Extending the R Language with a Scalable Matrix Summarization Operator

Sikder Tahsin Al-Amin, Siva Uday Sampreeth Chebolu, Carlos Ordonez
Department of Computer Science
University of Houston, TX 77204, USA

***Abstract*—Analysts prefer simpler interpreted languages to program their computations. Prominent languages include R, Python, and Matlab. On the other hand, analysts aim to compute mathematical models as fast as possible, especially with large data sets. Data summarization remains a fundamental technique to accelerate machine learning computations. Based on this motivation, we propose a novel summarization mechanism computed via a single matrix multiplication in the statistical R language. We show our summarization benefits a large family of linear models, including Linear Regression, PCA, and Naive Bayes. We present a subsystem that enables exploiting summarization by detecting Gramian matrix products in R. We optimize the existing R source code by overriding the internal R matrix multiplication algorithm using ours. Our solution can be plugged into R and help solving where a similar matrix multiplication appears, much faster and without RAM limitations. Moreover, our solution can be benefited from the parallel processing ability of the summarization matrix. We present an experimental validation showing our subsystem incurs little overhead since it works on source code while providing much faster speeds compared to the R language built-in functions. To round up our comparisons, we also compare our subsystem with Spark in parallel machines. For our solution, we assume that data can be in the HDFS, disk, or already partitioned. Our solution triumphs Spark in most cases proving we can also compete in the big data space.**

***Index Terms*—R, Summarization Operator, Matrix Multiplication, Machine Learning.**

I. Introduction

Machine Learning has been gaining much traction recently due to an explosion in the availability of data and processing power. There are a lot of languages and technologies like Python, R, Matlab, Java, C++, and many more for building machine learning models. Even having the advantage of compiling at faster speeds, Java and C++ still suffer from many limitations like memory-management, complicated and strict syntax, and little scope to create new operators. On the other hand, Python and R offer extensive library support to analysts outside standard built-in libraries. Moreover, advancement in hardware, simplicity, machine independency, and library support has made analysts high-level languages like Python or R as their favorite choice for data analysis.

Although deep learning is currently popular and being used by many data analysts [1], [8], it is still computationally expensive and analysts often rely on trial and error to find a working model. On the other hand analytic languages like Python, R provide comprehensive libraries to support machine learning and statistical computation. However, they are not designed to scale to large data sets. Also, there are issues of memory management and speed for large data sets. These factors create a dilemma for the analyst. In this work, we focus on the R language and present an efficient way to compute several ML models exploiting the R language interpreter and summarization mechanism. Data summarization is also another popular technique among machine learning practitioners to accelerate ML computations [10], [12], [14].

Our contributions are: (1) We introduce a new matrix multiplication operator in the R language that helps summarize the large data sets without RAM limitation. (2) We detect the Gramian (or Gram) matrix products and overrides the internal R matrix multiplication algorithm using ours. (3) We present parallel processing aspects to compute summarization to compete with other existing big data solutions. Though modifying the R parser is not easy, we understand how R does it, and we modify the parser to override the matrix multiplication operator. We combine R with C++ for efficiency and overcome the main memory limitation. Our summarization benefits a large family of linear models including Linear Regression (LR), Principal Component Analysis (PCA), and Naïve Bayes (NB) [4]. Analysts compute the models much faster than R built-in functions as well as write short, elegant R source code. Finally, given the fact that there are many R scripts, it is easier to plug in our optimization than rewriting them to work in some Hadoop platform (e.g. Spark).

II. Preliminaries

A. 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. We use $d \times n$ for convenience of math notation but in practice X is stored $n \times d$. 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 $\mathbf{R}^d$. In the case of predictive models, we augment X with a $(d+1)$ dimension: an output variable Y for regression or discrete attribute G for the class (most commonly binary), making X a $(d+1) \times n$ matrix and we call it $\mathbf{X}$. We represent Θ as a statistical or ML model (LR, PCA, NB).

