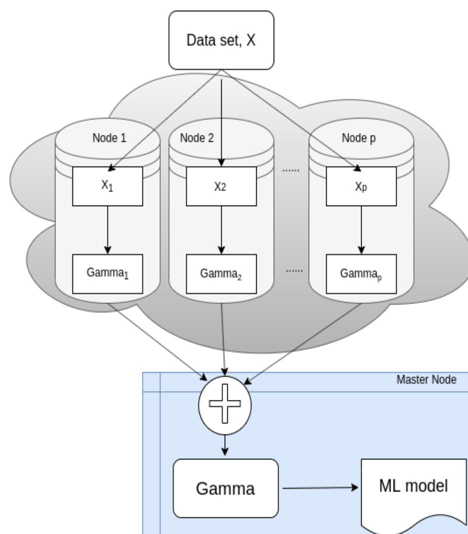


Figure 1: System architecture to compute ML and statistical models by parallel data summarization.

225 *3.3. Parallel Algorithm to Compute Machine Learning and Statistical Models*

226 Here, we present our generalized parallel algorithm. We propose a 3-phase
227 algorithm to compute the summarization matrix and show how machine
228 learning and statistical models can be computed exploiting it. Figure 1 shows
229 our system architecture for N processing nodes. Our proposed generalized
230 3-phase algorithm is given below:

231 1. Phase 0: Pre-process the data set. Partition the data set to the N
232 processing nodes.
233 2. Phase 1: Compute summarization matrix in parallel across N nodes: Γ
234 or Γ^k. This phase will return N partial (local) summarization matrices
235 (Γ_I or Γ_I^k, $I = 1, 2, ..., N$)
236 3. Phase 2: Add the partial summarization matrices to get final Γ or Γ^k
237 on the master node. Compute machine learning or statistical model (Θ)
238 based on Γ or Γ^k.

239 *3.3.1. Phase 0:*

240 Here, we partition the data set X to the N processing nodes. The data
241 set X can be either a full data set or data subsets. We assume data set X
242 can be either be in a parallel cluster (disk), cloud (remote cluster, HDFS), or

243 in a local machine. In any case, data to be analyzed must be partitioned into
244 the processing nodes. We split and transfer the data set X into N processing
245 nodes ($X_I, I = 1, .., N$). There are several partitioning strategies available
246 like row-based or column-based partitioning. Here, we use the row-based
247 partitioning (horizontal partitioning) as our summarization matrix (Γ) is
248 $O(d^2)$ and we need all the d columns in each node. If we choose column-based
249 (vertical partitioning) or block-based partitioning, the summarization matrix
250 in different nodes may end up having different sizes. We use n/N as the data
251 set size in each partition (row-based). So, each node in the parallel cluster
252 has the same number of rows except for the N-th node which is important
253 for our parallel algorithm to work efficiently.

254 *3.3.2. Phase 1:*

255 This phase computes our summarization matrix in parallel. In each node,
256 we compute a local summarization matrix and at the end of this phase, we
257 send it to the master node. The whole procedure is shown in Algorithm 1.
258 The input for this phase is the partitioned data set (X_I) from the previous
259 phase. We assume X_I can be of any size and it may or may not fit in the local
260 main memory. To address this issue, we optimize the technique to read data
261 in blocks so that we can handle large files. For each node, the partitioned
262 data set (X_I) is read into $b = 1...b$ blocks of same size (m) where $m < |X_I|$.
263 The block size depends on the number of records (n_I) in X_I. As discussed in
264 [7], we define the block size as $\log n_I$. As $\log n_I \ll n_I$, even if n_I is very large,
265 each block will easily fit in the main memory. Processing data one block at a
266 time has many benefits. It is the key to being able to scale the computations
267 without increasing memory requirements. External memory (or out-of-core)
268 algorithms do not require that all of the data be in RAM at one time. Data
269 is processed one block at a time, with intermediate results updated for each
270 block. When all the data is processed, we get the final result. According
271 to Algorithm 1, we read each block (b) into the main memory and compute
272 Gamma for that block (Γ_b). This partial Gamma is added to the Gamma
273 computed up to the previous block ($b - 1$). We iterate this process until no
274 blocks are left and get the Gamma (Γ_I) for that node. As each node has all
275 the d columns, the size of each Γ_I will be $(d + 2) \times (d + 2)$.

