A different VIM: Visualizing Incremental Machine Learning

Sikder Tahsin Al-Amin, Carlos Ordonez, Mohammad Imtiaz Nur, Aisha M. Farooque, Guoning Chen
Department of Computer Science, University of Houston
Houston, TX, USA

ABSTRACT
Computing machine learning models is challenging when data sets do not fit in the main memory. Incremental learning has been used to continuously extend the existing model’s knowledge. After a few iteration, it is hard to determine if the model has saturated or if it is still learning from the data. On the other hand, data summarization has been a fundamental technique in data mining that has promise with more demanding data science applications. With these motivations in mind, we present a system, VIM, that computes the machine learning models in an incremental manner and visualizes the learning of the model parameters to the user. Specifically, machine learning models are computed and visualized as the data set is being summarized. Our system works for a wide spectrum of machine learning models and can handle data sets larger than the RAM size. There is a GUI which: (1) guides the user to upload data sets and choose the machine learning model, (2) offers interactive visualization of the model parameters, and (3) helps getting an approximate model without reading the whole data.

1 INTRODUCTION
With the advancement of big data, machine learning model nowadays are usually trained with a large amount of data. In many practical scenarios, the samples in the data set can change over time mostly due to the addition of new data samples and removal of existing data samples. It is still computationally intensive to re-compute 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 [4], [6]. In our work, we focus on the computation of incremental machine models in a single machine when the data set size is large. On the other hand, accelerating machine learning algorithms does not always mean adding new hardware and memory. Therefore, processing and analyzing large volumes of data, specially when it is larger than the main memory, becomes non-feasible using a traditional serial approach. Data summarization has been fairly popular among data scientists to accelerate the computation of machine learning models [1], [2].

Our contributions are the following: (1) We present a system that can compute several incremental machine models using data summarization for large data sets in a single machine. (2) We visualize the learning of the model parameters as the data is being summarized and model is being computed. (3) Our solution can get an approximate model faster than the built-in Python incremental library. (4) We provide a GUI to help the user navigate our system and display our results. In this work, we focus on the incremental computation of three popular and fundamental machine learning models: Linear Regression (LR) [5], Principal Component Analysis (PCA) [5], and Naïve Bayes (NB). The models were solved by computing the summarization matrix first while we read the data set in blocks and compute the models from the summarization matrix for each block. From a system angle, our system can show the learning of the model parameters as we read through the data set. Also, the system can show the evaluation metrics on the test data to help determine along with the model parameters to see if the model has been saturated.

2 SYSTEM OVERVIEW
2.1 Definitions
The input matrix is defined as $X$ which is a set of $n$ column vectors. And $\Theta$ 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 $X$. We augment $X$ with an extra row of $n$ is and call that as matrix $Z$, $(d + 1) \times n$ dimension.

2.2 Theory and Algorithm
Here, we first review the Gamma summarization matrix ($\Gamma$) [3] and computation of several ML models ($\Theta$) exploiting $\Gamma$. The main algorithm has two phases. Our algorithm efficiently compute these two phases together in each iteration (for each block of data).

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

2.2.1 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 [3], the Gamma ($\Gamma$) is defined below in Eq. 2. The $\Gamma$ 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.

\[
\Gamma = \begin{bmatrix}
1^T & Y^T \\
L^T & X^T \\
X & YY^T
\end{bmatrix}
\]

Now, from [3], we need $k$-Gamma $(\Gamma^k)$ for classification/clustering models which is given in Eq. 3 ($k$=number of classes/clusters).

Here, we need only a few parameters, $n, L, L^T, diag(Q)$. Both $L$ and $diag(Q)$ 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
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. To connect Python with C++ we use the Python ‘SWIG’ library. However, our solution can be integrated to any language that supports API call to C/C++.

2.3 System Architecture

We assume the user needs to compute the machine learning models quickly and accurately. We assume visualizing the learning of model parameters can help them understand how the model is behaving, when to stop the model (if needed), and satisfy their needs. Our system offers incremental machine learning algorithms for large datasets as a service and provides a graphical visualization that will be easily understandable to the user. Our system splits the work among Python and C++.

Fig 1 shows the data flow diagram of our proposed system. Our system has three main parts: (1) Host Module, (2) Processing, and (3) Visualization. Here, Python will act as the host program as it is interactive and popular in analytics. Moreover, it can run on any machine and it has many extensive library support. The computation is divided between Python and C++. The main processing will be done (computing summarization matrix, Phase 1) in C++ while the other parts will be done in Python. C++ works at the back end and it is hidden from the user. As mentioned above, a Python library (‘SWIG’) will handle the data passing through Python and C++. We emphasize that we only need to compute the summarization in C/C++ as Python is slower for this kind of processing. After processing is done in C++, it returns the output as a matrix back to the Python module. As the size of our summarization matrix is concise, transferring data to Python will be fast and it will easily fit in the main memory. Python processes the matrix and compute machine learning models exploiting it as mentioned earlier. While

represented in two columns ($L$ and $Q$). We still need to store the value of $n$ in a row, which makes the $k$-Gamma $(d+1) \times 2k$.

