



Incremental and accurate computation of machine learning models with smart data summarization

Sikder Tahsin Al-Amin¹ · Carlos Ordonez¹

Received: 29 June 2021 / Revised: 3 October 2021 / Accepted: 22 November 2021 /

Published online: 19 January 2022

© The Author(s), under exclusive licence to Springer Science+Business Media, LLC, part of Springer Nature 2022

Abstract

Nowadays, data scientists prefer “easy” high-level languages like R and Python, which accomplish complex mathematical tasks with a few lines of code, but they present memory and speed limitations. Data summarization has been a fundamental technique in data mining that has promise with more demanding data science applications. Unfortunately, most summarization approaches require reading the entire data set before computing any machine learning (ML) model, the old-fashioned way. Also, it is hard to learn models if there is an addition or removal of data samples. Keeping these motivations in mind, we present incremental algorithms to smartly compute summarization matrix, previously used in parallel DBMSs, to compute ML models incrementally in data science languages. Compared to the previous approaches, our new smart algorithms interleave model computation periodically, as the data set is being summarized. A salient feature is scalability to large data sets, provided the summarization matrix fits in RAM, a reasonable assumption in most cases. We show our incremental approach is intelligent and works for a wide spectrum of ML models. Our experimental evaluation shows models get increasingly accurate, reaching total accuracy when the data set is fully scanned. On the other hand, we show our incremental algorithms are as fast as Python ML library, and much faster than R built-in routines.

Keywords Data mining · Incremental computation · Summarization · Machine learning

1 Introduction

Machine learning models in many applications usually need to be trained with a large amount of data. In many practical scenarios, the samples in the data set can change over time, due to the addition of new data samples (e.g., those that have been collected and labeled recently) and removal of existing data samples (e.g., those that are too noisy or incorrectly labeled). Recomputing the model from scratch whenever there is a change in the data set is time-consuming and not a “clever” way. Also, it is still computationally intensive

✉ Sikder Tahsin Al-Amin
stahsin.cse@gmail.com

¹ Department of Computer Science, University of Houston, Houston, TX 77204, USA

to recompute the model parameters on the entire data set when only a few samples in the data set change. Rather, it is more smart and efficient to update the model by including or excluding the influence of specific data samples, which is known as incremental and decremental learning. The goal is to adapt a pre-computed model to the changes in the data set, without recomputing the model on all the data, where the changes can include addition and deletion of data samples.

Nowadays, deep learning (LeCun et al., 2015) is widely used among machine learning and data science practitioners. Although using deep learning yields better accuracy in most cases, it is still computationally expensive than traditional algorithms and a kind of “black box” in nature. Moreover, they are most suited for images and text data and they cannot provide a statistical explanation of their accuracy. So, many data scientists are still using traditional machine learning algorithms and statistics for data analysis and data pre-processing (Al-Amin et al., 2020; David et al., 2020; Levatic et al., 2017; Osojnik et al., 2018; Totad et al., 2012). In fact, in most data science projects, data scientists iterate many times before a good model is computed as it is never accurate the first time. However, for an average data scientist with limited memory in one machine, large data sets become a problem (e.g. bigger than RAM). They do not have parallel cluster set up at home, and clouds are expensive. So, they have to rely on their single machine for large scale data analysis. Also, sometimes data scientists want to see “early results” instead of waiting until the completion of the full data set to see how their model is behaving, or they may need an approximate accurate model with fewer data points. Moreover, most ML algorithms require multiple pass to the data set or recomputing the model from scratch when new data points are added or removed with the original data set. With these motivations in mind, we propose an incremental and accurate computation of ML models that can handle large data sets in a single machine, and does not need to recompute the models from scratch with the addition of new data points.

Our contributions are the following: (1) We present intelligent incremental algorithms that will compute the ML models efficiently with smart data summarization. (2) We provide properties and proof sketch of our solution, giving it a solid theoretical background. (3) We improve and optimize the technique to read the data set from disks in blocks and discuss the optimal block size. (4) We compare against R built-in routines and Python scikit-learn incremental library, the most popular machine learning library nowadays. We focus on the incremental computation of four popular and fundamental machine learning models: Linear Regression (LR) (Hastie et al., 2001), Principal Component Analysis (PCA) (Hastie et al., 2001), Naïve Bayes (NB) (Hastie et al., 2001) and K-means (KM) (Hastie et al., 2001). This article is a significant step forward and is fundamentally different from our previous work where the summarization and computation of machine learning models utilizing it was introduced (Chebolu et al., 2019). The models were solved by computing the summarization matrix first (full pass) while we compute the models incrementally here. Also, we can solve all the models with one-pass to the data set which was not possible for some models before (e.g. K-means). Experimental results in Section 4 prove that our incremental computation can handle more data than Chebolu et al. (2019), performs better than R, and is competitive with Python scikit-learn incremental library. Also, we can get an accurate model with very minimal relative error without reading the full data set. In summary, to our knowledge, there does not exist an approach which (1) supports both incremental and decremental summarization matrix, and (2) updates model parameters based on the summarization matrix. Here, we propose such an approach satisfying both (1) and (2) in this paper.

This is an outline of the rest of this article. Section 2 is a reference section that introduces the definitions used throughout the paper. Section 3 presents our theoretical research contributions where we present our incremental algorithm. Section 4 presents an experimental

Table 1 Basic symbols and their description

Symbol	Description
X	Data set
d	Number of records/rows in X
n	Number of attributes/columns in X
Y	Dependent Variable
X	Augmented X with Y
Z	Augmented X with 1s and Y
Γ	Gamma Summarization Matrix
Γ^k	k -Gamma Summarization Matrix
Θ	Machine learning model
b	Blocks to read data
m	Total number of blocks

evaluation comparing our solution to state-of-the-art data science systems including the R package, and Python. We discuss closely related work in Section 5. Conclusions and directions for future work are discussed in Section 6.

2 Definitions

The input matrix is defined as X which is a set of n column vectors. And Θ is used to represent a statistical or machine learning model in a general manner. All the models take a $d \times n$ matrix X as input. 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} . We augment \mathbf{X} with an extra row of n 1s and call that as matrix Z , $(d + 2) \times n$ dimension (Table 1).

2.1 Example

An example of Z matrix is shown in Fig. 1. We show this example to make the Z matrix clear to the reader which is at heart of our summarization matrix computation. Here, “ $d1, \dots, d4$ ” are the input dimensions (d), and “ Y ” is the output variable Y , “ $n1, \dots, n8$ ” are the column vectors, and 1 is row of n 1s, making Z a $(d + 2) \times n$ matrix as mentioned above. We use this Z matrix to show an example of our algorithms in Section 3.

	[n1]	[n2]	[n3]	[n4]	[n5]	[n6]	[n7]	[n8]
[1]	1	1	1	1	1	1	1	1
[d1]	1	2	3	2	9	-2	7	6
[d2]	2	-1	1	1	5	9	-9	9
[d3]	-5	-4	0	5	9	-6	3	9
[d4]	3	6	7	1	7	8	-4	-2
[Y]	0	0	0	0	1	1	1	1

Fig. 1 Example of Matrix Z (in R)

3 Theory and algorithm

3.1 Summarization matrix and ML model computation

Here, we first review the Gamma summarization matrix (Γ) (Chebolu et al., 2019; Ordonez et al., 2016) and computation of several ML models (Θ) exploiting Γ . The main algorithm had two phases where after computing Phase 1, Phase 2 is computed.

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

3.1.1 Phase 1

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 Ordonez et al. (2016), the Gamma (Γ) is defined below in (2).

$$n = |X|; L = \sum_{i=1}^n x_i; Q = XX^T = \sum_{i=1}^n x_i \cdot x_i^T. \tag{1}$$

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

As mentioned in Section 2, X is defined as a $d \times n$ matrix, and Z is defined as a $(d+2) \times n$ matrix with extra row of n 1s and output variable Y . From Ordonez et al. (2016), we can easily understand that the 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., $\sum_i z_i \cdot z_i^T$. Here, we evaluate the later one. So, in short, the Gamma computation can be defined as:

$$\Gamma = ZZ^T = \sum_{i=1}^n z_i \cdot z_i^T \tag{3}$$

Now, from Chebolu et al. (2019), we need k -Gamma (Γ^k) for classification/clustering models which is given in (4) (k =number of classes/clusters). Here, we need only a few parameters, $n, L, L^T, \text{diag}(Q)$. Both L and $\text{diag}(Q)$ can be represented as a single vector and we do not need to store Q as a matrix. Hence, the k -Gamma matrix can be represented as a single matrix of size $d \times 2k$, where each Gamma is represented in two columns (L and Q). We still need to store the value of n in a row, which makes the k -Gamma $(d + 1) \times 2k$. Hence, we are using minimal memory to store k -Gamma even if the value of k is very large. So, the storage of optimized k -Gamma matrix is shown in (4).

$$\Gamma^k = \begin{bmatrix} n_1 & 0 & \dots & n_k & 0 \\ L_{11} & Q_{11_1} & \dots & L_{1_k} & Q_{11_k} \\ \dots & \dots & \dots & \dots & \dots \\ L_{d_1} & Q_{dd_1} & \dots & L_{d_k} & Q_{dd_k} \end{bmatrix} \tag{4}$$

3.1.2 Phase 2

Both Γ and Γ^k provide summarization for a different set of machine learning models (Θ). We briefly discuss how to compute each model (Θ) below. The details of the model computation are discussed in Chebolu et al. (2019).

