Principal Component Analysis using CKKS Homomorphic Scheme

Samanvaya Panda

International Institute of Information Technology, Hyderabad
samanvaya.panda@research.iiit.ac.in

Abstract. Principal component analysis(PCA) is one of the most popular linear dimensionality reduction techniques in machine learning. In this paper, we try to present a method for performing PCA on encrypted data using a homomorphic encryption scheme. In a client-server model where the server performs computations on the encrypted data, it (server) does not require to perform any matrix operations like multiplication, inversion, etc. on the encrypted data. This reduces the number of computations significantly since matrix operations on encrypted data are very computationally expensive. For our purpose, we used the CKKS homomorphic encryption scheme since it is most suitable for machine learning tasks allowing approximate computations on real numbers. We also present the experimental results of our proposed Homomorphic PCA(HPCA) algorithm on a few datasets. We measure the R2 score on the reconstructed data and use it as an evaluation metric for our HPCA algorithm.

Keywords: Homomorphic Encryption · Principal Component Analysis(PCA) · CKKS scheme · Goldschmidt's Algorithm

1 Introduction

With the rise of outsourcing of computational tasks through cloud computing and services, quintillion bytes of data are produced every day. Data analytics performed on these data provide several insights to the clients at the same to the cloud servers. This has raised several privacy issues and concerns among individuals, organizations, and government officials. With ever-increasing privacy issues, performing machine learning tasks on encrypted data has become the need of the hour. Several privacy-preserving machine learning tasks have been accomplished using fully homomorphic encryption(FHE) schemes and secure multiparty computations(MPC) based schemes, demonstrating their potential. Due to the interactive nature of MPC-based schemes, people started looking for FHE-based schemes as an alternative.

Gentry in [12][13] provided the construction of a leveled-HE scheme. Then the construction was converted to a FHE scheme using bootstrapping that allowed an arbitrary number of computations on the ciphertext. However, the scheme was not efficient to be used in practice. Various improvements have been made

since then, and FHE schemes have become efficient and practical. One such scheme that was introduced recently is the CKKS[7] homomorphic encryption scheme. CKKS scheme supports approximate arithmetic over encrypted data. It supports computations on real and complex values, which makes it most suitable for machine learning tasks. Application of CKKS scheme in various machine learning techniques have been already demonstrated in [10], [2], [3], [14], [15], [8], [16].

This paper focuses on using the CKKS homomorphic encryption scheme to perform principal component analysis(PCA) - a popular linear dimensionality reduction technique. Dimensionality reduction techniques transform higher dimensional data into a representation with a few intrinsic dimensions while retaining the original data's properties. When used as input to a machine learning model, such a low dimensional representation reduces the model's complexity and makes it simpler. For this reason, dimensionality reduction has numerous applications in fields such as data visualization, data compression, noise removal, pre-processing technique. Few attempts have been made in the past to perform PCA on encrypted data. Lu et al. in [19] and Rathee et al. in [22] performed PCA using the BGV scheme. However, both of them performed PCA on categorical datasets with relatively fewer attributes($\leq$ 20). Also, they provided experimentation results for the first principal component only. In [19] and [22] computing subsequent principal components requires communication between the client and the server because the client computes the eigenvalue. This makes their algorithm interactive.

Contributions

In this paper, we propose a technique to perform PCA using the CKKS homomorphic encryption scheme. Unlike in [19], [22] where they converted the real dataset to an integer dataset through appropriate scaling, we take advantage of using the CKKS scheme that can handle real numbers. We also propose a sub-ciphertext packing technique in which every vector is packed as a sub-ciphertext within a ciphertext. The length of the sub-ciphertext is almost equal to the size of the vector. The primary advantage of using such a packing technique is that most of the operations become polynomial in the sub-ciphertext length rather than polynomial in ciphertext's length. Since the sub-ciphertext length is almost the same as that of the original vector, computations become polynomial in the initial vector's length.

We compute the norm of vectors homomorphically, which makes our algorithm non-interactive for computing subsequent principal components. Our proposed Homomorphic PCA(HPCA) algorithm does not require performing any matrix operations like matrix-vector multiplication and matrix-matrix multiplication on encrypted data, significantly reducing computations.

We provide an implementation of our HPCA algorithm in SEAL-Python[24](A python binding for SEAL[23] library). Other than just categorical datasets with a few dimensions, we provide experimentation results on higher-dimensional datasets. Also, we perform computations to find more than just one principal

component as opposed to in [19], [22] which only considered the computation of the first principal component. We also measure the R2 score of the reconstructed data as an evaluation metric for our HPCA algorithm. We do not use bootstrapping in our HPCA algorithm implementation because it is not provided as an API in the SEAL library. Instead, we re-encrypt some of the encrypted parameters to eliminate noise in them. The re-encryption procedure could be ideally replaced by bootstrapping without making any changes in other parts. However, the algorithm's runtime may increase since bootstrapping in the CKKS scheme is a costlier operation than re-encryption.

2 Preliminaries

2.1 CKKS Homomorphic Encryption Scheme

The CKKS(Cheon-Kim-Kim-Song) scheme[7] is a leveled homomorphic encryption scheme that relies on the hardness of RLWE(Ring Learning With Errors) problem for its security. Unlike other HE schemes, CKKS supports approximate arithmetic on real and complex numbers with predefined precision. The main idea behind the CKKS scheme is that it treats noise generated upon decryption as an error in computation for real numbers. This makes it an ideal candidate for performing machine learning tasks where most of the computations are approximate. With the use of bootstrapping technique as mentioned in [6] and [5], the CKKS scheme becomes a FHE(fully homomorphic encryption) scheme.