276 *3.3.3. Phase 2:*

277 Here, we compute the machine learning or statistical model exploiting the
278 summarization matrix computed in Phase 1. However, this phase is computed

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

Data: Partitioned Data Set $(X_I, I = 1, 2..N)$ from Phase 0
Result: Γ_I

1 Read X_I into $b = 1, 2, ..., b$ blocks;
2 **while** *next(b)* **do**
3   read(b) ;
4   Γ_b = Gamma(b) ;
5     $\Gamma_I = \Gamma_b + \Gamma_I$;
6 **end**
7 **return** Γ_I to the master node

279 in the master node only, meaning no parallel processing is needed in this phase.
280 At the end of Phase 1, all the Γ_Is $(\Gamma_1, \Gamma_2, ..., \Gamma_N)$ are computed in parallel.
281 We need to combine them to get the final Γ. So, all the partial Γ_Is are sent
282 to a master node to perform the addition (sequential) or we can perform it
283 in a hierarchical binary tree manner. Hierarchical processing performs the
284 addition in multiple levels (bottom-up) until we get the final addition at the
285 top level. Here, we perform sequential processing where all the partial Γ_Is
286 are transferred to the main memory of the master node. Now, to get the final
287 summarization matrix (Γ), we compute $\Gamma = \Gamma_1 + \Gamma_2 + ... + \Gamma_N$. Similarly, for
288 Γ^k, $\Gamma^k = \Gamma^k_1 + \Gamma^k_2 + ... + \Gamma^k_N$. Now, utilizing this Γ or Γ^k, we compute the
289 machine learning or statistical models (Θ) as mentioned in Section 3.2.

290 One assumption for our algorithm to work is, we assume data points have
291 no specific order. Here, we are computing Phase 1 in parallel and we need
292 all the points from k classes/groups in each node to compute the k-Gamma
293 matrix. If the points are sorted by k-groups, then our solution would not work.
294 Hence, it is expected that the k-groups are represented in each node. Also, our
295 solution may not work if there are no intermediate computations that have
296 sums of variables or sums of cross-products of variables. In practical terms,
297 this means computing averages, covariances or correlations. 90% of models
298 have some intermediate computations like these (i.e. not all). However, there
299 may be other computations that cannot be helped by our summarization
300 matrix. For instance, gradient descent, kurtosis, logarithms of variables.

301 *3.4. Integrating with Data Science Languages: R*

302 Here, we discuss how we integrate our algorithm with a data science
303 language. In this paper, we choose R as our choice of data science language.
304 However, our solution applies to other programming languages that provide
305 an API to call C++ code. We choose and recommend data science languages
306 because SQL queries work for tables and relational data, and they do not have
307 subscripts. Also, SQL queries are slow for analytics, especially the parallel
308 JOIN operation, and UDFs are not portable. On the other hand, Spark is
309 not easy to debug and Java is slower than C++. Moreover, data science
310 languages like R and Python are more popular among analysts nowadays.

311 Before running our algorithm, we can perform getting subsets from the
312 original data set. For example, we can filter rows from the original data set
313 based on some criteria (e.g. age, gender) or choose columns based on our
314 need (selection and projection respectively, in a similar manner to a DBMS).
315 This way, we can build the data subset in R or other languages and then
316 apply our algorithm to the subsets. This should be done before Phase 0.

317 Key insight: Phase 1 must work in C++ (or C). As mentioned in Section
318 3.1, we evaluate the $\sum z_i * z_i^T$ to compute our summarization matrix. This
319 sum of vector outer products must be computed block by block in C++,
320 not in the host language. Computing $z_i * z_i^T$ in a loop in any analytic
321 language (e.g. Python, R) is slow, usually one-row-at-a-time. On the other
322 hand, computing $Z * Z^T$ with traditional matrix multiplication is also slow
323 due to Z^T materialization, even in RAM. We use an external library that
324 facilitates extending the host language with C++ functions to compute our
325 summarization matrix. For example, R has Rcpp library that can be used to
326 integrate C++ code with R and accelerate computation by replacing an R
327 function with its C++ equivalent function. In Rcpp, only the reference gets
328 passed to the other side but not the actual value which makes it efficient to
329 integrate the C++ code.

330 For Phase 2, statistical or machine learning model computation can be
331 done efficiently in one machine by calling the existing functions from the data
332 science languages. In other words, Phase 2 uses the data science language
333 "as is" where Phase 1 exploited C++. It would be difficult and error-prone to
334 reprogram all the ML models. Instead, our solution requires just changing
335 certain steps in each numerical method, rewriting their equations based on the
336 data summaries (1 or k). For example, in Listing 1, we present an example
337 of R source code to compute descriptive statistics such as mean, variance,

(Eq. 4, 5 respectively) and the LR model (mentioned in Eq. 10) from the summarization matrix (*gamma*) computed in Phase 1.

Listing 1: Example of R source code (Phase 2) to compute the descriptive statistics and the LR model.

```
d_plus2 = length(gamma[1,])
d = d_plus2 - 1
n = gamma[1,1]
L = gamma[2:d,1]
Q = gamma[2:d,2:d]

#descriptive statistics
mu = L/n #mean
sigma = Q/n - L*L/(n^2) #variance

#LR model
XYT = gamma[2:d, d_plus2]
Beta = solve(Q) * XYT
```

Our algorithm has the potential to work in Python. In the same manner as R, we can use Python with C++ to compute our summarization matrix. Python has 'SWIG' library that can be used to expose the C++ functions to the Python environment. Then we can use the NumPy library to compute the models in Python, similar to R. As an extra benefit, our solution gives the flexibility to the analyst to compute data summaries in a parallel cluster (local or cloud), but explore many statistics and models locally. Moreover, our parallel solution is simple, elegant, more general and we did not need any complicated library. However, analysts should have some good knowledge of the internals of the algorithm.