LR We can get the column vector of regression coefficients ($\hat{\beta}$), from the above mentioned Γ with:

$$\hat{\beta} = Q^{-1}(XY^T) \tag{5}$$

Equation (5) will yield a $d \times 1$ matrix, the regression coefficients. To get the intercept along with it, we need to modify (5) as: $\hat{\beta} = \tilde{Q}^{-1}(\mathbf{X}Y^T)$, where \tilde{Q} is a $(d + 1) \times (d + 1)$ matrix.

PCA PCA can be computed on the covariance matrix (V), or the correlation matrix (ρ). We compute ρ , the correlation matrix as $\rho = UD^2U^T = (UD^2U^T)^T$. We can also compute the covariance matrix as $V = Q/n - LL^T/n^2$. Then we compute PCA from the ρ by solving the Singular Value Decomposition on it ($svd(\rho)$). We express ρ in terms of sufficient statistics in (6), where a, b are indexes/subscripts (e.g. the term Q_{ab} represents a particular entry like $Q[a, b]$).

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

NB Here, we need the k -Gamma matrix. We assume, there are k number of classes (where $g = 1, \dots, k$), and for each class (g), we compute n_g, L_g, Q_g as discussed in Phase 1. The output of NB model is three parameters: mean (C), variance (R), and the prior probabilities (W). We can compute these parameters from the Γ^k matrix for each class label ($g = 1, \dots, k$) with the following statistical relations.

$$W_g = \frac{n_g}{n}; C_g = \frac{L_g}{n_g}; R_g = \frac{Q_g}{n_g} - \text{diag} \frac{[L_gL_g^T]}{n_g^2} \tag{7}$$

KM We assume, there are total k number of clusters and we denote them by $j = 1, \dots, k$. Similar to NB, for each cluster (j), we first compute n_j, L_j, Q_j from the k -Gamma matrix (notice that, we used g to denote the classes in NB, and j as the clusters in KM). Based on these, we compute the model parameters: mean (C), variance (R), and prior (W) for each cluster (j) as presented in (8).

$$C_j = \frac{L_j}{n_j}; R_j = \frac{Q_j}{n_j} - \text{diag} \frac{[L_jL_j^T]}{n_j^2}; W_j = \frac{n_j}{n} \tag{8}$$

Then, the algorithm iterates executing two steps starting from random initialization until cluster centroids become stable. Step 1 determines the closest cluster for each point using Euclidean distance ($d(x_i, C_j) = (x_i - C_j)^T(x_i - C_j)$) and adds the point to it. And Step 2 updates all the centroids C_j by computing the mean vector of points belonging to cluster j . The K-means algorithm stops when centroids change by a marginal fraction in consecutive iterations which is measured by the quantization error (Chebolu et al., 2019).

3.2 Our smart incremental algorithm to compute ML models

Here, we present our main contribution. We introduce a general, smart algorithm to efficiently compute the two phases together that were introduced in Section 3.1. The block diagram of our incremental computation of the ML models using the summarization matrix can be shown in Fig. 2. The mathematical symbols are discussed in Section 2. We explain in technical detail how Γ is exploited in intermediate computations to compute the models (Θ) incrementally. We present two algorithms. Algorithm 1 shows incremental computation of models (e.g. LR, PCA) using Gamma matrix and Algorithm 2 shows incremental

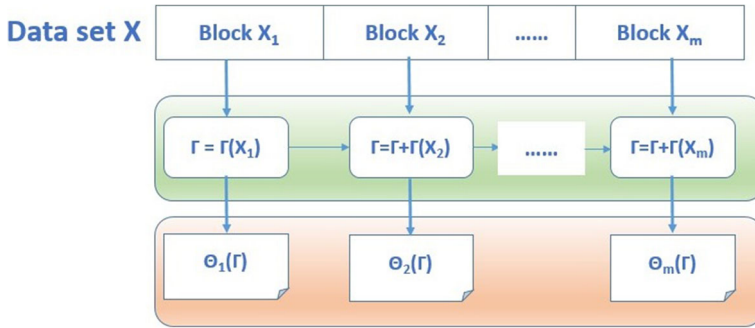


Fig. 2 Incremental Computation of ML models

computation of models (e.g. NB, KM) using k -Gamma matrix. First, we discuss the properties of our algorithms.

Property 1 Θ must be recomputed at each iteration (i). For iteration i , $\Theta(\Gamma_i) \neq \Theta(\Gamma_{i-1}) + \Theta(\Gamma_\Delta)$.

Property 2 Iterative algorithms (e.g. KM, HMM) only work on Γ_k and require X to compute Θ at each iteration.

Property 3 One-pass algorithms (e.g. LR, PCA, NB) work on both Γ and Γ_k , and does not require X to compute Θ . So, $\Theta(\Gamma_i) = \Theta(\Gamma_{i-1} + \Gamma_\Delta)$

Property 4 Matrix Γ or Γ^k can be decremental too.

Property 1 is fundamental to compute models (Θ) at each iteration (i). Our model computation is not distributive. That means, at iteration i , we can not reuse the model computed at the previous iteration ($\Theta(\Gamma_{i-1})$) to compute the model at the current iteration ($\Theta(\Gamma_i)$). It is not possible to converge Γ_{i-1} and Γ_Δ separately and then add them to get the correct model. For example, in the case of LR, $\beta_i \neq \beta_{i-1} + \beta_\Delta$. Similarly, for means in NB and KM, $\mu_i \neq \frac{\mu_{i-1} + \mu_\Delta}{2}$. So, Θ must be recomputed at each iteration based on Γ or Γ^k .

Property 2 states that iterative algorithms (e.g. K-means, Hidden Markov Model) only work on Γ^k . And they also require to read X at each iteration to compute the model Θ . We need to read the entire X at each iteration along with the Γ^k to increase the accuracy of the model. Otherwise, the accuracy of Θ may go down. So, only computing Γ^k is not enough to build a good model.

Property 3 is relevant to one-pass algorithms which work on both Γ and Γ^k . And we do not need to read X at each iteration. Popular one-pass algorithms presented here such as LR, PCA, and NB can achieve up to 99% accuracy (presented in Section 4) based on only Gamma assuming $\Theta(\Gamma)$ is 99% accurate.

We explain Property 4 in more technical details which states that our summarization matrix (Γ or Γ_k) is decremental too. Instead of adding new records to the data set, there can be cases where records are deleted from the data set based on some conditions or filters. In that case, we update the model without recomputing the whole data set. That is, we perform decremental operation on Gamma to compute or update the models. The main steps are the following which will be the same for k -Gamma too. (1) Read the deleted records (X_{del}).

(2) Compute partial Gamma (Γ_{Δ}) on the deleted records ($\Gamma_{\Delta} = \text{gamma}(X_{del})$). (3) Perform $\Gamma = \Gamma - \Gamma_{\Delta}$. (4) Compute the model, $\Theta = \text{compute}(\Gamma)$. Hence, we can update the models without reading the whole data set. However, instead of adding or removing rows from the data set, there can be addition or deletion of columns from the data set. Whether we have to recompute the model by reading the whole data set or Gamma can perform incrementally or decrementally, is an aspect for future research.

3.2.1 Incremental algorithm

Now, we explain the algorithm in details. From Algorithms 1 and 2, the input data set X is read into m blocks ($m \ll n$) of the same size ($|m| > 1$), $X = \{X_1, X_2, \dots, X_m\}$. Later we discuss how to determine the block size in detail. Processing data one block at a time has many benefits. It is the key to being able to scale the computations without increasing memory requirements. And we can handle data sets that are larger than the main memory. Now, we read each block (X_b) into the main memory and compute Gamma (Γ_{Δ} or Γ_{Δ}^k) for that block. However, for KM, there is one extra step before computing Gamma to get the nearest cluster j ($j = 1, \dots, k$) for each data point in the block X_b . Computing Gamma or k -Gamma matrix is discussed in Section 3.1. This partial Gamma (Γ_{Δ} or Γ_{Δ}^k) is added to the Gamma computed up to the previous block (X_{b-1}) and we get the Gamma (Γ or Γ^k) for up to b^{th} block. Finally, we compute the models (Θ) based on this Gamma (Γ or Γ^k). We iterate phase 1 and phase 2 together (inside “While” loop) until there are no more blocks left. So, the main difference between Algorithm 1 and 2 is that we are using one full Gamma (2) for LR and PCA in Algorithm 1 while we are using k -Gamma (4) for NB and KM in Algorithm 2.