Let $N = \phi(M)$ be the degree of the *M-th* cyclotomic polynomial $\Phi_M(X)$. If N is chosen as a power of 2 then $M = 2N$ and the *M-th* cyclotomic polynomial $\Phi_M(X) = X^N + 1$. Let $\mathcal{R} = \mathbb{Z}[X]/\Phi_M(X) = \mathbb{Z}[X]/(X^N + 1)$ be the ring of polynomials defined for the plaintext space. Let $\mathcal{R}_q = \mathcal{R}/q\mathcal{R} = \mathbb{Z}_q[X]/(X^N + 1)$ be the residue ring defined for the ciphertext space. Let $\mathbb{H}$ be a subspace of $\mathbb{C}^N$ which is isomorphic to $\mathbb{C}^{N/2}$. Let $\sigma : \mathcal{R} \to \sigma(\mathcal{R}) \subseteq \mathbb{H}$ be a canonical embedding. Let $\pi : \mathbb{H} \to \mathbb{C}^{N/2}$ be a map that projects a vector from a subspace of $\mathbb{C}^N$ to $\mathbb{C}^{N/2}$.

The CKKS scheme provides the following operations:-

KeyGen(N) Let $s(X) \in \mathbb{Z}_q[X]/(X^N+1)$ be the secret polynomial and $p(X) = (-a(X)\cdot s(X)+e(X), a(X))$ be the public polynomial where $a(X) \in \mathbb{Z}_q[X]/(X^N+1)$ is a polynomial chosen uniformly random and $e(X) \in \mathbb{Z}_q[X]/(X^N + 1)$ is a small noisy polynomial. Let $r(x) = (-a(X) \cdot s(X) + b \cdot s(X)^2 + e(X), a(X))$ be the relinearisation key where $b \in \mathbb{Z}_q$ is a large integer.

Encode(z) To encode a message vector $z \in \mathbb{C}^{N/2}$ to a message polynomial $m(X) \in \mathcal{R}$, we first expand the message vector z from $\mathbb{C}^{N/2}$ to $\mathbb{H}$ by applying $\pi^{-1}(z)$. Then we appropriately scale the vector by multiplying a scaling factor Δ followed by random rounding to $\lfloor \Delta \cdot \pi^{-1}(z) \rceil$. Scaling is done to achieve predefined precision since precision bits may be lost due to rounding. To obtain the message polynomial we apply the inverse of canonical embedding σ^{-1} and get $m(X) = \sigma^{-1}(\lfloor \Delta \cdot \pi^{-1}(z) \rceil) \in \mathcal{R}$.

Decode($m(X)$) To decode a message polynomial $m(X) \in \mathcal{R}$ to a message vector $z \in \mathbb{C}^{N/2}$, we first apply the canonical embedding σ to get $z = \lfloor \Delta \cdot \pi^{-1}(z) \rceil \in \mathbb{H}$. Then we divide it by the scaling factor Δ to obtain $\Delta^{-1} \lfloor \Delta \cdot \pi^{-1}(z) \rceil \approx \pi^{-1}(z)$. To obtain the message vector, we project the vector using π and get $\pi(\pi^{-1}(z)) = z \in \mathbb{C}^{N/2}$.

Encrypt($m(X), p(X)$) To obtain the ciphertext polynomial $c(X)$ corresponding to the message polynomial $m(X) \in \mathcal{R}$, we apply the RLWE encryption and get $c(X) = (c_0(X), c_1(X)) = (m(X), 0) + p(X) = (m(X) - a(X) \cdot s(X) + e(X), a(X)) \in (\mathbb{Z}_q[X]/(X^N + 1))^2$.

Decrypt($c(X), s(X)$) To obtain the message polynomial corresponding to the ciphertext polynomial $c(X) \in (\mathbb{Z}_q[X]/(X^N+1))$, we apply the RLWE decryption using the secret polynomial $s(X)$ and get $m(X) \approx c_0(X) + c_1(X) \cdot s = m(X) + e(X)$

Apart from the above operations, it also provides specialised ciphertext operations which include:-

Multiply($c(X), c'(X)$) Multiplication of two ciphertexts $c(X)$ and $c'(X)$ generates a ciphertext $c_m(X) = (c_0c'_0, c'_0c_1 + c_0c'_1, c'_0c'_1) = (c_0(X), c_1(X), c_2(X))$. Then ciphertext is relinearised and modulus is switched subsequently.

Relinearise($c_m(X), r(X)$) Relinearisation reduces the size of the ciphertext after multiplication of two ciphertexts. Let $c_m(X) = (c_0(X), c_1(X), c_2(X))$ be the resultant ciphertext after multiplication of two ciphertexts. After relinearisation, we obtain the ciphertext $c_r(X) = (c_0(X), c_1(X)) + \lfloor b^{-1} \cdot c_2(X) \cdot r(X) \rceil mod(q)$

Modulus Switching(c(X)) Modulus switching is rescaling of the ciphertext after multiplication in the RNS system. In addition to rescaling(dividing by scale and rounding), we the take ciphertext modulus of the next prime in the chain(lower level). The ciphertext obtained after modulus switching $c_s(X) = \lfloor \Delta^{-1} \cdot c(X) \rceil mod(q_{l-1}) = \lfloor \frac{q_{l-1}}{q_l} \cdot c(X) \rceil mod(q_{l-1})$.

Some other operations that are also supported by CKKS scheme are:-

- add($c(X), c'(X)$) - to add 2 ciphertext polynomials.
- rotateLeft($c(X), i$) - to rotate the ciphertext polynomial by i positions left.
- rotateRight($c(X), i$) - to rotate the ciphertext polynomial by i positions right.

2.2 Principal Component Analysis(PCA)

Principal component analysis(PCA) is an unsupervised dimensionality linear reduction technique. The PCA method searches for dimensions along which variance is maximized and from which one can reconstruct the original data with

minimal reconstruction error. Let u be the dimension that maximizes the variance of a vector x after projection in that direction. So, PCA could be formulated as maximization of variance problem:-

$$\begin{aligned} \max_{u} \quad & \frac{1}{n}\sum_{i=1}^{n}(u^T x_i - u^T \mu)^2 \\ \max_{u} \quad & \frac{1}{n} u^T \Sigma u \\ \text{subject to} \quad & \|u\| = 1 \end{aligned} \tag{1}$$

where $\Sigma = \sum_{i=1}^{n}(x_i - \mu)(x_i - \mu)^T$ is the covariance matrix and $\mu = \sum_{i=1}^{n} u_i$ is the mean vector.