3.5. Time and Space Complexity

We assume N to be the number of processing nodes under a shared-nothing architecture, and $d \ll n$, $N \ll n$. Let, m be the number of records in each block and b be the total number of blocks per processing node and each block size is fixed. Here, the time complexity is proportional to the block size as we are computing Γ in blocks per node. So, for each block time complexity of computing Γ will be $O(d^2b)$. For a total of m blocks, it will be $O(md^2b)$ in each node. When all the blocks are read, $mb = n$. In our case of parallel computation, the time complexity will be $O(md^2b/N)$ per processing node. In the case of the k-Gamma matrix, we only compute L and diagonal of Q of the whole Gamma matrix. So, for Γ^k, it will be $O(mdb/N)$ in each

374 processing node. On the other hand, when transferring all the partial Γs,
375 if we transfer to the master node all at once: $O(d^2)$, for sequential transfer:
376 $O(d^2 N)$, for hierarchical binary tree fashion: $O(d^2 N + log_2(N)d^2)$. Finally,
377 we take advantage of our summarization matrix to accelerate computing the
378 ML and statistical models. So, the time complexity of this part does not
379 depend on n and is $\Omega(d^3)$. Most common machine learning models including
380 LR with least squares and SVD have time complexity $\Omega(d^3)$ multiplied by a
381 certain number of iterations.

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

390 4. Experimental Evaluation

391 This section presents our experimental evaluation. First, we present our
392 experimental setup and data sets used for the experiments. Then, we present
393 the computation of descriptive statistics and statistical tests. For machine
394 learning models, we compare our proposed solution with Spark, a popular
395 Hadoop big data system, and parallel DBMS, to make sure our solution
396 is competitive with other parallel systems. We also present the trade-off
397 between computation in a single machine and a parallel cluster. All the time
398 measurements were taken five times and we report the average excluding the
399 maximum and minimum value. The standard deviations are very small and
400 the highest standard deviation recorded is 2.03 seconds.

401 *4.1. Experimental Setup*

402 *4.1.1. Hardware and Software*

403 We performed our experiments using an 8-node parallel cluster each with
404 Pentium(R) Quadcore CPU running at 1.60 GHz, 8 GB RAM, and 1 TB
405 disk space. We choose R as a representative of data science language and
406 we develop our solution using R and C++. We used the standard UNIX
407 commands to split and transfer the data set among the processing nodes. For
408 parallel comparison, we used the Spark-MLlib library for comparing with

Table 2: Base data sets description

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

Table 3: Time (in Sec) to compute descriptive statistics on data subsets in parallel $N = 8$ machines(M = Millions)

Data set	d	n	Phase 0	Phase 1	Phase 2	Total
YearPrediction-Subset1	90	1M	18	14	9	41
YearPrediction-Subset2	90	5M	102	67	9	178
YearPrediction-Full	90	10M	213	135	9	357

409 Spark, and we used Vertica to compare with a parallel DBMS. In the case
410 of Spark, we programmed the models using Scala, for Vertica, we used a
411 previous solution developed with UDFs and SQL queries.

412 *4.1.2. Data Sets*

413 Machine learning or statistical models does not work well on raw data as it
414 may have noise, missing values, outlier values, and so on which can overfit or
415 underfit the models. Also, large public data sets for computing all the models
416 are not available. Therefore, we had to use common data sets available and
417 replicate them to mimic large data sets. We used two data sets presented
418 in Table 2 as our base data sets: YearPredictionMSD and CreditCard data
419 set, obtained from the UCI machine learning repository. We sampled and
420 replicated the data sets in random order to get varying n (data set size). And
421 for lower d, we chose it randomly from the original data set.

422 *4.2. Computing Descriptive Statistics on Data Subsets*

423 As mentioned in Section 3.2, we can compute descriptive statistics and
424 statistical tests utilizing our summarization matrix. We can get subsets of
425 the original data set based on some filter (e.g. gender, age group, location)
426 and see the descriptive statistics there before applying any machine learning
427 models. Here, Table 3 shows how our summarization performs on data subsets
428 compared to the original data set in parallel $N = 8$ machines. We take the
429 YearPrediction data set and generate data subsets from there. We generate
430 10 data subsets (YearPrediction-Subset1) and 2 data subsets (YearPrediction-
431 Subset2) of equal sizes from the original data set. We report the time of one

Table 4: Time (in Seconds) to compute mean comparison on data subsets in parallel $N = 8$ machines (M=millions)

Data set	d	n	Partition	Compute Γ	Stat test
YearPrediction-subset3	90	1M	18	14	9
YearPrediction-subset4	90	9M	199	122	9
YearPrediction-subset5	90	5M	102	67	9
YearPrediction-subset6	90	5M	102	67	9

432 representative data subsets from each of them (the other subsets have almost
433 the same time measurements, within 1 sec variation) and also for the full
434 data set. For each of them, the time for Phase 0, Phase 1, and Phase 2 from
435 Section 3.3 is reported in Table 3. From Phase 0 and Phase 1, we can see that
436 our solution scales well and there is almost no performance penalty regardless
437 of data subset or the full data set. As for Phase 2, we compute the descriptive
438 statistics in one machine for data subsets (Eq. 7) using the summarization
439 matrix. For that, we first need to send the partial summarization matrices
440 to the master node, add them, and then compute the statistics. Computing
441 the statistics is fast utilizing R run-time and is done in less than a second
442 (< 1 sec). And sending partial matrices from $N = 8$ machines takes equal
443 time (< 1 sec for each machine). We round up the time and put 9 seconds
444 for Phase 2 in each case in Table 3. As our summarization matrix depends
445 on d, and not on n, this time does not change if the n gets bigger.