$$
\Gamma^k = \begin{bmatrix}
  n_1 & 0 & \cdots & n_k & 0 \\
  L_{11} & Q_{11} & \cdots & L_{1k} & Q_{1k} \\
  \vdots & \vdots & \cdots & \vdots & \vdots \\
  L_{d1} & Q_{d1} & \cdots & L_{dk} & Q_{dk}
\end{bmatrix}
$$

(3)

2.2.2 Phase 2: Both $\Gamma$ and $\Gamma^k$ provide summarization for a different set of machine learning models ($\Theta$). We briefly discuss how to compute each model ($\Theta$) below. The details of the model computation are discussed in [2].

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

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

(4)

Eq. 4 will yield a $d \times 1$ matrix, the regression coefficients. To get the intercept along with it, we need to modify Eq. 4 as: $\hat{\beta} = Q^{-1}(XY^T)$, where $Q$ is a $(d+1) \times (d+1)$ matrix.

$PCA$: PCA can be computed on the covariance matrix ($V$), or the correlation matrix ($\rho$). We compute $\rho$, the correlation matrix as $\rho = UDU^T = (UDU^T)^T$. We can also compute the covariance matrix as $V = Q/n - LL^T/n^2$. Then we compute PCA from the $\rho$ by solving the Singular Value Decomposition on it ($solv(\rho)$). We express $\rho$ in terms of sufficient statistics in Eq. 5, where $a,b$ are indexes/subscripts (e.g. the term $Q_{ab}$ represents a particular entry like $Q(a,b)$).

$$
\rho_{ab} = \frac{(nQ_{ab} - L_AL_B)}{(\sqrt{nQ_{aa} - L_A^2/n}Q_{bb} - L_B^2)}
$$

(5)

$NB$: Here, we need the $k$-Gamma matrix. We assume, there are $k$ number of classes (where $g = 1, \ldots, k$), 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 $\Gamma^k$ matrix for each class label ($g = 1, \ldots, k$) with the following statistical relations.

$$
W_g = \frac{n_g}{n}, C_g = \frac{L_g}{n_g}, R_g = \frac{Q_g}{n_g} - diag\left[\frac{L_gVT}{n^2}\right]
$$

(6)

2.2.3 Integration with Python: The input data set $X$ is read into $m$ blocks ($m \ll n$) of the same size ($|m| > 1$), $X = \{X_1, X_2, \ldots, X_m\}$. We read each block ($X_b$) into the main memory and compute the summarization matrix ($\Gamma^k$ or $\Gamma^k_b$) for that block as mentioned in Phase 1. This partial Gamma ($\Gamma^k_a$ or $\Gamma^k_b$) is added to the Gamma computed up to the previous block ($X_{b-1}$) and we get the Gamma ($\Gamma$ or $\Gamma^k$) for up to $b^{th}$ block. This phase 1 is computed in C++ as 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 Python is slow, usually one row at a time. Based on the summarization matrix computed here, we compute the machine learning models in Phase 2 as discussed earlier. We compute this Phase 2 in Python. Reprogramming all the models in C++ is an overdoing and will be error-prone. As our model computation
computing the machine learning models, we visualize the learning of the model parameters to the user using a GUI. We use plots, and graphs in the GUI to visualize the model learning to the user.

From a system perspective, our proposed system can show the learning of the machine learning models quickly and accurately. Users can also interact with the system to explore the models. For example, users can stop the computation at any time and get an approximate model there. Also, users have the option to visualize the evaluation metrics. Our system will divide the data set into train and test set, and visualize the respective evaluation metrics for each model based on the test data set. Our system has the potential to be released as a standard Python package that can be easily installed in any machine, solving all the dependencies (SWIG, C++ in this case).

### 2.4 Benchmarking

Table 1: Benchmarking performance to get approximate machine learning models ($M=$Millions; err=error)

<table>
<thead>
<tr>
<th>$\Theta$</th>
<th>Data set (X)</th>
<th>$n$</th>
<th>$d$</th>
<th>% of $X$</th>
<th>Rel. err.</th>
<th>Our Sol.</th>
<th>Python</th>
</tr>
</thead>
<tbody>
<tr>
<td>LR</td>
<td>YearPrediction</td>
<td>0.5M</td>
<td>91</td>
<td>38</td>
<td>3.1E-2</td>
<td>14.9</td>
<td></td>
</tr>
<tr>
<td>PCA</td>
<td>YearPrediction</td>
<td>0.5M</td>
<td>91</td>
<td>8</td>
<td>1.8E-3</td>
<td>6.6E-2</td>
<td></td>
</tr>
<tr>
<td>NB</td>
<td>SkinNonskin</td>
<td>0.2M</td>
<td>4</td>
<td>2</td>
<td>1.1E-3</td>
<td>4.5E-3</td>
<td></td>
</tr>
</tbody>
</table>

