PEM: A Practical Differentially Private System for Large-Scale Cross-Institutional Data Mining

Yi Li[1], Yitao Duan[2], and Wei Xu[1]

[1] Tsinghua University, Beijing, 100084, China
[2] Netease Youdao, Beijing, 100193, China

Abstract. Privacy has become a serious concern in data mining. Achieving adequate privacy is especially challenging when the scale of the problem is large. Fundamentally, designing a practical privacy-preserving data mining system involves tradeoffs among several factors such as the privacy guarantee, the accuracy or utility of the mining result, the computation efficiency and the generality of the approach. In this paper, we present PEM, a practical system that tries to strike the right balance among these factors. We use a combination of noise-based and noise-free techniques to achieve provable differential privacy at a low computational overhead while obtaining more accurate result than previous approaches. PEM provides an efficient private gradient descent that can be the basis for many practical data mining and machine learning algorithms, like logistic regression, k-means, and Apriori. We evaluate these algorithms on three real-world open datasets in a cloud computing environment. The results show that PEM achieves good accuracy, high scalability, low computation cost while maintaining differential privacy.

1 Introduction

With the increasing digitization of society, more and more data are being collected and analyzed in many industries, including e-commerce, finance, health care and education. At the same time, however, privacy concerns are becoming more and more acute as our ever-increasing ability to extract valuable information from the data may also work against people's fundamental rights to privacy. How to make good use of the data while providing an adequate level of privacy is an urgent problem facing both data mining and security communities.

Fundamentally, the problem boils down to striking a balance among three factors that are often at odds with one another: privacy, utility, and efficiency. Previous works largely focus on one or two of these aspects and, as a result, fail to provide practical solutions that can be used in real-world systems (to be elaborated later). Also, due to the diversity of applications regarding goals, algorithms, and data partition situations, a general solution that supports all the situations appears to be extremely hard, if not entirely impossible.

In this paper we target the cross-institutional mining problem where each institution (referred to as a *client* here) collects data independently and together they would like to mine collective data to extract insights. This setup can be found in many real-world situations and typically it is more beneficial to include more data in the analysis. For example, to study the side effects of a drug, it is more accurate and timely if one could pool all the data from multiple relevant institutions (medical research institutes, hospitals, pharmaceutical companies and so on). The same applies to quick public health threats detection, educational data mining etc. Due to privacy concerns, however, institutions are prohibited by law from sharing their data.

This problem is called the horizontally partitioned database model in the privacy-preserving data mining community and has been studied extensively. Many solutions are algorithm-specific, such as clustering [25] and recommendation [3], and they are not scalable to handle large-scale problems. Some solutions are data-specific, e.g. [21]. Other solutions either make impratical assumptions (e.g., [27] assumes all the participants are semi-honest and the exact results do not reveal privacy), or involve heavy computation and communication cost that makes it unacceptable (e.g., [2] is based on secure multi-party computation and is very slow even for vector addition). In addition, early works tend to have a less rigorous notion of privacy.

The privacy we provide is differential privacy [12], a rigorous and strong notion that limits the probability that an adversary can distinguish between two datasets that differ only by one record. A holistic privacy solution needs to protect any information that is released from its owner. In a distributed setting where data is partitioned among multiple institutions, general differential privacy doctrine would add noise in every of these occasions [3,21,22,24]. This approach does poorly for iterative data mining algorithms, as these algorithms would require noise for all intermediate results at each iteration, resulting in a final result too noisy to be useful.

The key idea of our approach is that we can use efficient and noise-free cryptographic primitives to reduce or even eliminate the noise for the intermediate results, preserving final accuracy while achieve the same level of differential privacy. We adopt a secret sharing over small field approach, similar to that of P4P [9] and Sharemind [5]. These frameworks are noise-free and efficient, but lacking explicit mechanisms to protect the intermediate results, and supports only limited operations (e.g. addition). Our noise adding mechanisms compensate the limitation.

Concretely, we make the following contributions:

1) We combine a natural noise addition mechanism with an efficient noise-free cryptographic primitive, making it differentially private. We show that utilizing our framework significantly reduces the amount of noise necessary for maintaining differential privacy. We can use *orders of magnitude* lower noise and thus improve resulting model accuracy dramatically.

2) We provide a complete solution, together with easy to use programming APIs and flexible options that enable differential privacy and performance optimizations. PEM automatically determines the noise level. This enables users to easily implement tradeoff between privacy, scalability and accuracy.

3) We design mechanisms to deal with common distributed system issues such as fault tolerance and load balancing.

4) The system provides a differentially private gradient method that can be the basis of many machine learning algorithms. We also implement a number of commonly used machine learning algorithms such as k-means, Apriori etc.

Our goal is to provide a practical solution with provable privacy for many real-world analysis. We evaluate the tradeoff between accuracy and privacy using three real-word datasets. Our results show that PEM not only guarantees differential privacy, achieving similar results as the original non-private algorithms, but also introduce very low computation overhead. We are adding more machine learning algorithms into PEM, and we will release PEM open source.

The paper is organized as follows. Section 2 reviews recent research on privacy-preserving computation. Section 3 formulates the problem and introduces import definitions. Section 4 introduces the goals of our system and describes some details about the system design. Section 5 gives some proofs why our system works. Section 6 illustrates three kinds of algorithms implemented in our system. Section 7 shows the experiment results. We conclude in Section 8.

2 Related Work

People have proposed many definitions of privacy over the years. Earlier versions, such as *k-anonymity* and *l-diversity* [16], are vulnerable to simple attacks [4,12]. Differential privacy is a popular definition that has strong privacy guarantee while still achievable in practice [12]. There are extended versions of differential privacy definitions such as *pan privacy* [11] and *noiseless privacy* [4]. These extensions often come with restriction to the datasets or use cases. Thus we adopt the general differential privacy definition.

The general approach to achieving differential privacy is to add noise to the released results [10,22]. Adding noise inevitably affects the accuracy and people have explored several noise-free methods [4,8]. These works make use of the adversary's uncertainty about the data thus eliminating the need for external noise. However, both [8] and [4] make strong assumptions about the data distribution to maintain differential privacy.

