Faster PCA and Linear Regression through Hypercubes in HElib ⋆

Deevashwer Rathee[1], Pradeep Kumar Mishra[2], and Masaya Yasuda[3]

[1] Department of Computer Science and Engineering,
Indian Institute of Techonology (BHU) Varanasi 221005, India.
deevashwer.student.cse15@iitbhu.ac.in
[2] Graduate School of Mathematics, Kyushu University,
744 Motooka Nishi-ku, Fukuoka 819-0395, Japan.
p-mishra@math.kyushu-u.ac.jp
[3] Institute of Mathematics for Industry, Kyushu University,
744 Motooka Nishi-ku, Fukuoka 819-0395, Japan.
yasuda@imi.kyushu-u.ac.jp

Abstract. The significant advancements in the field of homomorphic encryption have led to a grown interest in securely outsourcing data and computation for privacy critical applications. In this paper, we focus on the problem of performing secure predictive analysis, such as principal component analysis (PCA) and linear regression, through *exact* arithmetic over encrypted data. We improve the plaintext structure of Lu et al.'s protocols (from NDSS 2017), by switching over from *linear* array arrangement to a *two*-dimensional hypercube. This enables us to utilize the SIMD (Single Instruction Multiple Data) operations to a larger extent, which results in improving the space and time complexity by a factor of matrix dimension. We implement both Lu et al.'s method and ours for PCA and linear regression over `HElib`, a software library that implements the Brakerski-Gentry-Vaikuntanathan (BGV) homomorphic encryption scheme. In particular, we show how to choose optimal parameters of the BGV scheme for both methods. For example, our experiments show that our method takes 45 seconds to train a linear regression model over a dataset with 32k records and 6 numerical attributes, while Lu et al.'s method takes 206 seconds.

Keywords: Leveled homomorphic encryption · PCA · Linear regression · Hypercube arrangement.

1 Introduction

In the recent years, the cloud computing paradigm has grown in popularity as an economical solution for outsourcing data and computation. It enables ubiquitous access to shared storage and computational resources over the internet, and hence it is being adopted by many organizations. However, storing data on the cloud raises security and privacy concerns, since the cloud service provider can not only access the data but also share it with other parties. This makes it difficult to keep control of the data for applications that have privacy as a principal concern. A great solution to address these concerns is homomorphic encryption that enables computation on encrypted data. Using homomorphic encryption, a client can upload its sensitive data on the cloud in encrypted format, and the cloud can operate on that data without ever decrypting the data.

The concept of homomorphic encryption was first proposed by Rivest et al. in 1978 [19]. But the first construction of a *fully homomorphic encryption (FHE)* scheme, that allows arbitrary computation on ciphertexts, came around 30 years later through the ground-breaking work of Gentry [10]. Gentry's work showed that it is theoretically plausible to do any number of operations on ciphertexts, but the scheme was too inefficient to be practical yet for any application. Since then, a lot of work (e.g., [20,4,17,3,21]) has been done that saw major improvements in both theory and practice. However, the currently known FHE schemes are still regarded as impractical for real applications. On the other hand, there are some partially homomorphic schemes ([18,1]) available that are practical, but they offer limited functionality. At present, *somewhat* homomorphic encryption (SwHE) and its *leveled* improvement (called *leveled*-FHE), that allow a limited number of operations, have attracted a lot of attention from various communities. Despite this limitation, they are applicable in various scenarios and provide reasonable performance (e.g., see [17,11,23,24,5]). Since the complexity of algorithms grows linearly with circuit depth in *leveled*-FHE, as opposed to exponential growth in SwHE, it is more suitable for applications requiring larger circuit depth.

In this paper, we use leveled-FHE for performing predictive statistics such as PCA and linear regression, through *exact* arithmetic over encrypted data (cf., approximate arithmetic). We choose a variant [11] of the leveled BGV

scheme [3] as our cryptosystem, and leverage the software library `HElib` [14] for its implementation. Many works have been proposed to address the problem of performing statistical analysis over encrypted data (see [12,22,2,16]). The solution of [2] only addresses model evaluation, while the methods of [12,22] can only perform statistics on data with very low dimension. Recently, Lu et al. [16] proposed a solution to perform PCA and linear regression on data with up to 20 numerical attributes. Their solution achieves much better results than any previous work by utilizing the linear array structure of the plaintext slots. Our main aim is to improve Lu et al.'s method for further efficiency. Our contribution in this paper is two-fold:

1. Firstly, we improve upon the plaintext structure used in Lu et al.'s method [16] by utilizing a *two*-dimensional hypercube structure. This gives us benefits in terms of both space and time complexity. As a result, we reduce the complexity of their methods by a factor of data dimension. In the process, we also develop some general-purpose procedures for matrix operations that are significantly faster than the previously known solutions in `HElib`.
2. Secondly, we address the problem of optimal parameter selection in `HElib`, which is non-trivial for our application and was not discussed in [16] carefully. In this paper, we describe how to choose optimal parameters for our method as well as Lu et al.'s. We also compare the performance of our method with Lu et al.'s method for performing PCA and linear regression over `HElib`.

Notation We use the notation $\mathsf{ord}_G(g)$ to denote the order of an element g in the group G. We switch to $\mathsf{ord}(g)$ for concise representation whenever there's no confusion. We also use $[\cdot]_q$ to denote reduction modulo q in the interval $(-q/2, q/2]$. We denote by χ a discrete Gaussian distribution with zero mean and variance σ^2. We use $[n]$ to denote the set $\{0, \ldots, n-1\}$. The row vectors of a matrix $\mathbf{X}$ are represented by $\mathbf{x}_i^\mathsf{T}$. The matrix entries are represented by non-bold lowercase roman letter with subscripts e.g. $x_{i,j}$. We denote the set of primes by $\mathbb{P}$.

2 Preliminaries

2.1 The BGV Cryptosystem

In this work, we use the Ring-LWE variant of the BGV scheme [11], which is defined over the polynomial ring $\mathbb{A} = \mathbb{Z}[X]/\Phi_m(X)$, where $\Phi_m(X)$ is the m-th cyclotomic polynomial.

Plaintext Space The plaintext space is defined by the ring $\mathbb{A}_t = \mathbb{A}/t\mathbb{A}$, where t is a prime. Under modulo t, the polynomial $\Phi_m(X)$ factors into ℓ irreducible polynomials, each of degree $s = \phi(m)/\ell$ such that $\Phi_m(X) = F_1(X) \cdot F_2(X) \cdots F_\ell(X) \pmod t$. Each factor $F_i(X)$ corresponds to a plaintext slot (each slot is isomorphic to the finite field $\mathbb{F}_{t^s}$) and the following isomorphism holds:

$$\mathbb{A}_t \simeq \mathbb{Z}_t[X]/F_1(X) \times \cdots \times \mathbb{Z}_t[X]/F_\ell(X) \simeq \mathbb{F}_{t^s} \times \cdots \times \mathbb{F}_{t^s}$$

Therefore, a polynomial $a(X) \in \mathbb{A}_t$ can be represented as the vector $(a \bmod F_i)_{i=1}^\ell$. `HElib` provides high level interfaces that allow conversion between a vector of plaintext values $(a^{(i)})_{i=1}^\ell \in (\mathbb{F}_{t^s})^\ell$ and a polynomial $a(X) \in \mathbb{A}_t$ (native plaintext space of the BGV scheme) through encoding and decoding routines. Hence, given two polynomial encodings $a(X) = \mathsf{Encode}\big((a_i)_{i=1}^\ell\big)$ and $b(X) = \mathsf{Encode}\big((b_i)_{i=1}^\ell\big)$, we have:

$$\begin{cases} \mathsf{Decode}(a + b \bmod (t, \Phi_m)) = (a_i + b_i \bmod (t, F_i))_{i=1}^\ell \\ \mathsf{Decode}(a \cdot b \bmod (t, \Phi_m)) = (a_i \cdot b_i \bmod (t, F_i))_{i=1}^\ell \end{cases}$$

Each slot in the vector representation corresponds to a unique conjugacy class of $\mathbb{Z}_m^*/\langle t \rangle$. An isomorphism exists between the polynomial ring and the vector of plaintext slots, and the plaintext slots are isomorphic to one another. This imparts automorphic mappings of the form $\kappa : a(X) \to a(X^k)$, where $k \in \mathbb{Z}_m^*/\langle t \rangle$, that allow data movement among the slots.

The structure of $\mathbb{Z}_m^*/\langle t \rangle$ can be represented by a set of generators $\{f_1, \ldots, f_n\}$, where the order of f_i in $\mathbb{Z}_m^*/\langle t, f_1, \ldots, f_{i-1} \rangle$ is m_i. Each slot has a unique representative in the slot-index representative set $\mathrm{T} = \left\{ \prod_{i=1}^n f_i^{e_i}, 0 \le e_i \le m_i - 1 \right\}$ which can be indexed by the vector of exponents $(e_1, \ldots, e_n)$. Therefore, the plaintext slots can be mapped to an n-dimensional *hypercube*, whose i-th dimension is of size m_i. The i-th dimension is labelled as a *good* dimension if $m_i = \mathsf{ord}_{\mathbb{Z}_m^*}(f_i)$, otherwise it is labelled as a *bad* dimension. *Good* dimensions lead to more efficient data movement (see [14, Section 4] for more details).

Data Movement The basic data movement operation is rotation and all other operations that move data depend on it (See [15, Section 4] for more details). Rotation comes in two flavours depending on the arrangement of plaintext slots. There are two ways to arrange the plaintext slots:

- *Hypercube* arrangement: As described before, the plaintext slots natively assume the structure of an n-dimensional *hypercube* $\mathbf{V}$ and each slot is indexed by some $\mathbf{e} = (e_1, \ldots, e_n)$, where e_i ranges over $[m_i]$. The dimensions of this *hypercube* can be rotated independently through the $\mathsf{rotate1D}$ procedure. A call to "$\mathtt{rotate1D}(\mathbf{V}, i, k)$" will move the content of slot $(e_1, \ldots, e_i, \ldots, e_n)$ to the slot $(e_1, \ldots, e_i + k, \ldots, e_n)$ of $\mathbf{V}$, where addition is done modulo m_i.
- *Linear Array* arrangement: In this arrangement, the plaintext slots are presented to the application in the form of a *linear array*. The number of elements in the array is $\ell = |\mathbb{Z}_m^*/\langle t \rangle|$. It is made possible by ordering the slots lexicographically over the vector of indices of *hypercube*. The position k of this *linear array* is identified by $\mathbf{e}^{(k)} = (e_1^{(k)}, \ldots, e_n^{(k)})$, where $k \in [\ell]$. A call to "$\mathtt{rotate}(\mathbf{v}, k)$" will move the content of slot j (resp., $\mathbf{e}^{(j)}$) to the slot $j + k$ (resp., $\mathbf{e}^{(j+k)}$) of $\mathbf{v}$, where addition is done modulo ℓ, thereby rotating the array by k positions. The addition over the respective index vectors can be seen as addition with carry over n-digit numbers, where i-th digit has base m_i and n-th digit is least significant.