From the above formulation, it is evident that the solution to the maximization problem would be the largest eigenvector of the covariance matrix Σ. To find the largest eigenvector, we use an iterative technique called the Power method. The power method finds the dominant eigenvector of a given matrix A by repeatedly multiplying a random vector u. As the number of iterations i increases, the product $A^i u$ converges to the largest eigenvector.

Algorithm 1 First principal component(Power Method)

Input: X : Data Matrix of row vectors.
Output: λ, w: Largest eigenvalue and corresponding eigenvector of X.

```
w ←$ ℝ^d
for i = 1 to t do
    s = Σ_{x∈X} x^T (x · w)
    λ = ‖s‖
    w = s/‖s‖
end for
return λ, w
```

In algorithm 1, instead of using covariance matrix Σ, we use the sum of the outer product of x_i's. In this approach, we do not require to store the covariance matrix Σ and only store a vector. Also, we are not required to perform any matrix operations, which could be useful for us in the homomorphic setting.

If we desire to find subsequent($2^{nd}, 3^{rd}, \cdots\ l^{th}$largest) eigenvectors then we need to use the Eigen shift procedure. The Eigen shift procedure would remove the largest eigenvalue and corresponding eigenvector from A.

By combining the Eigen shift procedure with the power method, we would be able to find out l largest eigenvectors of covariance matrix Σ, which would be the top l principal components of the given data matrix X.

Algorithm 2 Eigen Shift Procedure

Input: Σ : Covariance Matrix, λ: Largest eigen value of Σ, w: Eigen vector corresponding to λ.
Output: Σ': Shifted covariance Matrix
$\Sigma' \leftarrow \Sigma - \lambda \cdot w^T w$
return Σ'

2.3 Goldschmidt's Algorithm

Goldschmidt's algorithm[20][4] is an iterative algorithm that finds the value of a fraction. To find the value of the fraction a_0/b_0, it multiplies a series of variables $r_0, r_1, \cdots$ to both numerator(a_0) and denominator(b_0) such that the value of the resultant denominator converges to 1 and the value of the resultant numerator tends to the desired result.

$$\frac{a_0}{b_0} = \frac{a_0}{b_0} \cdot \frac{r_0}{r_0} \cdot \frac{r_1}{r_2} \cdots \frac{r_\alpha}{r_\alpha}, \quad b_0 \cdot r_0 \cdot r_1 \cdots r_\alpha \to 1$$

An initial guess of value the $r_0 \approx 1/b_0$ is required. An good approximation of r_0 is considered to be when $3/4 \leq r_0 b_0 \leq 3/2$. The successive values of the fraction after each iteration are estimated as:-

$$\frac{a_i}{b_i} = \frac{a_{i-1}}{b_{i-1}} \cdot \frac{r_{i-1}}{r_{i-1}}, \quad \text{and } r_i = 2 - b_i \ \ \forall i = 1, 2, \cdots, \alpha$$

Algorithm 3 Top l principal component(Power Method)

Input: X : Data Matrix of row vectors.
Output: Λ, W: Largest l-eigenvalues and corresponding eigenvectors of X.
$W \leftarrow \{\}$
$\Lambda \leftarrow \{\}$
for components = 1 to l **do**
  $r \xleftarrow{\$} \mathbb{R}^d$
  for $i = 1$ to t **do**
    $s1 = \sum_{x \in X} x^T (x \cdot r)$
    $s2 = \sum_{w \in W, \lambda \in \Lambda} \lambda w^T (w \cdot r)$
    $s = s1 - s2$
    $\lambda = \|s\|$
    $r = \frac{s}{\|s\|}$
  end for
  $W \leftarrow W \cup r$
  $\Lambda \leftarrow \Lambda \cup \lambda$
end for
return Λ, W

Using Goldschmidt's algorithm, we can find the square root and its inverse simultaneously. The fused multiply-add version of Goldschmidt's algorithm[27] is mentioned in algorithm 4.

Algorithm 4 Goldschmidt's Algorithm

Input: val
Output: x: The square root of val, h: The inverse square root of val.

```
y ≈ 1/√val
x ← val · y
h ← y/2
for i = 1 to l do
    r ← 0.5 − xh
    x ← x + xr
    h ← h + hr
end for
return x, 2h
```

2.4 R2 Score

R2, also known as the coefficient of determination measures the goodness of fit of a model. It computes the amount of variance captured by the dependent variables in a model. Let y_i be a true output value and y_i' be the corresponding output predicted by the model. Then the coefficient of determination(R2 score) is defined as :-

$$R2 = 1 - \frac{SS_{res}}{SS_{total}} \tag{2}$$

where $SS_{res} = \sum_i (y_i - y_i')^2$ is the sum of squares of residuals and $SS_{total} = \sum_i (y_i - \bar{y})^2$ is the total variance.

3 Vector Operations

3.1 Norm and inversion by norm

Computing the norm of a vector requires a square root operation to be performed. Since we can not perform square root directly on ciphertext, we use Goldschmidt's algorithm to find the square root of a number as mentioned in [4]. The advantage of using Goldschmidt's algorithm is that along with the square root, it also finds the inverse of the square root, which is precisely what we need. But the Goldschmidt's algorithm requires a good initial approximation of $\frac{1}{\sqrt{x}}$ for faster convergence.

We could use a good initial guess for $\frac{1}{\sqrt{x}}$ using the fast inverse square root algorithm [18] [26]. But such approximations are difficult to realize in the homomorphic setting because it requires conversion from IEEE representation(single and double precision floating point) to integer and vice-versa. Instead, we use a linear approximation of $\frac{1}{\sqrt{x}}$ in a given interval and use it as an initial guess for Newton's method. Then we perform few iterations of Newton's method to improve our guess.