Many systems use cryptography to achieve differential or other types of privacy. They are based on either homomorphic encryption (HE) or secure multiparty computation (MPC) [28]. The problem with HE or MPC is that both make expensive use of large integer operations and are orders of magnitude slower than the non-private version, making them impractical. For example, Rastogi *et al.* proposes an algorithm PASTE, which uses HE for noise generation to mine distributed time-series data without any trusted server [21]. SEIPA performs privacy-preserving aggregation of network statistics using Shamir's secret sharing scheme [23], which involves heavy polynomial calculation, making it slow in large datasets [6]. DstrDP [26] and SDQ [29] use HE and secure equality check to garble the noise from participants. Other approaches, including [2,5,14,19], also use expensive cryptograph techniques.

The other trend is to take advantage of the property of application algorithm or statistics of the dataset to avoid the expensive cryptographic operations. Chaudhuri *et al.* shows that sampling is helpful for increasing the accuracy of principal component analysis while preserving differential privacy [7]. Shokri *et al.* implements a system for privacy-preserving deep learning [24], using distributed selective SGD algorithm. However, the convergence of the algorithm strongly depends on the dataset. PINQ assumes a trusted third party to implement privacy-preserving queries [18] by adding Laplace noise. However, a single trusted third party is not only a strong assumption but also a performance and security bottleneck. P4P relaxes the trust assumption by allowing non-colluding semi-honest servers [9]. However, the noise-free approach in P4P still assumes too much about the dataset.

We combine these methods into a coherent system: we take the relaxed trust model, adding (reduced amount of) Laplace noise to achieve provable differential privacy, and leverage the properties of algorithms (such as sampling) to further reduce the amount of noise.

3 Preliminaries

3.1 Problem Formulation

In PEM, there are n clients. Each client $C_i (i = 1, 2, \ldots, n)$ has a subset of records D_i to contribute to the computation. The goal of our computation is to use the union of all D_i's, which we denote as D, to compute some statistics, $f(D)$. During the computation, each client wants to ensure that no information about the individual records in the dataset is leaked to other clients or any third pary. Specifically, we want to support any iterative computation where each iteration can be expressed in the form of

$$f(D) = g(\sum_{d \in D} h(d)). \tag{1}$$

where both h and g can be nonlinear. This simple form can support a surprisingly large number of widely-used data mining algorithms, including k-means, expectation maximization (EM), singular value decomposition (SVD), etc.

Assume we have m independent *servers*, $S_1, \ldots, S_m$, and an *aggregator* server. We make the following key assumptions about the servers, which usually hold for real-world applications:

1. All servers are *semi-honest*, which means each server follows the protocol but may attempt to learn about all private data when it has a chance.
2. There are at least two servers that do not conspire with other servers to break the protocol. We can achieve this goal by using servers from different administrate domains.
3. All communications are secure so that adversaries cannot see / change any useful information in the communication channels. We can ensure this assumption using encryption and message authentication methods.

In a realistic scenario, the clients can be collaborating institutions (e.g., hospitals, schools) that do not wish their data exposed to any party. The servers and aggregator can be cloud computing facilities that are owned by different service providers. The non-colluding assumption upholds as the service providers are prohibited by law to leak their customer data. The number of servers, m, does not have to be very large. Using more servers increases security but also the cost. $m = 2$ or 3 is enough for many situations.

3.2 Definitions

We summarize important definitions for readers not familiar with the field.

Differential Privacy [12]. An algorithm K gives ϵ-differential privacy if for all *adjacent datasets* D, D' and all $S \subseteq Range(K)$,

$$Pr[K(D) \in S] \leq e^{\epsilon} \cdot Pr[K(D') \in S) \tag{2}$$

where adjacent datasets are two data sets that differ in at most a single record. Intuitively, with differential privacy, we make the distributions of $K(D)$ and $K(D')$ nearly undistinguishable by perturbing the outputs.

Laplace Distribution [12]. A random variable X follows Laplace distribution $Lap(\lambda)$ if the probability density function (PDF) is

$$Pr\{X = x\} = \frac{1}{2\lambda} e^{-|x|/\lambda} \tag{3}$$

where λ is a parameter that determines the variance. The mean and variance of $Lap(\lambda)$ are 0 and $2\lambda^2$, respectively. We denote $\mathbf{Lap}^d(\lambda)$ as a d-dimensional vector consisting of d independent $Lap(\lambda)$ random variables.

Sensitivity. For a function $f : \mathcal{D} \to \mathbb{R}^n$, the L_1-sensitivity $S(f)$ is defined as:

$$S(f) = \max_{D_1, D_2} \|f(D_1) - f(D_2)\|_1 \tag{4}$$

where D_1 and D_2 are two neighboring datasets differing in at most one row. It can be shown that $f(D) + r$ is ϵ-differentially private if $r \sim \mathbf{Lap}^d(S(f)/\epsilon)$.

4 System Design

Differential privacy provides a rigorous definition and tuneable parameters for us to specify the level of protection that we desire. However, prior works such as [3,13,24] indicate that direct application of the noisy response approach may not allow us to find a spot where acceptable level of privacy and utility coexist. In all the cases a large ϵ must be used, meaning that there is essentially little privacy, if we want the results to have any sensible usage.