B. Matrix Multiplication in the R Language

There are several ways to compute the matrix multiplication in R. The most common way of doing that in R is by using the $\% * \%$ operator. This operator multiplies two matrices if they are compatible. If one argument is a vector, it will be promoted to either a row or column matrix to make the two arguments compatible. If both are vectors of the same length, it will return the inner product (as a matrix). Another method is the sum of the vectorized product of rows. In this case, we slice the rows of the first matrix and the columns of the second matrix as separate vectors. Then we compute the sum of the products of corresponding rows and columns and store it in a new matrix, like using two loops. An example of $\% * \%$ in R to get the mean and variance of X with n rows is given below:

```
mu  = colSums(X) /n
var = (t(X) %*%  X)/n - (t(mu) %*% mu)
```

In this work, we will create a new operator $(*)$ different from this $\% * \%$ operator and we will give it a different meaning: Gramian matrix or Gram matrix multiplication operator.

III. A POWERFUL MATRIX MULTIPLICATION IN R

First, we review the summarization matrix from our previous work. Then, we present our R scripts to compute the ML models and discuss how we detect the Gramian matrix product and override the internal R matrix multiplication. Finally, we discuss the parallel processing aspects of our solution.

A. Summarization Matrix

We first review the Gamma summarization matrix (Γ) [4], [12] and computation of several ML models (Θ) exploiting Γ. The main algorithm had 2 phases.

- Phase 1: Computation of Summarization Matrix; one Γ matrix or k-Γ matrices.
- Phase 2: Computation of model Θ based on the Γ matrix (or matrices).

1) Phase 1: 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 [12], 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:

$$\Gamma = \begin{bmatrix} \text{n} & \text{L}^T & \mathbf{1}^T \cdot Y^T \\ \text{L} & \text{Q} & \text{X Y}^T \\ \text{Y} \cdot \mathbf{1} & \text{Y X}^T & \text{Y Y}^T \end{bmatrix} = \begin{bmatrix} \text{n} & \sum x_i^T & \sum y_i \\ \sum x_i & \sum x_i x_i^T & \sum x_i y_i \\ \sum y_i & \sum y_i x_i^T & \sum y_i^2 \end{bmatrix} \tag{1}$$

X is defined as a $d \times n$ matrix, and Z is defined as a $(d+2) \times n$ matrix ($\mathbf{X}$ augmented with extra row of n 1s). From [12], 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 [4], 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. Here, we need only a few parameters out of the whole Γ, namely, n, L, L^T, Q. 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.

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

2) Phase 2: Now, we briefly discuss how to compute the ML models using the Γ and Γ^k matrix. A detailed explanation can be found in [4].

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

b) Principal Component Analysis (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: $\rho_{ab} = (nQ_{ab} - L_a L_b)/(\sqrt{nQ_{aa} - L_a^2}\sqrt{nQ_{bb} - L_b^2})$

c) Naïve Bayes (NB): Here, we need the k-Gamma matrix. 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.

$$W_G = \frac{n_G}{n};\ C_G = \frac{L_G}{n_G};\ R_G = \frac{Q_G}{n_G} - diag\frac{[L_G L_G^T]}{n_G^2} \tag{3}$$

B. Automatic Optimization of R programs

Here, we present the R source code to compute the summarization matrix and the models step by step in the R interpreter. While the summarization matrix can be exploited to compute many models, we are showing some representative models here. A detailed explanation of how we parse, detect, and compute summarization is given below.

1) Exploratory Statistics: We can compute exploratory statistics from Gamma like mean, variance, and correlation based on n, L, and Q. These are common computations and can tell a lot about the data. The analyst can also exploit these statistics to compute other models as they appear frequently in machine learning models. Analysts may also get a subset of data (e.g. separate data based on gender or age) and see the statistics there using Γ. The R source code to compute mean, variance, and correlation is given below in Listing 1. How we detect Γ computation in the R source code, compute it escaping main memory limitation, and hide the details from the user is discussed in Section III-C.

