Machine Learning Classification over Encrypted Data

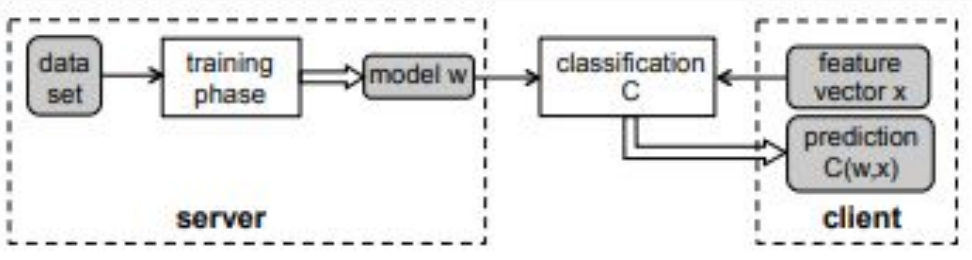
Team 3

Overview

ML is being used in a lot of different domains. However, some domains like Medical Imaging, Face Recognition requires the privacy of the data as well as the classifier.

Consider the setup where we have a classifier C which learns from trains the model w. Now we test this on an unseen vector x to output a prediction.

In case of sensitive data we need to ensure the privacy of the model w and the feature x. Ideally, the hospital and the client run a protocol at the end of which the client learns one bit (“yes/no”), and neither party learns anything else about the other party’s input. Neither the clients data is revealed, nor the training data of the hospital.



Formal Contribution

In this work, we construct efficient privacy-preserving protocols for two of the most common classifiers: hyperplane decision and Naïve Bayes

Our main work is to identify a set of core operations over encrypted data that underlie many classification protocols, which are comparison, argmax, and dot product. We use efficient protocols for each one of these.

The input and output of all our building blocks are data encrypted with additively homomorphic encryption. In addition, we provide a mechanism to switch from one encryption scheme to another

Homomorphic Encryption

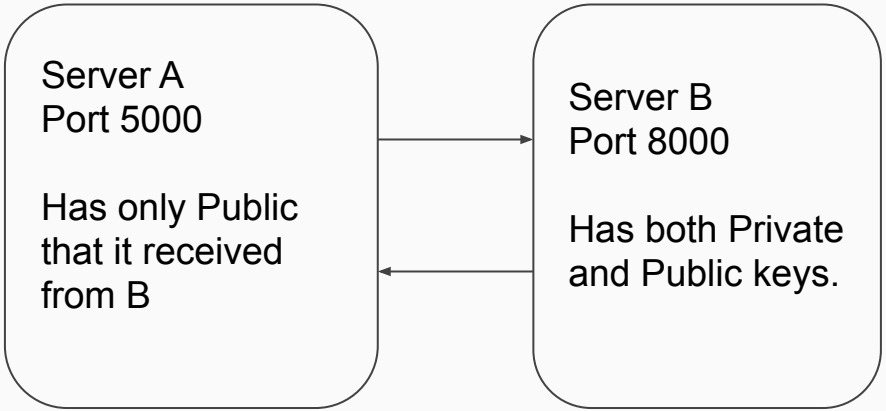
- The purpose of homomorphic encryption is to allow computation on encrypted data. Thus data can remain confidential while it is processed, enabling useful tasks to be accomplished with data residing in untrusted environments.
- A homomorphic cryptosystem is like other forms of public encryption in that it uses a public key to encrypt data and allows only the individual with the matching private key to access its unencrypted data.
- In practice, most homomorphic encryption schemes work best with data represented as integers and while using addition and multiplication as the operational functions. This means that the encrypted data can be manipulated and analyzed as though it's in plaintext format without actually being decrypted.
- Types
    - Partially Homomorphic
    - Somewhat Homomorphic
    - Levelled Fully Homomorphic
    - Fully Homomorphic

Architecture

To accurately represent and test this privacy preserving aspect, we had created entities running on separate ports. With only one of these (8000 port) holding the private keys.

All further operations, data transfers and models built considered this separation.
Example: any decrypt operation is impossible on Server A (5000 port)

Time bottlenecks like the dgk key generation was made faster using multithreading by processing iterations concurrently. This provided a speed up of approximately 4 times



Why not just FHE ?

FHE is not a viable alternative as there is **huge overhead** associated with it both in terms of time (slow computationally) and space (large storage needed for keys)

Thus for simple models it's better to combine basic building blocks built using partial homomorphic encryption only.

Homomorphic only for addition, which makes them efficient.

