Fast Machine Learning in Data Science with a Comprehensive Data Summarization

Sikder Tahsin Al-Amin, Carlos Ordonez
Department of Computer Science
University of Houston
USA

***Abstract*—Machine learning algorithms must be able to handle large volume in big data. Nowadays, data science languages such as Python and R, are widely popular to compute machine learning models. Unfortunately, model computation can be slow, especially when the data set does not fit in the main memory or it needs to be iteratively analyzed. With these motivations in mind, we present theory and algorithms to produce a multidimensional data set summary. We show our data summaries preserves essential statistical properties of the data set and it can be computed with an accelerated Gramian matrix multiplication. That is, our data summaries represents a lossless compression and we accelerate the expensive Gramian matrix multiplication in Python with C++ code. Our solution also works for a subset of the original data set obtained by variable selection without much loss on accuracy and without recomputing all intermediate matrices. We also consider parallel processing aspects leveraging our recently introduced low-cost parallel architecture. Our experimental evaluation shows that our Gramian matrix multiplication mechanism is superior to Python and it can work beyond RAM limitations. On the other hand, our computation of the machine learning model is competitive with the Python and R on a single machine, but it outperforms Spark in parallel machines.**

***Index Terms*—Machine Learning, Matrix Multiplication, Data Science, Parallel Processing, Data Summarization**

I. Introduction

Machine learning (ML) is at the heart of data science. Data nowadays is growing at a speed that has never been seen before. Machine learning algorithms must be able to handle this increased volume of data. Data science practitioners use various tools and programming languages to analyze data or build a machine learning model. For example, Python, R, JavaScript, Matlab are frequently used in data science nowadays. These languages offer huge library support and they are easy to learn. Moreover, portability, strong community, and easy integration feature with other languages have made them the "go-to" choice for machine learning or data analysis. However, processing large data sets in these languages remains a bottleneck especially when the processing is done in a single machine with limited memory. Though big companies can move the processing to the cloud with larger memory and faster processing speed, it is not the case for average data science practitioners, mostly due to budget limitations. They have to rely on their single machine with limited memory to handle the large data sets. Hence, it is important to analyze these large data sets or apply machine learning models, and get the results back as soon as possible.

On the other hand, matrix multiplication is a common technique that has been used in many areas to perform certain operations [16]. Many research has been conducted to optimize the matrix multiplication techniques and use it in the desired applications. One of the dominant uses of matrix multiplication is in data summarization [5], [8]. Despite being a common and essential computation, most data science languages become slow when the matrices are large, or fail when they are bigger than the main memory. With these motivations in mind, here, we present a technique to accelerate the computation of several machine learning models by computing data summaries from large data sets in data science languages (e.g. Python). And our data summaries are computed using our efficient Gramian matrix multiplication mechanism.

Our contributions are the following: (1) We present an efficient way to generate data summaries for large data sets using fast Gramian matrix multiplication. (2) We discuss several ways to store the multiple data summaries and provide an innovative way to store them. (3) We explore and present a wide variety of machine learning algorithms whose computation can be accelerated using our method. (4) We also show how our solution can accurately work with a variable selection method without recomputing everything. In this paper, we use Python as our choice of data science language, combined with C++ to escape the limitations of Python and provide an efficient and scalable solution. Experimental evaluation shows our solution is competitive with current state-of-the-art solutions both in a single machine and parallel platforms.

This is the organization of our paper. In Section 2, we give a brief overview of the mathematical definitions we used for this paper. Section 3 discusses our theoretical contribution where we present our approach. Section 4 presents experimental evaluation where we compare our method with current state-of-the-art libraries and tools. We discuss closely relation work in Section 5. Conclusions and directions for future work are discussed in Section 6.

II. Definitions

A. Mathematical Definitions

We represent our input data set as matrix X, where $X = \{x_1, ..., x_n\}$, a set of n column vectors. We represent Θ as a statistical or a machine learning (ML) model (e.g. Linear Regression, Naïve Bayes, and so on), and all the models take a $d \times n$ matrix X as input. 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 classification (most commonly binary), making X a $(d+1) \times n$ matrix and we call it $\mathbf{X}$. And, we define Z as $\mathbf{X}$ augmented with an extra rows of n 1s, making Z a $(d+2) \times n$ matrix.

B. Matrix Multiplication in Data Science Languages

Matrices are represented in different ways across popular data science languages. As for Gramian matrix multiplication (a matrix multiplied by itself), there is no dedicated operator in any language. The most traditional way to compute it in any language is to use nested loops which is not efficient and has high time complexity when the matrix size is large. In Python, NumPy library is widely used for matrix multiplication while R provides a built-in matrix multiplication operator ("%*%"). Also, JavaScript's math.js library provides 'math.multiply()' function to multiply two matrices. Though the methods discussed above are sufficient for regular size matrices, they must work in RAM and begin to fail when the matrix size is larger, or they do not fit in the main memory. In the next section, we discuss how our method of Gramian matrix multiplication can be efficiently incorporated with these languages beyond RAM limitation to accelerate the computation of ML models.