Listing 1: R source code to compute the mean, variance and correlation.

```
x = read.csv('YearPredictionMSD.csv')
Z = Z(x)
gamma = t(Z) * Z

d_plus2 = length(gamma[1,])
d = d_plus2 - 1
```

```
n = gamma[1,1]
L = gamma[2:d,1]
Q = gamma[2:d,2:d]

mu = L/n #mean
variance = Q/n - L*L/(n^2) #variance
corr.matrix = matrix(nrow = length(Q[1,]),
            ncol = length(Q[1,]),0)
for (a in 1:length(Q[1,])) {
  denom_1 = sqrt((n*Q[a,a]) - (L[a]*L[a]))
  for (b in 1:length(Q[1,])) {
    numerator = ((n*Q[a,b]) - (L[a]*L[b]))
    denom_2 = sqrt((n*Q[b,b]) - (L[b]*L[b]))
    corr.matrix[a,b] = numerator/(denom_1*denom_2)}
}
return list(mu,variance,corr.matrix)
```

2) Linear Regression (LR): For LR, we can get the regression coefficient ($\hat{\beta}$) as mentioned in Section III-A. The R source code is shown in Listing 2. First, we compute Γ (*gamma*) on the data set (*x*). From $gamma$ we compute Q and XYT. And then we compute $Beta$ by getting the inverse of Q and multiplying with XYT. We use R built in function $solve()$ to get the inverse of Q.

Listing 2: R source code to compute the LR model.

```
x = read.csv('YearPredictionMSD.csv')
Z = Z(x)
gamma = t(Z) * Z

d_plus2 = length(gamma[1,])
Q = gamma[1:(d_plus2-1), 1:(d_plus2-1)]
XYT = gamma[1:(d_plus2-1), d_plus2]
Beta = solve(Q) * XYT
return Beta
```

3) Principal Component Analysis (PCA): Similar to Listing 1, first we detect and compute Γ (*gamma*) from the data set (*x*). Then, from the $gamma$, we get the values of n, L, and Q, and from these values we get the correlation matrix ($corr.matrix$) as given in Listing 1. Finally, we compute the SVD of the correlation matrix using the R built-in function ($svd(corr.matrix)$). As the R code is similar to Listing 1 except for the $svd()$ computation part, we are not showing the R code due to page limitation.

4) Naïve Bayes (NB): We compute NB based on Γ^k matrices. However, we have to split the data set based on class labels (k) and compute Γ for each class. So, here the Γ computation is different than LR and PCA as we split the data set and create a subset based on class labels. Then for each $gamma\_class$ we get the values on n, L and Q. Finally, we compute *prior*, *mu* and *sigma* (from R source code in Listing 3) as given in Eq 3 for each $gamma\_class$ and return them as a list to the $model$.

Listing 3: R source code to compute the NB model.

```
credit = read.csv('creditcard.csv')
nGlobal = dim(credit)[1]
gamma = by(credit, list(class=credit$V31),
        FUN = function(x){
  xj <- as.matrix(subset(x, select= -V31))
  Z = Z(xj)
  gamma = t(Z) * Z
})
model = apply(gamma, 1, function(x){
  x_list = unlist(x)
  gamma_class = matrix(x_list,
            nrow = sqrt(length(x_list)),
            ncol = sqrt(length(x_list))) #G=class
  d_plus1 = nrow(gamma_class)
  n = gamma_class[1,1]
  L = gamma_class[2:d_plus1,1]
  Q = diag(gamma_class[2:d_plus1,2:d_plus1])

  prior = n/nGlobal
  mu = L/n
  sigma = Q/n - (L/n)^2)
  return(list(prior, mu, sigma))
})
return model
```

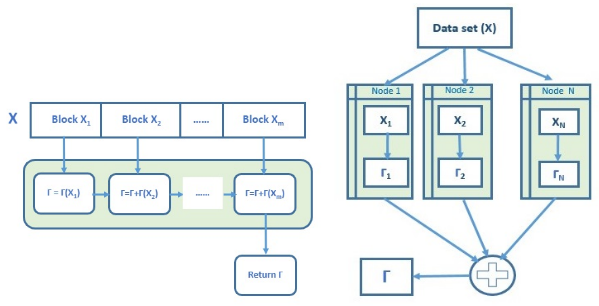
C. Extending the R Language with an Efficient Gramian Matrix Multiplication Operator

First, we discuss several ways to program Γ in plain R source code. We show this to justify why our solution with a multiplication operator is needed. The ”brute-force” approach will be using three nested loops. We are not showing the source code of this as it is inefficient and have high time complexity. Another approach will be using a vectorized multiplication technique. Γ computed this way is slightly more efficient than the previous one. However, it is still slow and nobody would attempt to do this unless they have Γ in mind or they know well how matrix multiplication works (i.e. not the average analyst). So, it is important to have our solution with C++ that can compute Γ faster and efficiently than R itself.