To address this problem, it is clear that we must reduce noise as much as possible. Considering the distributed setting that we are dealing with, we are motivated to adopt the following design principle: use efficient cryptographic

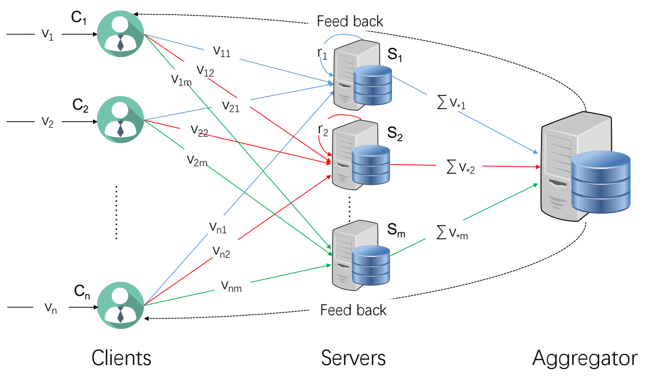


Fig. 1. The overview of PEM architecture.

tools whenever possible to eliminate the need for noise. Instead of adding noise at every client, we adopt secret sharing over small field to perform the aggregation. This paradigm allows us to only perturb the final aggregates, as secret sharing itself protects each client's input. Since the number of servers is far less than the number of clients, the final noise is much smaller.

4.1 System Architecture

There are three roles in PEM, and Figure 1 shows an overview of the system.

Clients. The clients are the owners of the data sources. The PEM client module uses secret sharing protocol to privatize the data while providing useful tools for privacy tracing and model updates.

Servers. The servers compute partial sums and add noise to the sums. Each server is logically a single component, while it can be implemented as a distributed system for scalability and fault tolerance.

Aggregator. The aggregator performs all data aggregation and application-specific computation. It completes an iteration of model update based on the noisy aggregates and sends the model back to the clients.

4.2 Data Processing Protocol

On the start of the job, PEM distributes important parameters and the application executables to all the nodes, such as the privacy parameter ϵ and the list of participants. Then the data processing follows four major steps in PEM:

1) (Clients-Servers) Using secret sharing to aggregate client data. Each client secret-shares each data vector into m random vectors, one for each server. Specifically, for a vector $\mathbf{v}_i$ held by client C_i (here we assume $\mathbf{v}_i$ is an integer vector, while real-value vectors can be converted by discretization), the client can first generate $m-1$ random vectors $\mathbf{v}_{ij} \in \mathbb{Z}_\phi^d (j = 1, 2, \ldots, m-1)$, where ϕ

is a 32- or 64-bit prime and $\mathbb{Z}_\phi$ is the additive group of integers module ϕ. Then it computes the m-th vector as $\mathbf{v}_{im} = \mathbf{v}_i - \sum_{j=1}^{m-1} \mathbf{v}_{ij} \mod \phi$. Obviously, the sum of the m vectors equals to $\mathbf{v}_i \mod \phi$. The client C_i sends a random vector $\mathbf{v}_{ij}$ to the server $S_j(j = 1, 2, \dots, m)$, and S_j receives a random vector from each client. Each server computes a sum vector using vectors from all clients.

2) (Servers) Generating Laplace noise to achieve privacy. To preserve differential privacy, each server automatically determines the level of noise, generates Laplace noise following the approaches in [10] and adds it to its partial sum. Each server $S_j(j = 1, 2, \dots, m)$ generates a d-dimensional random vector $\mathbf{r}_j$ following $\mathbf{Lap}^d(\lambda)$, and sends $\sum_{i=1}^{n} \mathbf{v}_{ij} + \mathbf{r}_j$ to the aggregator.

3) (Servers - Aggregator) Computing the sum of all vectors. At this step, each server sends its perturbed partial sum to the aggregator for final aggregation. On receiving the sums from servers, the aggregator sums up the vectors and obtains the final sum, $\mathbf{v} = \sum_{i=1}^{n} \mathbf{v}_i + \sum_{j=1}^{m} \mathbf{r}_j$. The noise prevents the aggregator from guessing the original sum.

4) (Aggregator - Clients) Update model and complete an iteration. The aggregator runs the application code to generate or update the model. As many learning algorithms can be written in the form in Eq. 1, the sum obtained from the previous step is sufficient for a single iteration in the model update. Then the aggregator sends back necessary information, e.g., the latest models, back to each client for the next iteration.

4.3 ϵ-splitting and dynamic L setting

Privacy parameter (ϵ) splitting. As mentioned above, the servers add noise $r \sim \mathbf{Lap}^d(\lambda)$ to a d-dimensional sum. This works in many cases. However, sometimes we want to apply different noise levels to each dimension. For example, when we use k-means for clustering, we should calculate the sum of and the count of each class to get the mean of these records. As the sensitivity of the records and the sensitivity of the count of each class are different, it is reasonable to add noise of different levels to them. PEM allows users to specify different privacy parameters to different dimensions. Specifically, users can assign privacy parameter as $[d_1 : \epsilon_1, d_2 : \epsilon_2, \dots, d_k : \epsilon_k]$ where d_i is the subset of dimensions, as long as $\forall_{i,j} d_i \cap d_j = \emptyset$, $\cup_i d_i = \{1, 2, \dots, d\}$ and $\sum_{i=1}^{k} \epsilon_i = \epsilon$. Servers automatically compute the amount of noise according to the manner of ϵ splitting.

Dynamic sensitivity (L) setting. PEM splits the ϵ(s) equally to each iteration in advance, and thus the noise levels of different iterations can be calculated independently according to the sensitivity L and other parameters. However, sometimes the vectors to be added up from the clients may change over iterations, leading to a change of the sensitivity. In this case, L is set dynamically over each iteration, making the servers generate noise of different levels. PEM allows users to specify L for each iteration. Then the λ will change accordingly and the servers will update the level of noise automatically.