To the best of our knowledge, there is little work putting together visualization of complex math models as they are gradually computed. Mostly, users call Python or R libraries in their code, and they have to wait until the computation ends. So, we cannot compare our system end-to-end with another system. Here, we compare our algorithm utilizing summarization matrix with the Python incremental library. Table 1 shows the result of our experimental outcome. We use data sets obtained from UCI machine learning repository where $n$ and $d$ shows the original size of the data sets. For each model, we show the amount of the data needed to get an approximate model ($\%$ of $X$ column), the relative error of our solution for the approximate model, and the relative error from Python incremental library at the same point. These relative errors are computed from the model parameters [3]. We can see that our solution can get an approximate model faster than Python and with much less relative error. However, the final error after reading the whole data set between our solution and Python is less than 0.01%.

### 3 SYSTEM DEMONSTRATION

We will run our system on a portable computer with Python and C++ installed. Our program provides a GUI to the user to load or select data sets (path of the data set location), choose the desired machine learning algorithm, and visualize performance and results. Users can choose the type of machine learning model and it will be processed in Python and C++ in an incremental manner. Users can also select the block size (or chunk size) to read the data. However, a default block size of 1000 will be set as the recommended block size which is explored in [3]. The visualization will be shown mostly in the GUI itself. For the demonstration, we will use Python 3.6 and C++ 11. Python will be connected to C++ with a standard Python package (SWIG). A pre-load data set of small size will be available to the user for faster demonstration purposes. If the data set size is big (> 8Gb), our system will show warning to the user. The output shown by our system is concise and is updated as it read through the data set. However, our system has the potential to work both from the command prompt and using a GUI.

#### 3.1 Points to Emphasize

In our system demonstration we will emphasize the following points: (1) Our system is simple, can be easily installed solving the dependencies, and the GUI is easy to use. (2) Our system is portable. That is, we do not need any system-specified libraries and computation is done C++ and Python, and works on any system supporting them. (3) Our prototype is ideal for data that is standard for any machine learning model. (4) Users will choose the type of incremental machine learning model from Python GUI, processing will be performed in C++ and Python, and visualization will be shown in Python. They can also choose the block size to read the data set. (5) Connection of C++ and Python is performed with standard Python packages (SWIG) and the connection and computation are hidden from the user. (6) We use a single machine for processing. However, it is possible to do the processing in parallel. The system is ran as an stand-alone application in the machine. (7) Our system is easy to modify for the developer. To extend the system, only Python module needs to be changed without worrying about C++. (8) User will be provided guidance on how to use the system.

#### 3.2 Demonstration Scenarios

Here, we walk-through the demonstration scenarios. Our system is easy to use. To run our system, C++ will be connected to Python using SWIG library.

![Figure 2: Example of visualizing the learning of the model parameters](image-url)

Once the user runs our system as an application in any OS (e.g., Windows or Linux), the Python GUI will provide options to paste the file-path of a new data set or choose from existing data sets. For example, the user can choose YearPrediction or Skin-NonSkin dataset available from UCI machine learning respiratory or paste...
the file-path of any dataset of their own in CSV format. However, small data sets will be loaded for the demonstration but the user can provide the file-path of a data set of any size. The user can also choose the block size (or chunk size) of their own or choose the default one. Providing small block size will show a warning as the models will take more time to saturate. The user will also have the option to show the evaluation metrics or not. After that, the system will start the computation and Python will start reading the data set in blocks. As the summarization matrix and model parameters are computed, the results will be shown in the GUI. For example, we show a demo scenario in Figure 2 of computing Linear Regression in an incremental manner. We can see that our system is showing the learning of the model parameters (regression coefficients and intercepts). The X-axis show the % of data that is being read and the Y-axis shows the values. Similarly, the bottom images shows the evaluation metrics for the model on the test data set. Here, we show the R-Squared value as well as the cross validation score.

As the model is being computed, the user have the option to wait until the model is computed on the full data set or they can stop the execution at any time if they think the model is saturated. In any case, the user will be shown the report of the model performance in a new window. For example, Figure 3 shows a demo report of the above mentioned model performance (LR). We can see that it notifies the user that the computation was stopped by the user, and the % of data read at that time. Next, it shows the values of the model parameters (coefficient and intercept), as well as the values of the evaluation metrics (R-Squared and CV-Score). Although here we present the demo scenario for the Linear Regression model, the PCA and Naïve Bayes model will also behave the same way. Only the model parameters and the evaluation metrics (if applicable) will be changed.

There are many benefits of our system. Most users need to go through applying several machine learning models on their data before finalizing a model. They can explore the models by visualizing them using our system and see how their model is behaving on the data. Also, our system will benefit the users who are running these models on large data sets. They do not have to wait until the completion, they can stop the computation at any time if they think the model is saturated and get an approximate model. As for future work, we intend to integrate more models to our system.

REFERENCES