We have other high level data movement operations such as total-sum and replicate. As mentioned before, these operations depend on rotation. Hence, they have different impact depending on the type of slot arrangement. They are defined as follows:

- For *hypercube* arrangement: Given an n-dimensional *hypercube* $\mathbf{V} = (\mathbf{V}[e_1, \ldots, e_n])$, where e_j ranges over $[m_j]$, the function "$\mathsf{TS1D}(\mathbf{V}, j)$" outputs:
$$\mathbf{W}_{\mathsf{TS}}[e_1, \ldots, e_j, \ldots, e_n] = \textstyle\sum_{k=0}^{m_j - 1} \mathbf{V}[e_1, \ldots, k, \ldots, e_n].$$
Let $\mathbf{e}^{(i)}$ be a vector of indices excluding the index for j-th dimension i.e. $\mathbf{e}^{(i)} = (e_1^{(i)}, \ldots, e_{j-1}^{(i)}, e_{j+1}^{(i)}, \ldots, e_n^{(i)})$, where i ranges over $[\ell/m_j]$. Given a set $\{k_i\}_{i=0}^{\ell_j - 1}$ with $\ell_j = \ell/m_j$ indices, the function "$\mathsf{replicate1D}\big(\mathbf{V}, j, \{k_i\}_{i=0}^{\ell_j-1}\big)$" outputs:
$$\mathbf{W}_{\mathsf{replicate}}[e_1^{(i)}, \ldots, e_j, \ldots, e_n^{(i)}] = \mathbf{V}[e_1^{(i)}, \ldots, k_i, \ldots, e_n^{(i)}],$$
where e_j ranges over $[m_j]$. Both "$\mathsf{TS1D}$" and "$\mathsf{replicate1D}$" have a running time of $O(\log m_j)$ rotations and additions.
- For *linear array* arrangement: Given a vector $\mathbf{v} = (\mathbf{v}[i])_{i=0}^{\ell-1}$, the functions "$\mathsf{TS}(\mathbf{v})$" and "$\mathsf{replicate}(\mathbf{v}, k)$" respectively output:
$$\mathbf{w}_{\mathsf{TS}}[j] = \textstyle\sum_{k=0}^{\ell-1} \mathbf{v}[k] \text{ and } \mathbf{w}_{\mathsf{replicate}}[j] = \mathbf{v}[k].$$
These functions have a running time of $O(\log \ell)$ rotations and additions.

For details on underlying algorithms, see [15, Section 4].

Ciphertext Space The ciphertext space is defined by vectors over the ring $\mathbb{A}_q = \mathbb{A}/q\mathbb{A}$, where q is an odd modulus that changes over homomorphic evaluation. The scheme with level parameter L is parametrized by a chain of moduli $q_0 < q_1 < \cdots < q_{L-1}$ and freshly encrypted ciphertexts are defined over $\mathbb{A}_{q_{L-1}}$. Ciphertexts defined over $\mathbb{A}_{q_l}$ are called level-l ciphertexts.

The modulus q_l (also called level-l modulus) is defined as the product of $l+1$ small primes p_i of same size chosen such that $m \equiv 1 \bmod p_i$ (`HElib` provides an additional half-prime optimization. See [11, Section 3] for details). This is done so that for all i, $\Phi_m(X)$ factors linearly under modulo p_i.

For efficient arithmetic over ciphertexts, a polynomial $a(X) \in \mathbb{A}_{q_l}$ (in coefficient representation) is represented as a $(l+1) \times \phi(m)$ matrix $\mathsf{DoubleCRT}^l(a)$ (in evaluation representation), whose (i,j)-th entry is the evaluation of $a(X)$ at j-th root of $\Phi_m(X)$ modulo p_i. Addition and multiplication in $\mathbb{A}_{q_l}$ is done entry-wise modulo the appropriate primes p_i. For ease of representation, in the rest of description we ignore the double CRT representation of polynomials lying in ciphertext space and describe the scheme as if we were operating on polynomials directly.

Key Generation Given a parameter w, a random low norm polynomial $\mathfrak{s} \in \mathbb{A}_{q_{L-1}}$ having coefficients in $\{-1, 0, 1\}$ is chosen such that its Hamming weight is exactly w. Secret key is set as $\mathsf{sk} = (1, -\mathfrak{s})$. To generate public key, a uniformly random polynomial $a \in \mathbb{A}_{q_{L-1}}$ is chosen and a low-norm error polynomial $e \in \mathbb{A}_{q_{L-1}}$ is sampled from χ. The public key is set as $\mathsf{pk} = (a, b)$, where $b = [a \cdot \mathfrak{s} + t \cdot e]_{q_{L-1}}$.

In addition to this, the public key also has key switching matrices of the form $W[\mathfrak{s}' \to \mathfrak{s}]$ that transform a ciphertext decryptable by $\mathfrak{s}'$ into a ciphertext decryptable by $\mathfrak{s}$. Specifically, we have $W[\mathfrak{s}^2 \to \mathfrak{s}]$ and $W[\mathfrak{s}(X^k) \to \mathfrak{s}]$, where $k \in \mathbb{Z}_m^*/\langle t \rangle$, which are used in multiplication and data movement respectively to get "canonical" ciphertexts of the form $\mathbf{c} = (c_0, c_1) \in (\mathbb{A}_{q_l})^2$, that are decryptable by a secret key of the form $\mathsf{sk} = (1, -\mathfrak{s})$.

Encryption To encrypt a polynomial $m \in \mathbb{A}_t$, a random low norm polynomial $r \in \mathbb{A}_{q_{L-1}}$ having coefficients in $\{-1, 0, 1\}$ is chosen and two low-norm error polynomials $e_0, e_1 \in \mathbb{A}_{q_{L-1}}$ are sampled from χ. The ciphertext is computed as: $\mathbf{c} = (c_0, c_1) = \mathsf{Enc}(m, \mathsf{pk}) = (b \cdot r + t \cdot e_0 + m, a \cdot r + t \cdot e_1) \in (\mathbb{A}_{q_{L-1}})^2$.

Decryption To decrypt a level-l ciphertext, we first compute the noise polynomial $m' = [\langle \mathbf{c}, \mathsf{sk} \rangle]_{q_l} = [c_0 - c_1 \cdot \mathfrak{s}]_{q_l}$. Then, the message $m \in \mathbb{A}_t$ is recovered by computing $m' \bmod t = \mathsf{Dec}(\mathbf{c}, \mathsf{sk})$. For the decryption procedure to work, the norm of noise polynomial m' should be sufficiently smaller than q_l.

Homomorphic Operations and Noise Control The noise associated with a ciphertext should be considerably small compared to the ciphertext modulus to successfully recover the message. But, homomorphically operating on ciphertexts leads to an increase in the noise term. The freshly encrypted ciphertext is valid w.r.t. the largest modulus q_{L-1}. When the noise term grows too much, we modulus-switch to a ciphertext that is valid w.r.t. smaller moduli to decrease the noise magnitude. As we operate on a ciphertext, we need to switch to smaller moduli until the ciphertext is defined over $\mathbb{A}_{q_0}$. Beyond this point, we can not modulus-switch any further, and if the noise grows too much now, the ciphertext will be rendered useless.

In `HElib`, every l-th level ciphertext is represented as the tuple $\mathbf{c} = ((c_0, c_1), l, \nu)$, where ν is an estimate of noise magnitude of the ciphertext. This estimate helps the library in automatically switching to lower levels when needed (for details on noise estimate, see [14, Section 3.1.4]). The homomorphic operations are defined as follows:

- *Addition*: Before *adding* ciphertexts $\mathbf{c} = ((c_0, c_1), l, \nu)$, $\mathbf{c}' = ((c'_0, c'_1), l', \nu')$ encrypting messages $m, m' \in \mathbb{A}_t$ respectively, we bring them to the same level l'' (if $l \neq l'$) by reducing the larger one modulo the smaller of the two moduli if the noise doesn't overflow, otherwise we modulus-switch the larger one to the smaller level. Then, we add the ciphertexts to get:

$$\mathbf{c}_{\mathsf{add}} = (([c_0 + c'_0]_{q_{l''}}, [c_1 + c'_1]_{q_{l''}}), l'', \nu + \nu').$$

- *Multiplication*: Given two ciphertexts $\mathbf{c} = ((c_0, c_1), l, \nu)$, $\mathbf{c}' = ((c'_0, c'_1), l', \nu')$ encrypting $m, m' \in \mathbb{A}_t$, we first perform modulus switching on them to bring their noise magnitude below a preset constant (refer [11, Appendix C.2] for details). Then, we reduce the larger one modulo the smaller of the two moduli to bring them to the same level l''. Having brought the ciphertexts to the same level, we perform their tensor product to get:

$$\mathbf{c}'_{\mathsf{mult}} = ((c_0 c'_0, c_0 c'_1 + c'_0 c_1, c_1 c'_1), l'', \nu\nu').$$

  $\mathbf{c}'_{\mathsf{mult}}$ is a ciphertext decryptable by $\mathsf{sk}' = (1, -\mathfrak{s}, \mathfrak{s}^2)$. We perform key-switching on $\mathbf{c}'_{\mathsf{mult}}$ using $W[\mathfrak{s}^2 \to \mathfrak{s}]$ to get a canonical ciphertext $\mathbf{c}_{\mathsf{mult}} = ((c''_0, c''_1), l'', \nu'')$ with noise magnitude ν''.
  We can also multiply the ciphertext with a scalar. So, given a scalar $\alpha \in \mathbb{A}_t$ and a ciphertext $\mathbf{c} = ((c_0, c_1), l, \nu)$ encrypting $m \in \mathbb{A}_t$, we can multiply them to get the ciphertext $\mathbf{c}_{\mathsf{scalar-mult}} = ((\alpha \cdot c_0, \alpha \cdot c_1), l, \nu')$ encrypting $\alpha \cdot m \in \mathbb{A}_t$. The noise estimate ν' is computed as $\nu' = \nu \cdot \nu_\alpha$, where ν_α is the maximum norm possible for the scalar α.
- *Automorphism*: As mentioned in Section 2.1, data movement is made possible by automorphic mappings of the form $\kappa : a(X) \to a(X^k)$, where $k \in \mathbb{Z}^*_m / \langle t \rangle$. Given a ciphertext $\mathbf{c} = ((c_0, c_1), l, \nu)$ encrypting $m \in \mathbb{A}_t$, by applying automorphism (note that the noise doesn't change), we get:

$$\mathbf{c}'_{\mathsf{auto}} = ((\kappa(c_0), 0, \kappa(c_1)), l, \nu).$$

  $\mathbf{c}'_{\mathsf{auto}}$ is a ciphertext decryptable by $\mathsf{sk}' = (1, -\mathfrak{s}, -\kappa(\mathfrak{s}))$. We perform key-switching on $\mathbf{c}'_{\mathsf{auto}}$ using $W[\mathfrak{s}(X^k) \to \mathfrak{s}]$ to get a canonical ciphertext $\mathbf{c}_{\mathsf{auto}} = ((c'_0, c'_1), l, \nu')$ with noise magnitude ν'.

The BGV scheme has a ring homomorphism between the plaintext and ciphertext space. Therefore, we have:

$$\left.\begin{array}{l} \mathsf{Dec}(\mathbf{c}_{\mathsf{add}}, \mathsf{sk}) = m + m' \\ \mathsf{Dec}(\mathbf{c}_{\mathsf{mult}}, \mathsf{sk}) = m \cdot m' \\ \mathsf{Dec}(\mathbf{c}_{\mathsf{auto}}, \mathsf{sk}) = \kappa(m) \end{array}\right\} \in \mathbb{A}_t.$$

2.2 Principal Component Analysis (PCA)

PCA is a dimensionality-reduction tool, that takes a number of possibly correlated variables and transforms them into a smaller number of uncorrelated variables, while retaining most of the information. These uncorrelated variables are

called principal components, and they do not necessarily have a physical interpretation. PCA can be seen as a rotation of the original axes to a new set of orthogonal axes, that are aligned in the direction of maximum variation.