Algorithm 1 Incremental algorithm to compute ML models with Gamma.

Data: $X = \{x_1, x_2, \dots, x_n\}$

Result: Θ

```

1 Read  $X$  into  $m$  blocks as  $X_1, X_2, \dots, X_m$ ;
2 while  $\text{next}(X_b)$  do
3   | read( $X_b$ ) ;
4   |  $\Gamma_{\Delta} \leftarrow \text{gamma}(X_b)$  //phase 1;
5   |  $\Gamma \leftarrow \Gamma + \Gamma_{\Delta}$  ;
6   |  $\Theta \leftarrow \text{compute}(\Gamma)$  //phase 2 ;
7 end
```

Algorithm 2 Incremental algorithm to compute ML models with k -Gamma.

Data: $X = \{x_1, x_2, \dots, x_n\}, k$

Result: Θ

```

1 Read  $X$  into  $m$  blocks as  $X_1, X_2, \dots, X_m$ ;
2 while  $\text{next}(X_b)$  do
3   | read( $X_b$ ) ;
4   |  $\text{getNearestClusters}()$  //only for KM ;
5   |  $\Gamma_{\Delta}^k \leftarrow k\text{-gamma}(X_b, k)$  //phase 1 ;
6   |  $\Gamma^k \leftarrow \Gamma^k + \Gamma_{\Delta}^k$  ;
7   |  $\Theta \leftarrow \text{compute}(\Gamma^k)$  //phase 2 ;
8 end
```

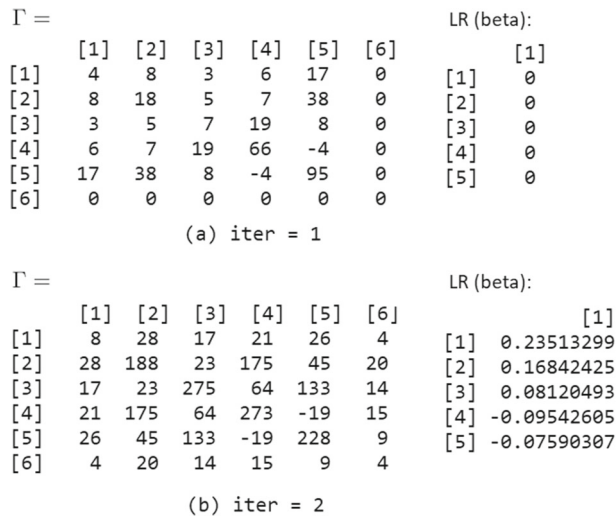


Fig. 3 Example of our Incremental Gamma algorithm with LR model (in R)

To illustrate an example, Fig. 3 shows the steps of our Algorithm 1 for the input data set in Fig. 1. We assume the block size is 4 to show this example, so there will be 2 iterations of our algorithm. For the LR model, we show the state of Γ and model parameter β (regression coefficients) after each iteration. Here, from matrix Γ , entry [1,1] is the value of n . Entry [2,1] to [5,1] is the value of L . And Q is a $d \times d$ matrix which ranges from [2,2] to [5,5]. Based on this Γ , we get the value of regression coefficients (β) using (5).

3.3 Parallel processing aspects

The summarization matrix proposed in Section 3.1 is fully parallel and it has been implemented for DBMSs before (Ordóñez et al., 2019). For our incremental algorithm, let us assume we want to compute our solution in N nodes (number of machines). The steps are shown in Algorithm 3. Though we present this algorithm for Gamma (2), it will be similar for k -Gamma (4). Phase 1 can be easily parallelized as we can compute the summaries for partial data sets in each machine. This is similar to our previously presented Algorithms 1 and 2. However, Phase 2 is difficult and will be significantly different. Computing Θ incrementally and in parallel requires a barrier. To compute the model (Θ), we need to send the Γ_I s or Γ_I^k s (where Γ_I = partial summarization matrix in machine I) from N machines to one central node because parallelizing Θ is difficult and assuming $d \ll n$, it is best to it on one node. Then we need to update the Θ and maybe broadcast the Θ depending on the ML model. In this way, the accuracy of LR and NB will be good because they do not need the previous Θ . For PCA, it is dependent on small principal components and KM requires redistributing points or iterations over the full data set to converge. So, the accuracy will not be always good for PCA and KM. Overall, computing the ML models incrementally in parallel is a deep topic and beyond the scope of this paper. We leave this for future work.

Algorithm 3 Parallel Incremental Algorithm to compute ML models.

Data: $X = \{x_1, x_2, \dots, x_n\}$
Result: Θ
Partition X into $I = 1, \dots, N$ machines ;
//Phase 1: parallel
for $I = 1, \dots, N$ machine (in parallel) **do**
 Read partitioned X_I into m blocks ;
 while $next(X_{I_b})$ **do**
 read(X_{I_b}) ;
 $\Gamma_{\Delta} \leftarrow \text{gamma}(X_{I_b})$;
 $\Gamma_I \leftarrow \Gamma_I + \Gamma_{\Delta}$;
 send_to_a_central_machine(Γ_I).
 end
end
//Phase 2: In central machine
Collect Γ_I s from N machines and compute $\Gamma = \Gamma_1 + \dots + \Gamma_N$;
 $\Theta = \text{compute}(\Gamma)$;
Redistribute Θ to N machines if necessary.

3.4 Error analysis of model approximation

In this section, we discuss how error goes down as we read more data points and how to determine the block size. First, we present the following two propositions.

Proposition 1 *According to the central limit theorem in statistics, sampling distribution of the sample means approaches a normal distribution as the sample size gets larger (Rumsey, 2011). In other words, for a sufficiently large sample size (blocks), the closer distribution of the sample means will be to a normal distribution. We assume that the standard error of the sample mean is denoted by \hat{s} (Rumsey, 2011), defined as, $\hat{s} = \frac{\sigma}{\sqrt{n}}$; where n is the sample size (blocks) and σ is the standard deviation.*

Proposition 2 *When estimating a global mean/variance by a sample mean/variance, the larger the sample size (blocks) - the greater the likelihood for possible mean values to cluster closely around the global mean, indicating the sampling error tends to smaller.*

Proof Sketch For linear regression, we assume the error is Gaussian. For simplicity, we assume LR can be viewed as $Y = \beta_0 + \beta_1 X + \epsilon$. We have already stated that in general, the mean of a block will provide a good estimate of the overall mean. In the same way, the unknown coefficients β_0 and β_1 in LR define the overall regression line. We seek to obtain coefficients estimates $\hat{\beta}_0$ and $\hat{\beta}_1$ such that the linear model fits the data well as they are similar to true values β_0 and β_1 . To compute the standard error associated with $\hat{\beta}_0$ and $\hat{\beta}_1$, we use following formulas from James et al. (2013). These equations tell us how this deviation shrinks with n , the more blocks we have, the smaller the standard error.

$$\hat{s}(\hat{\beta}_0)^2 = \sigma^2 \left[\frac{1}{n} + \frac{\bar{x}^2}{\sum_{i=1}^n (x_i - \bar{x})^2} \right], \hat{s}(\hat{\beta}_1)^2 = \frac{\sigma^2}{\sum_{i=1}^n (x_i - \bar{x})^2} \quad (9)$$

However, one extreme case is that the LR model will not work if the variables are correlated. That is, if it is a straight line, the matrix can not be inverted. We need some variability in the data.

The PCA finds top q (conventional symbol is top k , but it will conflict with our k = number of classes/clusters) principal components from all the dimensions. These top components capture 90% variance of the data set (Hastie et al., 2001). We assume error goes down for these top q components as we read more data points. It is impossible that all the dimensions show this behavior separately. We know from Section 3.1 that we need to compute covariance or correlation matrix from Γ first to compute PCA. From Proposition 2 presented above, it is clear that error will go down for covariance and correlation matrix as we read more data points. However, for both LR and PCA, dimensions that are 100% correlated would induce noise and should be eliminated. Total correlation is bad because it uncovers redundant data. Also, constant dimensions have the same value for all the points and they should be eliminated too.