4.4 Automatically computing the noise level (λ)

A key feature of PEM is the automatical computation of λ based on user settings, such as ϵ and L. As mentioned, each server generates noise following Laplace noise $\mathbf{Lap}^d(\lambda)$, where λ is the noise level. In PEM, the servers automatically compute the noise level according to iteration count T, ϵ and L. As different parameter setting means different ways to generate noise, we consider the following cases:

- In the simplest case, where there is no *ϵ-splitting* or *dynamic L setting*, each server S_j computes $\lambda = T \cdot L/\epsilon$ and generates noise $\mathbf{r}_j = \mathbf{Lap}^d(\lambda)$.
- If *ϵ-splitting* is specified, i.e., the privacy parameter is assigned as $[d_1 : \epsilon_1, d_2 : \epsilon_2, \ldots, d_k : \epsilon_k]$, each server S_j computes $\lambda_k = T \cdot L_k/\epsilon_k$ where L_k is the sensitivity of elements in d_k and generates noise $\mathbf{r}_j = [\mathbf{r}_{j,1}, \mathbf{r}_{j,2}, \ldots, \mathbf{r}_{j,k}] = [\mathbf{Lap}^{|d_1|}(\lambda_1), \mathbf{Lap}^{|d_2|}(\lambda_2), \ldots, \mathbf{Lap}^{|d_k|}(\lambda_k)]$.
- If *dynamic L setting* is specified, i.e., the sensitivity of iteration t is $L(t)$, each server S_j computes $\lambda(t) = T \cdot L(t)/\epsilon$ and generates noise $\mathbf{r}_j = \mathbf{Lap}^d(\lambda(t))$.

4.5 Optimizations

PEM is a federated system across multiple administrative domains. To handle the privacy-related requirements, we make the following extensions.

Extending servers to clusters. Although these operations are simple, we can still accelerate them to make PEM more efficient. We can extend each server to a cluster (e.g., a Spark cluster) to parallize the communications and computations such that the servers are unlikely to be bottlenecks.

Handling client failures. If a server does not hear from a client for an extended period, it informs the aggregator about the client's failure, and the aggregator notifies all servers to remove the client.

Intuitive configurations and programing interfaces. The application developers only need to provide a few intuitive parameters, such as the sensitivity L, privacy strength ϵ, and the switch of sparse vector technique. PEM automatically determines other settings. The programmers only need to implement three functions: *init*, *mapInClient* and *reduceInAggregator* (see Sec. 6 for details).

Online processing. PEM also supports online processing. Clients send the data streams to servers for online processing. We model a data stream as a sequence of *mini-batches*, to reduce the communication overhead. In a mini-batch iteration, each client buffers the data records locally and sends them out periodically.

Algorithm specific optimizations. There are often algorithm specific optimizations to further reduce the required level of noise, such as *sampling* and *sparse vector technique* [12]. Sampling is commonly used to deal with imbalanced data, and can introduce more uncertainty about the raw data, reducing the level of noise required for differential privacy [15]. Sparse vector technique selects the queries above a pre-defined threshold τ and set the unselected elements to 0. Thus we can compress the proposed sparse vector and save communication cost with a small accuracy loss [12]. PEM allows users to enable these optimizations, and PEM automatically performs them and adjusts the noise level accordingly.

5 Analysis

Provable differential privacy. We first prove that we can make the algorithms differentially private to the aggregator and clients, and then prove that the algorithms are also differentially private to the servers. Formally, we determine the parameter λ using the sensitivity method [10]. Assuming the sensitivity is no more than L, i.e. $|\mathbf{v}_i - \mathbf{v}_j| \leq L$, we have the following theorem.

Theorem 1 *For any $\boldsymbol{v} \in \mathbb{R}^d$, $\boldsymbol{v} + \sum_{j=1}^{m} \boldsymbol{r}_j$ is ϵ-differentially private if, for $j = 1, 2, \ldots, m$, $\boldsymbol{r}_j$ is drawn independently from $\boldsymbol{Lap}^d(L/\epsilon)$.*

Proof. According to [10], if we add only one of $\mathbf{r}_j (j = 1, 2, \ldots, m)$, the algorithm is already ϵ-differentially private. As $\mathbf{r}_j (j = 1, 2, \ldots, m)$ are independent, adding another random noise to the result can still achieve ϵ-differential privacy.

We now show that the mechanism in PEM for calculating the sum is differentially private. Specifically, in PEM, as long as there are at least two servers that do not collude with others, the noisy sum calculated by the aggregator is ϵ-differentially private to all roles. The reason is straightforward: both the aggregator and the clients see the true vector sum plus at least two pieces of noise generated independently according to $\mathbf{Lap}^d(L/\epsilon)$. By Theorem 1, the aggregation is ϵ-differentially private to the aggregator and clients. Meanwhile, according to [9], PEM leaks no information beyond the intermediate sums to the servers as the servers' view of the intermediate sums is uniformly random and contains no information about the raw data. On the other hand, as there are at least two semi-honest servers generating noise faithfully, each server sees the result perturbed by at least one piece of noise following $\mathbf{Lap}^d(L/\epsilon)$ even if it excludes its own noise from the sum, which is enough for preserving ϵ-differential privacy.

Cost and complexity. In PEM, there are m servers and n clients. We have $m \ll n$ in general. To preserve the privacy of data, m servers generate independent Laplace noise. As the variance of Laplace distribution $Lap(\lambda)$ is $2\lambda^2$, the variance of the sum of m independent noises is $2m\lambda^2$. Comparing to the methods where all clients add noise (e.g. [3,24]), where the overall variance of the sum is $2n\lambda^2$, our method reduces much noise as $m \ll n$.

As most of operations in PEM are vector addition and noise generation, the computational overhead grows nearly linearly with the number of clients and the data dimensionality. Thus, PEM is suitable for processing high-dimensional vectors. In PEM, noise generation and addition is orders of magnitudes faster, comparing to many homomorphic-encryption-based approaches, especially those related to high-dimensional vectors.

6 Sample Algorithms in PEM

To implement privacy-preserving data mining algorithms in PEM, users only need to implement the following necessary functions:

```
**Function** *Init()*:
    Initialize $\epsilon$ and $L$ as constants. Initialize the iteration count $T$, the weight $\mathbf{w}$,
    loss function $\phi$ and the learning rate $\eta$.

**Function** *mapInClient*$(\mathbf{x}, y, \mathbf{w}, \phi)$:
    return $\partial\phi(\mathbf{x}, y, \mathbf{w})/\partial\mathbf{w}$.

**Function** *reduceInAggregator*$(\mathbf{v}, \mathbf{x}, \eta, t)$:
    update $\mathbf{w}$ as $\mathbf{w} = \mathbf{w} - \eta\frac{1}{\sqrt{t}} \cdot \frac{1}{\sum_{i=1}^{n} b_i}\mathbf{v}$.
```

Algorithm 1: The user functions of gradient descent.