Let $\mathbf{X}$ be a data matrix, with N records and d numerical attributes, which is defined as:

$$\mathbf{X} = \underbrace{\left.\begin{pmatrix} — & \mathbf{x}_0^\mathsf{T} & — \\ & \vdots & \\ — & \mathbf{x}_{N-1}^\mathsf{T} & — \end{pmatrix}\right\}}_{d \text{ attributes}} N \text{ records}$$

The problem of finding the principal components of $\mathbf{X}$ is the same as finding the eigen vectors of its covariance matrix $\boldsymbol{\Sigma}$, which is defined as:

$$\boldsymbol{\Sigma} = \frac{1}{N}\mathbf{X}^\mathsf{T}\mathbf{X} - \boldsymbol{\mu}\boldsymbol{\mu}^\mathsf{T}, \text{ where } \boldsymbol{\mu}^\mathsf{T} = \frac{1}{N}\sum_{i=0}^{N-1}\mathbf{x}_i^\mathsf{T}.$$

To find the eigen vectors of $\boldsymbol{\Sigma}$, a technique called the Power Method is used. PowerMethod is an iterative technique that is used to find the dominant eigen-vector of a matrix, and is described in Algorithm 1. The intuition behind the

Algorithm 1 PowerMethod$(\boldsymbol{\Sigma}, T)$

Input:
- $\boldsymbol{\Sigma}$: covariance matrix of $\mathbf{X}$
- T: number of iterations

Output:
- $\mathbf{u}_1, \lambda_1$: dominant eigen-vector of $\boldsymbol{\Sigma}$ and its eigen-value

1: Choose a random vector $\mathbf{v}^{(0)}$ of size d
2: **for** $i = 1$ to T **do**
3: $\quad \mathbf{v}^{(i)} = \boldsymbol{\Sigma}\mathbf{v}^{(i-1)}$
4: **end for**
5: return $\mathbf{u}_1 = \mathbf{v}^{(T)}/\|\mathbf{v}^{(T)}\|$ and $\lambda_1 = \|\mathbf{v}^{(T)}\|/\|\mathbf{v}^{(T-1)}\|$

Power Method is that by multiplying the initial vector $\mathbf{v}$ repeatedly by $\boldsymbol{\Sigma}$, $\mathbf{v}$ is stretched in the direction of the $\boldsymbol{\Sigma}$'s dominant eigen-vector $\mathbf{u}_1$, to the point where the vector lies almost entirely in the direction of $\mathbf{u}_1$. Power Method can also be used to find the k-th dominant eigen-vector of $\boldsymbol{\Sigma}$ by using the EigenShift procedure, which is described in Algorithm 2. For input k, the Eigen Shift procedure outputs a matrix $\boldsymbol{\Sigma}_k$ whose most dominant eigen-vector is

Algorithm 2 EigenShift$(\boldsymbol{\Sigma}, \{\mathbf{u}_i, \lambda_i\}_{i=1}^{k-1})$

Input:
- $\boldsymbol{\Sigma}$: covariance matrix of $\mathbf{X}$
- $\{\mathbf{u}_i, \lambda_i\}$: i-th dominant eigen-vector of $\boldsymbol{\Sigma}$ and its eigen-value

Output:
- $\boldsymbol{\Sigma}_k$: k-shifted covariance matrix of $\mathbf{X}$

1: $\boldsymbol{\Sigma}_1 = \boldsymbol{\Sigma}$
2: **for** $i = 1$ to $k - 1$ **do**
3: $\quad \boldsymbol{\Sigma}_{i+1} = \boldsymbol{\Sigma}_i - \lambda_i \mathbf{u}_i \mathbf{u}_i^\mathsf{T}$
4: **end for**
5: return $\boldsymbol{\Sigma}_k$

$\mathbf{u}_k$, where $\mathbf{u}_k$ is the k-th most dominant eigen-vector of $\boldsymbol{\Sigma}$ and λ_k is its associated eigen-value. Therefore, combining these two methods, the K dominant eigen-vectors (and their associated eigen-values) of the covariance matrix $\boldsymbol{\Sigma}$ can be found to get the K principal components of $\mathbf{X}$.

2.3 Linear Regression (LR)

Linear Regression is a statistical procedure that models a relationship between the target (dependent) variable y and one or more input (independent) variables $\mathbf{x}$ using a linear equation. Given a dataset $(\mathbf{X}, \mathbf{y}) = \{(\mathbf{x}_i^\mathsf{T}, y_i)\}_{i=0}^{N-1} \in \mathbb{Z}^{N \times d}$,

where $\mathbf{x}_i^\mathsf{T}$ are the input variables and y_i is the target variable, the aim is to find a vector of weights $\mathbf{w}$ such that $\mathbf{y} \approx \mathbf{X}\mathbf{w}$. We obtain $\mathbf{w}$ by minimizing the least-squares error defined by:

$$\mathbf{w} = \arg\min_{\mathbf{w}^*} \frac{1}{N} \sum_{i=1}^{N} \|y_i - \mathbf{x}_i^\mathsf{T}\mathbf{w}^*\|^2.$$

The most popular solution is an iterative technique called Gradient Descent. Gradient Descent is useful when we are operating on data with very large dimension d. For smaller dimensions, there is a one-step analytical solution called Normal Equation method, which computes $\mathbf{w}$ as:

$$\mathbf{w} = (\mathbf{X}^\mathsf{T}\mathbf{X})^{-1}\mathbf{X}^\mathsf{T}\mathbf{y}.$$

We leverage an iterative division-free technique from [13] for matrix inversion, which only involves matrix multiplications and additions. This technique, described in Algorithm 3, allows us to compute inverse of encrypted matrices. There can be different choices for the input α to the function $\mathsf{InvertMatrix}$, but taking α close to the largest eigen value

Algorithm 3 $\mathsf{InvertMatrix}(\mathbf{M}, \alpha, T)$

Input:
- $\mathbf{M}$: given matrix
- α: a constant close to the dominant eigen-value of $\mathbf{M}$
- T: number of iterations

Output:
- $\mathbf{M}^{-1}$: inverse of matrix $\mathbf{M}$

1: $\mathbf{A}^{(0)} = \mathbf{M}, \mathbf{R}^{(0)} = \mathbf{I}$ ▷ $\mathbf{I}$ is the identity matrix
2: $\alpha^{(0)} = \alpha$
3: **for** $i = 1$ to T **do**
4: $\quad \mathbf{R}^{(i)} = 2\alpha^{(i-1)}\mathbf{R}^{(i-1)} - \mathbf{R}^{(i-1)}\mathbf{A}^{(i-1)}$
5: $\quad \mathbf{A}^{(i)} = 2\alpha^{(i-1)}\mathbf{A}^{(i-1)} - \mathbf{A}^{(i-1)}\mathbf{A}^{(i-1)}$
6: $\quad \alpha^{(i)} = \alpha^{(i-1)}\alpha^{(i-1)}$
7: **end for**
8: return $\mathbf{M}^{-1} = \mathbf{R}^{(T)}/\alpha^{(T)}$

of matrix $\mathbf{M}$ ensures a stable quadratic convergence to $\mathbf{M}^{-1}$.

3 Matrix Operations

In the previous section, we described statistical procedures that we will compute securely in the cloud. It is evident that these procedures require matrix operations. In this section, we describe the method used in [16] and propose a new method for matrix operations over encrypted matrices. We also draw a complexity comparison between the two methods in terms of homomorphic operations.

3.1 Lu et al.'s Method (Linear Array Arrangement)

Lu et al. proposed a matrix multiplication method in [16] that assumes a linear array arrangement of plaintext slots, and works along the lines of column-order matrix-vector multiplication method described in [15, Section 4.3]. Given the one-dimensional array arrangement, a vector can be packed directly in the plaintext slots (followed by zeros if the length of vector is less than ℓ) and encrypted. Their proposed techniques work on matrices encrypted in row-order i.e. each row vector is encrypted separately. The encryption of a vector $\mathbf{u} \in \mathbb{Z}_t^d$ and a matrix $\mathbf{X} \in \mathbb{Z}_t^{d \times d}$ are represented as $\mathsf{ct}(\mathbf{u})$ and $\mathsf{CT}(\mathbf{X}) = \{\mathsf{ct}(\mathbf{x}_i^\mathsf{T})\}_{i=0}^{d-1}$ respectively.

Outer Product of Vectors Given ciphertexts of vectors $\mathbf{u}$ and $\mathbf{v}$, to compute their outer product homomorphically, the function described in Algorithm 4 is used.

Matrix-Vector Multiplication Lu et al. have used the column-order matrix vector multiplication method of [15] for matrix-vector multiplication. They took advantage of the fact that their application only requires dealing with *symmetric* matrices. Therefore, they could use this method, described in Algorithm 5, even on matrices encrypted in row-order.

Algorithm 4 $\mathsf{OuterProduct}(\mathsf{ct}(\mathbf{u}), \mathsf{ct}(\mathbf{v}))$

Input:
- $\mathsf{ct}(\mathbf{u}), \mathsf{ct}(\mathbf{v})$: ciphertexts of vectors $\mathbf{u}, \mathbf{v} \in \mathbb{Z}_t^d$

Output:
- $\mathsf{CT}(\mathbf{uv})$: ciphertexts for rows of matrix $\mathbf{uv} \in \mathbb{Z}_t^{d \times d}$

1: **for** $i = 0$ to $d - 1$ **do**
2: $\quad \mathsf{ct}(\mathbf{uv}_i^\mathsf{T}) = \mathsf{replicate}(\mathsf{ct}(\mathbf{u}), i) \cdot \mathsf{ct}(\mathbf{v})$
3: **end for**
4: return $\mathsf{CT}(\mathbf{uv})$ ▷ $\mathsf{CT}(\mathbf{uv}) = \{\mathsf{ct}(\mathbf{uv}_i)\}_{i=0}^{d-1}$

Algorithm 5 $\mathsf{MatVecMul}(\mathsf{CT}(\mathbf{X}), \mathsf{ct}(\mathbf{u}))$

Input:
- $\mathsf{CT}(\mathbf{X})$: ciphertexts for rows of a *symmetric* matrix $\mathbf{X} \in \mathbb{Z}_t^{d \times d}$
- $\mathsf{ct}(\mathbf{u})$: ciphertext of vector $\mathbf{u} \in \mathbb{Z}_t^d$

Output:
- $\mathsf{ct}(\mathbf{Xu})$: ciphertext of vector $\mathbf{Xu} \in \mathbb{Z}_t^d$

1: **for** $i = 0$ to $d - 1$ **do**
2: $\quad \mathsf{ct}(\mathbf{Xu}) \mathrel{+}= \mathsf{ct}(\mathbf{x}_i^\mathsf{T}) \cdot \mathsf{replicate}(\mathsf{ct}(\mathbf{u}), i)$
3: **end for**
4: return $\mathsf{ct}(\mathbf{Xu})$

Matrix Addition & Multiplication Matrix addition and multiplication are computed in such a way that the layout is preserved i.e. output matrix is also encrypted in row order. Matrix addition is fairly straight-forward, and is done by adding the corresponding rows of the two matrices. Matrix multiplication is performed by the MatMul function described in Algorithm 6.

Algorithm 6 $\mathsf{MatMul}(\mathsf{CT}(\mathbf{X}), \mathsf{CT}(\mathbf{Y}))$

Input:
- $\mathsf{CT}(\mathbf{X}), \mathsf{CT}(\mathbf{Y})$: ciphertexts for rows of matrices $\mathbf{X}, \mathbf{Y} \in \mathbb{Z}_t^{d \times d}$

Output:
- $\mathsf{CT}(\mathbf{XY})$: ciphertexts for rows of matrix $\mathbf{XY} \in \mathbb{Z}_t^{d \times d}$

1: **for** $i = 0$ to $d - 1$ **do**
2: $\quad$ **for** $j = 0$ to $d - 1$ **do**
3: $\quad\quad \mathsf{ct}(\mathbf{xy}_i^\mathsf{T}) \mathrel{+}= \mathsf{replicate}(\mathsf{ct}(\mathbf{x}_i^\mathsf{T}), j) \cdot \mathsf{ct}(\mathbf{y}_j^\mathsf{T}))$
4: $\quad$ **end for**
5: **end for**
6: return $\mathsf{CT}(\mathbf{XY})$ ▷ $\mathsf{CT}(\mathbf{XY}) = \{\mathsf{ct}(\mathbf{xy}_i^\mathsf{T})\}_{i=0}^{d-1}$

3.2 Our Method (Hypercube Arrangement)

We leverage a *two*-dimensional ($n = 2$, ref. Section 2.1) *hypercube* arrangement of plaintext slots to optimize matrix operations over encrypted matrices and vectors. Let the size of the *first* and the *second* dimension be $m_1 = \mathsf{ord}(f_1)$ and $m_2 = \mathsf{ord}(f_2)$ respectively. The *hypercube* can be seen as a matrix having m_1 rows and m_2 columns. For the rest of discussion, we treat our *hypercube* arrangement of plaintext slots as a $m_1 \times m_2$ matrix $\mathbf{V}$.