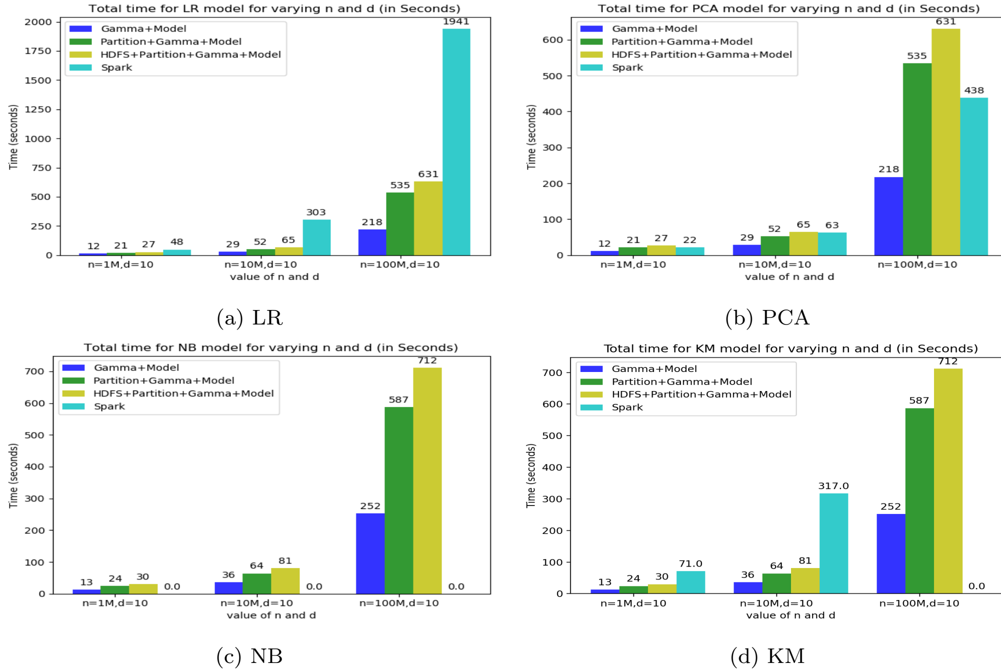
446 On the other hand, for statistical tests, we perform the mean comparison
447 test as mentioned in Section 3.2. Table 4 shows the time to perform the mean
448 comparison test in parallel $N = 8$ machines. To perform this test, we split the
449 data set as 10%-90% and 50%-50%. We partition the data sets and compute
450 the summarization matrix as mentioned above. We can see our solution is
451 scalable for this part. The partial summarization matrices are sent to the
452 master node (each machine takes < 1 second) and added to get the final
453 summarization matrix for each subset (total ~ 8 seconds). Based on this
454 matrix, we perform the mean comparison test. First, we get the mean, std.
455 deviation, and total number of points from the final summarization matrix
456 of each subset (Eq. 9) and then perform the test using Eq. 8. As we are
457 computing this part in R run-time, it is fast and takes less than 1 second for
458 each subset.

Table 5: Time (in Seconds) to compute the ML models with our solution and in Spark ($N = 8$ nodes; M=Millions)

Θ (Data set)	n	d	Partition	R Export	(N=8) $\Gamma + \Theta$	Spark Partition	(N=8) Θ
LR	1M	10	9	6	12	7	41
(Year-	10M	10	23	13	29	17	286
Prediction)	100M	10	317	96	218	161	1780
PCA	1M	10	9	6	12	7	15
(Year-	10M	10	23	13	29	17	46
Prediction)	100M	10	317	96	218	161	277
NB	1M	10	11	6	13	7	Crash
(credit-	10M	10	28	17	36	25	Crash
card)	100M	10	335	125	252	231	Crash
KM	1M	10	11	6	13	7	64
(credit-	10M	10	28	17	36	25	392
card)	100M	10	335	125	252	231	Stop

4.3. Comparison with Hadoop Parallel Big Data System: Spark

Here, we compare our solution with Spark for machine learning models, a popular parallel data processing engine developed to provide faster and easy-to-use analytics. For that, we partition the data sets using HDFS and then run the models using the Spark-MLlib library (with Scala), 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. Moreover, we do not assume that the data set is already in the processing machines. As mentioned in Section 3.3, we assume data to be analyzed can be stored in the (1) disk of a large machine (local file system), (2) HDFS, or (3) already partitioned in the processing machines. If data is in the file system, we need to partition the data set among N machines. Nowadays, data can be also in the HDFS (or cloud) as it is a popular platform to store huge data sets. In that case, we have to export the data and then partition it among N machines. It is possible to read directly from the HDFS using some library but we are not exploring that here. Finally, if the data set is already partitioned in the processing machines, no partitioning is needed. We do not show similar experiments based on data storage for descriptive



(a) LR

(b) PCA

(c) NB

(d) KM

Figure 2: Total time (in Seconds) for ML models using different approaches (M=millions).

statistics and statistical tests as it would be redundant and trivial.