III. Theory and Algorithm

In this section, we give an overview of computing the data summaries and propose how we can compute a wide range of machine learning models exploiting our data summaries. Later, we discuss how our solution works in more technical details and present the parallel processing aspects of our solution.

A. Matrix Multiplication to Compute Data Summaries

1) Compute Data Summaries: Here, we first review the data summaries (single and multiple) and present an efficient way to store the multiple data summaries. We represent the data summaries as a matrix, commonly known as summarization matrix. Our summarization matrix, named Gamma (Γ) [3], [5] is computed based on sufficient statistics. From [5], we define n, L, Q as: $n = |X|$, $L = \sum_{i=1}^{n} x_i$, and $Q = XX^T = \sum_{i=1}^{n} x_i \cdot x_i^T$, where X is the input data set, a $d \times n$ matrix, n counts the total number of points in the data set, L is the linear sum of x_i, and Q is the sum of vector outer products of x_i. Now, the Gamma matrix (Γ) is defined below in Eq. 1. The size of this matrix is $(d+2) \times (d+2)$.

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

Now, we define multiple data summaries (k-Gamma, Γ^k) [5], where the major difference between the two forms of summarization matrix is that we do not require parameters off the diagonal in Γ^k. Here, we need only a few parameters out of the whole Γ, namely, $n, L, L^T, diag(Q)$. The multiple data summaries or Γ^k is presented in Eq. 2.

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

We use Gramian matrix multiplication to compute both the aforementioned summarization matrices efficiently. Notice that 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 [5], our data summaries 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: $\Gamma = ZZ^T = \sum_{i=1}^{n} z_i \cdot z_i^T$. Here, we evaluate the later one (more discussion on Section III-D).

2) Storage Mechanism of Multiple Data Summaries: Multiple data summaries are mostly needed for a classification or clustering model [5]. There are several ways to store the multiple data summaries on memory or disk. The naive approach will be storing one matrix for each class/cluster (k). However, as given in Eq. 2, Q is diagonal, and off the diagonal elements are zero. So, we have to store a lot of unnecessary zeroes for this approach. Another approach to store the multiple data summaries is to store everything in a single matrix [2]. That is, for each class/cluster, we store the L and Q in two separate columns, and the matrix size will be $d \times 2k$. For example, for $k = 2$, the matrix columns will be $\{L_1, Q_1, L_2, Q_2\}$. Though it saves space compared to the previous approach, it becomes less user-friendly when d is high and it can not represent multi-dimensional data.

Here, we propose an efficient way to store the multiple data summaries. Our novel approach uses two tensors, one tensor each for L and Q. Tensors can be represented as a multi-dimensional array and they are dynamic. That is, tensors will transform when interacting with other mathematical entities (e.g. matrices, vectors) which can be leveraged to compute the ML models efficiently. In our case, each tensor will have the L and Q for each class respectively and will be a size of $d \times k$. This way, we can easily access the respective L and Q even if the d is high. As our tensors are two-dimensional, we represent (store) them with matrices. For example, for $k = 2$, we will have $L = [[L_1], [L_2]]$, and $Q = [[Q_1], [Q_2]]$.

B. ML Model Computation from Data Summaries:

Now, we give a brief explanation of computing several machine learning models using our data summaries. However, not all the ML models can be benefited from our solution like HMM, time series algorithms and so on. Here, we explore a variety of ML models beyond our previous work [5]. First, we compute the ML models on the full data set where each data set is represented as matrix X, of size $d \times n$.

a) Regression Analysis: Regression is one of the most common models in machine learning. One of the basic regression models, Linear regression, involves the use of a best-fit line. It can be computed using the least-squares estimation

technique. We can utilize our data summaries to compute the regression coefficients ($\hat{\beta}$) [5] as given in Eq. 3.

$$\hat{\beta} = (XX^T)^{-1}(XY^T) = Q^{-1}(XY^T) \quad (3)$$

However, if there is a high correlation between independent variables, a regularization technique is often introduced to reduce the complexity by adding a penalized estimation. Here, we also explore how Ridge regression can be computed using our data summaries. Eq. 4 shows how to compute the ridge coefficients ($\hat{\beta}_{ridge}$) utilizing our summarization matrix. A penalty term (λI) is added to compute the coefficients with linear regression. Here, $\lambda > 0$ is the penalty parameter and I is a $d \times d$ identity matrix. The value of λ varies for each data set and is often determined using the cross-validation technique which is beyond the scope of this paper.