In our work, we build the new operator ($*$) by overriding an existing operator in the R language. Though this $*$ operator is not used for matrix multiplication in R, we give this $*$ operator a different meaning: Gramian or Gram matrix multiplication. Instead of rebuilding the matrix multiplication operator from scratch, we built it using R's extensibility features, and our changes are a minimal change in the R parser. First, we parse the source code to detect a special type of matrix multiplication, $t(X) * X$ or $X * t(X)$. We emphasize that computing $X * X^T$ is a common form of matrix multiplication and it appears on many ML models as it is related to the covariance matrix (e.g. gaussian mixture, PCA, SVM, time series models). We presented some of the representative models in Section III-B. We use an R language parser that will parse the R source code for this special case of matrix multiplication. If the parser detects this multiplication operation, it calls our Gamma package inside the evaluation. This is where we modify the R's evaluation. The grammar to override the operator is given below.

```
<Gramian>::= <matrix> * t(<matrix>)
<Gramian>::= t(<matrix>) * <matrix>
```

The grammar shows “matrix” can be on both left and right sides. We do not require the analyst to compute Gamma in a specific way as Γ is $O(d^2)$ [12]. Specifying it in one fixed way may get confusing to the analyst and writing it the other way will have the same complexity. So, the data set (X) can be stored as $d \times n$ or $n \times d$. Our parser will be able to handle both kinds of operations. If the parser detects any such operation



(a) (b)

Fig. 1: Compute the summarization matrix in (a) one machine (b) parallel N machines.

in the file, then it calls Gamma based on the model being computed i.e., it calls Gamma function if the model is LR or PCA, else it calls the k-Gamma if the model is NB. To compute Γ or Γ^k, we adapted the scalability of the matrix multiplication from [4]. We show the full process in Fig 1(a) for a single machine. In short, we read the data in blocks and perform the matrix multiplication in C++ and return the result to the R language. As we are reading the data in blocks, we compute Γ for each block ($\Gamma_\triangle$), and add it with the Γ computed up to the previous block ($\Gamma = \Gamma + \Gamma_\triangle$). This way we update Γ in RAM incrementally and it can be maintained in the RAM. Reading data by blocks also gives us the flexibility to read data sets bigger than the RAM. But all these details are hidden from end-user, who will only see the R source code like the ones in Listing 1, 2, and 3. Though we present Gamma computation in general, our solution can help computing $X * X^T$ as a particular case.

In this paper, our prime focus is to make the Γ computation transparent and how an analyst can easily make changes to the algorithm as he tries different things using the Γ matrix. Source codes in R are similar to macros, which when executes behave as though analysts had typed those commands into the command window. Thus, all variables created in the program are added to the R workspace of that current session. This can be handy while computing several models on the same data set that require the same type of Γ matrix. For example, from Listing 2, the model computation steps are unique for each model. So, we can reuse the Γ computed from the LR model while computing PCA and vice versa as Γ is already in the workspace.

D. Parallel Processing Aspects (Big Data)

The summarization matrix proposed in Section III-A is fully parallel and it has been implemented for DBMSs before [13]. Here, we explain how it can work outside a DBMS architecture. Let us assume we have parallel N processing nodes ($d << n$ and $N << n$). Computing $t(Z) * Z$ or $Z * t(Z)$ can be easily parallelized as we can compute this on partial data sets in each node and then add them on the master node to get the final result. We emphasize that only computing $t(Z) * Z$ or $Z * t(Z)$ can be parallelized and then the remaining of the scripts in Listing 1, 2, and 3 should be done in one machine as R by default works better in a single machine. The idea is to first detect the Γ operation in the source code as before. Then instead of computing Γ in one machine, we partition the data set X $(X_1, X_2, ..., X_N)$ and transfer the partial data sets to N machines. Then in each machine, we compute local Γ $(\Gamma_1, \Gamma_2, ..., \Gamma_N)$ on the partial data set using the methodology discussed in Section III-C. Each local Γ is a $d \times d$ matrix. Next, we send all the local Γs to the master node and we compute the final summarization matrix with a simple matrix addition: $\Gamma = \Gamma_1 + \Gamma_2 + ... + \Gamma_N$. Then, this Γ can be used to compute the models using the R scripts given in Section III-B. The full process to compute Γ in parallel is shown in Fig 1(b).