- Goldwasser Micali
- Paillier
- DGK

Comparison: Private Unencrypted Data

Setup

Party	A	B
Input	x	y and K_{DGK}
Output	$\delta_A \in \{0,1\}$	$\delta_B \in \{0,1\}$
Constraints	$\delta_A \oplus \delta_B = (x \leq y)$ $0 \leq x, y < 2^\ell$	

- Intuitive Idea: When comparing two integers x and y bitwise, the obvious approach is to scan both bit rows from left (the most significant part) to right searching for the first differing bit. The outcome of the comparison of these differing bits will determine the comparison result of both integers.

- The DGK protocol is used for its xor and additive homomorphism.

DGK cryptosystem

Keygen

1. Generate distinct primes vp and vq of size t.
2. Generate a prime u just greater than 2^l.
3. Generate p such that vp | p - 1 and u | p - 1.
4. Generate q such that vq | q - 1 and u | q - 1.
5. Get n = p x q.
6. Generate h of order vp x vq in Zn*.
7. Generate g of order u x vp x vq in Zn*.
8. Private Key: p, q, vp, vq
9. Public Key: n, g, h, u, t

Encryption

1. Generate a random number r of size 2.5t .
2. c = (g^m * h^r) % n

If the plaintext is negative, we simply find the c for the abs(m) and then take the modulo inverse of it to get the answer. (Homomorphism)

Decryption:

1. m' = c^(vp*vq) % n
2. If m' == 1, m = 0.
3. Else query the data lookup table with m'. (Not needed for our purpose)

- First, we compute h_r and g_r of order $LCM(p-1,q-1) = (p-1)(q-1)/GCD(p-1,q-1) = 2up'_r v_p q'_r v_q$ in $\mathbb{Z}_n^*$ with Algorithm 4.83
- h must have order $v_p v_q$ in $\mathbb{Z}_n^*$, so we set $h = h_r^{2up'_r q'_r} \pmod{n}$
- g must have order $uv_p v_q$ in $\mathbb{Z}_n^*$, so we set $g = g_r^{2p'_r q'_r} \pmod{n}$

Specifically, I use **Algorithm 4.83**, Chapter 4: Selecting an element of maximum order in $\mathbb{Z}_n^*$, where $n = pq$:

INPUT: two distinct odd primes, p, q, and the factorizations of $p-1$ and $q-1$.

OUTPUT: an element α of maximum order $lcm(p-1; q-1)$ in $\mathbb{Z}_n^*$, where $n = pq$.

1. Use Algorithm 4.80 with $G = \mathbb{Z}_p^*$ and $n = p-1$ to find a generator a of $\mathbb{Z}_p^*$.
2. Use Algorithm 4.80 with $G = \mathbb{Z}_q^*$ and $n = q-1$ to find a generator b of $\mathbb{Z}_q^*$.
3. Use Gauss's algorithm (Algorithm 2.121) to find an integer α, $1 \leq \alpha \leq n-1$, satisfying $\alpha \equiv a \pmod{p}$ and $\alpha \equiv b \pmod{q}$.
4. Return α.

For completeness, here are the two algorithms referenced above in Algorithm 4.83:

Algorithm 4.80, Chapter 4 : Finding a generator of a cyclic group:

INPUT: a cyclic group G of order n, and the prime factorization $n = p_1^{e_1} p_2^{e_2} \cdots p_k^{e_k}$

OUTPUT: a generator α of G

1. Choose a random element α in G
2. For i from 1 to k do the following:
   - Compute $b \leftarrow a^{n/p_i}$ (N.B. $\pmod{n}$)
   - If $b = 1$ then go to step 1.
3. Return α.

Algorithm 2.121, Chapter 2 (Gauss's algorithm): The solution x to the simultaneous congruences in the Chinese Remainder Theorem (Fact 2.120) may be computed as $\sum_{i=1}^{k} a_i N_i M_i \pmod{n}$, where $N_i = n/n_i$ and $M_i = N_i^{-1} \pmod{n_i}$. These computations can be performed in $O((\lg n)^2)$ bit operations.