$$\hat{\beta}_{ridge} = (XX^T + \lambda I)^{-1}(XY^T) = (Q + \lambda I)^{-1}(XY^T) \quad (4)$$

b) Principal Component Analysis (PCA): First, we compute the correlation matrix (ρ) as $\rho = UD^2U^T = (UD^2U^T)^T$ utilizing our data summaries [5]. We express ρ in terms of our summarization matrix as given in Eq. 5. Then we compute PCA from the ρ by solving SVD ($SVD(\rho)$) on it.

$$\rho_{ab} = (nQ_{ab} - L_aL_b)/(\sqrt{nQ_{aa} - L_a^2}\sqrt{nQ_{bb} - L_b^2}) \quad where\ a = 1,..,d\ and\ b = 1,..,d \quad (5)$$

c) Naïve Bayes (NB): As mentioned above, for classification/clustering models, we need the multiple data summaries or k-Gamma matrix. First, we compute n_G, L_G, Q_G for each class from our k-Gamma matrix [5]. The output of the model is: mean (C), variance (R), and the prior probabilities (W), which is computed for each class in Eq. 6.

$$W_G = \frac{n_G}{n};\ C_G = \frac{L_G}{n_G};\ R_G = \frac{Q_G}{n_G} - diag\frac{[L_GL_G^T]}{n_G^2} \quad (6)$$

d) Linear Discriminant Analysis (LDA): LDA can be used as a linear classifier. Here, we focus on LDA for two classes. Our assumption is that each attribute in data follows a normal distribution and has the same variance. Similar to Naïve Bayes (NB), we use k-Gamma matrix to compute n_G, L_G, Q_G for each class. Now, we compute mean (C) as given in Eq. 7. As for variance (R), it is computed across all classes, unlike NB. Though, in theory, the denominator part subtracts k to get an unbiased estimator, we omit it here as $n >> k$ in our case - the difference becomes negligible.

$$C_G = \frac{L_G}{n_G};\ R = \sum_{G=1}^{k}(\frac{Q_G}{\sum n_G} - diag\frac{[L_GL_G^T]}{(\sum n_G)^2}) \quad (7)$$

C. ML models on Projection of Data Summaries from Variable Selection

Variable selection model is useful when data set dimension is high. The data set may contain variables that are redundant, or they may have low predictive accuracy. The search for the best subsets of explanatory variables that are good predictors of Y is called variable selection. In this work, we represent the set of selected variables as a d-dimensional vector $p \in \{0,1\}^d$, such that $p_i = 1$ if the variable is selected and $p_i = 0$ otherwise. We present p as an index to project matrices on selected variables such as Γ_p or β_p. Below we discuss variable selection for regression analysis.

a) Regression Analysis: The input is a $d \times n$ matrix X as mentioned before. For selecting the variables, we compute the correlations between each dimension and Y. To adapt this with our data summaries, we consider Y as another dimension and compute our summarization matrix (Γ). Now to get the correlation between each X_i and Y, one way to do it is by computing the full correlation matrix using Eq. 5, and extracting the relevant row from there. Here, we reduce the complexity (from $O(d^2)$ to $O(d)$) by computing only the correlations with Y as given in Eq. 8.

$$\rho_{aY} = \frac{nQ_{aY} - L_aL_Y}{\sqrt{nQ_{aa} - L_a^2}\sqrt{nQ_{YY} - L_Y^2}} \quad where\ a = 1,..,d \quad (8)$$

Now, we sort these correlations based on their absolute value and select the top values based on our needs. These values (p) tell us which variables are the good predictors of Y. Based on these values, we get our projected summarization matrix Γ_p from the original Γ. We emphasize that we do not have to recompute the full summarization matrix, rather a projection operation on the original Γ matrix gives us the projected Γ_p with fewer dimensions. Finally, utilizing this Γ_p, we can compute the regression coefficients for linear ($\hat{\beta}_p$) and ridge ($\hat{\beta}_{ridge_p}$) regression from Eq. 3 and Eq. 4 respectively. Later, we show in Section 4 that our variable selection method is accurate, and has marginal accuracy loss.

$$p(x_{ih}|G) = \frac{1}{\sqrt{2\pi\sigma_{Gh}^2}}exp[-\frac{(x_{ih} - \mu_{Gh})^2}{2\sigma_{Gh}^2}] \quad (9)$$

D. Integration with a Data Science Language

Here, we discuss how our approach can be integrated with a data science language. We use Python as our choice of the data science language. Python is widely popular among data scientists, mostly due to its huge library support, intuitive syntax, and ease of use. We structure our computation into two main phases:

1) Phase 1: Computation of the data summaries, and
2) Phase 2: Computation of the ML models utilizing the data summaries.