– *init*: initialize the parameters, including privacy parameter ϵ and the sensitivity L. Note that ϵ can be an array and L may change over the iterations.
– *mapInClient*: map the records of the local database to the vectors.
– *reduceInAggregator*: analyze the sum of the vectors from the clients and update the parameters.

We present three algorithms we have implemented on PEM: logistic regression, k-means and Apriori as examples.

6.1 Gradient descent

Gradient descent (GD) is a commonly-used algorithm for finding a local minimum of a function. Given an optimization problem, we want to minimize the loss function $\phi(\mathbf{x}, y, \mathbf{w})$, where $\mathbf{x}$ is the input, y is the expected output and $\mathbf{w}$ is the parameter. In GD, to estimate the optimal $\mathbf{w}$, we first calculate the gradient $\nabla\phi(\mathbf{w})$ on the dataset, then update $\mathbf{w}$ as $\mathbf{w}_{t+1} = \mathbf{w}_t - \eta_t \cdot \nabla\phi(\mathbf{w}_t)$ where η_t is the learning rate in iteration t. We set η_t to $O(\frac{1}{\sqrt{t}})$, like [30] does.

In PEM, servers add Laplace noise to the sum of the gradients from the clients. The formula for update becomes $\mathbf{w}_{t+1} = \mathbf{w}_t - \eta_t \frac{1}{\sum_{i=1}^{n} b_i}(\sum_{i=1}^{n} \nabla\phi_i(\mathbf{w}_t) + \sum_{j=1}^{m} \mathbf{r}_j)$ where b_i is the size of sub-dataset D_i held by client C_i, $\nabla\phi(\mathbf{w})$ is the sum of gradients on D_i and $\mathbf{r}_j$ is the Laplace noise. Usually, GD consists of multiple iterations. Assuming the number of iterations is T, $\mathbf{r}_j$ follows $\mathbf{Lap}^d(TL/\epsilon)$. Algorithm 1 shows the user functions for gradient descent.

6.2 k-means

k-means is a simple yet popular algorithm for clustering. In standard k-means, if we want to partition the d-dimensional records to l clusters, we first initialize l centroids $\mathbf{c}_1, \mathbf{c}_2, \ldots, \mathbf{c}_l$ of different clusters. In each iteration, we assign each record $\mathbf{x}$ to the cluster whose centroid $\mathbf{c}$ is the closest to it. Then we update each centroid to the mean of the records in its cluster: $\mathbf{c}_i = \frac{1}{|S_i|}\sum_{\mathbf{x}\in S_i} \mathbf{x}$ where $\mathbf{c}_i$ is the centroid of cluster i, S_i is the set of records belonging to cluster i and $|S_i|$ is the number of elements in S_i. In one iteration, $\mathbf{c}_1, \mathbf{c}_2, \ldots, \mathbf{c}_l$ should

```
**Function** *Init()*:
    Initialize the iteration count $T$ and the number of centroids $l$. Then initialize
    the rest parameters as follows: $\mathbf{c} = [\mathbf{c}_1, \mathbf{c}_2, \ldots, \mathbf{c}_l]$, $d_1 = \{1 \text{ to } ld\}$,
    $d_2 = \{ld+1 \text{ to } ld+l\}$, $\epsilon = [d_1 : \epsilon_1, d_2 : \epsilon_2]$ and $L = [d_1 : 2L_{\mathbf{x}}, d_2 : 2]$.

**Function** *mapInClient*$(\mathbf{x}, \mathbf{c})$:
    First we initialize the gradient $\mathbf{g} := [\mathbf{g}_1, \mathbf{g}_2, \cdots, \mathbf{g}_l, g_1, g_2, \cdots, g_l]$, where
    $\mathbf{g}_i = \mathbf{0}$ and $g_i = 0$ for each $i \in \{1 \text{ to } l\}$. Then we calculate
    $k = findCluster(\mathbf{x}, \mathbf{c})$ and set $\mathbf{g}_i = \mathbf{x}$ and $g_i = 1$. Return $\mathbf{g}$ as the result.

**Function** *reduceInAggregator*$(\mathbf{v})$:
    Represent $\mathbf{v}$ as $[\mathbf{v}_1, \mathbf{v}_2, \cdots, \mathbf{v}_l, v_1, v_2, \cdots, v_l]$ and update the centroid
    $\mathbf{c} = [\mathbf{v}_1/v_1, \mathbf{v}_2/v_2, \ldots, \mathbf{v}_l/v_l]$.
```
Algorithm 2: The user functions of k-means.

be updated all together, which means that the clients should jointly calculate $(\sum_{\mathbf{x}\in S_1} \mathbf{x}, \sum_{\mathbf{x}\in S_2} \mathbf{x}, \ldots, \sum_{\mathbf{x}\in S_l} \mathbf{x}, |S_1|, |S_2|, \ldots, |S_l|)$.

As the change of a record belonging to cluster i may change $\sum_{\mathbf{x}\in S_i} \mathbf{x}$ and $|S_i|$ simultaneously, the servers in PEM should not only add noise to perturb the centroids, but also add noise to perturb the number of records of each cluster. Specifically, we modify the formula for centroid update as $\mathbf{c}_i = \frac{1}{|S_i|+\sum_{j=1}^{m} \mathbf{r}_{j,2}}(\sum_{\mathbf{x}\in S_i} \mathbf{x} + \sum_{j=1}^{m} \mathbf{r}_{j,1})$ where $\mathbf{r}_{j,1}$ is the noise for the sum of records and $\mathbf{r}_{j,2}$ is the noise for the number of records in each cluster.