The Proposition 2 mentioned above tells us that sufficiently larger blocks tend to produce more accurate results. Hence, for random blocks, the mean and variance will be the same as the entire data set for larger block sizes. This is totally applicable to Naïve Bayes and K-means as we are computing mean (C) and variance (R) of these two models given in (7) and (8). So, as we read more blocks of n , the standard error goes down. However, one extreme case of KM is that it will not work for very large k (number of clusters) as it uses distance to determine the closest cluster for each point.

3.4.1 Determining block size

We have discussed that for sufficiently large blocks, the error goes down as we read more data points. Now the question is how large does the block size has to be so that we get a good accuracy, and minimize error. That is, how many data points are needed to re-estimate the ML model (Θ). For example, only 1 point is too small, and total n (number of records) points will have 100% accuracy but it is not incremental. Estimation of mean and variance can be explained using the central limit theorem but it does not give a proper bound. So, we need a sharper bound like Chernoff bound. We define block size as n_b where $1 \leq n_b \leq n$, and accuracy α . For confidence $1 - \delta$, utilizing Chernoff bounds, we get:

$$\begin{aligned} \delta \geq 2 \exp\left(-\frac{\alpha^2}{2 + \alpha} \cdot n_b\right) &\Leftrightarrow \exp\left(\frac{\alpha^2}{2 + \alpha} \cdot n_b\right) \geq \frac{2}{\delta} \\ &\Leftrightarrow n_b \geq \frac{2 + \alpha}{\alpha^2} \ln \frac{2}{\delta} \end{aligned} \quad (10)$$

The block size should at least satisfies the above-mentioned equation for error going down (typical values of δ are 90%, 95%, and 99%). It is clear from the above equation that getting high accuracy is costly but getting high confidence is comparatively easy because of “ln”. For example, if we want $\alpha = 0.05$ with 95% confidence, we need to set $n_b \geq 3020$. Similarly, for $\alpha = 0.01$ with 99% confidence, we need to set $n_b \geq 106330$. However, having $\alpha = 0.01$ with 99% confidence is very hard as we need very large sample size. So, we define $\alpha = 0.01$ with 99% confidence as the best case scenario and $\alpha = 0.05$ with 95% confidence as the worst case scenario. Also, in this way, the sample size does not depend on the total data points.

3.5 Integration with data science languages

We integrate our solution into a data science language. While our solution applies to any language which provides an API call to C or C++ code (e.g. Python, R), we will discuss our solution based on R here. The only dependency is a library to read the data set in blocks. Data science languages like Python or R are most popular among data analysts nowadays. The reason lies behind SQL queries being slow, UDFs are not portable among DBMSs, Spark is not easy to debug, and Java is slower than C++.

For our algorithm, Phase 1 must work in C++ (or C), and Phase 2 works in the data science language. As Phase 1 does the “heavy” processing, the sum of vector outer products ($z_i * z_i^T$) can be computed block by block efficiently in C++. Computing this operation using traditional loops in R or any other data science language is slow, usually one row at a time. However, R has a dedicated matrix multiplication operator ($\% * \%$) that is commonly used by R analysts, is still very slow for multiplying large dimensional matrices (Ordenez et al., 2016). We use Rcpp (Eddelbuettel, 2013), an R add-on package that facilitates extending R with C++ functions to compute Phase 1 efficiently. Rcpp can be used to accelerate the computation by replacing an R function with its C++ equivalent function. In Rcpp, only the reference gets passed to the other side but not the actual value when we pass the values. So, memory consumption is very efficient and the run time is the same. An equivalent of Rcpp library in Python is SWIG (Beazley, 1996), which automatically generates the bindings between C/C++ code and Python. Moreover, C++ code can be optimized with compiler flags. Our summarization matrix is also ideal for multi-core CPU and GPU computation, but this is an aspect of future work.

Model computation in Phase 2 can be efficiently done by calling existing R (or other data science languages) functions. While we are exploiting C++ in Phase 1, we are using the data science language “as is” in Phase 2. Reprogramming all the models in C++ is an overdoing and will be error-prone. As our model computation requires just changing certain steps in the numerical method, we write the equations based on the data summaries and program them in the data science language efficiently. Even for high d , model computation is fast in any data science language. We use the existing functions available in R to compute models based on (5), (6), (7), and (8). These functions are available in Python too.

As mentioned earlier, any language that API call to C/C++ will be benefited from our solution. For instance, in JavaScript, Emscripten (Zakai, 2011) provides numerous methods to connect and interact between JavaScript and compiled C or C++. So, phase 1 in C or C++ can be easily exploited in JavaScript using Emscripten “Module” object. Phase 2 can be computed in JavaScript itself as the common matrix operations and other functions are available here too. Moreover, JavaScript runs on any OS and web browsers. Integrating our solution with JavaScript will allow to run our solution on various platforms.

However, there are some drawbacks to our solution - not all the ML models can be benefited utilizing our summarization mechanism. For example, SVM, Decision Tree (C4.5) can not be computed using our solution. Similarly, neural network or time series models like ARIMA may not be fully benefited from our solution. In general, if there is no covariance or correlation in the model computation - our method does not help much. As for counts and sums or averages (means) computation - our method can partially benefit such models.

3.6 Time and space complexity

Here, we discuss the complexity of our algorithms in terms of n and d . The time complexity of computing Gamma in Algorithm 1 is $O(d^2n)$. On the other hand, we only compute L

and diagonal of Q of the k -Gamma matrix in Algorithm 2. So, the time complexity of k -Gamma is $O(dn)$. This time complexity applies to all the models utilizing the k -Gamma except K-means. The time complexity of K-means would be $O(kdn)$, where k is the number of clusters. We take advantage of Gamma to accelerate computing the machine learning models. So, the time complexity of this part does not depend on n and is $\Omega(d^3)$, which for a dense matrix may approach $O(d^4)$, when the number of iterations in the factorization numerical method is proportional to d .

In the case of space complexity and memory analysis, our algorithm uses very little RAM. Space required by Gamma matrix in the main memory is $O(d^2)$. And it is $O(kd)$ for k -Gamma matrix where k is the number of classes/clusters. Here, we are reading X only once. That is all the models except K-means are computed in one-pass. It is not possible to compute K-means in one-pass because we initialize it randomly and it computes the model until convergence is achieved. In our incremental computation, we are adding the new Γ with the previous one for each block. So, space does not depend on the number of blocks either and its fixed.

4 Experimental evaluation

We present a detailed experimental evaluation in this section. First, we prove the accuracy of our model by comparing with R language standard function with the original data sets. We also show how relative error decreases for the models as we read more blocks of data. Then we compare our algorithms with the previous full pass Gamma version (Chebolu et al., 2019), standard R functions, and Python Scikit-learn incremental library. Finally, we show how the model can be obtained with incremental computation without reading the full data set and the linear time complexity for a single data set.

However, we do not evaluate the parallel processing of Γ . Computing Γ in a parallel with an array DBMS has already been done in Ordonez et al. (2019) in inter-node and intra-node parallelism manner exploiting both CPU and GPU. And computing the models incrementally in parallel is not trivial because accuracy must be measured and controlled with incremental learning.

4.1 Experimental setup

4.1.1 Hardware and software

We conducted our experiments on a machine with Intel Pentium(R) Quadcore CPU running at 1.60 GHz, 8 GB RAM, 1 TB disk, and Linux OS. We developed our algorithms using standard R and C++. And we compared our solution with R and Python.

4.1.2 Data sets

The data sets used for our experiments are summarized in Table 2, obtained from UCI machine learning repository (Dua & Graff, 2017). For computing accuracy and error analysis (Section 4.2), we use the data sets “as is”. And for comparing execution speed with other systems (Section 4.5), we sampled, replicated, and shuffled the original data sets to get varying n (data set size) and d (dimensions). For varying d , we chose d randomly from the original data set. Also, due to replication our machine learning models may suffer from overfitting or underfitting. To overcome this, we randomize the data sets (X) first. For each

Table 2 Base data sets description

Data set	d	n	Description	Model Applied
YearPredictionMSD	90	515K	Rain or not	LR, PCA
CreditCard	30	285K	Raise in credit line	NB, KM
Iris	3	150	Class of iris plant	KM
SuperConductivity	81	22K	Predict critical temp	LR, PCA
SkinNonskin	4	250K	Skin or not	NB, KM
HouseholdPower	7	2M	Power consumption time series	LR

row (vector) in the data set, we assign a 64bit random number and sort the data set based on the random number. The data set X is now randomized (shuffled) and we apply our algorithms on this randomized data set. We randomize the data set using Python Pandas library and using Pandas chunking feature to read data set in chunks. So, it does not matter if the data set size exceeds RAM capacity while randomizing (shuffling) the data set.