For computing Newton's method on $\frac{1}{\sqrt{x}}$ we have $f(x) = x^{-2} - b$. The derivative would be $f'(x) = -2x^{-3}$. In each iteration, the update would be :-

$$\begin{aligned} x_{i+1} &= x_i - \frac{f(x_i)}{f'(x_i)} \\ \implies x_{i+1} &= \frac{x_i}{2}(-bx_i^2 + 3) \end{aligned} \tag{3}$$

For linear approximation of $\frac{1}{\sqrt{x}}$, we use constrained linear regression which is formulated as the following minimization problem :-

$$\begin{aligned} \min_{w} \quad & \frac{1}{n}\sum_{i=1}^{n}(y_i - w^T x_i)^2 \\ \text{subject to} \quad & y_i - w^T x_i \geq 0 \quad \forall\, i = \{1, 2, \cdots n\} \end{aligned} \tag{4}$$

The additional constraints $y_i - w^T x_i \geq 0\ \forall i$ are necessary because $\frac{1}{\sqrt{x}}$ would have a positive and a negative value. The constraint ensures that the initial guess is a positive value and Newton's method doesn't diverge towards the negative value. The minimization problem in equation 4 is solved using SLSQP(Sequential Least Squares Quadratic Programming) solver.

3.2 Ciphertext Packing

We consider all vectors as row vectors and thus consider only row-wise packing of vectors in a ciphertext. We partition the number of ciphertext slots equally among all the vectors such that the number of zeros present in each partition is the same. Let d be the dimension of each vector and N be the total number of ciphertext slots. Then the number of partitions in the ciphertext would be N/k where k is a factor of N larger than or equal to d. In this paper, we consider N to be a power of 2. So finding k would be equivalent to calculating the closet power of 2 greater than or equal to d. This can be done very efficiently using binary search in $O(\log\log(N))$ steps.

Suppose $u_1, u_2, \cdots u_j$ are the vectors to be packed in a ciphertext where $j = N/k$. Let $z = k - d$ be the number of trailing zeros for each vector. Then the vectors $u_1, u_2, \cdots u_j$ would packed in a ciphertext as :-

$$c = [u'_1, u'_2, \cdots, u'_j] \tag{5}$$

where $u'_i = \text{Ciphertext}(u_i || \underbrace{0, 0, \cdots, 0}_{z\ times})$, $\forall\, i = \{1, 2, \cdots, j\}$.

Each vector, along with trailing zeros, could be thought of as a sub-ciphertext of size k. Instead of partitioning the number of ciphertext slots into $\lceil N/d \rceil$ partitions, we instead preferred N/k partitions because the former doesn't guarantee equipartition of ciphertext slots among all vectors. If there are many vectors, then the last vector would spill over to the next ciphertext leaving behind trailing zeros at the end of each ciphertext which is not ideal for our operations. Hence, we distribute the zeros equally among all the vectors packed in a ciphertext.

If a particular row vector needs to be packed in an entire ciphertext, then the row vector is appended with $z = k - d$ trailing zeros to form a sub-ciphertext. This sub-ciphertext would be then be repeated in each partition. For example, let v be a vector that has to be packed into an entire ciphertext. Then the sub-ciphertext $v' = \text{Ciphertext}(v||\underbrace{0, 0, \cdots, 0}_{z\ times})$. Then the ciphertext would be:-

$$c = [v', v', \cdots, v']$$

In [19], each vector was packed into a separate ciphertext, whereas in [22] the entire dataset was packed into a single ciphertext. Our sub-ciphertext packing technique is somewhat in between those two packing techniques. Partitioning the ciphertext into sub-ciphertext helps to achieve an overall reduction in operations. The operations that were earlier polynomial in the length of ciphertext(both in [19] and [22]) now would become polynomial in the length of sub-ciphertext, which is almost equal to the vector's length. Another advantage of using sub-ciphertext packing is that it provides much parallelism as operations are performed independently on each ciphertext.

3.3 Vector operations on ciphertext and sub-ciphertexts

Sum of elements Since we consider the size of ciphertext to be a power of 2, to add all the elements in a ciphertext homomorphically, we need to rotate the ciphertext by increasing power of 2 and add it to itself.

Algorithm 5 Sum(c)

Input: c : Ciphertext.
Output: c': Sum of all the elements in ciphertext c.

```
temp ← Ciphertext()
c' ← c
for i = 0 to log(N) − 1 do
    temp ← rotateLeft(c', 2^i)
    c' ← c' + temp
end for
return c'
```

Partial sum of ciphertext Partial sum of a ciphertext is the sum of all the elements within a sub-ciphertext for every sub-ciphertext in any given ciphertext. Suppose $c = [u'_1, u'_2, \cdots, u'_j]$ is a ciphertext and $u'_i = (u_i||0, 0, \cdots, 0)$ be a sub-ciphertext . Then partial sum of c would result in a ciphertext $c' = [Su_1, Su_2, \cdots, Su_j]$ where $Su_i = \left(\sum_{q=1}^{k} u_{iq}, \sum_{q=1}^{k} u_{iq}, \cdots, \sum_{q=1}^{k} u_{iq}\right)$.

Algorithm 6 PartialSum(c)

Input: c : Ciphertext.
Output: c': Co-ordinate wise sum of all sub-ciphertexts in c.

```
init ← Ciphertext(1, 1, ··· ||0, 0, ··· ) {1st half is all 1's and 2nd half is all 0's}
c' ← Ciphertext(c)
for i = log (k) − 1 to 0 do
   temp ← rotateRight(init, 2^i)
   s1 ← multiply(init, c')
   s2 ← multiply(temp, c')
   s2 ← rotateLeft(s2, 2^i)
   c' ← add(s1, s2)
   if i > 0 then
      temp ← rotateLeft(temp, 2^i + 2^(i−1))
      init ← multiply(temp, init)
   end if
end for
for i = 0 to log (k) do
   temp ← rotateRight(c', 2^i)
   c' ← add(c', temp)
end for
return c'
```

Let $c = [(1, 2, 3, 4), (2, 3, 4, 5)]$ be a ciphertext with $j = 2$ sub-ciphertexts. Then the partial sum of ciphertext would be $c' = [(10, 10, 10, 10), (14, 14, 14, 14)]$.