The main bottleneck of this method is partitioning the data set X into N machines. We explain in more detail why this is a bottleneck in Section IV-D. Partitioning a large file among several machines has been studied and implemented efficiently by HDFS [15] and parallel DBMSs [17] and it is beyond the scope of this paper. However, as each local Γ is a $d \times d$ matrix, transferring them to the master node is much faster.

E. Time Complexity Analysis

From [4], it is clear that the computation time for Γ is $O(d^2n)$ and Γ^k is $O(dn)$ if the processing happens in a single machine. For parallel computation, the complexity will be changed to $O(d^2n/N)$ and $O(dn/N)$ respectively. The time complexity of the parser is $O(1)$ as the number of lines in the R source code is independent of d or n. We take advantage of Γ (or Γ^k) to accelerate computing the ML models. So, the time complexity of this part does not depend on n and is $\Omega(d^3)$. In the case of parallel computation, transferring all the local Γs to the master node at once is: $O(d^2)$, for sequential transfer: $O(d^2N)$, and for hierarchical binary tree fashion: $O(d^2N + \log_2(N)d^2)$.

IV. Experimental Evaluation

Here, we compare our proposed solution with built-in R and our previous approach in [4]. We also compare with Spark running in parallel clusters to show our solution can compete in the big data space.

A. Experimental Setup

The system used for the experiments is one with Pentium(R) Quadcore CPU running at 1.60 GHz with Linux Ubuntu as the operating system. The system has 8 GB of Physical Memory and 1 TB of storage space. For parallel processing, we perform our experiments using an 8-node parallel cluster each with the same configuration as above. We developed our solutions using standard R and C++.

TABLE I: Base data sets description

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

The data sets are summarized in Table I. They are obtained from the UCI machine learning repository. We have already showed in [4] that we can get more than 99% accurate model using ”original” data sets by our summarization mechanism. Hence, we are not repeating the same experiments here.

B. Parsing R Source Code

We use R to parse the R source code and detect the Gamma computation. How we detect the Gamma computation is discussed in section III-C. The parsing is fast even though we do it in R ($< 1\ sec$) and not in C++. And the time to override the “*” is also done in a fraction of seconds ($< 1\ sec$). We also detect what type of model the code is computing. Based on the type of model computation, we call Gamma or k-Gamma. We also import (load) the previous library to compute Gamma in C++ [4]. That time is also minimal ($< 1\ sec$) and negligible.We emphasize that we do not need an LR or recursive parser because we assume Gamma appears once in one R line and therefore there is no recursion in the grammar. However, an entire matrix equation does indeed require a recursive grammar, but that is handled by the R grammar.

C. Benchmarking

TABLE II: Time to compute $X * t(X)$ using multiplication operators in R (in Seconds) (M= Millions).

Matrix ($n \times d$)	**%*% (R)**	**Vectorized mul. (R)**	**Gamma**
$0.5M \times 91$	10	292	5
$1M \times 9$	1	6	1
$1M \times 91$	66	588	10
$10M \times 30$	Stop	Stop	1000

First, we compare our Gamma matrix with the standard matrix multiplication operators in R as discussed in Section II-B to show why our multiplication scheme is needed. We present the result of multiplying a matrix with itself ($X * t(X)$) for varying n and d in Table II. Here, it is clear that our Gamma computation is much faster than the other two methods. And when the matrix dimension is high, the other two methods fail to compute the multiplication in a reasonable time and they do not scale well. We put ”Stop” if the computation is not finished in 30 minutes.

TABLE III: Time to compute LR and PCA on YearPrediction-MSD Data set (in Seconds).