The records themselves and record-count obviously carry different amount of privacy information, and thus we add different levels of noises to the records dimensions and the count dimension. Using PEM's ϵ-spliting feature, we set the values of ϵ_1 and ϵ_2 as the users want, as long as their sum is ϵ. Denote L_1 to be the sensitivity of the sum of records $(\sum_{\mathbf{x}\in S_1} \mathbf{x}, \sum_{\mathbf{x}\in S_2} \mathbf{x}, \ldots, \sum_{\mathbf{x}\in S_l} \mathbf{x})$ and L_2 to be the sensitivity of the number of records in the clusters $(|S_1|, |S_2|, \ldots, |S_l|)$. It is easy to see that $L_1 = 2 \cdot L_{\mathbf{x}}$ and $L_2 = 2 \cdot 1$, where $L_{\mathbf{x}}$ is the maximum 1-norm of $\mathbf{x}$. There is a coefficient 2 here is because the change of an record may affect at most two clusters. Then we have $\mathbf{r}_{j,1} \sim \mathbf{Lap}^{ld}(TL_1/\epsilon_1)$ and $\mathbf{r}_{j,2} \sim \mathbf{Lap}^{l}(TL_2/\epsilon_2)$ where T is the number of iterations. Algorithm 2 shows the pseudo-code.

6.3 Apriori

Apriori is an algorithm for frequent itemset mining [17]. The *support* of a set of items (itemset) is the fraction of records containing the itemset respect to the database D. If the support of an itemset is above a preassigned minimum support, we call the itemset a *large itemset*. The target of frequent itemset mining is to find all the large itemsets. We denote I_k as the set of large itemsets of length k. To find large itemsets I_k, Apriori uses a function called *apriori-gen*, which takes I_{k-1} as an argument and generates candidates of k-itemsets denoted by I_k^*. Then Apriori calculates the count of each itemset in I_k^* and reserves the

```
Function Init():
    Set $l_m$ as the length of the longest record, $I_1$ as the set of atomic items, and
    minsup. Initialize the iteration count $T$ and the privacy parameter $\epsilon$. Then set
    the sensitivity function $L(k, I_k^*)$ accordingly.

Function mapInClient($\mathbf{x}, I_{k-1}$):
    Set $I_k^* = $ apriori-gen($I_{k-1}$) and return count($I_k^*, \mathbf{x}$).

Function reduceInAggregator($\mathbf{v}$):
    Here the parameter $\mathbf{v}$ is count($I_k^*, \mathbf{x}$), i.e. the output of mapInClient. First we
    set $I_k^* = $ apriori-gen($I_{k-1}$) and $I_k$ as the empty set $\{\}$. Then for each element
    $e \in I_k^*$, if $\mathbf{v}[e] > minsup$, put it in $I_k$. $I_k$ is the set of large items of length $k$.
```

Algorithm 3: The user functions of Apriori.

itemsets whose supports are above the minimum support. The set of reserved large k-itemsets from I_k^* is I_k.

As the change of a record will affect the count of itemsets, we should add noise to the count of the itemsets to preserve privacy. In PEM, in the situation where the dataset is distributed in multiple clients, *apriori-gen* can be done locally in each client given I_{k-1} and the count of each itemset in I_{k-1}. Filtering I_k^* to get I_k involves no raw data and thus can be done in the aggregator without privacy issue. We only need to add noise to the count of each itemset in I_k^*. Formally, we represent I_k^* as $[t_1, t_2, \ldots, t_{|I_k^*|}]$ where $t_i \in I_k^*$ is a candidate large itemset. In each iteration for generating I_k, $count(I_k^*, D) = [c_1, c_2, \ldots, c_{|I_k^*|}]$ is calculated, where c_i is the count of t_i respect to the dataset D. The sensitivity L of k-itemset counts is different for different k values. A record of length l contains at most $\binom{l}{k} = \frac{l!}{k!(l-k)!}$ itemsets of length k. The change of a record may affect the counts of at most $2 \cdot \binom{l}{k}$ itemsets. Then if we know the length l_m of the longest record beforehand, we can calculate the sensitivity of $count(I_k^*, D)$ as $L_k = \min(2 \cdot \binom{l_m}{k}, |I_k^*|)$. Algorithm 3 shows the user functions. We enable *dynamic L setting* and thus λ is modified in each iteration along with the sensitivity L_k.

7 Evaluation

7.1 Evaluation Setup

We empirically evaluate PEM in a cloud computing environment. We setup one aggregator and two servers ($m = 2$). Each of them runs on a virtual machine (VM) node. Each VM has 8 Xeon CPU cores, 16GB RAM and 10GE ethernet. We emulate $100 \sim 1000$ clients, each of which uses a sperate VM node with the same configuration. We use three open datasets from *UCI Machine Learning Repository* [1] for evaluation. We partition each dataset evenly onto the clients, emulating a horizontally partitioned dataset setting.

- The **Adult** dataset contains information of many people, including gender, age and salary. We clean the dataset and finally get 48,842 data records, each of which has 124 dimensions.
- The **Nursery** dataset is derived from a hierarchical decision model originally for nursery schools. There are 21,960 instances and 8 categorical attributes.
- The **Mushroom** dataset contains information of hypothetical samples corresponding to 23 species of gilled mushrooms. There are 8,124 instances with 22 attributes, each of which describes some shape information.

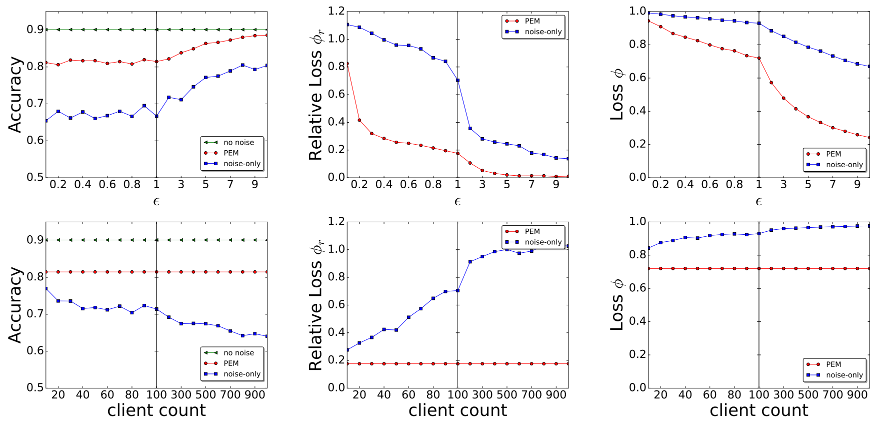
In our evaluation, we compare our algorithms with no-privacy versions. Meanwhile, many approaches add adequate noise on all clients, which means $100 \sim 1000$ Laplace noises in our experiment setting. We call the approach *noise-only approach*, and we show the comparison with ours. Finally we show the performance overhead of PEM.