Lemma 1. *Let k be the size of sub-ciphertext. Then the multiplicative depth required by the algorithm 6 is* $\log(k)$

Sum of sub-ciphertexts Addition of all the sub-ciphertexts is the coordinate-wise sum of all sub-ciphertexts in a ciphertext. Suppose $c = [u'_1, u'_2, \cdots, u'_j]$ is a ciphertext with j sub-ciphertexts. Then the sum of all sub-ciphertexts in ciphertext c would result in a ciphertext $c' = [\sum_{i=1}^{j} u'_{i1}, \sum_{i=1}^{k} u'_{i2}, \cdots, \sum_{i=1}^{k} u'_{ij}]$

Let $c = [(1, 2, 3, 4), (2, 3, 4, 5)]$ be a ciphertext with $j = 2$ sub-ciphertexts. Then the sum of sub-ciphertexts would be $c' = [(1 + 2, 2 + 3, 3 + 4, 4 + 5), (1 + 2, 2 + 3, 3 + 4, 4 + 5)] = [(3, 5, 7, 9), (3, 5, 7, 9)]$.

Algorithm 7 SubSum(c)

Input: c : Ciphertext.
Output: c': Coordinate-wise sum of all sub-ciphertexts in c.
temp ← Ciphertext()
$c' \leftarrow c$
for $i = 0$ to $\log(j) - 1$ **do**
temp ← rotateLeft$(c', 2^i \cdot k)$
$c' \leftarrow c' +$ temp
end for
return c'

Inner Product Let v be ciphertext packed with a vector in all its sub-ciphertexts. The inner product of v and a ciphertext c packed with j sub-ciphertexts can be found by multiplying each element co-ordinate wise and then performing a partial sum on the resultant ciphertext.

Algorithm 8 InnerProduct(c, v)

Input: c : Ciphertext, v : Vector packed in an entire Ciphertext.
Output: c': Inner product of j vectors with v.
$c' \leftarrow$ multiply(c, v)
$c' \leftarrow$ PartialSum(c')
return c'

Lemma 2. *Let k be the size of each sub-ciphertext. Then the multiplicative depth of algorithm 8 is* $\log k + 1$

4 Homomorphic Evaluations

After defining all the vector operations and ciphertext packing technique, we will now move forward and describe the methods for performing PCA using CKKS homomorphic scheme. This section would first define the homomorphic version of Goldschmidt's algorithm and the power iteration method essential for performing PCA. Finally, we bundle together all the methods to produce a single method for performing PCA homomorphically.

4.1 Homomorhpic Goldschmidt's algorithm

Goldschmidt's algorithm [27] is an iterative algorithm that computes the square root and its inverse simultaneously. It converges faster than Newton's method. Similar to newton's method, Goldschmidt's algorithm also requires a good initial approximation(of $\frac{1}{\sqrt{x}}$) for faster convergence. We use algorithm 9 to obtain a good initial approximation of $\frac{1}{\sqrt{x}}$.

Approximation of $\frac{1}{\sqrt{x}}$ Algorithm 9 provides us with an initial approximation of $\frac{1}{\sqrt{x}}$. It is a homomorphic adaptation of the fast square root inverse algorithm[18]. The approach is similar to the one in [18] where we first find a linear approximation of $\frac{1}{\sqrt{x}}$ and then use that as an initial guess for newton's method and improve upon our approximation in a few iterations.

Algorithm 9 InvNormApprox(*norm*)

Input: *norm* : Ciphertext with sum of squares.
Output: *guess*: Approximate inverse of norm of *c*.

```
neg_half ← Ciphertext(-0.5) {A ciphertext with all its entries as -0.5}
three_half ← Ciphertext(1.5) {A ciphertext with all its entries as 1.5}
guess ← linearApprox(norm)
for i = 1 to iterations do
    sq ← multiply(sq, square(guess))
    sq ← multiply(multiply(guess, neg_half), sq)
    temp ← multiply(three_half, guess)
    guess ← add(temp, sq)
end for
return guess
```

Lemma 3. *Let l_1 be the number of iterations in algorithm 9. Then the multiplicative depth of algorithm 9 is* $3l_1 + 1$

Lemma 4. *Let l_1, l_2 be the number of iterations in algorithms 9 and 10 respectively. Let k be the size of the sub-ciphertext. Then the multiplicative depth of algorithm 10 is* $\log k + 3(l_1 + l_2) + 2$

4.2 Homomorphic Power Method

The homomorphic power method computes the top l principal components of the data matrix X homomorphically. It finds the largest eigenvector of the covariance matrix and uses the Eigen shift procedure to find the subsequent eigenvectors. It uses algorithm 10 to compute the norm and its inverse.

Lemma 5. *Let l_1, l_2, l_3 be the number of iterations in algorithms 9, 10, 11 respectively. Let k be the size of the sub-ciphertext and l be the number principal components. Then the multiplicative depth of the algorithm 11 is* $ll_3(2\log(k) + 3(l_1 + l_2) + 7)$

4.3 Homomorphic PCA

We perform PCA(principal component analysis) homomorphically using algorithm 12. In algorithm 12, the client first computes the mean vector and subtracts the mean vector from all other vectors to center the data matrix about

Algorithm 10 Goldschmidt(c)

Input: c : Ciphertext,
Output: x: Norm of c, h: Inverse of norm of c
half ← Ciphertext(0.5) {A ciphertext with all its entries as 0.5}
neg_one ← Ciphertext(-1.0) {A ciphertext with all its entries as -1.0}
norm ← ReEncrypt(PartialSum(square(c)))
y ← ReEncrypt(InvNormApprox(norm))
x ← multiply(norm, y)
h ← multiply(norm, half)
for $i = 1$ to *iterations* **do**
temp_r ← multiply(multiply(x, h), neg_one)
r ← add(temp_r, half)
x ← add(x, multiply(x, r))
h ← add(h, multiply(h, r))
if depth(x) ≤ 2 **then**
x ← ReEncrypt(x)
h ← ReEncrypt(h)
end if
end for
two ← Ciphertext(2.0) {A ciphertext with all its entries as 2.0}
h ← multiply(h, two)
return x, h