		LR			PCA		
n	d	R	Γ-R	Γ-RLang	R	Γ-R	Γ-RLang
1M	91	630	74	55	575	66	60
10M	91	Fail	720	569	Fail	726	570
1M	9	24	6	7	21	9	7
10M	9	285	91	78	205	91	78
100M	9	Fail	941	820	Fail	1018	802

TABLE IV: Time to compute Naïve Bayes on CreditCard data set (in Seconds).

n	d	R	Γ-R	Γ-RLang
1M	30	158	40	32
10M	30	Crash	399	312
100M	30	Crash	1132	Stop

Now, we compare the performance of the models computed by our solution with the models computed by the currently available best packages in R, and with the models computed by the Γ-R package [4]. We want to show that our solution is much faster than R built-in packages and it performs almost as good as our previous solution. To compute our models we need to parse the R code first, detect a Gamma computation, and then call our method. We report the time including these computations done in Section IV-B. In Table III, we present a time comparison of the LR and PCA model computations with our solution (Γ-RLang) and the aforementioned standards (R and Γ-R). Table IV shows the time to compute the NB model. As the size of the data set grows, the standard R packages crash due to insufficient main memory, but our solution performs almost as good as the Γ-R package. However, NB script from Listing 3 shows that we are using built-in R functions ($apply(), unlist()$) to compute the models that tends to make the computation slower than the previous one (Γ-R) where we could use C++ functions to escape these pitfalls. As the analyst is developing the script in R, we used the R built-in functions for the model computation. In short, our solution provides the analysts faster execution than R built-in routines and greater flexibility to analyze the data sets than our previous solution without much time compensation.

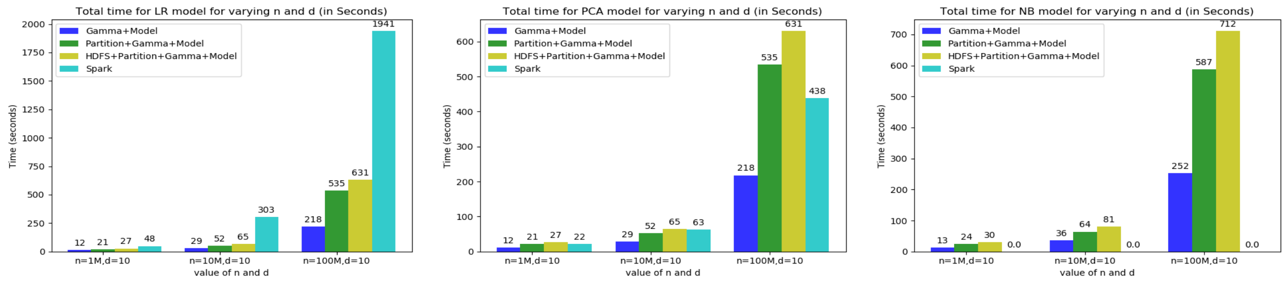
D. Comparison with a Parallel Big Data System: Spark

Here, we compare our solution with Spark, 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 used the available functions in MLlib and we emphasize that we used the recommended settings and parameters as given in the library documentation.

Table V compares our solution with Spark-MLlib in parallel $N = 8$ machines. Here, we are taking data sets with varying n (n=1M, 10M, 100M) and medium d (d=10) to demonstrate how large data sets perform on both. For our solution, we assume that data to be analyzed can be stored in the file system, in HDFS, or already partitioned in disks. If data is in the file system, we need to partition the data set among N machines. Data can be also in HDFS 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. Finally, if the data set is already partitioned, we do not have to do any partitioning. From Table V, the ’Partition’ column is the time to partition X among N processing nodes. We used the standard and fastest UNIX commands available to perform this operation. The ’Export from HDFS’ column is the time to export the data set X from HDFS to the local machine. And the ’Compute Γ and Θ’ 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 ML model (Θ) from it. In the Spark part of Table V, ’HDFS 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 ’Compute Θ’ column.

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

Θ				Our solution	(N=8)	Spark	(N=8)
(Data set)	n	d	Partition	Export from HDFS	Compute Γ and Θ	HDFS Partition	Compute 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



(a) LR (b) PCA (c) NB

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

Despite HDFS being faster to partition the data sets, the time to compute the model (Θ) is much slower in Spark. However, Spark uses a similar algorithm as ours to compute PCA. It computes $X^T.X$ by computing the outer product of each row of the matrix itself, then adds all the result up which is similar to compute Q of Γ. In the case of LR, Spark trains the model using Stochastic Gradient Descent (SGD) which solves the least square regression formulation. This results in slower execution of the model as shown in Table V. For NB, Spark implements multinominal NB that takes RDD of labeled point and an optional smoothing parameter as input. The major drawback of this model is, having negative values in the data set crashes the model showing "illegalArgumentException" which has happened for the Creditcard data set here.