4.2 Asymptotic error analysis

Here, we first establish our incremental computation is accurate and show error goes down as discussed in Section 3.4. We emphasize that the term ‘accuracy’ discussed here is not the prediction accuracy and similarly, the term ‘error’ is not the prediction error. We refer both accuracy and error as the accuracy of our model and the relative error of our model respectively. To compute accuracy and to show error going down, we use the real-world data sets “as is”, meaning we did not alter the data sets (except for Iris data set, where original n is very small). Table 3 shows the results of the experiments that were performed to prove the accuracy of our method. We used two different data sets for each model to show our algorithm is accurate for any data set. All the data sets were obtained from the UCI machine learning repository. We compared the accuracy of our incremental algorithms (with summarization) with the standard ML models in R, computed on non-summarized original data set. And we also compare our solution with the relative error of Python scikit-learn incremental library utilizing partial fit. The details of computing incremental models

Table 3 Final relative error of the models on respective data sets for our solution and Python incremental library

Model	Data set	n	d	Our (R) Solution	Python (sklearn)
LR	YearPredictionMSD	515K	90	4.6E-11	5.7E-1
	SuperConductivity	22K	81	2.5E-9	9.5E-1
PCA	YearPredictionMSD	515K	90	3.5E-5	1.3E-3
	SuperConductivity	22K	81	1.3E-5	2.4E-2
NB	CreditCard	285K	30	0	1.7E-3
	SkinNonskin	150k	4	0	2.6E-2
KM	Iris	1M	4	4.7E-2	1.6E-2
	SkinNonskin	150k	4	1.6E-4	2.6E-2

in Python is discussed later. We take the final value after the algorithm provides the final result in both cases. For each model, we have a different way of measuring the accuracy with the common underlying metric being ‘relative error’. For LR, we take the maximum relative error of the coefficient β and for PCA we take the relative error of maximum Eigenvalue. On the other hand, for both NB and KM, we take the relative error of the model parameter C (means). From Table 3, we can see the relative error for all our models is close to zero, almost negligible. On the other hand, Python has also low relative error but it seems to converge its solution rather quickly than ours with higher relative errors (compared to our solution).

As mentioned in Algorithms 1 and 2 in Section 3.2, we compute the models at each iteration. First, we compute the Gamma based on the block and add the Gamma with previously computed Gamma up to the previous block. Then we use this Gamma to compute the models. The block size represents a very low portion of the original data. So, at the beginning when we compute the models the error is very high. As we read more data points (number of blocks increases), the error goes down as discussed in Section 3.4. In Fig. 4, we show how relative error goes down for models LR and PCA as the number of blocks increases for two different data sets. For LR, we take the maximum relative error of the coefficient β among $d/2$ dimensions. We are taking $d/2$ dimensions because we need a stable LR model without numerical issues. It is impossible that for a fairly large d , all the d dimensions will show the same behavior. First, we compute β from the standard library in R for the full data set and then we compute beta using our method and take the maximum relative error for each block. Similarly, for PCA, we take the relative error of the maximum Eigenvalue from the model. Both plots show that error is high at the beginning. As n increases, the error goes down. We also show a 2% error line for a better understanding. On the other hand, Fig. 5 shows the relative error vs number of blocks plot for NB and KM models for different data sets. For Naïve Bayes, we calculate the maximum relative error of the μ . Similar to before, first we compute the value of μ using the standard library in R and then we compute using Gamma and we take the maximum relative error. Also, for K-means we take the maximum relative error of the model parameter C (means). Both plots show the error going down with the increase of blocks as presented in Section 3.4. We can see error stabilizes after reading few initial blocks. As the error scale here is very small compared to LR, we show a 0.01% error line for a better understanding.

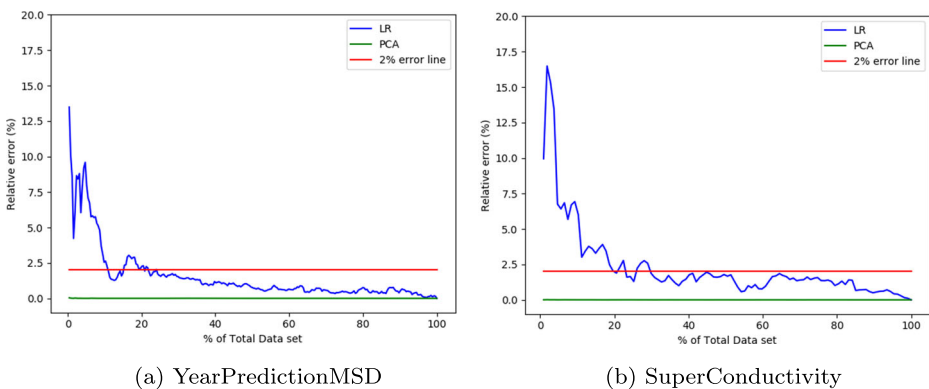


Fig. 4 How relative error decreases as data set size grows for LR and PCA on different data sets

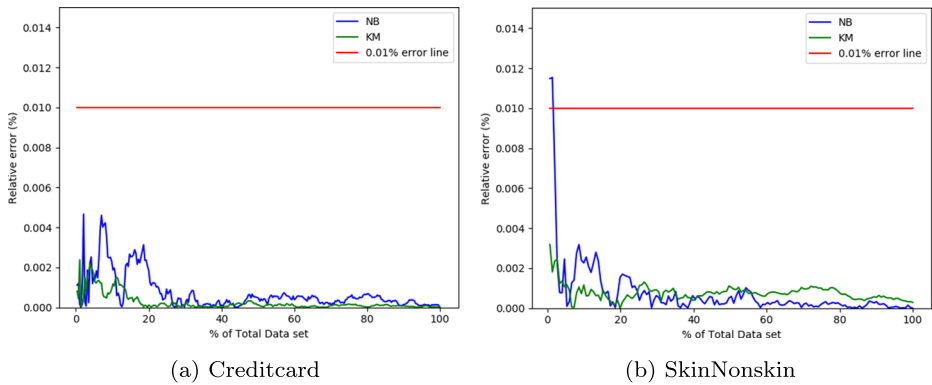


Fig. 5 How relative error decreases as data set size grows for NB and KM on different data sets

We use a default block size above the minimum discussed in Section 3.4. For all the models, we can see that the error is less than 1% after reading 10% of the data set except for LR which requires estimating error per variable. Also, the maximum error does not exceed more than 5% after reading 10% of the data set. This proves that we can get a fairly accurate model by our method even without reading half the data set.

4.3 Data set with concept drift

Here, we further evaluate our solution with a data set that has concept drift, meaning the variance of data changes over time. We use a public time series data set, HouseholdPower (Table 2), which records the electric power consumption of one household with a one-minute sampling rate over a period of almost 4 years. We had to pre-process and clean the data set to make it compatible with our solution as the data set had some missing values and also, we do not need the date and time columns to build the model. To show the advantages of our method, we compute the LR model on the data set and show how our model behaves when new batch of data comes (kind of a streaming fashion). Figure 6 shows the changes in relative error in each increment as new batch of data comes. We consider two scenarios. (1)

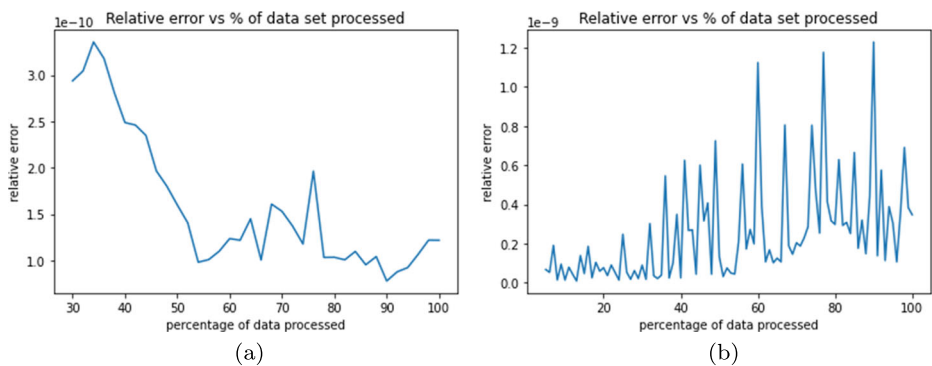


Fig. 6 How relative error changes as we read more data with concept drift (for LR model): (a) 2% increment after 30% initial model (b) 1% increment after 5% initial model