1) Processing Mechanism: The processing mechanism in a single machine can be shown in Algorithm 1. We use Python combined with C++ to compute the data summaries and the ML models. For Phase 1, as mentioned above, we are computing the data summaries using matrix multiplication. The main computation is given in Eq. 10. That is, we are computing the matrix multiplication using the sum of vector outer products.

$$\Gamma = \Gamma + z_i \cdot z_i^T \quad (10)$$

In our solution, Phase 1 must work in C++ (or C). The sum of vector outer products must be computed block by block in C++, not in the host language. Computing $z_i \cdot z_i^T$ in a loop in any other analytic language is slow, usually one-row-at-a-time. Computing $Z \cdot Z^T$ with traditional matrix multiplication is slow due to Z^T materialization, even in RAM. Initially, our algorithm reads the data set X into a fixed size of blocks (Algo. 1, line 1). We assume there are total of m blocks ($X = X_1, X_2, ...X_m$) and $|m| << n$. After each block is read, our algorithm performs the matrix multiplication using Eq. 10 to compute the partial summarization matrix for that block and add it to the previously computed summarization matrix (Algo. 1, line 3-4). When all the blocks are read, we get the final summarization matrix. Reading the data set by blocks can be beneficial especially when the data set size is larger than the main memory. As block size (usually $\log(n)$ or $\sqrt{n}$) is much smaller than the total size, each block of data easily fits into the main memory. Due to the additive feature of our summarization matrix, partial results from each block are easily added with the previously computed portions. This way, we are escaping the main memory limitation and as our summarization matrix is $O(d^2)$, it easily fits in the RAM. Also, after each block is processed, the matrix is updated without any extra memory requirement. So, the time complexity of Phase 1 is $O(d^2 n)$, and the space complexity is $O(d^2)$ for Gamma and $O(kd)$ for k-Gamma matrix.

On the other hand, Phase 2 computes the machine learning models utilizing the data summaries computed in Phase 1 (Algo. 1, line 6). These model computations can be efficiently done in any host data science language which is Python in our case. We use the available standard functions offered by Python and its libraries (e.g. NumPy) to compute the ML models efficiently. It would be too difficult and error-prone to reprogram all the ML models. Instead, our solution requires just changing certain steps in each numerical method, rewriting their equations based on the data summaries as discussed in Section III-B. Computing ML models this way requires less amount of code, easily understandable, and have faster processing.

Algorithm 1 Processing Steps (Single machine).

1: Read X into m ($X_1, X_2, ..X_m$) blocks.
2: **for** each block $b = 1...m$ **do**
3: Compute data summary on the block b (Γ_b).
4: Add it to the global data summary, Γ =Γ + Γ_b
5: **end for**
6: Compute the ML model (Θ) based on Γ.

So, Phase 1 is exploiting C++ while Phase 2 uses the Python language "as is". The detail of how we combine Python with C++ is discussed later. In general, our solution applies to any language that can call C++ via library or API.

2) Calls from the Host Language: In general, the syntax to call our solution for any ML model from any language is:

```
theta = f (dataset)
```

Here, f is the function name based on the ML model, *dataset* can be any numerical data set with CSV file, and the final model will be stored in *theta*. As mentioned before, we are combining Python, our choice of data science language with C++. There are two ways we can do it for our solution: (1) we can expose the summarization matrix computed in C++ (Phase 1) to Python (for Phase 2), or (2) call the C++ functions from Python prompt to execute Phase 1. Here, we explore the second one. For this, we use the Python SWIG library to call the C++ functions from the Python interpreter. We call the respective C++ function (for Gamma or k-Gamma) based on the model to compute the summarization matrix, with the data set as the input parameter. Our C++ code load the data set in blocks and compute the summarization matrix as mentioned in Algorithm 1. This way, we can use plain C++ code to compute Phase 1 without any overhead introduced by the library. After the summarization matrix is computed, we return it to the host language. We convert the object returned by C++ to a NumPy array using SWIG library. However, we can also store the matrix as a CSV file and load it as a NumPy array in Python interpreter to reuse, which has almost no overhead due to Gamma being $O(d^2)$. Finally, based on the summarization matrix, the respective ML model is computed in Python as discussed in Section III-A. An example of Python source code is shown in Listing 1 to compute the LR model.