Fig 2 shows the total time to compute the ML models using different approaches discussed above. The plots are generated based on Table V. We simply add the times to get the total time for different approaches. We can see that if the data set is already partitioned, computing the models utilizing the Γ matrix is fast in all cases. Computing models by partitioning the data set or exporting from HDFS and then partitioning takes a bit more time. On the other hand, Spark is mostly slow compared to any of our approaches. As Spark crashes during the execution of NB, there is no plot for Spark in Fig 2(c). If we analyze the plots more carefully, we can see that only computing model with Γ (Gamma+Model) is at least $2X$ faster than other approaches. That is, the data set is already partitioned among N machines. In the other two methods, we have to perform partitioning the data set (when data is in the file system), or export from HDFS and then perform partitioning (when data is in HDFS). Both cases are a bit slower. The reason behind that is we are partitioning the text (.csv) files, not binary files. This is a bottleneck and taking a long time as we mentioned in Section III-D. However, it is due to the file format and not a shortcoming of our solution. We present Table VI to prove our aforementioned argument. It shows the time to load the data set in parallel for both our method and a parallel columnar DBMS. Parallel DBMSs can send blocks efficiently in parallel among the processing nodes. However, it is still much slower than our method. Moreover, R can read binary files and as we can call C++ code from R, it is possible to read binary files efficiently in R (but CSV is most common).

TABLE VI: Time to load data in parallel (in Seconds).

n	d	Our method (N = 8 nodes)	DBMS (N = 8 nodes)
1M	10	9	29
10M	10	23	141
100M	10	317	1711

V. Related Work

We first discuss the general related work of other researchers. Many systems improved the efficiency of the R language for large data sets. S. Sridharan et al. [16] addresses the limitation of R by systematically cataloging where time is spent when running R programs. According to S. Sridharan et al. when analyzing large data sets, R programs spends most time in processor stalls, trigger garbage collector frequently and creating a large number of unnecessary temporary objects. Ricardo [6], developed by IBM, combines the data management capabilities of Hadoop and Jaql with the statistical functionality provided by R. El-Khamara et al. in [9] argued that it is possible to enable massive parallelism with existing R solutions with little to no modification. Also, Subramanian et al. in [18] propose a framework that provides users with access to high-performance computing resources with R through a web user interface. With higher data volume and varied data types, many new machine learning models and algorithms aiming at scalable applications [2], [5], [11], [13]. Large-scale distributed classification methods for Spark was proposed by [11], where the authors used the Newton methods for solving logistic regression and SVM. On the other hand, for handling large data sets in Python, two libraries was proposed in [5].

We conclude our discussion comparing with our previous works. Summarization of large data sets was first proposed for DBMS in [12] combining UDFs and SQL queries. Being a mathematical language, it is much easier to manipulate vectors and matrices in R than in DBMS. We use Rcpp [7] to integrate R and C++. Also, any average analyst will mostly use R for computing ML models than using SQL and UDF. Taylor et al. proposed iotools [3], which provides a set of tools for input and output intensive data processing in R. This enables to process data block by block and allows scalability in R. Our work is scalable, simpler, and has more impact as $X * X^T$ appears almost everywhere in ML model computation. Algorithms to compute the models discussed in this paper were first proposed in [4] utilizing R. In this paper, we adapted the algorithms from [4] and express them in a more mathematical way which can be used in the R interpreter. Also, we provide exploratory statistics that can help the analysts to explore data sets, combine multiple models, and compute other models using our solution. Moreover, we present the parallel processing aspect here which was were not investigated in [4]. Parallel processing helps our solution to handle even larger data sets than before.