A matrix $\mathbf{X} \in \mathbb{Z}_t^{d \times d}$ can be packed in $\mathbf{V}$ by putting the (i, j)-th entry of the matrix in (i, j)-th position of the hypercube i.e. $\mathbf{V}[i, j] = x_{i,j}$ (all other positions are filled with zeros). There are two ways a vector $\mathbf{u} \in \mathbb{Z}_t^d$ can be packed in $\mathbf{V}$:

1. *Row-wise* packing: If $\mathbf{u}$ is supposed to act like a row-vector in the computation, then we pack the vector $\mathbf{u}$ (row-wise) in all the rows of $\mathbf{V}$ i.e. $\mathbf{V}[i, j] = u_j$, where $i \in [m_1]$ and $j \in [d]$. Given $\mathbf{u} = \{u_j\}_{j=0}^{d-1}$, we have:

$$\mathbf{V} = \begin{bmatrix} u_0 & \dots & u_{d-1} \\ \vdots & \ddots & \vdots \\ u_0 & \dots & u_{d-1} \end{bmatrix} \Bigg\}_{m_1}$$

2. *Column-wise* packing: Similarly, if $\mathbf{u}$ is supposed to act like a column-vector in the computation, then we pack the vector $\mathbf{u}$ (column-wise) in all the columns of $\mathbf{V}$ i.e. $\mathbf{V}[i,j] = u_i$, where $i \in [d]$ and $j \in [m_2]$. Given $\mathbf{u} = \{u_i\}_{i=0}^{d-1}$, we have:

$$\mathbf{V} = \underbrace{\begin{bmatrix} u_0 & \dots & u_0 \\ \vdots & \ddots & \vdots \\ u_{d-1} & \dots & u_{d-1} \end{bmatrix}}_{m_2}$$

There are two major advantages of using the *hypercube* arrangement:

- *Space*-efficient: Using the *hypercube* arrangement, we can pack the whole matrix in a single plaintext, as opposed to the *linear array* arrangement, which requires d plaintexts. Let $\mathbf{V}_l = \{\mathbf{v}_i^\mathsf{T}\}_{i=0}^{d-1}$ and $\mathbf{V}_h$ be the packing of matrix $\mathbf{X} \in \mathbb{Z}_t^{d \times d}$ in the *linear array* and *hypercube* arrangement respectively. Then, we have:

$$\underbrace{\begin{bmatrix} [\, x_{0,0} & \dots & x_{0,d-1} \,] \\ \vdots & \ddots & \vdots \\ [\, x_{d-1,0} & \dots & x_{d-1,d-1} \,] \end{bmatrix}}_{\mathbf{V}_l = \{\mathbf{v}_i^\mathsf{T}\}_{i=0}^{d-1}} \quad \Big| \quad \underbrace{\begin{bmatrix} x_{0,0} & \dots & x_{0,d-1} \\ \vdots & \ddots & \vdots \\ x_{d-1,0} & \dots & x_{d-1,d-1} \end{bmatrix}}_{\mathbf{V}_h}$$

d-ciphertexts | 1-ciphertext

- *Time*-efficient: The *hypercube* arrangement offers a huge advantage by enabling us to operate on a dimension, independent of other dimensions. Therefore, given a set of positions $\{k_i\}_{i=0}^{d-1}$, we can replicate the k_i-th element of the i-th row through one replication operation with *hypercube* arrangement as opposed to d replication operations in case of *linear array*. Hence, we can utilize the SIMD operations to a greater extent. On performing replication, we get:

$$\underbrace{\begin{bmatrix} [\, x_{0,k_0} & \dots & x_{0,k_0} \,] \\ \vdots & \ddots & \vdots \\ [\, x_{d-1,k_{d-1}} & \dots & x_{d-1,k_{d-1}} \,] \end{bmatrix}}_{\{\mathsf{replicate}(\mathbf{v}_i^\mathsf{T}, k_i)\}_{i=0}^{d-1}} \quad \Big| \quad \underbrace{\begin{bmatrix} x_{0,k_0} & \dots & x_{0,k_0} \\ \vdots & \ddots & \vdots \\ x_{d-1,k_{d-1}} & \dots & x_{d-1,k_{d-1}} \end{bmatrix}}_{\mathsf{replicate1D}(\mathbf{V}_h, 2, \{k_i\}_{i=0}^{d-1})}$$

d-replications | 1-replication

  This property holds for all data movement operations such as rotation and total-sum. Similarly, we can also replicate the k_i-th element of the i-th column through a call to the function $\mathsf{replicate1D}\big(\mathbf{V}_h, 1, \{k_i\}_{i=0}^{d-1}\big)$.

We use the notation $\mathsf{ct}_1(\mathbf{u})$ and $\mathsf{ct}_2(\mathbf{u})$ to denote ciphertexts encrypting the vector $\mathbf{u} \in \mathbb{Z}_t^d$ row-wise and column-wise respectively. The encryption of matrix $\mathbf{X} \in \mathbb{Z}_t^{d \times d}$ is denoted by the notation $\mathsf{ct}(\mathbf{X})$.

Outer Product of Vectors The outer product of vectors $\mathbf{u} \in \mathbb{Z}_t^d$ and $\mathbf{v} \in \mathbb{Z}_t^d$ is computed by the function defined in Algorithm 7.

Algorithm 7 $\mathsf{OuterProduct1D}(\mathsf{ct}_2(\mathbf{u}), \mathsf{ct}_1(\mathbf{v}))$

Input:
- $\mathsf{ct}_2(\mathbf{u})$: column-wise encryption of $\mathbf{u} \in \mathbb{Z}_t^d$
- $\mathsf{ct}_1(\mathbf{v})$: row-wise encryption of $\mathbf{v} \in \mathbb{Z}_t^d$

Output:
- $\mathsf{ct}(\mathbf{uv})$: ciphertext of matrix $\mathbf{uv} \in \mathbb{Z}_t^{d \times d}$

1: $\mathsf{ct}(\mathbf{uv}) = \mathsf{ct}_2(\mathbf{u}) \cdot \mathsf{ct}_1(\mathbf{v})$
2: return $\mathsf{ct}(\mathbf{uv})$

Matrix-Vector Multiplication To multiply a matrix $\mathbf{X} \in \mathbb{Z}_t^{d \times d}$ with a vector $\mathbf{u} \in \mathbb{Z}_t^d$, our technique requires the matrix and the vector to be encrypted in one of the following ways:

1. $\mathbf{X}$ is encrypted and $\mathbf{u}$ is encrypted row-wise.
2. $\mathbf{X}^\mathsf{T}$ is encrypted and $\mathbf{u}$ is encrypted column-wise.

It turns out that our technique outputs vector $\mathbf{Xu}$ encrypted column-wise if the input vector $\mathbf{u}$ is encrypted row-wise and vice-versa. This doesn't pose a problem since we can easily convert between the two packings by multiplying the vector with identity matrix $\mathbf{I}$. For our application, we can do successive multiplications (without a need for conversion) with encryption of $\mathbf{X}$ since we only deal with *symmetric* matrices. Our technique for matrix vector multiplication is

Algorithm 8 $\mathsf{MatVecMul1D}(\mathsf{ct}(\mathbf{X}), \mathsf{ct}_{1/2}(\mathbf{u}))$

Input:
- $\mathsf{ct}(\mathbf{X})$: ciphertext of a *symmetric* matrix $\mathbf{X} \in \mathbb{Z}_t^{d \times d}$
- $\mathsf{ct}_{1/2}(\mathbf{u})$: row/column-wise encryption of $\mathbf{u} \in \mathbb{Z}_t^d$

Output:
- $\mathsf{ct}_{2/1}(\mathbf{Xu})$: column/row-wise encryption of $\mathbf{Xu} \in \mathbb{Z}_t^d$

1: $\mathsf{ct}_{1/2}(\mathbf{Xu}^*) = \mathsf{ct}(\mathbf{X}) \cdot \mathsf{ct}_{1/2}(\mathbf{u})$
2: $\mathsf{ct}_{2/1}(\mathbf{Xu}) = \mathsf{TS1D}(\mathsf{ct}_{1/2}(\mathbf{Xu}^*), 2)$
3: return $\mathsf{ct}_{2/1}(\mathbf{Xu})$ ▷ $\mathsf{ct}_{1/2}(\cdot) \rightarrow \mathsf{ct}_{2/1}(\cdot)$

defined in Algorithm 8 and the following examples demonstrate its working:

$$\underbrace{\begin{bmatrix}1&2\\2&4\end{bmatrix}}_{\mathsf{ct}(\mathbf{X})} \cdot \underbrace{\begin{bmatrix}e&g\\e&g\end{bmatrix}}_{\mathsf{ct}_1(\mathbf{u})} \rightarrow \underbrace{\begin{bmatrix}e&2g\\2e&4g\end{bmatrix}}_{\mathsf{ct}_1(\mathbf{Xu}^*)} \xrightarrow{\mathsf{TS1D}(2)} \underbrace{\begin{bmatrix}e+2g&e+2g\\2e+4g&2e+4g\end{bmatrix}}_{\mathsf{ct}_2(\mathbf{Xu})}$$

$$\underbrace{\begin{bmatrix}1&2\\2&4\end{bmatrix}}_{\mathsf{ct}(\mathbf{X})} \cdot \underbrace{\begin{bmatrix}e&e\\g&g\end{bmatrix}}_{\mathsf{ct}_2(\mathbf{u})} \rightarrow \underbrace{\begin{bmatrix}e&2e\\2g&4g\end{bmatrix}}_{\mathsf{ct}_2(\mathbf{Xu}^*)} \xrightarrow{\mathsf{TS1D}(1)} \underbrace{\begin{bmatrix}e+2g&2e+4g\\e+2g&2e+4g\end{bmatrix}}_{\mathsf{ct}_1(\mathbf{Xu})}$$

Note: We can switch between the two encrypted vector packings at the cost of a replication by using the $\mathsf{SwitchOrder1D}$ routine that comes directly from the $\mathsf{MatVecMul1D}$ function by replacing $\mathsf{ct}(\mathbf{X})$ with $\mathbf{I}$, where $\mathbf{I}$ is the identity matrix. Therefore, combining the two functions, we get a general-purpose function to do matrix vector multiplication.