Listing 1: Python source code to compute the LR model.

```
import numpy

def LRmodel(X):
    #Call C++ Gamma function
    gm = GammaMatrix(X) #instantiate an object
    gamma = gm.execute()#returns summarization matrix

    #Compute the LR model from Gamma
    d = len(gamma) #dimensions
    Q = gamma[1:d-1,1:d-1] #[row,column]
    XYT = gamma[1:d-1,d-1]
    invQ = numpy.linalg.inv(Q)
    beta = numpy.matmul(invQ, XYT)

    return beta

#Call from Python interpreter
theta = LRmodel("X.csv")
```

E. Parallel Processing Aspects

It is understood that not all the models can be computed in parallel using our proposed solution, especially the iterative models. Our parallel solution is inspired by our 3-phase parallel architecture presented in [9]. The pseudocode of our parallel solution steps is presented in Algorithm 2. Here, we introduce a new phase (Phase 0) before computing Phase 1 presented in Sec III-D. For parallel processing, we assume the processing is done in N machines, where $d << n$ and $N << n$. Our data summaries computation part presented

TABLE I: Baseline 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

in Eq. 10 can be easily parallelized as we can compute this on partial data sets in each machine and then add them on the master node to get the final data summary. In phase 0, we split the data set $X(X_1, X_2, ...X_N)$ and transfer the partitioned data sets to the N processing machines (Algo. 2, Line 1). We use row-based partitioning mechanism as all d columns must be presented in each machine. Then, for phase 1, we compute the local data summary $(\Gamma_1, \Gamma_2, ..\Gamma_N)$ on each machine as mentioned in Section III-D1 based on the partial data set (Algo. 2, Line 2-4). So, the time complexity of this phase is $O(d^2 n/N)$. After this phase, each machine will have a local summarization matrix which we store as a file on the disk. Finally, in phase 2, we send all the local summarization matrices to the master node. We use sequential transfer technique to send all the partial data summaries which has a time complexity of $O(d^2 N)$. Then in the master node, we perform a single matrix addition to get the final summarization matrix, i.e. $\Gamma = \Gamma_1 + \Gamma_2 + .. + \Gamma_N$ (Algo. 2, Line 5). Based on this Γ, we can compute the ML models using Python as mentioned before (Algo. 2, Line 6). We emphasize that only computing Eq. 10 is parallelized as we can compute this on partial data sets on each node and then add them on the master node. And as mentioned previously, if the model is iterative, and needs to recompute the data summaries in each iteration, our proposed parallel solution will not work, and it is a scope for future work.

Algorithm 2 Parallel Processing Steps.

1: Split and transfer X into N $(I = 1..N)$ machines.
2: **for** each machine I in parallel **do**
3: Compute local data summaries on partitioned X_I.
4: **end for**
5: Transfer and combine all the partial data summaries on the master node
6: Compute the ML model (Θ).

IV. Experimental Evaluation

In this section, we evaluate our solution with some existing data sets, and compare the performance with Python and R in a single machine, as well as with Spark and a DBMS in parallel machines.

A. Experimental Setup

We used Pentium(R) Quadcore CPU machine running at 1.60 GHz, 8 GB RAM, 1TB storage running on Linux OS for our experiments. Data sets used for the experiments are summarized in Table I, obtained from the UCI machine learning repository. As the large data sets are hardly available to public, we sampled and replicated the existing data sets from Table I to get varying n (data set size) and d (dimensions) as mentioned in [5]. We programmed our solution combining Python with C++. For comparison purposes, we used the existing libraries and functions available in the languages. All the time performance experiments were performed 3 times and we report the average time here.

B. Accuracy with Variable Selection

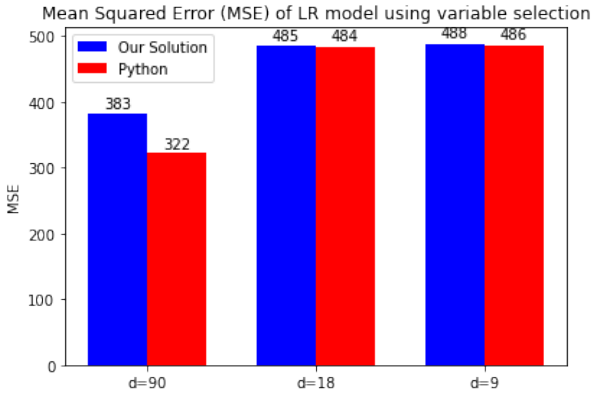

Fig. 1: Mean Squared Error (MSE) of LR model using variable selection with our solution and Python scikit-learn.