Algorithm 11 PowerMethod(X)

Input: X: Ciphertext packing of the original data matrix
Output: W: Top l principal components of X
$W \leftarrow \{\}$
$\Lambda \leftarrow \{\}$
neg_one ← Ciphertext(-1.0) {A ciphertext with all its entries as -1.0}
for components = 1 to l **do**
$r \xleftarrow{\$} \mathbb{R}^d$
for $i = 1$ to *iterations* **do**
$s1 \leftarrow \sum_{x \in X}$ multiply(x, InnerProduct(x, r))
$s1 \leftarrow$ SubSum($s1$)
$s1 \leftarrow$ ReEncrypt($s1$)
$s2 \leftarrow \sum_{\lambda \in \Lambda; w \in W}$ multiply(λ, multiply(w, InnerProduct(w, r))
$s2 \leftarrow$ ReEncrypt($s2$)
$s2 \leftarrow$ multiply($s2$, neg_one)
$s \leftarrow$ add(s1, s2)
eig_val, eig_inv ← Goldschmidt(s)
$r \leftarrow$ ReEncrypt(multiply(eig_inv, s))
end for
$W \leftarrow W \cup r$
$\Lambda \leftarrow \Lambda \cup$ eig_val
end for
return W

its mean. Then, the original data's principal components are obtained from the server using algorithm 11 by performing computations on encrypted data. Finally, the principal components are multiplied by the client to get the lower dimensional representation of the original data.

Algorithm 12 HPCA(X)

Input: X : Data Matrix with row vectors
Output: X_red: Reduced data matrix with k principal components of X
Server:
Recieve X' from Client.
$W \leftarrow$ PowerMethod(X')
Send W to Client.
Client:
$X\_tmp \leftarrow X -$ mean(X)
$X' \leftarrow$ Encrypt(X_tmp)
Send X' to Server.
Receive W from Server.
$W \leftarrow$ Decrypt(W)
$X\_red \leftarrow X\_tmp \cdot W$
return X_red

5 Implementation details and Results

We implemented all the procedures described in this paper using Python. We used SEAL-Python[24](A python binding for Microsoft SEAL library[23]) for the implementation of the CKKS scheme. All the experiments were run on a machine with Intel® Core™ i5-7200U CPU @ 2.50GHz having 4 cores. The machine ran on 64-bit Ubuntu 20.04.2 LTS with a memory of 7.6 GiB and a disk capacity of 1TB. We conducted experiments on seven datasets. Four of them were categorical datasets - air quality[11], Parkinsons telemonitoring[25], winequailty-red[9] and winequality-white[9]. The other three were image datasets - MNIST handwritten digits[17], Fashion-MNIST [28] and Yale face database[1]. We computed each dataset's first few principal components and verified our HPCA algorithm's efficiency by computing the R2 score on the reconstructed data. R2 score gives a measure of the variance captured in the reconstructed data and provides the goodness of fit.

Datasets were scaled appropriately so that eigenvalues are small enough to be handled properly. The Yale database was converted from three channels to a single channel and then resized from 195×231 pixels to 16×16 pixels using bicubic interpolation. Similarly, the MNIST handwritten and Fashion-MNIST datasets were resized from 28×28 pixels to 16×16 by trimming the images' outer boundaries as they contain 0's only. We also conducted few experiments by eliminating the last five features of the Parkinsons telemonitoring dataset

to make the number of features exact power of 2(from 21 to 16). For linear approximation of $\frac{1}{\sqrt{x}}$, we considered the interval $[0.001, 750]$ and obtained the coefficients for the line $y = ax + b$ as $a = -0.00019703$ and $b = 0.14777278$ using SLSQP solver. The negative slope illustrates the decreasing trend of $\frac{1}{\sqrt{x}}$ function. The values for number of iterations l_1, l_2 and l_3 for algorithms 9, 10 and 11 were fixed to be 2, 4, 4 respectively.

5.1 Parameter selection

We did not use bootstrapping in our implementation as it is not provided as an inbuilt API in the SEAL library. Instead, we re-encrypted some of the parameters to get rid of the noise in them. Re-encryptions are used to be ideally replaced by bootstrapping without making changes to any other part of the algorithm. We ensured that we do not have to re-encrypt any ciphertext from the input, making the number of re-encryptions independent of the dataset. For this, we observe that the maximum depth required by each ciphertext is $\log(k) + 2$ in algorithm 11. To further reduce the number of re-encryptions, we also restricted the re-encryption of eigenvectors. For this, we require a maximum depth of $\log(k) + 3$ in algorithm 11. So, the maximum depth used was $\log(k) + 3$.

Re-encryption could be thought of as a server sending a ciphertext back to the client in public-key setting. The client re-encrypts the ciphertext and sends it back to the server. This would require some bytes of communication between the server and the client. For $N = 16384$, we communicate about 128KB and for $N = 32768$, we communicate about 256KB of information in a single round trip. Using lemma 6, we compute the total communication cost between the client and server as shown in table 1.

Lemma 6. *Let l_2, l_3 be the number of iterations of algorithms 10, 11 respectively. Let l be the number of principal components and z be the maximum depth used. Then the number of re-encryptions required by algorithm 12 is $l(l_3(5 + \lceil \frac{3l_2}{z} \rceil))$*

N	z	l	Re-encryptions	Bits communicated(in MB)
16384	7	2	72	9.009
16384	8	2	56	7.007
32768	11	4	112	14.014
32768	11	5	140	17.517
32768	11	6	168	21.021

Table 1: Communication cost for re-encryption

In all of our experiments, we used the polynomial modulus degree(N) of 16384 and 32768. This gave us a total of 438, 881 bits respectively for coefficient modulus to achieve 128-bit security. The scale was chosen to be 2^{40} to achieve 20 bits of precision after the decimal point.