Table 5 presents the time to compute the ML models in the parallel cluster with our solution and Spark. For each entry, we round it up to the nearest integer value. From Table 5, the 'Partition' column is the time to partition X among N processing nodes. This is Phase 0 from our 3-phase algorithm discussed in Section 3.3. We used the standard and fastest UNIX commands available to perform this operation. The 'Export' column is the time to export the data set X from HDFS to the local machine. And the '$\Gamma + \Theta$' column is the time to compute Γ in parallel N machines, send them to the master node to compute the final summarization matrix, and compute the machine learning model (Θ) from it. This process is Phase 1 and 2 combined from our algorithm in Section 3.3. In the Spark part of Table 5, 'Partition' is the time to load and distribute the data set in HDFS among N machines. And we report the time to compute the models using Spark-MLlib in the 'Θ' column.

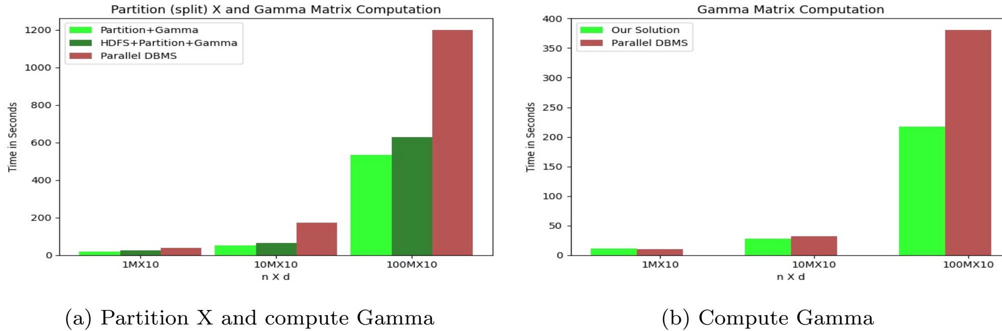
Fig 2 shows the total time based on Table 5 to compute the ML models using different approaches discussed above. We simply add the times to get the total time for different approaches. We can see that if the data set

496 is already partitioned in the processing machines, computing the models
497 utilizing our summarization matrix is fast in all cases. Partitioning the
498 data set first and then computing the ML models is a bit slower, but still
499 fast. Exporting the data set from HDFS adds more time over the previous
500 approach. On the other hand, Spark is mostly slow compared to any of
501 our approaches. For Linear Regression, Spark minimizes the specified loss
502 function with regularization. For PCA, the Spark-MLlib library uses a similar
503 algorithm as ours. It computes $X^T * X$ for large X by computing the outer
504 product of each row of the matrix by itself, then adding all the results up.
505 This is Q from our summarization matrix which Spark manipulates in the
506 main memory by each worker node. Still, Spark is slightly slower than our
507 method for computing only the PCA model. For Naïve Bayes, Spark-MLlib
508 implements the multinominal Naïve Bayes whose major drawback is having
509 negative values in the data set crashes the model. As Spark crashes showing
510 "illegalArgumentException" during the execution of NB, there is no plot for
511 Spark in Fig 2(c). And for K-means, Spark implements a parallelized variant
512 of k-means++ [8] which generates a k-means model and is roughly $O(k)$, so
513 this suffers a slower start with a large k. Also, it is expensive when the model
514 is trained. If we analyze the plots from Fig 2 more carefully, we can see that
515 when the data set is already partitioned, computing the models is at least
516 $2X$ faster than our other approaches. In the other approaches, we have to
517 partition the data set (data in disk), or export from HDFS and then perform
518 partition it (data in HDFS). This is slower because we are partitioning the
519 text (.csv) files, not binary files. This is a bottleneck and takes a long time.
520 However, it is due to the file format and not a shortcoming of our solution.
521 Moreover, R can read binary files and as we can call C++ code from R, it is
522 possible to read binary files efficiently in R (but CSV is most common).

523 However, there are some drawbacks to our solution. If some distribution
524 cannot be summarized with sufficient statistics from the Gaussian distribution,
525 then our approach would not work. For example, exponential distribution
526 may be inaccurate. But, in general, one or multiple Gaussian works well. Also,
527 it is not possible to get the original data set back from the summarization
528 matrix. That is, we cannot get X back from Γ or Γ^k.