VI. Conclusions

We introduced a new operator for the summarization of large data sets by matrix multiplication that can be used in R source code. Alongside developing the clean, elegant, and intuitive scripts, we also preserved the approaches to solve the main memory limitation in the R language and achieve high speed by computing the summarization matrix in the C++ code. Though the evaluation is difficult, we hide the details from the analysts and the analysts can perform analysis with standard and short R programs. Parsing and detecting expression where our summarization matrix can help can be done efficiently in a fraction of 1 second. Our parse just needs one pass to detect and evaluate certain sub-steps more efficiently. We also presented the parallel processing aspects to handle even larger data sets. Our proposal works for many ML models that have mean, variance, covariance, or correlation computations. We used LR, PCA, NB as common examples. Our time performance experiments show that our solution is faster and more scalable than the built-in functions in R. Furthermore, our parallel approach performs much better than Spark although partitioning the data set among the machines remains a bottleneck.

We believe that apart from looking for just one instance of matrix multiplication in a R program, future research should look for detecting R programs where it may appear multiple times. We want to explore the other statistical and ML computations that may be benefited: most salient example statistical tests like means comparison, chi-square, and so on. However, there may be other computations in an R program that do not benefit from our solution. For instance, $\log()$ on vectors, splitting variables in decision trees, histograms because they are not based on vector or matrix multiplications. It is likely our approach can work in other "analytic" languages like Matlab and Python, but the effort to modify their parsers and run-time evaluation may be significant. Nevertheless, as future work, we intend to study the applicability of our summarization operator in other areas of research such as image recognition, text processing, and mathematical optimization methods.

References

[1] Aizman, A., Maltby, G., Breuel, T.: High performance I/O for large scale deep learning. In: 2019 IEEE International Conference on Big Data (Big Data). pp. 5965–5967. IEEE (2019)

[2] Al-Amin, S.T., Ordonez, C.: Scalable machine learning on popular analytic languages with parallel data summarization. In: Big Data Analytics and Knowledge Discovery- DaWaK 2020. vol. 12393, pp. 269–284 (2020)

[3] Arnold, T., Kane, M.J., Urbanek, S.: iotools: High-Performance I/O Tools for R. The R Journal pp. 6–13 (2017)

[4] 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 DEXA. pp. 247–262 (2019)

[5] Crist, J.: Dask & numba: Simple libraries for optimizing scientific python code. In: 2016 IEEE International Conference on Big Data,. pp. 2342–2343 (2016)

[6] Das, S., Sismanis, Y., Beyer, K., Gemulla, R., Haas, P., McPherson, J.: RICARDO: integrating R and hadoop. In: Proc. ACM SIGMOD Conference. pp. 987–998 (2010)

[7] Eddelbuettel, D.: Seamless R and C++ Integration with Rcpp. Springer, New York (2013)

[8] Garg, S., Krishnan, R., Jagannathan, S., Samaranayake, V.A.: Distributed learning of deep sparse neural networks for high-dimensional classification. In: IEEE International Conference on Big Data, Big Data 2018. pp. 1587–1592. IEEE (2018)

[9] Khamra, Y.E., Gaffney, N., Walling, D., et. al: Performance evaluation of R with intel xeon phi coprocessor. In: IEEE International Conference on Big Data. pp. 23–30 (2013)

[10] Li, F., Nath, S.: Scalable data summarization on big data. Distributed and Parallel Databases **32**(3), 313–314 (2014)

[11] Lin, C., Tsai, C., Lee, C., Lin, C.: Large-scale logistic regression and linear support vector machines using spark. In: 2014 IEEE International Conference on Big Data. pp. 519–528 (2014)

[12] 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)

[13] Ordonez, C., Zhang, Y., Johnsson, S.L.: Scalable machine learning computing a data summarization matrix with a parallel array DBMS. Distributed and Parallel Databases **37**(3), 329–350 (2019)

[14] Shah, Z., Mahmood, A.N.: A summarization paradigm for big data. In: 2014 IEEE International Conference on Big Data. pp. 61–63 (2014)

[15] Shvachko, K., Kuang, H., Radia, S., Chansler, R.: The hadoop distributed file system. In: IEEE 26th Symposium on Mass Storage Systems and Technologies, MSST. pp. 1–10. IEEE Computer Society (2010)

[16] Sridharan, S., Patel, J.: Profiling r on a contemporary processor. Proceedings of the VLDB Endowment **8**, 173–184 (2014)

[17] 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)

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