5.2 Results

We computed the first few principal components for each dataset. Then we measured the R2 score of the reconstructed data to use it as an evaluation metric. We also compared the results obtained by performing PCA on un-encrypted data to verify the efficiency of our HPCA algorithm. An R2 score between $0.3 - 0.7$ is considered a good fit for the original data. We also measured the total time taken by all the procedures on different datasets. Table 2 summarizes all the results obtained after experimentation on different datasets [1].

Dataset	d	k	N	$j = \frac{N}{2k}$	n	l	Depth	R2 Score (Encrypted)	R2 Score (Un-encrypted)	Time Taken (in mins)
MNIST	256	256	32768	64	100	4	11	0.15667	0.3320	9.2965
	256	256	32768	64	200	4	11	0.1410	0.4124	12.9907
Fashion-MNIST	256	256	32768	64	100	4	11	0.4199	0.5013	9.2878
	256	256	32768	64	200	4	11	0.4111	0.4762	12.9968
Yale	256	256	32768	64	165	4	11	0.5292	0.5191	11.0622
	256	256	32768	64	165	5	11	0.5729	0.6012	15.3264
	256	256	32768	64	165	6	11	0.5790	0.6758	19.8646
Winequailty-white	11	16	16384	512	4898	2	7	0.2517	0.25502	2.4066
	11	16	32768	1024	4898	2	11	0.2544	0.25502	5.2634
Winequailty-red	11	16	16384	512	1599	2	7	0.1487	0.20001	1.4447
	11	16	32768	1024	1599	2	11	0.1463	0.20001	2.8728
Air Quality	13	16	16384	512	9357	2	7	0.6012	0.6062	3.3823
	13	16	32768	1024	9357	2	11	0.6054	0.6062	8.6394
Parkinsons	16	16	16384	512	9357	2	7	0.1509	0.1604	2.2068
	21	32	16384	256	9357	2	8	0.14488	0.1604	5.4160
	16	16	32768	1024	9357	2	11	0.1506	0.1604	5.8734

Table 2: Performance of HPCA(Homomorphic PCA) algorithm on different datasets

From table 2 we observe that our HPCA algorithm can capture variance to a considerable amount in different datasets. It does not perform well on the MNIST dataset, with a significant difference between the R2 score on encrypted and unencrypted data. It performs moderately well on Fashion-MNIST and winequality-red datasets. But it performs pretty well on the Yale face database, air quality, winequality-white and Parkinson's telemonitoring datasets. The datasets on which our HPCA algorithm's performance on encrypted data are similar to that of PCA on un-encrypted data are the datasets that either have a considerable difference between their successive eigenvalues of the variance is captured by the first few principal components. Figure 1 shows the first 4 eigenfaces obtained by the HPCA algorithm. Figure 2 shows how an image looks after reconstruction using a few principal components.

[1] Time taken mentioned in table 2 doesn't take into account the communication time required for re-encryption.

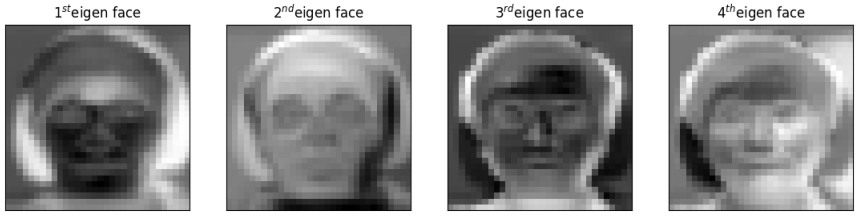

Fig. 1: First four eigenfaces for Yale face Database obtained using HPCA algorithm

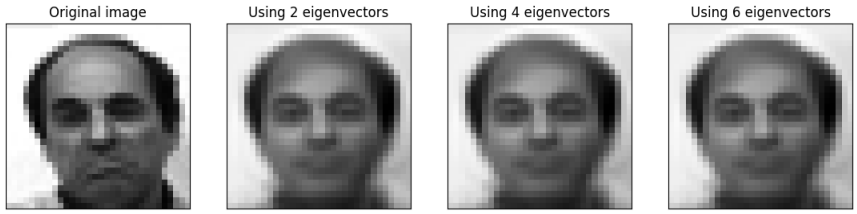

Fig. 2: Reconstruction of a image from Yale face database using principal components obtained from HPCA algorithm

6 Conclusion and Future work

This paper presents the HPCA algorithm that performs PCA on encrypted data using CKKS homomorphic encryption scheme. Our HPCA algorithm does not require any matrix operations and doesn't require us to store the original data's covariance matrix. This reduces the memory and computational requirements significantly. Calculation of the norm and its inverse are the most computationally heavy operations in the HPCA algorithm. Most of the previous works, including [22] and [19] do not compute norm or its inverse homomorphically. Instead, the client is required to calculate the norm after decryption. This makes their algorithm interactive in nature for computing subsequent principal components. We tried to overcome this problem and tried to reduce the client's computational burden by evaluating the norm and its inverse homomorphically on the client-side. With the use of bootstrapping, our algorithm would become totally non-interactive.

Numerical stability of various algorithms could be studied as appropriate scaling of data is required for obtaining the first few principal components. Iterative algorithms like the power method accumulate noise after each iteration. This seems to be the major drawback of our HPCA algorithm, which restricts us from computing only the first few principal components. It would be interesting to learn how noise grows for each component of the HPCA algorithm with each iteration and how it affects the maximum number of principal components found in a given setting. Other alternatives of power method like gradient descent could also be used instead. Finally, we measure the R2 score of the reconstructed data

for different datasets to demonstrate our algorithm's efficiency. The R2 score obtained from our HPCA algorithm is almost comparable to the R2 score obtained on most of the datasets without encryption. The implementation of all the algorithms can be found in [21]. A parallelized version of HPCA could also be developed to achieve faster runtime.

Acknowledgement

We would like to sincerely thank all the anonymous reviewers for their valuable feedback and Dr.Kannan Srinathan for providing the necessary motivation.