529 *4.4. Comparison with a Parallel Big Data System: DBMS*

530 Now, we compare our solution with a parallel columnar DBMS (Vertica)
531 running on N processing nodes. In general, parallel columnar DBMS performs
532 better than row DBMS when d is not large. However, it is feasible that a

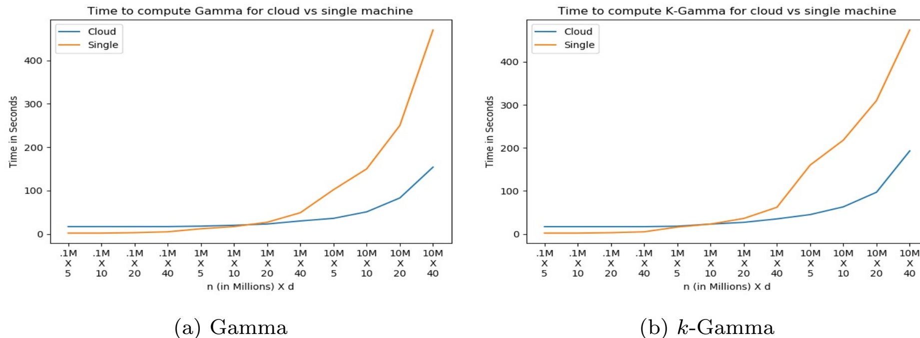


(a) Partition X and compute Gamma

(b) Compute Gamma

Figure 3: Time (in Sec) comparison for Γ on $N = 8$ nodes: our solution vs parallel DBMS for varying n and d (M=millions)

parallel row DBMS may be faster, but d would have to be very high, probably hundreds of columns and a dense matrix. We adapted the solution presented in [2] using UDFs and SQL queries which is the current best solution to compute the summarization matrix in a parallel DBMS. As there is no prior solution of Γ^k matrix in a DBMS, here we only compare our solution with the Γ matrix. We already know that the machine learning 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 N processing nodes which are shown in Fig 3. We compute Γ for varying $n = 1M, 10M, 100M$ and $d = 10$ in two cases: (1) when data is not partitioned (data in disk or HDFS), and (2) when data is already partitioned. We consider both cases to give the parallel DBMS a fair chance as it is often assumed for analytics that data is already stored in the DBMS. Fig 3a shows the comparison when data is in the disk. In this case, we split the data set into N processing nodes and then compute Γ. And, if the data set is in the HDFS, then we export the data first and then partition it as mentioned above. For DBMS, we used 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 in all cases than our solution when it has to partition the data first. On the other hand, Fig 3b shows the comparison to just compute Γ using N machines where the data set is already partitioned and loaded into DBMS. Our solution also performs better for Γ computation as n grows (Fig 3b). Also, DBMS solutions using UDFs are not portable and they require a lot of memory to scale up.



(a) Gamma

(b) k-Gamma

Figure 4: Time comparison for Γ and Γ^k in parallel cluster ($N = 8$ nodes) and single machine ($N = 1$ node) for varying n and d.

4.5. Understanding Trade-offs: Parallel Cluster and Single Machine

Here, we understand the trade-offs of computing our summarization matrix on a parallel cluster and a single machine. Though we have seen that parallel processing accelerates the computation, we may not need a large cluster each time, especially when the data set size is smaller. Instead, we can use a single machine to handle small data sets as parallel processing may introduce overhead and make the processing slower. Fig 4 shows the comparison to compute Γ and Γ^K in one machine and parallel ($N = 8$) machines. We can see that a 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 partitioning the data set and transferring the partial Γ_I (or Γ_I^k) matrices. On the other hand, parallel cluster seems to be 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 single machine and becomes the obvious choice for processing. This is because a single machine cannot scale as the data size grows due to limited memory. However, we should emphasize that these time measurements are only for partitioning the data set and computing the actual summarization matrix (Phase 0 and 1 from Section 3.3), and it does not include the time to compute the machine learning or statistical models (Phase 2 from Section 3.3).

5. Related Work

Here, we discuss the closely related work of other researchers as well as discuss the extension from our previous approaches.

5.1. Fast ML Algorithms

Research has developed fast algorithms mostly based on sampling, data summarization, and gradient descent [9], generally working in a sequential manner (data mining) [10]. Stochastic (incremental) gradient descent (SGD) [11], [12] 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). Also, MapReduce (MR) is another data mining technique used in big data analytics. Research has developed to classify big data [13], processing all-k-nearest-neighbor queries in parallel [14] using MapReduce. In [15] by Chu et al., a wide range of machine learning algorithms were parallelized by taking advantage of the summation form in a MapReduce framework. Using summation, the authors could achieve a 1.9 times speed up on a dual-processor without any special optimizations. However, MapReduce is not a suitable choice as not every algorithm can be implemented as an MR program and when we need to process data through iterations such as K-means. On the other hand, data summarization to accelerate the computation of machine learning models has received significant attention [2], [5], [16] [17]. Zhang et al. in [16] proposed to accelerate the computation of distance-based clustering: the sums of values and the sums of squares. Later Bradley et al. [17] exploited such summaries as multidimensional sufficient statistics for the K-means and EM clustering algorithms. Compared to our solution, these proposed techniques were useful only for one model (clustering). From a computational perspective, our Γ computation boils down to one matrix multiplication, whereas those algorithms work as aggregations. A more general data summarization capturing up to the fourth moment was proposed by William et al. in [18]. However, unlike our method, it relies on building histograms which are incompatible with most statistical methods. In summary, our summarization is a generalized algorithm that helps to compute statistical and complex ML models like LR, PCA, NB, and KM that could not be solved with older summaries.