We have already shown in [5] that we can get more than 99% accurate model from "original" data sets using our summarization mechanism. Here, we present the accuracy evaluation with the variable selection method for linear regression. As mentioned in Section 3, we select the top variables based on our need and compute the ML model based on the projected summarization matrix. Figure 1 shows the mean squared error (MSE) of the linear regression model for different d and fixed n $(n = 515K)$ with our solution and Python. First, we use the original data set $(d = 90)$, and split it into train and test set in a $70\%-30\%$ fashion. For both cases, we train the model on the train data set and test the prediction using the test data set. We use the built-in routine available in Python scikit-learn library to train in Python. We compute the MSE based on the test data set using both methods and plot it in Figure 1. Similarly, we compute the MSE selecting 20% of the variables $(d = 18)$ and 10% of the variables $(d = 9)$ following the same procedure. In each case, we can see that although there is some difference for $d = 90$ using our solution and Python, the error gap is much smaller when we are selecting variables. It is sometimes hard to interpret the data having high d (like $d = 90$), but if we can reduce the d (like $d = 9$, or even lower) without compromising much accuracy as we did here, we can interpret the data in a much easier way.

C. Time Performance Comparison on a Single Machine

First, we compare the computation of the data summaries using our solution with Python. We want to justify that our approach with vectorized outer products in Python and C++ is much faster and more efficient than Python itself. For this,

TABLE II: Computing the summarization matrix using matrix multiplication: Our solution vs Python (Time in Seconds, M = Millions).

Matrix	n	d	Our solution	Python
Gamma	1M	9	17	217
Gamma	10M	9	164	2151
Gamma	10M	90	979	Stop
k-Gamma	1M	9	21	115
k-Gamma	10M	9	195	789

we implement our solution using Eq. 10 both in Python and C++ as mentioned in Section III-D. For Python, we follow the same steps with NumPy and Pandas library. We read the data set using Pandas library as a data frame and then convert it to NumPy array for further processing. We use the Pandas library to give Python a fair chance as reading CSV files using a plain file reader or CSV reader is much slower in Python. Table II shows the time to compute the Gamma and k-Gamma for varying n and d matrix using our solution and Python. We see that our solution utilizing C++ is much faster than Python itself and can handle large data sets. The main reason behind that is we are escaping the bottleneck by computing the vectorized outer product in C++ as loops are much slower.When the data set size is bigger, Python could not finish the computation in 1 hour and we put "Stop" in the table. Also, naturally, Python cannot maintain a large array or matrix in the main memory when its size is bigger than the RAM itself, which is not the case for our approach.

Now, we compare our solution with the built-in ML models in Python and R. We use a popular Python machine learning library, scikit-learn, to compute the ML models in Python. We read the data set as a DataFrame using Pandas library and convert it to NumPy array before feeding it to the model. Also, we use the default functions available in R to compute the ML models. Table III shows the time to compute the ML models using our solution in Python and C++, our previous solution in R [5], and in Python and R itself. We see that our solution with Python is almost similar to our previous solution in R. We put N/A for LDA as we did not explore the model before with R. For LR and PCA, our solution computes the data summaries in C++ and then computes the model in Python using NumPy library as mentioned in Sec III-D. For any n and d, the computation of the data summaries part is the same for all models. And, computing the actual models from the summarization matrix in Python is fast and is done in less than 1 second. So, the overall time to compute the models is almost constant regardless of the model and number of records, n. For Python scikit-learn library, it works better when d and n both are small. When d and n get bigger, especially d, it suffers from high dimensionality and our solution eventually performs better. On the other hand, R works much slower than any of the other methods mostly due to reading the data set and higher data volume. We put "Fail" when Python or R could not finish the computation in 20 minutes.

D. Comparison with Parallel Big Data Systems: Spark and parallel DBMS

For parallel processing, we compare our solution with Spark-MLlib library - Spark's scalable machine learning library to compute ML models, and Vertica - a parallel DBMS. We used the available functions in MLlib and programmed the models using Scala for Spark, and we used our previously computed version of Gamma with SQL and UDFs for parallel DBMS [2]. Table IV compares our solution with Spark-MLlib and Vertica in parallel $N = 8$ machines. Here, we are comparing with two representative models from each data summary: linear regression (for Γ matrix), and Naïve Bayes (for k-Gamma matrix). All the machines have the same configuration as mentioned in Sec IV-A. Here, we are taking data sets with varying $n(1M, 10M, 100M)$, and $d = 10$ to demonstrate how large data sets perform on both.