Matrix Addition & Multiplication Matrix Addition is trivial for our packing and can be simply done by adding the ciphertexts of the two matrices. For matrix multiplication, we use the Fox Matrix multiplication method for hypercubes described in [9]. This method can be seen as an extension of the diagonal order matrix-vector multiplication described in [15, Section 4.3]. Suppose we want to multiply a matrix $\mathbf{X} \in \mathbb{Z}_t^{d \times d}$ with a matrix $\mathbf{Y} \in \mathbb{Z}_t^{d \times d}$. Given the hypercube structure, we can rotate the column vectors of $\mathbf{Y}$ by i positions, denoted by $\mathbf{Y} \lll_1 i$, with one rotation operation. Using one replication operation, we can extract the i-th diagonal of $\mathbf{X}$, defined as $\mathbf{d}_i(\mathbf{X}) = (x_{0,i}, x_{1,i+1}, \ldots, x_{d-1,i-1})$, and replicate it along the rows. Then, we can compute the product $\mathbf{XY}$ as $\sum_{i=0}^{d-1} \mathbf{d}_i(\mathbf{X}) \times (\mathbf{Y} \lll_1 i)$. The $\mathsf{MatMul1D}$ function implements our matrix multiplication procedure and is defined in Algorithm 9.

The following example demonstrates the working of $\mathsf{MatMul1D}$:

$$\underbrace{\begin{bmatrix}1&2\\3&4\end{bmatrix}}_{\mathsf{ct}(\mathbf{X})} \times \underbrace{\begin{bmatrix}e&f\\g&h\end{bmatrix}}_{\mathsf{ct}(\mathbf{Y})} \rightarrow \overbrace{\underbrace{\begin{bmatrix}1&1\\4&4\end{bmatrix} \cdot \begin{bmatrix}e&f\\g&h\end{bmatrix}}_{i=0}}^{\mathsf{ct}(\mathbf{d}_0(\mathbf{X}) \times (\mathbf{Y} \lll_1 0))} + \overbrace{\underbrace{\begin{bmatrix}2&2\\3&3\end{bmatrix} \cdot \begin{bmatrix}g&h\\e&f\end{bmatrix}}_{i=1}}^{\mathsf{ct}(\mathbf{d}_1(\mathbf{X}) \times (\mathbf{Y} \lll_1 1))}$$

Remark 1. Our matrix multiplication technique defined for square matrices can be easily extended to general matrices. Every matrix can be partitioned into square sub-matrices called blocks. We can then multiply the matrices block-wise using the Block Matrix Multiplication technique.

Algorithm 9 $\mathsf{MatMul1D}(\mathsf{ct}(\mathbf{X}), \mathsf{ct}(\mathbf{Y}))$

Input:
- $\mathsf{ct}(\mathbf{X}), \mathsf{ct}(\mathbf{Y})$: ciphertexts of matrices $\mathbf{X}, \mathbf{Y} \in \mathbb{Z}_t^{d \times d}$

Output:
- $\mathsf{ct}(\mathbf{XY})$: ciphertext of matrix $\mathbf{XY} \in \mathbb{Z}_t^{d \times d}$

1: **for** $i = 0$ to $d-1$ **do**
2: **for** $j = 0$ to $d-1$ **do**
3: $k_j = i + j \pmod d$
4: **end for**
5: $\mathsf{ct}_2(\mathbf{d}_i(\mathbf{X})) = \mathsf{replicate1D}(\mathsf{ct}(\mathbf{X}), 2, \{k_j\}_{j=0}^{d-1})$
6: $\mathsf{ct}(\mathbf{Y} \lll_1 i) = \mathsf{rotate1D}(\mathsf{ct}(\mathbf{Y}), 1, -i)$
7: $\mathsf{ct}(\mathbf{XY}) \mathrel{+}= \mathsf{ct}_2(\mathbf{d}_i(\mathbf{X})) \cdot \mathsf{ct}(\mathbf{Y} \lll_1 i)$
8: **end for**
9: return $\mathsf{ct}(\mathbf{XY})$

3.3 Comparison

In this subsection, we compare the complexities of the techniques described in Section 3.1 and Section 3.2, in terms of homomorphic operations. For details on effect of these operations on noise and their running time, we refer the readers to [15, Table 1]. In Table 1, we have summarized the complexities for all the matrix operations. It can be easily

Table 1. Complexity of Matrix Operations in terms of Homomorphic Operations. d is the size of matrix dimensions and "$-$" denotes that the operation is not used. The rows corresponding to our techniques are shown in bold face.

	Addition	Multiplication	Rotation	Scalar-multiply
OuterProduct	$O(d \log d)$	$O(d)^*$	$O(d \log d)$	$O(d)^*$
OuterProduct1D	$-$	$\boldsymbol{O(1)}^*$	$-$	$-$
MatVecMul	$O(d \log d)$	$O(d)^*$	$O(d \log d)$	$O(d)^*$
MatVecMul1D	$\boldsymbol{O(\log d)}$	$\boldsymbol{O(1)}^*$	$\boldsymbol{O(\log d)}$	$-$
SwitchOrder1D	$\boldsymbol{O(\log d)}$	$-$	$\boldsymbol{O(\log d)}$	$\boldsymbol{O(1)}^*$
MatMul	$O(d^2 \log d)$	$O(d^2)^*$	$O(d^2 \log d)$	$O(d^2)^*$
MatMul1D	$\boldsymbol{O(d \log d)}$	$\boldsymbol{O(d)}^*$	$\boldsymbol{O(d \log d)}$	$\boldsymbol{O(d)}^*$

$*$: represents that the depth complexity of the operation is $O(1)$

observed that for all the matrix operations, our method has decreased the complexity by a factor of matrix dimension i.e. d.

4 Secure computation in the cloud

Having defined the matrix primitives in the previous section, we are now ready to proceed to defining protocols for secure computation in the cloud. We showed in the previous section that our method for matrix operations can result in significant performance improvement. Therefore, we adapt the PrivatePCA and PrivateLR protocols proposed in [16] to work with our method for secure matrix operations. The protocols used are basically the same, except we have used our method for matrix operations. Before we proceed with their description, we first describe the input data that will be computed upon:

- We perform statistical procedures on datasets with entries in the real domain. Since the BGV scheme only works on integers, we first need to convert all the real attributes of a dataset into integers with fixed point precision. Given $x \in \mathbb{R}$, we scale it up by the magnifying constant M and then round it to the nearest integer i.e. $\lfloor Mx \rceil$. For the rest of description, we assume that we are computing statistics on datasets with integer attributes.
- Our setup assumes that the data rows could be coming from different sources that require that their data remains secure from other sources and the cloud. To keep the data secure, each source can encrypt their data rows separately, and send them to the cloud. But there is a problem that given the dataset $(\mathbf{X}, \mathbf{y}) = \{\mathbf{x}_i^\intercal, y_i\}_{i=0}^{N-1} \in \mathbb{Z}_t^{N \times d}$, it will be too inefficient to compute $\mathbf{X}^\intercal\mathbf{X}$ or $\mathbf{X}^\intercal\mathbf{y}$ homomorphically for any practical N. To overcome this problem, we use the fact that $\mathbf{X}^\intercal\mathbf{X} = \sum_{i=0}^{N-1} \mathbf{x}_i\mathbf{x}_i^\intercal$ and $\mathbf{X}^\intercal\mathbf{y} = \sum_{i=0}^{N-1} y_i\mathbf{x}_i^\intercal$. Therefore, each source can encrypt the matrix $\mathbf{x}_i\mathbf{x}_i^\intercal$ and the vector $y_i\mathbf{x}_i^\intercal$ for each data row independent of other sources, which can then be added in the cloud to reconstruct $\mathbf{X}^\intercal\mathbf{X}$ and $\mathbf{X}^\intercal\mathbf{y}$.

Remark 2. Since we use the protocols proposed by Lu et al., with the exception of underlying matrix operations, we defer the security analysis of the protocols to [16, Section VI].

4.1 PrivatePCA

In this subsection, we want to compute the principal components of the data matrix $\mathbf{X} \in \mathbb{Z}_t^{N \times d}$ securely in the cloud, which requires to compute the covariance matrix $\boldsymbol{\Sigma} = \frac{1}{N}\mathbf{X}^\mathsf{T}\mathbf{X} - \boldsymbol{\mu}\boldsymbol{\mu}^\mathsf{T}$. Since we can't perform division while computing $\boldsymbol{\mu}^\mathsf{T}$ and $\frac{1}{N}\mathbf{X}^\mathsf{T}\mathbf{X}$ in the cloud, we rather calculate $N^2\boldsymbol{\Sigma}$, multiplying $\mathbf{X}^\mathsf{T}\mathbf{X}$ by N and computing $N\boldsymbol{\mu}^\mathsf{T}$ in place of $\boldsymbol{\mu}^\mathsf{T}$. This works because scaling a matrix doesn't change the direction of its eigen-vectors, it only scales their associated eigen-values. To compute $N^2\boldsymbol{\mu}\boldsymbol{\mu}^\mathsf{T}$, we first compute $N\boldsymbol{\mu}^\mathsf{T}$ in row-wise packing, use SwitchOrder1D procedure to get $N\boldsymbol{\mu}$ in column-wise packing, and then input both the ciphertexts into OuterProduct1D to get an encrypted matrix for $N^2\boldsymbol{\mu}\boldsymbol{\mu}^\mathsf{T}$. Having computed $N^2\boldsymbol{\Sigma}$, all that is left is to multiply the vector $\mathbf{v}$ with matrix $N^2\boldsymbol{\Sigma}$ repeatedly. Since $N^2\boldsymbol{\Sigma}$ is a symmetric matrix, we can do it directly with MatVecMul1D procedure without the need of SwitchOrder1D procedure. The description of the protocol is given in Algorithm 10.

Algorithm 10 PrivatePCA$(\{\mathsf{ct}_1(\mathbf{x}_i^\mathsf{T}), \mathsf{ct}(\mathbf{x}_i\mathbf{x}_i^\mathsf{T})\}_{i=0}^{N-1}, T)$

Input:
- $\mathsf{ct}_1(\mathbf{x}_i^\mathsf{T})$: row-wise encryption of i-th row of $\mathbf{X}$
- $\mathsf{ct}(\mathbf{x}_i\mathbf{x}_i^\mathsf{T})$: ciphertext of outer-product of $\mathbf{x}_i^\mathsf{T}$ with itself
- T : number of iterations

Output:
- $\mathbf{u}_1, \lambda_1$: first principal component of $\mathbf{X}$ and its magnitude

Cloud:

$\mathsf{ct}_1(N\boldsymbol{\mu}^\mathsf{T}) = \sum_{i=0}^{N-1} \mathsf{ct}_1(\mathbf{x}_i^\mathsf{T})$ ▷ $N\boldsymbol{\mu}^\mathsf{T} \in \mathbb{Z}_t^d$
$\mathsf{ct}_2(N\boldsymbol{\mu}) = \mathsf{SwitchOrder1D}(\mathsf{ct}_1(N\boldsymbol{\mu}^\mathsf{T}))$
$\mathsf{ct}(N^2\boldsymbol{\mu}\boldsymbol{\mu}^\mathsf{T}) = \mathsf{OuterProduct1D}(\mathsf{ct}_2(N\boldsymbol{\mu}), \mathsf{ct}_1(N\boldsymbol{\mu}^\mathsf{T}))$
$\mathsf{ct}(N^2\boldsymbol{\Sigma}) = N \cdot \sum_{i=0}^{N-1} \mathsf{ct}(\mathbf{x}_i\mathbf{x}_i^\mathsf{T}) - \mathsf{ct}(N^2\boldsymbol{\mu}\boldsymbol{\mu}^\mathsf{T})$ ▷ $N^2\boldsymbol{\Sigma} \in \mathbb{Z}_t^{d \times d}$
for $i = 1$ to T **do**
  if i is odd **then**
    $\mathsf{ct}_2(\mathbf{v}^{(i)}) = \mathsf{MatVecMul1D}(\mathsf{ct}(N^2\boldsymbol{\Sigma}), \mathsf{ct}_1(\mathbf{v}^{(i-1)}))$
  else if i is even **then**
    $\mathsf{ct}_1(\mathbf{v}^{(i)}) = \mathsf{MatVecMul1D}(\mathsf{ct}(N^2\boldsymbol{\Sigma}), \mathsf{ct}_2(\mathbf{v}^{(i-1)}))$
  end if
end for
if T is odd **then**
  return $\mathsf{ct}_2(\mathbf{v}^{(T)})$ and $\mathsf{ct}_1(\mathbf{v}^{(T-1)})$
else if T is even **then**
  return $\mathsf{ct}_1(\mathbf{v}^{(T)})$ and $\mathsf{ct}_2(\mathbf{v}^{(T-1)})$
end if

Decryptor:

return $\mathbf{u}_1 = \mathbf{v}^{(T)}/\|\mathbf{v}^{(T)}\|$ and $\lambda_1 = \|\mathbf{v}^{(T)}\|/(N^2 \cdot \|\mathbf{v}^{(T-1)}\|)$

In this way, we can securely compute the *first* principal component of data matrix $\mathbf{X}$. Apart from the PowerMethod that we have in place in the form of PrivatePCA, we just need the EigenShift procedure to compute other principal components securely. The EigenShift procedure is easily performed in the cloud through simple operations like matrix addition and homomorphic multiplication, along with outer product on $\mathbf{u}_i$'s which can be done exactly like we did on $N\boldsymbol{\mu}$.