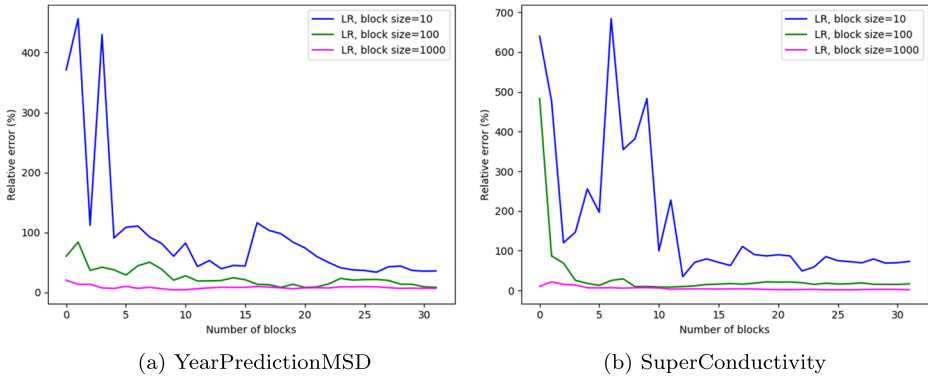


Fig. 7 Relative error vs block size for different data sets: LR model

First, we compute our summarization matrix on a good portion of the data set (30%), then compute the LR model incrementally for each 2% of data. (2) We compute the summarization matrix on a pre-mature data set (5%), and then compute the LR model incrementally for each 1% of data. For both scenarios, we compute the LR model with built-in R routines and compute the relative error of our method. From Fig. 6(a), we notice that the model behaves (more stable) well when we compute our initial summarization matrix on a good portion of the data (30%). However, when the initial summarization matrix is computed on a non-stable data set (5%) and the increment is too small (1%), the model oscillates too much (Fig. 6(b)). We emphasize that although this experiment behave like a data stream, we are not actually computing the models on real streaming data which is out of scope of this paper.

4.4 Impact of block size on accuracy

We discussed in Section 3.4 that we need a minimum block size for the error to go down. Here, we present how the block size affects the relative error for different data sets. We discuss only LR and NB models, one model from each algorithm. Figures 7 and 8 shows how relative error changes based on different block sizes. We take three different block sizes: 10, 100, and 1000 for each model. For each block size, we run the model for a fixed

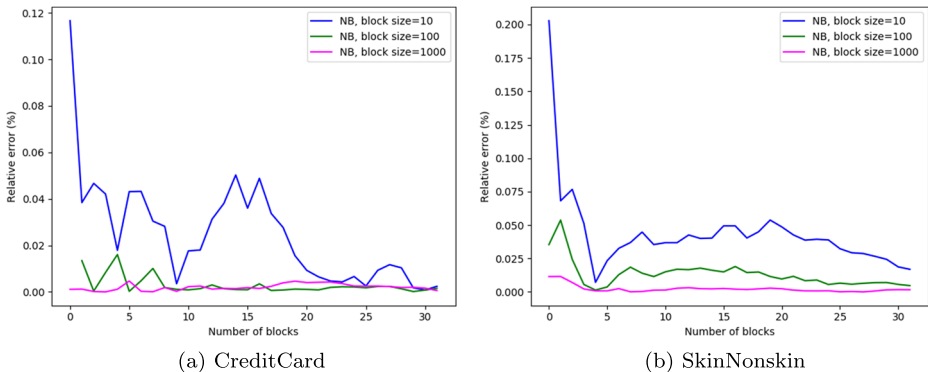


Fig. 8 Relative error vs block size for different data sets: NB model

number of blocks (32 blocks) and plot the relative errors. We can see that the relative error is very high and it oscillates too much in each case when the block size is very small (block size = 10). For block size=100, the relative error shows better behavior than the previous one but it is not stable. On the other hand, with a standard block size (block size=1000), the relative error does not oscillate much and shows a stable behavior. This is because the models can not learn properly when the block size is small. Also, in traditional ML algorithms, the model suffers from underfitting when data samples are very low and it is often recommended to get more samples or perform cross-validation. In our case, the smaller block size shows the same behavior and yields incorrect results.

4.5 Comparison with data science languages (R and Python)

We compare our solution with our previous method of full-pass Gamma (Chebolu et al., 2019), standard R, and Python incremental library. Our goal is to show that our solution has the same capabilities as existing systems without much compensation in execution time. In our previous solution with full-pass Gamma (Chebolu et al., 2019), first, we computed the full summarization matrix based on the entire data set, and then computed the ML models based on it. Also, the k -Gamma matrices were represented in a different way. More details are discussed on Section 5. For Python, scikit-learn (Pedregosa et al., 2011) with the “partial fit” method supports out-of-core learning when the data can not fit in the main memory. However, the exact algorithm as ours is not available for some models in scikit-learn incremental library. We took the most similar algorithm in that case. Also, we used the recommended parameter settings given in the documentation for all the models. To read the data set, we used the Python pandas library which is very fast, widely used, and supports reading data sets by chunks (blocks). On the other hand, to the best of our knowledge, we are not aware of any packages in R that support incremental or online learning. So, we compare with standard R functions that compute the models. For our algorithms, we use a default block size above the minimum discussed in Section 3.4.

Tables 4 and 5 compares the time to compute LR and PCA on YearPrediction data set. We can see that as the size of the data set increases, the inbuilt R packages crash. One of the main reasons can be - it tries to load the whole data set into the main memory, eventually resulting in untimely aborts of the program. In the case of Python, we use SGDRegressor that learns a linear model using stochastic gradient. However, we must emphasize that SGDRegressor yields a very high relative error with “partial fit” when training is done in batches. Incremental PCA in Python (Ross et al., 2008) has a similar implementation like our Γ matrix where a similar computation like Q (in Γ) is performed in orthogonal form. From both Tables, we see that our solution is much faster than R and they are competitive

Table 4 Time to compute LR on YearPrediction data set (in secs)

n	d	Our solution	Full pass gamma	R	Python
0.5M	9	7	7	25	15
1M	9	15	15	51	29
10M	9	159	153	357	287
100M	9	1689	1604	Fail	2878
0.5M	91	53	51	350	50
1M	91	110	105	670	100
10M	91	1068	1054	Fail	931

Table 5 Time to compute PCA on YearPrediction data set (in secs)

n	d	Our solution	Full pass gamma	R	Python
0.5M	9	8	9	26	8
1M	9	16	17	42	16
10M	9	167	178	391	152
100M	9	1742	1901	Fail	1533
0.5M	91	58	63	345	52
1M	91	118	127	726	103
10M	91	1156	1252	Fail	972

with Python. We must emphasize that Python is internally faster than R in most cases. Our solution in R computes the models in the almost same time as Python and can be easily implemented in Python to gain better speed.

In the case of Naïve Bayes, Table 6 compares the time to compute it for Creditcard data set. We see that R crashes for large values of n (10M, 100M) as it fails to load the data set in the main memory. Also, in the case of full-pass Gamma, all the intermediate Gamma for each class based on blocks are stored in a list in R. So, when the value of n is very large, the number of intermediate Gamma matrices increase, and R gives “subscript out of bounds” error. We discussed this problem in Section 3.2 and provided an optimization for our current solution. In Python, we use Gaussian NB with “partial fit” to compute the model. There are other implementations available in Python to compute NB incrementally. But Multinomial NB “Fails” if there exist negative values in the data set and Bernoulli NB is slightly slower than Gaussian based on our experiments. So, we are comparing with the fastest and accurate solution from Python. Our solution stores the k -Gamma in a single matrix and for each block, the partial Gamma is added to the full Gamma. So our solution can compute the models regardless of the number of blocks or n . Similarly, we show the comparison with K-means in Table 7. As usual, R crashes for large n (10M). On the other hand, our incremental algorithm performs better than full-pass Gamma in each case which also fails for large n due to the problem discussed above. In Python, we use “MinibatchKMeans” available in scikit-learn library with “partial fit” like other models to compute it incrementally. The scikit-learn MiniBatchKMeans is a variant of the K-means algorithm which uses mini-batches to reduce the computation time, while still attempting to optimize the same objective function. We used the default parameter settings for running the comparisons. We can see from Table 7 that our solution has almost the same or better performance than Python.

Besides, the summarization matrix can help the data scientists to explore other basic statistics like mean, variance, or correlation from the data set. They can compute any models that contain these computations. Also, they can get a subset of data (ex: separate data based on gender or age) and see the statistics there. However, computing the Gamma matrix directly from R or Python is not possible. We need to compute n , L , and Q separately in these languages. Though computing n and L is straightforward and fast in these languages,

Table 6 Time to compute Naïve Bayes on Creditcard data set (in secs)

n	d	Our solution	full-pass Gamma	R	Python
0.3M	30	8	9	68	9
1M	30	44	45	245	43
10M	30	508	Fail	Fail	415

Table 7 Time to compute K-means on Iris data set (in secs)

n	d	Our solution	full-pass Gamma	R	Python
0.15M	4	3	20	3	3
1M	4	39	237	9	43
5M	4	188	768	48	214
10M	4	386	Fail	Fail	425

they have to compute Q (i.e., XX^T) using a traditional matrix multiplication. As it is well known that the traditional default matrix multiplications are slow in these languages and they run out of memory for huge matrices, we are not showing the comparisons for this.