5.2. Parallel Processing in ML

Parallel processing in machine learning has received attention due the to the sheer volume of data. There is a large body of work on computing machine learning models in Hadoop Big Data systems, before with MapReduce [19] and currently with Spark [20]. Distributed implementation of Logistic regression

616 and linear SVM using Spark was discussed by Lin et al. in [21]. Spark-
617 MLlib [22] is a popular open-source platform for large-scale data processing
618 which well-suited for iterative machine learning tasks. In this paper, we
619 compared this library and had similar or better performance in most cases
620 for our algorithms as discussed in Section 4. On the other hand, computing
621 models with parallel DBMSs have received less attention [11] because they are
622 considered cumbersome and more difficult to program. A previous approach
623 similar to our method was developed using SQL queries and UDFs for parallel
624 DBMSs in [2]. However, it did not have the k-Gamma summarization matrix
625 and our comparison in Section 4 shows the solution is much slower than our
626 current solution. Also, SQL is mostly popular for transactions and query
627 processing and UDFs have portability issues which makes analytics in DBMS
628 less popular. In the case of data science languages, there are some available
629 packages in R and Python for parallel computing. Compared to other R
630 parallel libraries like Revolution R [23] that requires Windows operating
631 system or pbdR [24] that provides high-level interfaces to MPI requires a
632 complex set up process, our solution avoids the complex set up process and is
633 not dependent on any OS. From a "systems" angle, R combined with C++
634 did not exist and nobody thought we could insert efficient C++ code for
635 a very common computation on parallel machines. El-Khamara et al. in
636 [25] argued that it is possible to enable massive parallelism with existing R
637 solutions with little to no modification. Also, Subramanian et al. in [26]
638 propose a framework that provides users with access to high-performance
639 computing resources with R through a web user interface.

640 This article is a significant step forward from [5]. Here, we present how we
641 can get descriptive statistics from our summarization matrix for the full data
642 set or the data subsets. Our experiments showed computing these statistics
643 has almost no performance penalty if they are computed on data subsets or
644 full data set. Also, we discuss how we can perform statistical tests based
645 on our summarization matrix. Moreover, as a pre-processing step, we do
646 not assume that data can only be in the file system. Rather, we presented
647 experiments with the data set being in the file system, cloud (HDFS), and
648 already partitioned in the processing nodes.

649 6. Conclusions

650 We proposed an efficient way to compute machine learning and statistical
651 model with parallel processing. Our general, parallel summarization algorithm

652 can work with multiple programming languages and platforms. We present
653 how our summarization matrix can help to compute descriptive statistics,
654 perform statistical tests, and compute ML models on the full data set or
655 data subsets. We then justified why C++ code is required and how it can
656 be integrated with a data science language, presenting R as an example.
657 Computing summarization matrix is done with vector outer products in
658 C++ and the model computation is performed with existing R functions.
659 The experimental evaluation section provides a detailed experiment and
660 comparison of our solution. We do not assume data can only be stored in the
661 file system. Rather, we provide experiments with data being in the file system,
662 cloud, or already partitioned in the processing machines. Our experiments
663 prove that our solution is more scalable than Spark and faster than Spark in
664 most cases. However, our solution suffers only when partitioning the data
665 set as we are doing it in a simple approach using available UNIX commands
666 where Spark is using HDFS, a well established distributed file system. On
667 the other hand, our solution is way faster than the previous version of the
668 summarization matrix which was done with UDF and SQL queries. We also
669 showed that our summarization matrix can be used to compute models on
670 data subsets or full data set with almost no performance penalty.

671 As for future work, we want to explore other dimensions. We intend to
672 study how to accelerate computation with multicore CPUs and GPUs in a
673 single box. We also want to explore more ML models, including Logistic
674 Regression, LDA, and SVMs. Also, we want to explore if we can extend our
675 approach to window data and stream such as, BIRCH algorithm that has
676 been extensively used for stream clustering. Moreover, we want to compare
677 the tradeoffs when we integrate our solution with other popular languages
678 like Python or JavaScript. Finally, we want to see how our solution behaves
679 on a sliding window.

680 **References**

681 [1] H. Hu, Y. Wen, T. Chua, X. Li, Toward scalable systems for big data
682 analytics: A technology tutorial, IEEE Access 2 (2014) 652–687.

683 [2] C. Ordonez, Y. Zhang, W. Cabrera, The Gamma matrix to summarize
684 dense and sparse data sets for big data analytics, IEEE Transactions on
685 Knowledge and Data Engineering (TKDE) 28 (2016) 1906–1918.

686 [3] S. T. Al-Amin, C. Ordonez, L. Bellatreche, Big data analytics: Exploring
687 graphs with optimized SQL queries, in: Proc.DEXA Conference, pp.
688 88–100.