Remark 3. If the input data is known to be normalized beforehand, we can skip the computation of $N^2\boldsymbol{\mu}^\mathsf{T}\boldsymbol{\mu}$ since it will always be zero. Moreover, we don't need to multiply $\mathbf{X}^\mathsf{T}\mathbf{X}$ with N then.

4.2 PrivateLR

To perform Linear Regression on the dataset $(\mathbf{X}, \mathbf{y}) = \{\mathbf{x}_i^\mathsf{T}, y_i\}_{i=0}^{N-1} \in \mathbb{Z}_t^{N \times d}$, we need to compute $\mathbf{w} = (\mathbf{X}^\mathsf{T}\mathbf{X})^{-1}\mathbf{X}^\mathsf{T}\mathbf{y}$. Since we can construct encrypted matrices $\mathbf{X}^\mathsf{T}\mathbf{X}$ and $\mathbf{X}^\mathsf{T}\mathbf{y}$ in the cloud by adding the inputs $\mathbf{x}_i\mathbf{x}_i^\mathsf{T}$ and $y_i\mathbf{x}_i^\mathsf{T}$ respectively, all we need is to invert the matrix $\mathbf{X}^\mathsf{T}\mathbf{X}$ and multiply it with $\mathbf{X}^\mathsf{T}\mathbf{y}$. We use the procedure defined in Algorithm

3 for matrix inversion, that involves just matrix multiplications and additions. Hence, we can use our proposed matrix operations to compute Linear Regression securely. We also need the dominant eigen value λ of matrix $\mathbf{X}^\intercal\mathbf{X} \in \mathbb{Z}_t^{(d-1)\times(d-1)}$ for stable geometric convergence, which we can get through the PCA protocol before performing Linear Regression. For the scalar $\lambda \in \mathbb{Z}_t$, we use the notation $\mathsf{ct}(\lambda)$ to refer to the ciphertext of a plaintext $\mathbf{V}$ with λ packed in each plaintext slot i.e. $\mathbf{V}[i,j] = \lambda$. The PrivateLR protocol is described in Algorithm 11.

Algorithm 11 $\mathsf{PrivateLR}(\{\mathsf{ct}_1(y_i\mathbf{x}_i^\intercal), \mathsf{ct}(\mathbf{x}_i\mathbf{x}_i^\intercal)\}_{i=0}^{N-1}, \mathsf{ct}(\lambda), T)$

Input:
- $\mathsf{ct}_1(y_i\mathbf{x}_i^\intercal)$: row-wise encryption of i-th row of $\mathbf{X}$ multiplied with i-th element of $\mathbf{y}$
- $\mathsf{ct}(\mathbf{x}_i\mathbf{x}_i^\intercal)$: ciphertext of outer-product of $\mathbf{x}_i^\intercal$ with itself
- $\mathsf{ct}(\lambda)$: ciphertext of dominant eigen-value of $\mathbf{X}^\intercal\mathbf{X}$
- T : number of iterations

Output:
- $\mathbf{w}$: weight vector $\mathbf{w} \in \mathbb{Z}_t^{d-1}$

Cloud:

$\mathsf{ct}_1(\mathbf{X}^\intercal\mathbf{y}) = \sum_{i=0}^{N-1} \mathsf{ct}_1(y_i\mathbf{x}_i^\intercal)$ ▷ $\mathbf{X}^\intercal\mathbf{y} \in \mathbb{Z}_t^{(d-1)}$

$\mathsf{ct}(\mathbf{A}^0) = \mathsf{ct}(\mathbf{X}^\intercal\mathbf{X}) = \sum_{i=0}^{N-1} \mathsf{ct}(\mathbf{x}_i\mathbf{x}_i^\intercal)$ ▷ $\mathbf{X}^\intercal\mathbf{X} \in \mathbb{Z}_t^{(d-1)\times(d-1)}$

$\mathsf{ct}(\mathbf{R}^0) = \mathsf{ct}(\mathbf{I}),\ \mathsf{ct}(\alpha^{(0)}) = \mathsf{ct}(\lambda)$

for $i = 1$ to T **do**

  $\mathsf{ct}(\mathbf{B}) = 2 \cdot \mathsf{ct}(\alpha^{(i-1)}) \cdot \mathbf{I} - \mathsf{ct}(\mathbf{A}^{(i-1)})$

  $\mathsf{ct}(\mathbf{R}^{(i)}) = \mathsf{MatMul1D}(\mathsf{ct}(\mathbf{R}^{(i-1)}), \mathsf{ct}(\mathbf{B}))$

  $\mathsf{ct}(\mathbf{A}^{(i)}) = \mathsf{MatMul1D}(\mathsf{ct}(\mathbf{A}^{(i-1)}), \mathsf{ct}(\mathbf{B}))$

  $\mathsf{ct}(\alpha^{(i)}) = \mathsf{ct}(\alpha^{(i-1)}) \cdot \mathsf{ct}(\alpha^{(i-1)})$

end for ▷ $\mathsf{ct}(\lambda^{2^T}(\mathbf{X}^\intercal\mathbf{X})^{-1}) = \mathsf{ct}(\mathbf{R}^{(T)})$

$\mathsf{ct}_2(\lambda^{2^T}\mathbf{w}) = \mathsf{MatVecMul1D}(\mathsf{ct}(\lambda^{2^T}(\mathbf{X}^T\mathbf{X})^{-1}), \mathsf{ct}_1(\mathbf{X}^\intercal\mathbf{y}))$

return $\mathsf{ct}_2(\lambda^{2^T}\mathbf{w})$ and $\mathsf{ct}(\lambda^{2^T})$ ▷ $\mathsf{ct}(\alpha^{(T)}) = \mathsf{ct}(\lambda^{2^T})$

Decryptor:

return $\mathbf{w}$ from $\lambda^{2^T}\mathbf{w}$ by dividing with λ^{2^T}.

Remark 4. We have described the protocols as if each data row is coming from an independent source. In practice, we will have multiple data rows coming from a particular source. The sources can send just one ciphertext each for $\mathbf{x}_i^\intercal, \mathbf{x}_i\mathbf{x}_i^\intercal$ and $y_i\mathbf{x}_i^\intercal$ by performing addition corresponding to their data rows in the plaintext.

5 Choosing the right context

The right choice of context parameters can significantly improve the performance of our protocols. Choosing the right parameters is a non-trivial task in `HElib` and in most works using `HElib`, the authors do not describe how they chose the optimal parameters for their application. In this section, we describe the theory used to choose optimal parameters for our setup. Moreover, we found that the parameter choice in [16] is non-optimal. So, we describe how to choose the right parameters for their setup as well.

As described in Section 2.1, $\{f_1, \ldots, f_n\}$ is the generating set of $\mathbb{Z}_m^*/\langle t\rangle$. Let the order of f_i in $\mathbb{Z}_m^*/\langle t, f_1, \ldots, f_{i-1}\rangle$ be m_i. Therefore, we have the number of plaintext slots $\ell = |\mathbb{Z}_m^*/\langle t\rangle| = \prod_{i=1}^n m_i$. For efficient data movement operations, we basically have two main requirements:

1. For each $i \in [n]$, we should have $\mathsf{ord}_{\mathbb{Z}_m^*}(f_i) = m_i$, since it allows true rotations on slots.
2. ℓ should be kept close to the number of plaintext slots required by the application.

For more details on why we have these requirements, refer [14, Section 4]. Before proceeding to the solution that satisfies the above-mentioned requirements, we derive some results on which our solution will be based.

Theorem 1. *Let G be a finite abelian group. Suppose we have an element $g \in G$ and a subgroup N such that $\mathsf{ord}(g) = k$ and $|N| = K$. Then the following holds:*

$$\gcd(k, K) = 1 \implies \mathsf{ord}_G(g) = \mathsf{ord}_{G/N}(g).$$

Proof. It is clear from the definition of a quotient group that:

$$\mathsf{ord}_G(g) = \mathsf{ord}_{G/N}(g) \iff \langle g \rangle \cap N = \{e\},$$

where e is the identity element of G. Therefore, to prove the theorem, it is sufficient to prove that $\gcd(k, K) = 1$ implies $\langle g \rangle \cap N = \{e\}$. Let g' be an arbitrary element $\in \langle g \rangle \cap N$. This implies that $\mathsf{ord}(g')|k$ as well as $\mathsf{ord}(g')|K$. Since $\gcd(k, K) = 1$, g' has order 1, implying $g' = e$. Since we assumed g' to be a general element in $\langle g \rangle \cap N$, we have $\langle g \rangle \cap N = \{e\}$.

Corollary 1. *Let $g \in G$ be an element of a finite abelian group G with $\mathsf{ord}(g) = k$. Suppose we have n elements $g_1, \ldots, g_n \in G$ with $\mathsf{ord}(g_i) = k_i$. We denote the subgroup generated by the set $\{g_1, \ldots, g_n\}$ with N i.e. $N = \langle g_1, \ldots, g_n \rangle$. Then the following holds:*

$$\forall i \in [n], \gcd(k_i, k) = 1 \implies \mathsf{ord}_G(g) = \mathsf{ord}_{G/N}(g)$$

Proof. It is a well-known equality that given subgroups $N_1, N_2 \triangleleft G$, we have a subgroup $N_1N_2 = \{h_1h_2 | h_1 \in N_1, h_2 \in N_2\}$ such that $|N_1N_2| = |N_1| \cdot |N_2|/|N_1 \cap N_2|$. Therefore, we have a subgroup N generated by $\{g_i\}_{i=1}^n$ of order K which divides the product of all k_i's i.e. $K|(\prod_{i=1}^n k_i)$. Since $\forall i \in [n], \gcd(k_i, k) = 1$, we have $\gcd(k, K) = 1$. Hence, by Theorem 1, the proof is complete.

5.1 For Lu et al.'s Method

For our application using Lu et al.'s method, we need a one-dimensional array of size m_1, where m_1 is greater than and close to d. Moreover, this dimension should be a *good* dimension. These two conditions satisfy our above-mentioned requirements for efficient data movement. We achieve these conditions by choosing m and t such that:

1. $m = k \cdot m_1 + 1 \in \mathbb{P}$,
2. $\gcd(k, m_1) = 1$,
3. $\mathsf{ord}(t) = k$.