7.2 Performance of Algorithms

Distributed gradient descent. We use *logistic regression* as the example of gradient descent shown in Algorithm 1. With the **Adult** dataset, we construct a logistic regression model to predict whether each person has high income ($>$ 50K) or not ($\leq$ 50K). We preprocess the dataset using *one-hot encoding* and the sensitivity L is 28 here. Each accuracy number is obtained from a 10-fold validation with 1000 iterations. Using different values of ϵ, we compare the model prediction accuracy using different approaches including no privacy, noise-only (100 clients) and PEM approach. Using different numbers of clients, we compare the model prediction accuracy with $\epsilon = 1$. Figure 2(a) shows the comparison. As expected, the noise-only approach needs to add significant noise, reducing the prediction accuracy significantly. Using the severs to add noise significantly lowers the loss in model accuracy, even for small ϵ. In comparison, noise-only approach with too many clients will cause too much noise.

Distributed clustering. We use k-means for clustering as shown in Algorithm 2, with the **Nursery** dataset. There are 8 categorical attributes in the dataset. We preprocess using *one-hot encoding* and finally get 27 binary attributes. It is easy to see that $L_{\mathbf{x}} = 8$, and thus $L_1 = 16$. In this experiment, we partition the records into five clusters, i.e., $l = 5$. We split ϵ equally to ϵ_1 and ϵ_2, i.e. $\epsilon_1 = \epsilon_2 = \frac{1}{2}\epsilon$. We calculate the loss function ϕ as $\phi(D, \mathbf{c}) = \sum_{i=1}^{l} \sum_{\mathbf{x} \in S_i} \|\mathbf{x} - \mathbf{c}_i\|^2$. As our goal is to compare the loss of different approaches, we use the relative loss to evaluate the performance of different approaches, which is calculated as $\phi_r(D, \mathbf{c}) = \frac{\phi(D,\mathbf{c}) - \phi_0}{\phi_0}$ where ϕ_0 is the loss of the approach without noise. Each training has 50 iterations. Figure 2(b) shows the result.

Distributed frequent itemset mining. We perform distributed frequent itemset mining on the **Mushroom** dataset using Algorithm 3. Considering the itemsets and the counts of them, we define the loss function as $\phi(\mathbf{I}) = \frac{\sum_k \sum_{t \in I_k \cup I_{k0}} |I_k(t) - I_{k0}(t)|}{\sum_k \sum_{t \in I_k \cup I_{0k}} |I_k(t) + I_{k0}(t)|}$ where $\mathbf{I} = [I_2, I_3, \ldots]$, I_{k0} is the set of large itemsets of length k generated without noise, and $I_k(t)$ is the (perturbed) count of t for $t \in I_k$. In this experiment, we



(a) LR, Adult (b) k-means, Nursery (c) Apriori, Mushroom

Fig. 2. Evaluation of algorithms using the according datasets with different values of ϵ (client count = 100) and different numbers of clients (fixing $\epsilon = 1$).

consider the large itemsets of length no more than 4, which means $\mathbf{I} = [I_2, I_3, I_4]$. We set *minsup* to 0.01. Figure 2(c) shows the result.

7.3 Scalability

The main computation workload of PEM comes from two parts: 1) computation overhead: vector addition and noise generation; and 2) communication overhead. We show that the overhead is small in most of the cases. First, we increase the number of clients from 0 to 10000 and Figure 3(a) shows the computation time for a scalar on different roles. The aggregator computation time is independent of the number of clients, as it only receives a vector from each server. Thus the aggregator is unlikely to become the bottleneck. The server workload increases linearly with the number of clients. Fortunately, we can increase each logical server capacity by adding more nodes. Figure 3(b) shows that the computation time is linear to the dimensionality of the feature vector, as expected. Finally, we record the overall overhead of vector addition, including the computation and communication overhead. Figure 3(c) shows that, compared with the communication time, the computation (i.e. vector addition) time can be negligible.

8 Conclusion and Future Work

We present PEM as a practical tradeoff among privacy, utility and computation overhead. PEM is practical in that 1) it has simple assumptions: it only requires a few semi-honest servers and there is no restriction on the dataset itself; 2) it supports a large range of common applications; 3) PEM entails low computation overhead and is scalable to a large number of clients; and 4) all user-visible

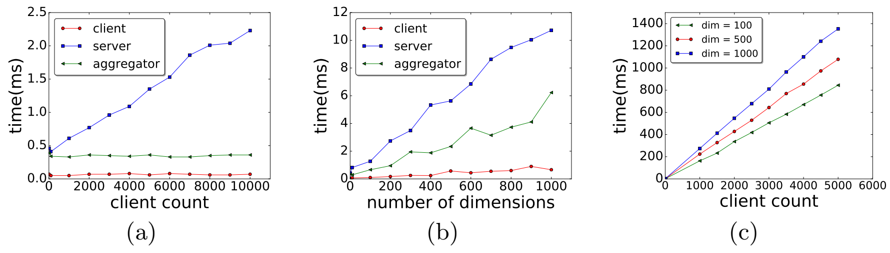


Fig. 3. (a) computation overhead w.r.t. number of clients. (b) computation overhead w.r.t the dimensionality of feature vectors. (c) overall overhead.

configuration parameters are intuitive and PEM automatically determines other internal parameters. Using algorithms on real datasets, we show that we can achieve the same level of privacy without the amount of accuracy degradation that previous systems suffer from. Our system also has low computation and communication cost, and thus is very practical.