References

1. Belhumeur, P.N., Hespanha, J.P., Kriegman, D.J.: Eigenfaces vs. fisherfaces: recognition using class specific linear projection. IEEE Transactions on Pattern Analysis and Machine Intelligence **19**(7), 711–720 (1997). https://doi.org/10.1109/34.598228
2. Boura, C., Gama, N., Georgieva, M., Jetchev, D.: Chimera: Combining ring-lwe-based fully homomorphic encryption schemes. Cryptology ePrint Archive, Report 2018/758 (2018), https://eprint.iacr.org/2018/758
3. Boura, C., Gama, N., Georgieva, M., Jetchev, D.: Simulating homomorphic evaluation of deep learning predictions. Cryptology ePrint Archive, Report 2019/591 (2019), https://eprint.iacr.org/2019/591
4. Cetin, G.S., Doroz, Y., Sunar, B., Martin, W.J.: Arithmetic using word-wise homomorphic encryption. Cryptology ePrint Archive, Report 2015/1195 (2015), https://eprint.iacr.org/2015/1195
5. Chen, H., Chillotti, I., Song, Y.: Improved bootstrapping for approximate homomorphic encryption. Cryptology ePrint Archive, Report 2018/1043 (2018), https://eprint.iacr.org/2018/1043
6. Cheon, J.H., Han, K., Kim, A., Kim, M., Song, Y.: Bootstrapping for approximate homomorphic encryption. Cryptology ePrint Archive, Report 2018/153 (2018), https://eprint.iacr.org/2018/153
7. Cheon, J.H., Kim, A., Kim, M., Song, Y.: Homomorphic encryption for arithmetic of approximate numbers. Cryptology ePrint Archive, Report 2016/421 (2016), https://eprint.iacr.org/2016/421
8. Cheon, J.H., Kim, A., Yhee, D.: Multi-dimensional packing for heaan for approximate matrix arithmetics. Cryptology ePrint Archive, Report 2018/1245 (2018), https://eprint.iacr.org/2018/1245
9. Cortez, P., Cerdeira, A., Almeida, F., Matos, T., Reis, J.: Modeling wine preferences by data mining from physicochemical properties. Decision Support Systems **47**(4), 547–553 (2009). https://doi.org/https://doi.org/10.1016/j.dss.2009.05.016, https://www.sciencedirect.com/science/article/pii/S0167923609001377, smart Business Networks: Concepts and Empirical Evidence
10. Crockett, E.: A low-depth homomorphic circuit for logistic regression model training. Cryptology ePrint Archive, Report 2020/1483 (2020), https://eprint.iacr.org/2020/1483

11. De Vito, S., Massera, E., Piga, M., Martinotto, L., Di Francia, G.: On field calibration of an electronic nose for benzene estimation in an urban pollution monitoring scenario. Sensors and Actuators B: Chemical **129**(2), 750–757 (2008). https://doi.org/https://doi.org/10.1016/j.snb.2007.09.060, https://www.sciencedirect.com/science/article/pii/S0925400507007691
12. Gentry, C.: A fully homomorphic encryption scheme. Ph.D. thesis, Stanford University (2009), https://crypto.stanford.edu/craig
13. Gentry, C.: Fully homomorphic encryption using ideal lattices. In: In Proc. STOC. pp. 169–178 (2009)
14. Han, K., Hong, S., Cheon, J.H., Park, D.: Efficient logistic regression on large encrypted data. Cryptology ePrint Archive, Report 2018/662 (2018), https://eprint.iacr.org/2018/662
15. Kim, A., Song, Y., Kim, M., Lee, K., Cheon, J.H.: Logistic regression model training based on the approximate homomorphic encryption. Cryptology ePrint Archive, Report 2018/254 (2018), https://eprint.iacr.org/2018/254
16. Kim, M., Song, Y., Wang, S., Xia, Y., Jiang, X.: Secure logistic regression based on homomorphic encryption: Design and evaluation. Cryptology ePrint Archive, Report 2018/074 (2018), https://eprint.iacr.org/2018/074
17. LeCun, Y., Cortes, C., Burges, C.: Mnist handwritten digit database. ATT Labs [Online]. Available: http://yann.lecun.com/exdb/mnist **2** (2010)
18. Lomont, C.: Fast inverse square root. Tech. rep., Purdue University (2003), http://www.matrix67.com/data/InvSqrt.pdf
19. jie Lu, W., Kawasaki, S., Sakuma, J.: Using fully homomorphic encryption for statistical analysis of categorical, ordinal and numerical data. Cryptology ePrint Archive, Report 2016/1163 (2016), https://eprint.iacr.org/2016/1163
20. Markstein, P.: Software division and square root using goldschmidt's algorithms (01 2004)
21. Panda, S.: Homomorphic pca. https://github.com/pandasamanvaya/Homomorphic_PCA (2021)
22. Rathee, D., Mishra, P.K., Yasuda, M.: Faster pca and linear regression through hypercubes in helib. Cryptology ePrint Archive, Report 2018/801 (2018), https://eprint.iacr.org/2018/801
23. Microsoft SEAL (release 3.6). https://github.com/Microsoft/SEAL (Nov 2020), microsoft Research, Redmond, WA.
24. Python binding for the Microsoft SEAL library. https://github.com/Huelse/SEAL-Python
25. Tsanas, A., Little, M., Mcsharry, P., Ramig, L.: Accurate telemonitoring of parkinson's disease progression by noninvasive speech tests. IEEE transactions on bio-medical engineering **57**, 884–93 (11 2009). https://doi.org/10.1109/TBME.2009.2036000
26. Wikipedia contributors: Fast inverse square root (2021), https://en.wikipedia.org/wiki/Fast_inverse_square_root
27. Wikipedia contributors: Goldschmidt's algorithm — Wikipedia, the free encyclopedia (2021), https://en.wikipedia.org/wiki/Methods_of_computing_square_roots
28. Xiao, H., Rasul, K., Vollgraf, R.: Fashion-mnist: a novel image dataset for benchmarking machine learning algorithms (2017)