As the landscape of big data has changed in recent years, we assume data can be either in the disk (file system), in the cloud (ex: HDFS), or already partitioned among the processing machines. If the data is in the file system, we need to partition the data first and then compute the ML models as mentioned in Sec III-E. As for the cloud (HDFS), we need to export the data first to a local machine, and then transfer it to the processing machines. And, if the data is already partitioned, we can compute the ML models directly. From Table IV, the 'partition' column shows the time to split the data set and transfer it to the N processing machines. Following our previously proposed parallel architecture [9], we used the standard and fastest UNIX commands to perform this operation. The 'HDFS Export' column shows the time to export X to a local machine. And, 'Γ + Θ' column is the time to compute local data summaries in each machine, transfer them to the master node, get the final summarization matrix, and compute the ML models from it. To get the local summarization matrices from each machine in the master node, we use the Python 'subprocess' library which transfers the files efficiently. From Table IV, despite HDFS being faster to partition the data sets, the time to compute the ML models (Θ) is much slower in Spark. The reason behind that is, Spark trains LR model using Stochastic Gradient Descent which solves least square regression formulation and results in slower execution of the model. And for NB, Spark implements multinominal NB that suffers from having negative values in the data set, which is the case for our CreditCard data set here. On the other hand, our SQL solution with DBMS is slow in all cases mostly due to the parallel JOIN operation which is the bottleneck. There was no k-Gamma proposed for DBMSs, so we put 'N/A' for Naïve Bayes. However, it is understood that if there are hundreds of machines, Spark's performance will become better.

V. Related Work

Accelerating the computation of machine learning (ML) models for large-scale data sets does not always mean adding new hardware and GPUs. Many techniques have been proposed to optimize the ML algorithms, and some of them have

TABLE III: Time (in Seconds) to compute the ML models with our solution, Python, and R (One machine) (M=Millions).

Θ			Python		R	
(Data set)	n	d	Our solution	Python	Prev. solution	R
LR	1M	9	18	5	15	24
(Year-	10M	9	165	41	153	285
Prediction)	1M	90	87	88	105	630
	10M	90	980	Fail	1054	Fail
PCA	1M	9	18	5	15	21
(Year-	10M	9	165	38	153	205
Prediction)	1M	90	87	105	127	575
	10M	90	980	Fail	1054	Fail
NB	1M	9	23	9	26	29
(creditcard)	10M	9	196	62	243	281
LDA	1M	9	23	13	N/A	112
(creditcard)	10M	9	196	124	N/A	487

TABLE IV: Big data benchmark to compute the ML models (parallel) (Time = Seconds; $N = 8$ nodes; M=Millions).

Θ				Our solution	(N=8)	Spark	(N=8)	DBMS	(N=8)
(Data set)	n	d	Partition	HDFS Export	Γ + Θ	HDFS Partition	Θ	Partition	Γ
LR	1M	10	9	6	10	7	41	10	28
(Year-	10M	10	23	13	23	17	286	32	141
Prediction)	100M	10	317	96	201	161	1780	381	935
NB	1M	10	11	6	12	7	Fail	11	N/A
(credit-	10M	10	28	17	29	25	Fail	38	N/A
card)	100M	10	335	125	229	231	Fail	405	N/A

gained more attraction. One such technique, data summarization, has been proposed in many research [1], [8], [14], [17]. The basic idea is to create a summary of the original data set that is much smaller in size but preserves the original statistical properties. Despite being explored by the research community, computing the ML models using data summaries in Python has not been done before. Python is widely popular among data science practitioners for data analysis and machine learning [12], [13], [15]. The scikit-learn library [11] of Python integrates a wide range of state-of-the-art machine learning algorithms for medium-scale supervised and unsupervised problems. However, the library does not work in parallel and most models suffer from high dimensionality. Here, we compare our solution with scikit-learn and achieve competitive performance for both accuracy and speed. Also, James et. al [6] proposed two libraries (Dask and Numba) to optimize Python code for numerical processing by translating it to machine code. Dask is an open source library for parallel computing in Python and Numba translates subset of Python and NumPy code into fast machine code. Similar to these approaches, we are computing the summarization part in C++ to achieve higher speed. To integrate Python with C++, we use SWIG [4], which compared to other available libraries, is easier to port interface to C/C++ and both Python and C++ code can be kept clean. As mentioned above, our solution computes the data summaries part using Gramian matrix multiplication in C/C++ and the model computation part is done in Python. Optimizing the matrix multiplication has been proposed both in sequential [7] and parallel [10] manner. However, they mostly work on the main memory. Sebastian et. al [16] explored if serverless computing is a feasible and beneficial approach to big data processing using matrix multiplication as example. Unlike these techniques, our Gramian matrix multiplication technique can work beyond RAM limits and works both in a sequential and parallel manner.