There are lots of future directions along the lines of privacy. Firstly, we are extending our system to support more operations, such as handling vertically partitioned datasets. Secondly, we will provide the flexibility allowing the clients to choose different trust assumptions, so that the application programmers can choose their own tradeoffs. Last but not least, we will provide a permission system allowing different clients to see different levels of private content, like CryptDB [20] does, but on a much larger scale.

Acknowledgement

This research is supported in part by the National Natural Science Foundation of China (NSFC) grant 61532001, Tsinghua Initiative Research Program Grant 20151080475, MOE Online Education Research Center (Quantong Fund) grant 2017ZD203, and gift funds from Huawei and Ant Financial.

References

1. A. Asuncion and D.J. Newman. UCI machine learning repository, 2007.
2. A. Ben-David, N. Nisan, and B. Pinkas. FairplayMP: a system for secure multi-party computation. In *CCS '08*. ACM, 2008.
3. A. Berlioz, A. Friedman, M.A. Kaafar, R. Boreli, and S. Berkovsky. Applying differential privacy to matrix factorization. In *RecSys*. ACM, 2015.
4. R. Bhaskar, A. Bhowmick, V. Goyal, S. Laxman, and A. Thakurta. Noiseless database privacy. In *Advances in Cryptology–ASIACRYPT 2011*. Springer, 2011.
5. D. Bogdanov, S. Laur, and J. Willemson. Sharemind: A framework for fast privacy-preserving computations. In *ESORICS*. Springer, 2008.
6. M. Burkhart, M. Strasser, and D. e.t.c. Many. SEPIA: Privacy-preserving aggregation of multi-domain network events and statistics. *Network*, 1, 2010.

7. K. Chaudhuri, A.D. Sarwate, and K. Sinha. A near-optimal algorithm for differentially-private principal components. *JMLR*, 14(1), 2013.
8. Y. Duan. Differential privacy for sum queries without external noise. In *ACM Conference on Information and Knowledge Management (CIKM)*, 2009.
9. Y. Duan, J. Canny, and J. Zhan. P4P: Practical Large-scale Privacy-preserving Distributed Computation Robust Against Malicious Users. In *Proceedings of USENIX Security*. USENIX Association, 2010.
10. C. Dwork, F. McSherry, K. Nissim, and A. Smith. Calibrating noise to sensitivity in private data analysis. In *Theory of cryptography*. Springer, 2006.
11. C. Dwork, M. Naor, T. Pitassi, G.N. Rothblum, and S. Yekhanin. Pan-Private Streaming Algorithms. In *ICS*, 2010.
12. C. Dwork and A. Roth. The algorithmic foundations of differential privacy. *Foundations & Trends in Theoretical Computer Science*, 9(34), 2014.
13. A. Friedman, I. Sharfman, D. Keren, and A. Schuster. Privacy-Preserving Distributed Stream Monitoring. In *NDSS*, 2014.
14. Q. Jia, L. Guo, Z. Jin, and Y. Fang. Privacy-preserving data classification and similarity evaluation for distributed systems. In *ICDCS*. IEEE, 2016.
15. N. Li, W. Qardaji, and D. Su. On sampling, anonymization, and differential privacy or, k-anonymization meets differential privacy. In *Proceedings of the 7th ACM Symposium on Information, Computer and Communications Security*. ACM, 2012.
16. A. Machanavajjhala, D. Kifer, J. Gehrke, and M. Venkitasubramaniam. l-diversity: Privacy beyond k-anonymity. *ACM TKDD*, 1(1), 2007.
17. M.H. Margahny and A.A. Mitwaly. Fast algorithm for mining association rules. *Proc.int.conf.very Large Databases (VLDB)*, 23(3), 1994.
18. F.D. McSherry. Privacy integrated queries: an extensible platform for privacy-preserving data analysis. In *Proceedings of SIGMOD*. ACM, 2009.
19. M. Pathak, S. Rane, and B. Raj. Multiparty differential privacy via aggregation of locally trained classifiers. In *NIPS*, 2010.
20. R.A. Popa, C.M.S. Redfield, N. Zeldovich, and H. Balakrishnan. Cryptdb: Protecting confidentiality with encrypted query processing. In *ACM SOSP*, 2011.
21. V. Rastogi and S. Nath. Differentially private aggregation of distributed time-series with transformation and encryption. In *Proceedings of SIGMOD*. ACM, 2010.
22. R. Sarathy and K. Muralidhar. Evaluating Laplace Noise Addition to Satisfy Differential Privacy for Numeric Data. *Transactions on Data Privacy*, 4(1), 2011.
23. A. Shamir. How to share a secret. *Communications of the ACM*, 22(11), 1979.
24. R. Shokri and V. Shmatikov. Privacy-preserving deep learning. In *ACM Conference on Computer and Communications Security*, 2015.
25. D. Su, J. Cao, N. Li, E. Bertino, and H. Jin. Differentially private k-means clustering. In *Proceedings of CODASPY*. 2016.
26. H. Takabi, S. Koppikar, and S. T. Zargar. Differentially private distributed data analysis. In *Collaboration and Internet Computing (CIC), 2016 IEEE 2nd International Conference on*, pages 212–218. IEEE, 2016.
27. K. Xu, H. Yue, L. Guo, Y. Guo, and Y. Fang. Privacy-preserving machine learning algorithms for big data systems. In *Distributed Computing Systems*, 2015.
28. Andrew C Yao. Protocols for secure computations. In *Foundations of Computer Science, 1982. SFCS'08. 23rd Annual Symposium on*. IEEE, 1982.
29. N. Zhang, M. Li, and W. Lou. Distributed data mining with differential privacy. In *Communications (ICC), 2011 IEEE International Conference on*, pages 1–5. IEEE, 2011.
30. M. Zinkevich. Online convex programming and generalized infinitesimal gradient ascent. *ICML*, 2003.