We believe there does not exist any system that is entirely good. Like every good system, our solution has some minor drawbacks too. For instance, we may not get the exact summarization each time if there are terabytes of data all with floating-point numbers. Because the system may truncate values after decimal points during matrix multiplication. Another shortcoming of our solution is that we can not get back the original data from the summarization matrix. That is, we can not get back X from Γ .

4.6 Incremental model computation and linear time complexity

Here, we present the experiments to compute the approximate ML models in an incremental manner and the linear speed up.

4.6.1 Incremental ML models from partial data set

Here, we show our solution can get an approximate model without reading the whole data set. Table 8 shows how our incremental algorithm performs to get a stable model with low relative error before reading even half of the data sets. As mentioned before, we are reading data set by blocks and they represent the samples of the total data set. For each block, we compare the relative error with the last two blocks. If both the difference values are less than a threshold, we stop our algorithm and get the model. We set the threshold value as 0.0001. We report both the relative error when we stop our algorithm and the final relative

Table 8 Getting approximate ML models from partial data set (M =Millions; err=error) (Time in Seconds)

Θ	Data set (X)	n	d	% of		Rel. err.		Final
				X	Time	Our Sol.	Python	Rel. err
LR	YearPrediction	0.5M	91	38	21	3.1E-2	14.9	4.6E-11
	SuperConductivity	0.1M	81	36	13	1.5E-2	21.1	2.5E-9
PCA	YearPrediction	0.5M	91	8	7	1.8E-3	6.6E-2	3.5E-5
	SuperConductivity	0.1M	81	11	4	3.0E-3	5.8E-1	1.3E-5
NB	CreditCard	1M	31	6	4	0	9.9E-1	0
	SkinNonskin	0.2M	4	2	1	1.1E-3	4.5E-3	0
KM	Iris	1M	4	10	3	2.9E-2	2.81	2.5E-2
	SkinNonskin	0.2M	4	2	1	2.4E-3	9.2E-1	1.6E-4

error when we read the full data set. The difference between both values is very small. Also, we report the percentage of the data set (X) that has been read in blocks and the time it takes to compute the models up to that portion. We can get an accurate model without even reading 25% of the data set except for LR which needs almost half of the data set to generate an accurate model. As for the time measurement from Table 8, obtaining the approximate model is much faster and can be done within seconds. Moreover, we compare our relative error with Python's scikit-learn incremental algorithms. We measure the relative error for Python at the same point (% of X read) when our solution has the approximate model. We notice that, our solution performs much better than Python in terms of getting an approximate model with lower relative error.

4.6.2 Linear time complexity

Figure 9 shows how time increases as we read more portion of the data sets. The X-axis is what percentage of the data set is read (in percentage) and the Y-axis shows the time to calculate the models using Algorithm 1 and 2 discussed in Section 3.2. We emphasize that we are not stopping the algorithms forcefully or by giving any condition, and we are not taking the accuracy into account. The algorithms are executed until all the blocks are read. Hence, we used a fairly large value of n and a default d for each data set. As we read more blocks, the data set size increases - and the time increases linearly. When the whole data set is read, we can see that the time is almost linear for each model. This shows that our incremental algorithm works evenly at each iteration and our algorithm can achieve linear speedup.

4.6.3 Memory utilization

We measure how much memory is needed to compute the models for large data sets. We already mentioned in Section 3.6 that our solution uses very little RAM. Table 9 shows the

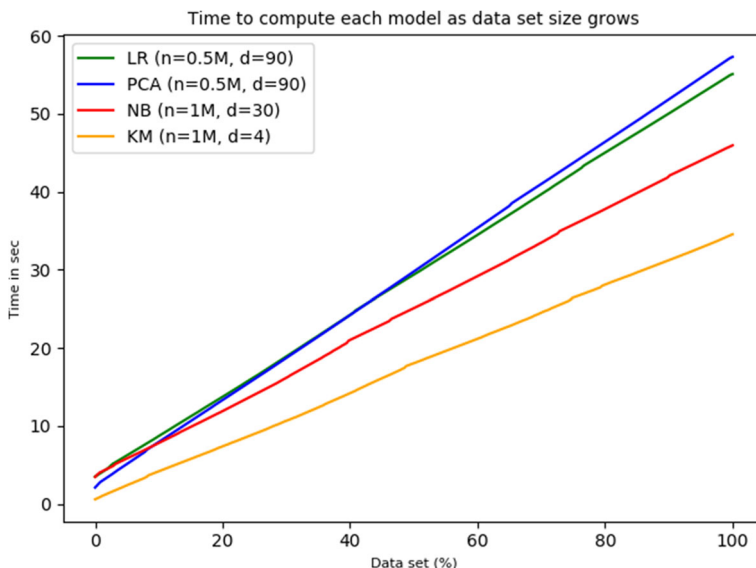


Fig. 9 How time increases as the data set size increases

Table 9 Maximum Memory Utilization

Model	Data Set	n	d	Max memory (%)
LR	YearPredictionMSD	100M	9	11.3
PCA	YearPredictionMSD	100M	9	11.3
NB	CreditCard	10M	30	11.7
KM	Iris	100M	4	11.4

percentage of the physical memory of the system that is used by our solution. We report the maximum value during the program execution time. We stopped all the other user-defined processes and everything else was set to default. We can see that our program uses a small portion of the memory even for large data sets as mentioned in Section 3.6. As a result, we can avoid “memory leak” which may cause all or part of the system to work incorrectly.

5 Related work

Here, we discuss the closely related works and our previous approaches.

5.1 Data summarization

Data summarization to accelerate the computation of machine learning models has received significant attention (Ahmed, 2019; Al-Amin et al., 2020). Summarization of scalable machine learning algorithms was done in a parallel manner in Ordonez et al. (2016). The authors introduced the Gamma summarization matrix and computed the models like LR, PCA, and VS (variable selection). However, this work was developed for a parallel DBMS and did not have any incremental algorithm. In this paper, we removed the use of DBMS completely which was the main focus on Ordonez et al. (2016). We adapted the algorithms and implemented them such that they are incremental, scalable, and more than 99% accurate in R. We also introduced k -Gamma matrix and solved more models like NB and KM which are significantly different from Ordonez et al. (2016). Moreover, our method can read arbitrarily large data set incrementally in blocks. A similar, but less general, data summarization to ours was pioneered in Zhang et al. (1996). to accelerate the computation of distance-based clustering: the sums of values and the sums of squares. Later Bradley et al. (Bradley et al., 1998) exploited such summaries as multidimensional sufficient statistics for the K-means and EM clustering algorithms. Also, data summarization has been used for hierarchical clustering of large data sets (Patra & Nandi, 2015) where the authors proposed a summarization scheme to speed up single-link clustering method. In short, our summarization is more general and it can help to compute more complex models like LR, PCA, NB, and KM that could not be solved with older summaries.

From a “systems” angle, R combined with C++ did not exist and nobody thought we could insert efficient C++ code for a very common computation. Gamma can be considered a fundamental incremental operator plugged into the language, like UDFs in a DBMS. But this approach is more efficient because we do not need to modify the R runtime interpreter. For example, for statistical analysis in R, RIOT (Zhang et al., 2010) package extends R and makes R programs I/O efficient. It features a flexible array storage manager and an optimization engine suitable for statistical and numerical operations but it does not support incremental computation. Another package Ricardo (Das et al., 2010) combines the data

management capabilities of Hadoop and Jaql with the statistical functionality provided by R. However, it was also not envisioned the machine learning models could compute large data sets incrementally on a single machine with limited RAM in R.