Since m is chosen to be a prime, $\mathbb{Z}_m^*$ is a cyclic group of order $m_1 \cdot k$. This ensures that it has an element f_1 of order m_1. If we choose an element of order k (exists because $\mathbb{Z}_m^*$ is cyclic) as t, then the number of plaintext slots $\ell = |\mathbb{Z}_m^*/\langle t \rangle| = m_1$. Given $\gcd(k, m_1) = 1$, by Corollary 1, we get $\mathsf{ord}_{\mathbb{Z}_m^*}(f_1) = \mathsf{ord}_{\mathbb{Z}_m^*/\langle t \rangle}(f_1) = m_1$. Therefore, $\mathbb{Z}_m^*/\langle t \rangle = \langle f_1 \rangle$ and we get a one-dimensional array with a *good* dimension of size m_1.

5.2 For Our Method

The optimal parameters for our method lead to a *two*-dimensional hypercube with *good* dimensions of size m_1 and m_2, where $m_1 = d$ and m_2 is greater than and close to d. These parameters satisfy the requirements for efficiency, and are achieved by choosing m and t such that:

1. $m = k \cdot m_1 \cdot m_2 + 1 \in \mathbb{P}$,
2. $\gcd(k, m_1) = \gcd(k, m_2) = \gcd(m_1, m_2) = 1$,
3. $\mathsf{ord}(t) = k$.

Since $\mathbb{Z}_m^*$ is a cyclic group of order $k \cdot m_1 \cdot m_2$, we can find f_1 and f_2 in $\mathbb{Z}_m^*$ with order m_1 and m_2 respectively. For a chosen t of order k, we have $\ell = |\mathbb{Z}_m^*/\langle t \rangle| = m_1 \cdot m_2$. Given $\gcd(k, m_1) = 1$ and $\gcd(k, m_2) = \gcd(m_1, m_2) = 1$, by Corollary 1, we get $\mathsf{ord}_{\mathbb{Z}_m^*}(f_1) = \mathsf{ord}_{\mathbb{Z}_m^*/\langle t \rangle}(f_1) = m_1$ and $\mathsf{ord}_{\mathbb{Z}_m^*}(f_2) = \mathsf{ord}_{\mathbb{Z}_m^*/\langle t, f_1 \rangle}(f_2) = m_2$ respectively. Therefore, $\mathbb{Z}_m^*/\langle t \rangle = \langle f_1, f_2 \rangle$ and we get a *two*-dimensional hypercube with *good* dimensions of size m_1 and m_2.

Remark 5. It gets relatively easier to find an optimal m for Lu et al.'s method as the matrix dimension grows. Therefore, for a considerably large d, we have to settle with relaxed parameters for our method. Despite the relaxed parameters, our method still vastly outperforms Lu et. al's because our method improves the complexity of their techniques by a factor of d (see Table 1).

5.3 CRT method for plaintext modulus

In `HElib`, the maximum plaintext precision is 60 bits, which is not sufficient for our applications. To address this problem, we split the plaintext modulus t into e different primes t_i each of bitsize ¡ 60 bits such that $t = \prod_{i=1}^{e} t_i$ for some $e \in \mathbb{Z}$. Now, we create e different instances of the cryptosystem, with i-th instance operating on plaintext modulus t_i, and encrypt our inputs in each one of them. Then, the protocol is performed for every instance and the decryption results are combined using the Chinese Remainder Theorem to get plaintext precision of $\prod_{i=1}^{e} \lfloor \log_2(t_i) \rfloor$ bits. Using the CRT method, we essentially have e ciphertexts as opposed to one. This does not pose a problem since the CRT method is completely parallelizable. Moreover, increasing e leads to a smaller t_i, and as a result, lesser number of levels L. Therefore, we get a more efficient implementation, given we have enough CPU cores.

Remark 6. In practice, we want to choose each t_i of p-bits, and to maintain security, m is chosen to be greater than a lower bound which depends on t and the number of levels L. To choose each t_i, we search the equivalence class of elements with order k for distinct p-bit members. The number m is chosen such that m_1 and m_2 are close to d while keeping m as close to the lower bound as possible.

6 Implementation Details and Results

We implemented the protocols defined in Section 4 using both our method and Lu et al.'s method. The implementations were written in C++ and compiled with g++ 4.8.5. We used `HElib` [14] for the implementation of the BGV scheme. Our experiments ran on a machine with Intel(R) Xeon(R) E5-4650@2.70GHz processor and 1TB memory running CentOS 7.5.1804. In all our experiments, we use CRT method with $e = 8$ (see Section 5.3), and have parallelized the implementations over 8 threads.

6.1 Experimental Setup

Our experimental setup is identical to the one used in [16]. We have conducted our experiments on five datasets from the UCI Machine Learning Repository [8]. For the analysis of the effect of number of iterations and different magnification constants on performance, experiments were performed on the *adult* dataset that has $N = 32561$ records with $d = 6$ numerical attributes each. More specifically, we took iteration numbers $T = \{3, 4, 5\}$ for PrivatePCA protocol and iteration numbers $T = \{1, 2, 3\}$ for PrivateLR protocol. Three magnification constants $M = \{10, 100, 1000\}$ were considered for both the protocols. Experiments on the other datasets serve to show the scalability of our approach and we have taken $T = 5, M = 1000$ and $T = 3, M = 1000$ for PrivatePCA and PrivateLR protocol, respectively on these datasets. Following the experimental setup of [16], we have also performed all our experiments on normalized datasets, with the assumption that the data is coming from a single source. Since our experimental setup is identical to Lu et al.'s, we defer to [16, Appendix A] for discussion on iteration-error tradeoffs.

6.2 Selection of Parameters

Our parameter selection basically involves choosing m and t_i that define the plaintext spaces $\mathbb{Z}[X]/(\Phi_m(X), t_i)$ for $1 \leq i \leq e$, and L which defines the number of levels. Given a dataset with each entry bounded by B, we get a lower bound on the bitsize p of t_i from the inequality $t_i > (B^2MNd)^{T/8}$ for the PrivatePCA protocol and $t_i > M^{1/8}(B^2MNd)^{2^T/8}$ for the PrivateLR protocol. An ephemeral p-bit prime following this inequality is sampled and a dry run is performed to find the required number of levels L.

According to the analysis of [11, Appendix C.3], we get κ-bit security from the BGV scheme if the following inequality holds:

$$\phi(m) > \frac{(L(\log \phi(m) + 23) - 8.5)(\kappa + 110)}{7.2}.$$

Since we only choose a prime m in our setup, we can replace $\phi(m)$ with $m - 1$ in the above inequality. Given L (from dry run) and a minimum security requirement κ, from the above inequality, we get a lower bound on m. Given the lower bound on m and bitsize of t_i, we choose m and the t_i's according to the method described in Section 5 for both our method and Lu et al.'s. In the experiments concerning PrivatePCA, we have taken $m_1 = d$ for Lu et al.'s method and $(m_1, m_2) = (d, d + 1)$ for our method. For experiments concerning PrivateLR, we have taken $m_1 = d - 1$ for Lu et al.'s method and $(m_1, m_2) = (d - 1, d)$ for our method.

Remark 7. We have chosen the best possible plaintext dimensions for our experiments. We could easily find such parameters since the matrix dimension was not too large (≤ 20). For operations on matrices with large dimensions, we can relax the upper bound on the size of plaintext dimensions, or we can use the block matrix multiplication method.

For other parameters in `HElib`, we use the default values such as $\sigma = 3.2$ (standard deviation parameter for error distribution), $H = 64$ (Hamming weight of secret key), $r = 1$ (lifting factor) and $c = 3$ (number of columns in key switching matrix). For all the experiments, we have maintained a minimum security of 128 bits.

6.3 Results and Analysis

In Table 2 and Table 3, we have compared the performance of our method with Lu et al.'s method for performing PrivatePCA and PrivateLR respectively. For PrivatePCA, we have only considered the time taken to compute the first principal component, and for PrivateLR, we exclude the time taken to compute the first eigen value λ_1. From the results of Table 2 and Table 3, the following observations can be made:

- *Encryption Time*: The encryption time for our method is d times faster than Lu et al.'s on an average because we require *one* ciphertext to encrypt a matrix, as opposed to d ciphertexts for their method.
- *Evaluation Time*: The difference in homomorphic evaluation time is in accordance with the complexities of matrix operations described in Table 1. As expected, the evaluation time of our method is d times faster than Lu et al.'s.
- *Decryption Time*: Since the output of both the protocols is a vector, our method outputs the same number of ciphertexts as Lu et al.'s method. Our decryption time is more, owing to the fact that the decoding operation takes more time for our method, given we are using more plaintext slots.
- *Depth Requirement*: Our method has comparatively less depth requirement. This is due to the fact that the MatVecMul procedure requires an additional scalar-multiplication operation, which adds moderate noise (see [15, Table 1]), compared to the MatVecMul1D procedure. This effect is more pronounced in PrivatePCA since it involves the matrix vector multiplication in each iteration, while PrivateLR involves it only once.

7 Conclusion and Future Work

We made the protocols for PCA and Linear Regression, proposed in [16], more efficient by improving their underlying matrix operations by a factor of data dimension. This was achieved by utilising the hypercube structure of the plaintext slots. Our techniques for matrix operations are flexible and can be used for any application involving matrix operations. In addition to this, we show how to choose optimal parameters for our method as well as for the method proposed in [16] by Lu et al. With our improved matrix procedures, we reduced the evaluation time of largest case ($d = 20, N = 1994, M = 1000, T = \{5\}$) for PCA from 149 seconds to 6.5 seconds, and of largest case ($d = 20, N = 1994, M = 1000, T = \{3\}$) for Linear Regression from 4400 seconds to 207 seconds. With this work, we show that the plaintext space can be utilized to a larger extent, leading to much faster PCA and Linear Regression computation.

A direction for future work is the extension of the hypercube structure from *two* to *three*-dimensions. As a result, we can reduce the complexity of matrix multiplication from $O(d \log d)$ to $O(\log d)$, where d is the matrix dimension. This can be done by constructing a multiplication procedure along the lines of DNS algorithm [7]. A limitation of this approach is that it will be harder for us to find optimal parameters. A future work can explore the situations where it is beneficial to use a higher dimension plaintext structure. In this paper, we've considered predictive statistics through *exact* arithmetic and controlled the high plaintext modulus growth by using the CRT method and leveraging multiple cores to limit the impact. Switching to a scheme that allows *approximate* arithmetic, like the one described in [6], can help resolve this problem. Encoding techniques that better utilise the finite field structure of a plaintext slot can also help in containing the large plaintext modulus.

8 Acknowledgement

This work was supported by JST CREST Grant Number JPMJCR14D6, Japan. A part of this work was also supported by JSPS KAKENHI Grant Number 16H02830.

Table 2. Performance comparison for the PrivatePCA protocol of our method (Section 4.1) with Lu et al.'s method ([16, Section V.C], with our optimized parameters) to find the first principal component. For all the experiments, we split the plaintext modulus into 8 primes $t_1, \ldots, t_8$ using our CRT method and use 8 threads to execute them all in parallel. The performance for our method is shown in bold face.