Computing data summaries to accelerate the computation of ML models has been proposed before for the R language in both sequential [5] and parallel [3] manner. Similar to our approach here, R utilizes C++ with its Rcpp library to accelerate Phase 1, and Phase 2 is computed in R. Unlike R, we used plain C++ code here while we used Rcpp code before. We used to read data in blocks in R using R libraries and pass it to Rcpp. Here, we are loading data directly in C++ in blocks, minimizing the overheads introduced by the libraries. Moreover, we explored more ML models compared to the previous ones, like Ridge regression and LDA. We proposed variable selection method that can compute the ML models on a subset of the original data set. We presented experimental evaluation that proved we can get accurate model with variable selection without much compensation on accuracy. Also, we proposed an innovative and efficient way to store multiple data summaries using tensors.

VI. Conclusions

We presented an efficient way to compute a wide spectrum of machine learning models for large data sets in both single and parallel machines. Our solution leveraged a fast Gramian matrix multiplication to compute data summaries. Based on these data summaries, machine learning models are computed with high accuracy, high speed, and without main memory limitations. We showed that our solution can compute the ML models on the whole data set utilizing the full data summaries, as well as on the subset of the data set (from variable selection) utilizing the projected data summaries without much compensation on accuracy. We discussed in depth how high speed is achieved by computing the summarization part in C++, which is available to Python via an external library. Main memory

limitation is tackled by reading the data set in blocks and the additive feature of our data summaries. Also, to be competitive within the big data space, we presented the parallel version of our solution. As for experimental evaluation, we presented interesting experiments to evaluate the time performance. Our solution performed much faster to compute the data summaries than Python NumPy following the same algorithm. To compute the ML models, we had similar performance as our previous solution in R, and was competitive with Python scikit-learn library, especially for large data sets. Furthermore, our parallel approach outperformed Spark and was able to handle larger data sets.

Our research proves that we can get a good performance computing ML models in Python beyond RAM limits. However, as for future work, we want to explore more models like Logistic Regression and Support Vector Machine. We also want to compare our solution with Stochastic Gradient Descent, the fundamental optimization in machine learning nowadays. Moreover, we plan to explore best ways to compute data summaries on sparse matrices.

REFERENCES

[1] Ahmed, M.: Data summarization: a survey. Knowl. Inf. Syst. **58**(2), 249–273 (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 - 22nd International Conference, DaWaK 2020. vol. 12393, pp. 269–284 (2020)

[3] Al-Amin, S.T., Ordonez, C.: Efficient machine learning on data science languages with parallel data summarization. Data Knowledge Engineering **136**, 101930 (2021)

[15] Tavenard, R., Faouzi, J., Vandewiele, G., Divo, F., Androz, G., Holtz, C., Payne, M., Yurchak, R., Rußwurm, M., Kolar, K., et al.: Tslearn, a machine learning toolkit for time series data. J. Mach. Learn. Res. **21**(118), 1–6 (2020)

[4] Beazley, D.M.: SWIG: an easy to use tool for integrating scripting languages with C and C++. In: Fourth Annual USENIX Tcl/Tk Workshop (1996)

[5] Chebolu, S.U.S., Ordonez, C., Al-Amin, S.T.: Scalable machine learning in the R language using a summarization matrix. In: Database and Expert Systems Applications DEXA. pp. 247–262 (2019)

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

[7] Drevet, C.É., Islam, M.N., Schost, É.: Optimization techniques for small matrix multiplication. Theoretical Computer Science **412**, 2219–2236 (2011)

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

[9] Ordonez, C., Al-Amin, S.T., Zhou, X.: A simple low cost parallel architecture for big data analytics. In: IEEE International Conference on Big Data,. pp. 2827–2832 (2020)

[10] Parasrampuria, U., Misra, C., Bhattacharya, S.: An optimized distributed recursive matrix multiplication for arbitrary sized matrices. In: IEEE International Conference on Big Data,. pp. 5798–5800 (2020)

[11] Pedregosa, F., Varoquaux, G., et al.: Scikit-learn: Machine learning in python. the Journal of machine Learning research **12**, 2825–2830 (2011)

[12] Raschka, S., Mirjalili, V.: Python machine learning: Machine learning and deep learning with python. Scikit-Learn, and TensorFlow. Second edition ed (2017)

[13] Sarkar, D., Bali, R., Sharma, T.: Practical machine learning with python. A Problem-Solvers Guide To Building Real-World Intelligent Systems. Berkely: Apress (2018)

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

[16] Werner, S., Kuhlenkamp, J., Klems, M., Müller, J., Tai, S.: Serverless big data processing using matrix multiplication as example. In: IEEE International Conference on Big Data. pp. 358–365 (2018)

[17] Zhuang, H., Rahman, R., Hu, X., Guo, T., Hui, P., Aberer, K.: Data summarization with social contexts. In: Proceedings of the 25th ACM International Conference on Information and Knowledge Management,. pp. 397–406 (2016)