5.2 Incremental algorithms

Incremental learning is a method of machine learning in which input data is continuously used to extend the existing model's knowledge. Several researchers have aimed to adapt new data without forgetting the existing knowledge for their learning model (Spokoiny & Shahar, 2008). The concept of incremental learning, particular challenges, and popular approaches were discussed briefly in Gepperth and Hammer (2016). There is a large body of works based on incremental and decremental Support Vector Machine (SVM) (Cauwenberghs & Poggio, 2000; Chen et al., 2019; Karasuyama & Takeuchi, 2009). Linear classifiers like linear SVM can be easily designed to support incremental and decremental algorithms because of their simplicity over kernel or other methods. Polikar et al. (2001) presented an algorithm for incremental training of neural network (NN) pattern classifiers. Like our work, the algorithm does not require access to previously used data during subsequent incremental learning sessions. The authors of He et al. (2011) proposed a general adaptive incremental learning framework that is capable of learning from continuous raw data (stream data). The authors did not compete for the best classification accuracy across all data sets with existing models, rather they investigated how to effectively integrate previously learned knowledge into currently received data to improve learning from new raw data. Similar to our idea, multiple data streams over a sliding window of most recent entries are computed in one-pass fashion by the authors of Altıparmak et al. (2008). The authors adopted the transform-based data summarization technique in terms of the representation quality of a single signal and they extended it for multiple streams. An incremental algorithm for clustering spatial data streams was proposed by Totad et al. (2012) which operates on two phases like our K-means algorithm. Incremental approaches have also been used in relational databases by in Tari et al. (2012) where extraction is then performed on both the previously processed data from the unchanged components as well as the updated data generated by the improved component.

6 Conclusions

We presented a smart, generalized incremental algorithm to compute machine learning models using summarization matrix. Our solution is intelligent, learn with fewer points like sampling and get more and more accurate as we read more data. Also, we can read an infinite amount of data and update the model upon arrival of new data without recomputing everything. Our incremental learning allows learning from a data stream. The solution can significantly reduce the storage requirement on updating the ML models without the loss of model accuracy and enhance computational efficiency. We also studied how to integrate our smart solution with a data science language, specifically R but it can be extended to any language supporting API call to C or C++. We justified why we need C++ and why we use existing analytic functions to compute models from the summarization matrix without reprogramming everything. We would like to note that it is not our intention to compete for the best speed across all the data sets of the proposed approach with those of existing methods. Rather we proposed a generalized solution that can work in any data science language. Experimental results proved that our model computation is more than 99% accurate

for original base data sets. We also achieve remarkably higher speed than R functions and almost the same speed as Python incremental library.

Our research opens many possibilities. First, we want to extend our solution for sparse data sets. We want to explore what happens if we add or remove dimensions from the data set. This is a much harder problem since we have to read the data set again. Also, we will tackle other ML models, including SVMs, LDA, HMMs. A good research direction is how to compute the ML models incrementally in parallel as we have already discussed that this is a much harder problem to tackle. Finally, we would like to analyze how our summarization matrix and ML models behave on data streams over sliding windows.

Declarations

Conflict of Interests The authors declare that they have no conflict of interest.

References

- Ahmed, M. (2019). Data summarization: A survey. *Knowledge and Information Systems*, 58(2), 249–273.
- Al-Amin, S. T., Chebolu, S. U. S., & Ordonez, C. (2020). Extending the R language with a scalable matrix summarization operator. In *IEEE International conference on big data, big data 2020* (pp. 399–405).
- Altıparmak, F., Tuncel, E., & Ferhatosmanoglu, H. (2008). Incremental maintenance of online summaries over multiple streams. *IEEE Transactions on Knowledge and Data Engineering*, 20(2), 216–229.
- Beazley, D. M. (1996). SWIG: An easy to use tool for integrating scripting languages with C and C++. In *Fourth annual USENIX tcl/tk workshop*.
- Bradley, P., Fayyad, U., & Reina, C. (1998). Scaling clustering algorithms to large databases. In *Proc. ACM KDD conference* (pp. 9–15).
- Cauwenberghs, G., & Poggio, T. A. (2000). Incremental and decremental support vector machine learning. In *Advances in neural information processing systems 13, papers from neural information processing systems (NIPS) 2000* (pp. 409–415). Denver: MIT Press.
- Chebolu, S. U. S., Ordonez, C., & Al-Amin, S. T. (2019). Scalable machine learning in the R language using a summarization matrix. In *Database and expert systems applications DEXA* (pp. 247–262).
- Chen, Y., Xiong, J., Xu, W., & Zuo, J. (2019). A novel online incremental and decremental learning algorithm based on variable support vector machine. *Cluster Computing*, 22(Supplement), 7435–7445.
- Das, S., Sismanis, Y., Beyler, K., Gemulla, R., Haas, P., & McPherson, J. (2010). RICARDO: Integrating R and hadoop. In *Proc. ACM SIGMOD conference* (pp. 987–998).
- David, J., Pessemer, T. D., Dekoninck, L., Coensel, B. D., Joseph, W., Botteldooren, D., & Martens, L. (2020). Detection of road pavement quality using statistical clustering methods. *Journal of Intelligent Information System*, 54, 483–499.
- Dua, D., & Graff, C. (2017). UCI machine learning repository. <http://archive.ics.uci.edu/ml>.
- Eddelbuettel, D. (2013). *Seamless r and c++ integration with rcpp*. New York: Springer.
- Gepperth, A., & Hammer, B. (2016). Incremental learning algorithms and applications. In *24th european symposium on artificial neural networks, ESANN*.
- Hastie, T., Tibshirani, R., & Friedman, J. (2001). *The Elements of Statistical Learning*, 1st edn. New York: Springer.
- He, H., Chen, S., Li, K., & Xu, X. (2011). Incremental learning from stream data. *IEEE Transactions on Neural Networks*, 22(12), 1901–1914.
- James, G., Witten, D., Hastie, T., & Tibshirani, R. (2013). *An introduction to statistical learning* Vol. 112. Berlin: Springer.
- Karasuyama, M., & Takeuchi, I. (2009). Multiple incremental decremental learning of support vector machines. In *23rd annual conference on neural information processing systems 2009* (pp. 907–915).
- LeCun, Y., Bengio, Y., & Hinton, G.E. (2015). Deep learning. *Nature*, 521(7553), 436–444.
- Levatic, J., Ceci, M., Koccev, D., & Dzeroski, S. (2017). Semi-supervised classification trees. *Journal of Intelligent Information System*, 49, 461–486.
- Ordonez, C., Zhang, Y., & Cabrera, W. (2016). 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.

- Ordonez, C., Zhang, Y., & Johnsson, S.L. (2019). Scalable machine learning computing a data summarization matrix with a parallel array DBMS. *Distributed and Parallel Databases*, 37(3), 329–350.
- Osojnik, A., Panov, P., & Dzeroski, S. (2018). Tree-based methods for online multi-target regression. *Journal of Intelligent Information System*, 50, 315–339.
- Patra, B. K., & Nandi, S. (2015). Effective data summarization for hierarchical clustering in large datasets. *Knowledge and Information Systems*, 42(1), 1–20.
- Pedregosa, F., Varoquaux, G., Gramfort, A., & Michel, V. (2011). Scikit-learn: Machine learning in Python. *Journal of Machine Learning Research*, 12, 2825–2830.
- Polikar, R., Upda, L., Upda, S. S., & Honavar, V.G. (2001). Learn++: an incremental learning algorithm for supervised neural networks. *IEEE Trans. Syst. Man Cybern. Part C*, 31(4), 497–508.
- Ross, D. A., Lim, J., Lin, R., & Yang, M. (2008). Incremental learning for robust visual tracking. *International Journal of Computer Vision*, 77, 125–141.
- Rumsey, D. (2011). *Statistics For Dummies. –For dummies*. Hoboken: Wiley.
- Spokoyny, A., & Shahar, Y. (2008). Incremental application of knowledge to continuously arriving time-oriented data. *Journal of Intelligent Information System* 1–33.
- Tari, L., Tu, P. H., Hakenberg, J., Chen, Y., Son, T. C., Gonzalez, G., & Baral, C. (2012). Incremental information extraction using relational databases. *IEEE Transactions on Knowledge and Data Engineering*, 24(1), 86–99.
- Totad, S. G., Geeta, R. B., & Reddy, P.V.G.D.P. (2012). Batch incremental processing for fp-tree construction using fp-growth algorithm. *Knowledge and Information Systems*, 33(2), 475–490.
- Zakai, A. (2011). Emscripten: an llvm-to-javascript compiler. In C. V. Lopes, & K. Fisher (Eds.) *Companion to the 26th annual ACM SIGPLAN conference on object-oriented programming, systems, languages, and applications, OOPSLA* (pp. 301–312). ACM.
- Zhang, T., Ramakrishnan, R., & Livny, M. (1996). BIRCH: An efficient data clustering method for very large databases. In *Proc. ACM SIGMOD conference* (pp. 103–114).
- Zhang, Y., Zhang, W., & Yang, J. (2010). I/o-efficient statistical computing with RIOT. In *Proc. ICDE*.

Publisher's note Springer Nature remains neutral with regard to jurisdictional claims in published maps and institutional affiliations.