Dataset details			**Exp. Settings**		**BGV settings**			**Performance (seconds)**			
Dataset	N	d	M	T	$\lfloor \log_2(t_i) \rfloor$	m	L	Encryption	Hom. Eval.	Decryption	Total time
adult	32561	6	10	3	11	12703	11	0.4788	5.4683	0.1319	6.079
						11047	**9**	**0.0912**	**0.8562**	**0.2713**	**1.2187**
				4	12	14503	13	0.5767	8.8194	0.1363	9.5324
						13063	**11**	**0.1161**	**1.529**	**0.2893**	**1.9344**
				5	15	18583	17	1.2227	22.8318	0.2323	24.2868
						15331	**13**	**0.1469**	**2.2915**	**0.5591**	**2.9975**
			100	3	11	12703	11	0.4788	5.4683	0.1319	6.079
						11047	**9**	**0.0912**	**0.8562**	**0.2713**	**1.2187**
				4	14	14683	13	0.5288	9.0437	0.1364	9.7089
						14071	**11**	**0.1199**	**1.7285**	**0.296**	**2.1444**
				5	17	23011	21	1.4851	30.8704	0.2728	32.6283
						15331	**13**	**0.1423**	**2.3695**	**0.5595**	**3.0713**
			1000	3	11	12703	11	0.4788	5.4683	0.1319	6.079
						11047	**9**	**0.0912**	**0.8562**	**0.2713**	**1.2187**
				4	15	14683	13	0.6396	8.9445	0.2	9.7841
						14071	**11**	**0.1328**	**1.7843**	**1.1814**	**3.0985**
				5	19	25603	23	1.6289	33.835	0.4118	35.8757
						19447	**17**	**0.2769**	**5.5374**	**0.6144**	**6.4287**
auto-mpg	398	7	1000	5	15	19069	17	1.3713	33.1703	0.2885	34.8301
						15401	**13**	**0.1583**	**2.5215**	**1.3696**	**4.0494**
winequality	4898	12	1000	5	18	26293	23	3.4114	84.1018	0.5672	88.0804
						19501	**15**	**0.2865**	**5.3242**	**2.2644**	**7.8751**
forestfires	517	13	1000	5	16	20749	19	3.074	82.9019	0.4014	86.3773
						15107	**13**	**0.1651**	**2.9428**	**1.4105**	**4.5184**
communities	1994	20	1000	5	17	22741	21	5.1348	148.784	0.6661	154.5849
						18061	**15**	**0.2803**	**6.3988**	**2.09**	**8.7691**

References

1. Boneh, D., Goh, E.J., Nissim, K.: Evaluating 2-dnf formulas on ciphertexts. In: Proceedings of the Second International Conference on Theory of Cryptography. pp. 325–341. TCC'05, Springer-Verlag, Berlin, Heidelberg (2005). https://doi.org/10.1007/978-3-540-30576-7_18, http://dx.doi.org/10.1007/978-3-540-30576-7_18
2. Bost, R., Popa, R.A., Tu, S., Goldwasser, S.: Machine learning classification over encrypted data. In: 22nd Annual Network and Distributed System Security Symposium, NDSS 2015, San Diego, California, USA, February 8-11, 2015 (2015), https://www.ndss-symposium.org/ndss2015/machine-learning-classification-over-encrypted-data
3. Brakerski, Z., Gentry, C., Vaikuntanathan, V.: (leveled) fully homomorphic encryption without bootstrapping. In: Proceedings of the 3rd Innovations in Theoretical Computer Science Conference. pp. 309–325. ITCS '12, ACM, New York, NY, USA (2012). https://doi.org/10.1145/2090236.2090262, http://doi.acm.org/10.1145/2090236.2090262
4. Brakerski, Z., Vaikuntanathan, V.: Fully homomorphic encryption from ring-lwe and security for key dependent messages. In: Rogaway, P. (ed.) Advances in Cryptology – CRYPTO 2011. pp. 505–524. Springer Berlin Heidelberg, Berlin, Heidelberg (2011)
5. Cheon, J.H., Kim, M., Kim, M.: Optimized search-and-compute circuits and their application to query evaluation on encrypted data. IEEE Transactions on Information Forensics and Security **11**(1), 188–199 (Jan 2016). https://doi.org/10.1109/TIFS.2015.2483486
6. Cheon, J.H., Kim, A., Kim, M., Song, Y.: Homomorphic encryption for arithmetic of approximate numbers. In: Takagi, T., Peyrin, T. (eds.) Advances in Cryptology – ASIACRYPT 2017. pp. 409–437. Springer International Publishing, Cham (2017)

Table 3. Performance comparison for the PrivateLR protocol of our method (Section 4.2) with Lu et al.'s method ([16, Section V.C], with our optimized parameters). We excluded the computational time required to compute the largest eigen-value λ_1. For all the experiments, we split the plaintext modulus into 8 primes $t_1, \ldots, t_8$ using our CRT method and use 8 threads to execute them all in parallel. The performance for our method is shown in bold face.

Dataset details			**Exp. settings**		**BGV settings**			**Performance (Seconds)**			
Dataset	N	d	M	T	$\lfloor \log_2(t_i) \rfloor$	m	L	Encryption	Hom. Eval.	Decryption	Total time
adult	32561	6	10	1	11	10531	9	0.4964	3.8267	0.0733	4.3964
						8311	**7**	**0.1959**	**0.6444**	**0.1471**	**0.9874**
				2	15	13121	11	0.5524	28.1256	0.1198	28.7978
						10771	**9**	**0.2719**	**4.7969**	**0.2767**	**5.3455**
				3	26	23741	21	1.8307	159.8920	0.2256	161.9483
						20731	**19**	**0.7658**	**34.0543**	**0.6125**	**35.4326**
			100	1	11	10531	9	0.4964	3.8267	0.0733	4.3964
						8311	**7**	**0.1959**	**0.6444**	**0.1471**	**0.9874**
				2	17	15031	13	0.6564	33.8865	0.1153	34.6582
						14011	**11**	**0.2912**	**6.5338**	**0.3211**	**7.1461**
				3	30	27011	23	1.8077	188.1300	0.2712	190.2089
						23011	**21**	**0.7817**	**38.3839**	**0.6418**	**39.8074**
			1000	1	11	10531	9	0.4964	3.8267	0.0733	4.3964
						8311	**7**	**0.1959**	**0.6444**	**0.1471**	**0.9874**
				2	19	17011	15	1.2712	60.1418	0.1906	61.6036
						14731	**13**	**0.3093**	**7.4470**	**0.3506**	**8.1069**
				3	33	28111	25	2.0766	206.4470	0.2745	208.7981
						26431	**23**	**0.9210**	**45.4241**	**0.6954**	**47.0405**
auto-mpg	398	7	1000	3	26	24007	21	2.0630	228.9190	0.2190	231.2010
						21211	**19**	**0.8031**	**49.1406**	**0.6096**	**50.5533**
winequality	4898	12	1000	3	30	26203	23	3.5929	1169.4800	0.3569	1173.4298
						23629	**21**	**0.8884**	**95.7648**	**3.1740**	**99.8228**
forestfires	517	13	1000	3	27	25117	23	3.8555	1158.9700	0.3843	1163.2098
						22621	**21**	**0.8640**	**122.2040**	**2.9421**	**126.0101**
communities	1994	20	1000	3	29	27361	23	6.0348	4399.6500	0.4428	4406.1276
						26981	**21**	**0.9311**	**207.5400**	**4.2580**	**212.7291**

7. Dekel, E., Nassimi, D., Sahni, S.: Parallel matrix and graph algorithms. SIAM Journal on Computing **10**(4), 657–675 (1981). https://doi.org/10.1137/0210049, https://doi.org/10.1137/0210049
8. Dheeru, D., Karra Taniskidou, E.: UCI machine learning repository (2017), http://archive.ics.uci.edu/ml
9. Fox, G., Otto, S., Hey, A.: Matrix algorithms on a hypercube i: Matrix multiplication. Parallel Computing **4**(1), 17 – 31 (1987). https://doi.org/https://doi.org/10.1016/0167-8191(87)90060-3, http://www.sciencedirect.com/science/article/pii/0167819187900603
10. Gentry, C.: Fully homomorphic encryption using ideal lattices. In: Proceedings of the Forty-first Annual ACM Symposium on Theory of Computing. pp. 169–178. STOC '09, ACM, New York, NY, USA (2009). https://doi.org/10.1145/1536414.1536440, http://doi.acm.org/10.1145/1536414.1536440
11. Gentry, C., Halevi, S., Smart, N.P.: Homomorphic evaluation of the aes circuit. In: Safavi-Naini, R., Canetti, R. (eds.) Advances in Cryptology – CRYPTO 2012. pp. 850–867. Springer Berlin Heidelberg, Berlin, Heidelberg (2012)
12. Graepel, T., Lauter, K., Naehrig, M.: Ml confidential: Machine learning on encrypted data. In: Proceedings of the 15th International Conference on Information Security and Cryptology. pp. 1–21. ICISC'12, Springer-Verlag, Berlin, Heidelberg (2013). https://doi.org/10.1007/978-3-642-37682-5_1, http://dx.doi.org/10.1007/978-3-642-37682-5_1
13. Guo, C., Higham, N.: A schur-newton method for the matrix pth root and its inverse. SIAM Journal on Matrix Analysis and Applications **28**(3), 788–804 (2006). https://doi.org/10.1137/050643374, https://doi.org/10.1137/050643374
14. Halevi, S., Shoup, V.: Design and implementation of a homomorphic-encryption library (2013), http://people.csail.mit.edu/shaih/pubs/he-library.pdf
15. Halevi, S., Shoup, V.: Algorithms in helib. In: Garay, J.A., Gennaro, R. (eds.) Advances in Cryptology – CRYPTO 2014. pp. 554–571. Springer Berlin Heidelberg, Berlin, Heidelberg (2014)

16. Lu, W., Kawasaki, S., Sakuma, J.: Using fully homomorphic encryption for statistical analysis of categorical, ordinal and numerical data (this is the full version of the conference paper presented at NDSS 2017). IACR Cryptology ePrint Archive, Report 2016/1163 (2016), https://eprint.iacr.org/2016/1163
17. Naehrig, M., Lauter, K., Vaikuntanathan, V.: Can homomorphic encryption be practical? In: Proceedings of the 3rd ACM Workshop on Cloud Computing Security Workshop. pp. 113–124. CCSW '11, ACM, New York, NY, USA (2011). https://doi.org/10.1145/2046660.2046682, http://doi.acm.org/10.1145/2046660.2046682
18. Paillier, P.: Public-key cryptosystems based on composite degree residuosity classes. In: Stern, J. (ed.) Advances in Cryptology — EUROCRYPT '99. pp. 223–238. Springer Berlin Heidelberg, Berlin, Heidelberg (1999)
19. Rivest, R.L., Shamir, A., Adleman, L.: A method for obtaining digital signatures and public-key cryptosystems. Commun. ACM **21**(2), 120–126 (Feb 1978). https://doi.org/10.1145/359340.359342, http://doi.acm.org/10.1145/359340.359342
20. Smart, N.P., Vercauteren, F.: Fully homomorphic encryption with relatively small key and ciphertext sizes. In: Proceedings of the 13th International Conference on Practice and Theory in Public Key Cryptography. pp. 420–443. PKC'10, Springer-Verlag, Berlin, Heidelberg (2010). https://doi.org/10.1007/978-3-642-13013-7_25, http://dx.doi.org/10.1007/978-3-642-13013-7_25
21. Smart, N.P., Vercauteren, F.: Fully homomorphic simd operations. Des. Codes Cryptography **71**(1), 57–81 (Apr 2014). https://doi.org/10.1007/s10623-012-9720-4, http://dx.doi.org/10.1007/s10623-012-9720-4
22. Wu, D., Haven, J.: Using homomorphic encryption for large scale statistical analysis (2012), https://crypto.stanford.edu/~dwu4/FHE-SI_Report.pdf
23. Yasuda, M., Shimoyama, T., Kogure, J., Yokoyama, K., Koshiba, T.: Secure pattern matching using somewhat homomorphic encryption. In: Proceedings of the 2013 ACM Workshop on Cloud Computing Security Workshop. pp. 65–76. CCSW '13, ACM, New York, NY, USA (2013). https://doi.org/10.1145/2517488.2517497, http://doi.acm.org/10.1145/2517488.2517497
24. Yasuda, M., Shimoyama, T., Kogure, J., Yokoyama, K., Koshiba, T.: Secure statistical analysis using rlwe-based homomorphic encryption. In: Foo, E., Stebila, D. (eds.) Information Security and Privacy. pp. 471–487. Springer International Publishing, Cham (2015)