689 [4] F. Li, S. Nath, Scalable data summarization on big data, Distributed
690 and Parallel Databases 32 (2014) 313–314.

691 [5] S. T. Al-Amin, C. Ordonez, Scalable machine learning on popular
692 analytic languages with parallel data summarization, in: Big Data
693 Analytics and Knowledge Discovery - 22nd International Conference,
694 DaWaK 2020, volume 12393, pp. 269–284.

695 [6] S. U. S. Chebolu, C. Ordonez, S. T. Al-Amin, Scalable machine learning
696 in the R language using a summarization matrix, in: Database and
697 Expert Systems Applications DEXA, pp. 247–262.

698 [7] C. Ordonez, E. Omiecinski, Accelerating EM clustering to find high-
699 quality solutions, Knowledge and Information Systems (KAIS) 7 (2005)
700 135–157.

701 [8] D. Arthur, S. Vassilvitskii, k-means++: the advantages of careful seeding,
702 in: Proceedings of the Eighteenth Annual ACM-SIAM Symposium on
703 Discrete Algorithms, SODA, pp. 1027–1035.

704 [9] R. Gemulla, E. Nijkamp, P. Haas, Y. Sismanis, Large-scale matrix
705 factorization with distributed stochastic gradient descent, in: Proc.
706 KDD, pp. 69–77.

707 [10] Z. Xie, Y. Xu, Q. Hu, Uncertain data classification with additive kernel
708 support vector machine, Data Knowl. Eng. 117 (2018) 87–97.

709 [11] J. Hellerstein, C. Re, F. Schoppmann, D. Wang, E. Fratkin, A. Gorajek,
710 K. Ng, C. Welton, The MADlib analytics library or MAD skills, the
711 SQL, Proc. of VLDB 5 (2012) 1700–1711.

712 [12] O. F. Reyes-Galaviz, W. Pedrycz, Z. He, N. J. Pizzi, A supervised
713 gradient-based learning algorithm for optimized entity resolution, Data
714 Knowl. Eng. 112 (2017) 106–129.

715 [13] C. Banchhor, N. Srinivasu, Integrating cuckoo search-grey wolf optimiza-
716 tion and correlative naive bayes classifier with map reduce model for big
717 data classification, Data Knowl. Eng. 127 (2020) 101788.

[14] P. Moutafis, G. Mavrommatis, M. Vassilakopoulos, S. Sioutas, Efficient processing of all-k-nearest-neighbor queries in the mapreduce programming framework, Data Knowl. Eng. 121 (2019) 42–70.

[15] C. Chu, S. Kim, Y. Lin, Y. Yu, G. Bradski, A. Ng, K. Olukotun, Mapreduce for machine learning on multicore, in: Proc. NIPS Conference, pp. 281–288.

[16] T. Zhang, R. Ramakrishnan, M. Livny, BIRCH: An efficient data clustering method for very large databases, in: Proc. ACM SIGMOD Conference, pp. 103–114.

[17] P. Bradley, U. Fayyad, C. Reina, Scaling clustering algorithms to large databases, in: Proc. ACM KDD Conference, pp. 9–15.

[18] W. DuMouchel, C. Volinski, T. Johnson, D. Pregybon, Squashing flat files flatter, in: Proc. ACM KDD Conference.

[19] A. Behm, V. Borkar, M. Carey, R. Grover, C. Li, N. Onose, R. Vernica, A. Deutsch, Y. Papakonstantinou, V. Tsotras, ASTERIX: towards a scalable, semistructured data platform for evolving-world models, Distributed and Parallel Databases (DAPD) 29 (2011) 185–216.

[20] M. Zaharia, M. Chowdhury, M. Franklin, S. Shenker, I. Stoica, Spark: Cluster computing with working sets, in: HotCloud USENIX Workshop.

[21] C. Lin, C. Tsai, C. Lee, C. Lin, Large-scale logistic regression and linear support vector machines using spark, in: 2014 IEEE International Conference on Big Data, Big Data 2014, Washington, DC, USA, October 27-30, 2014, IEEE Computer Society, 2014, pp. 519–528.

[22] X. Meng, J. K. Bradley, B. Yavuz, et al., Mllib: Machine learning in apache spark, J. Mach. Learn. Res. 17 (2016) 34:1–34:7.

[23] J. Rickert, Big data analysis with revolution r enterprise, Revolution Analytics (2011).

[24] G. Ostrouchov, W.-C. Chen, D. Schmidt, P. Patel, Programming with big data in r, 2012.

747 [25] Y. E. Khamra, N. Gaffney, D. Walling, E. A. Wernert, W. Xu, H. Zhang,
748 Performance evaluation of R with intel xeon phi coprocessor, in: Pro-
749 ceedings of the 2013 IEEE International Conference on Big Data, pp.
750 23–30.

751 [26] R. Subramanian, H. Zhang, Parallel R computing on the web, in: 2019
752 IEEE International Conference on Big Data (Big Data), pp. 3416–3423.