Protocol

1) B sends the encrypted bits $[y_i]$, $0 \leq i < \ell$ to A.
2) For each i, $0 \leq i < \ell$, A computes $[x_i \oplus y_i]$ as follows:
   if $x_i = 0$ then $[x_i \oplus y_i] \leftarrow [y_i]$
   else $[x_i \oplus y_i] \leftarrow [1] \cdot [y_i]^{-1} \bmod n$.
3) A chooses a uniformly random bit δ_A and computes $s = 1 - 2 \cdot \delta_A$.
4) For each i, $0 \leq i < \ell$, A computes $[c_i] = [s] \cdot [x_i] \cdot [y_i]^{-1} \cdot (\prod_{j=i+1}^{\ell-1} [x_j \oplus y_j])^3 \bmod n$.
5) A blinds the numbers c_i by raising them to a random exponent r_i of $2t$ bits: $[c_i] \leftarrow [c_i]^{r_i} \bmod n$, and sends them in random order to B.
6) B checks whether one of the numbers c_i is decrypted to zero. If he finds one, $\delta_B \leftarrow 1$, else $\delta_B \leftarrow 0$.

$$c_i = s + x_i - y_i + 3 \sum_{j=i+1}^{\ell-1} (x_j \oplus y_j)$$

To show that in Protocol 1 indeed $\delta_A \oplus \delta_B = (x \leq y)$, we distinguigh two cases:

- If $\delta_A = 0$ then $s = 1$ so $s + x_i - y_i$ is only zero when $x_i = 0$ and $y_i = 1$. Thus when B finds $c_i = 0$ (in which case $\delta_B = 1$), we have $x < y$, and otherwise $x \geq y$.
- If $\delta_A = 1$ then $s = -1$ so $s + x_i - y_i$ is only zero when $x_i = 1$ and $y_i = 0$. Thus when B finds $c_i = 0$, we have $x > y$, and otherwise $x \leq y$.

DEMO

Comparison: Comparing Encrypted Data

Veu11 Comparison Protocol

1. This protocol implements comparison of 2 encrypted numbers.
2. Two party protocol.
3. DGK cryptosystem for comparing unencrypted integers.
4. Paillier and Goldwasser Micali Cryptosystems.
5. Party A has both the encrypted numbers.
6. Party B has the private key for Paillier and Goldwasser Micali.
7. This forms the base for all the comparison protocols.
8. Further used in argmax primitive.

Input A: $[\![a]\!]$ and $[\![b]\!]$, the bit length l of a and b, the secret key SK_{QR}, public key PK_P
Input B: Secret key SK_P, public key PK_{QR}, the bit length l
Output A: $(a \leq b)$

1: A: $[\![x]\!] \leftarrow [\![b]\!] \cdot [\![2^l]\!] \cdot [\![a]\!]^{-1} \bmod N^2$
2: A chooses a random number $r \leftarrow (0, 2^{\lambda+l}) \cap \mathbb{Z}$
3: A: $[\![z]\!] \leftarrow [\![x]\!] \cdot [\![r]\!] \bmod N^2$
4: A sends $[\![z]\!]$ to B
5: B decrypts $[\![z]\!]$
6: A: $c \leftarrow r \bmod 2^l$
7: B: $d \leftarrow z \bmod 2^l$
8: With A, B privately computes the encrypted bit $[\![t']\!]$ such that $t = (d < c)$ using DGK
9: A encrypts r_l and sends $[\![r_l]\!]$ to B
10: B encrypts z_l
11: B: $[\![t]\!] \leftarrow [\![t']\!] \cdot [\![z_l]\!] \cdot [\![r_l]\!]$
12: B: sends $[\![t]\!]$ to A
13: A decrypts and outputs t

DEMO

Paillier cryptosystem

Key generation works as follows:

1. Pick two large prime numbers p and q, randomly and independently. Confirm that $\gcd(pq, (p-1)(q-1))$ is 1. If not, start again.
2. Compute $n = pq$.
3. Define function $L(x) = \frac{x-1}{n}$.
4. Compute λ as $\operatorname{lcm}(p-1, q-1)$.
5. Pick a random integer g in the set $\mathbb{Z}^*_{n^2}$ (integers between 1 and n^2).
6. Calculate the modular multiplicative inverse $\mu = (L(g^\lambda \mod n^2))^{-1} \mod n$. If μ does not exist, start again from step 1.
7. The public key is (n, g). Use this for encryption.
8. The private key is λ. Use this for decryption.

Decisional composite residuosity assumption

Informally the DCRA states that given a composite *n* and an integer *z*, it is hard to decide whether *z* is a *n*-residue modulo n^2 or not, i.e., whether there exists *y* such that

$$z \equiv y^n \pmod{n^2}.$$

Encryption

Encryption can work for any m in the range $0 \leq m < n$:

1. Pick a random number r in the range $0 < r < n$.
2. Compute ciphertext $c = g^m \cdot r^n \mod n^2$.

Decryption

Decryption presupposes a ciphertext created by the above encryption process, so that c is in the range $0 < c < n^2$:

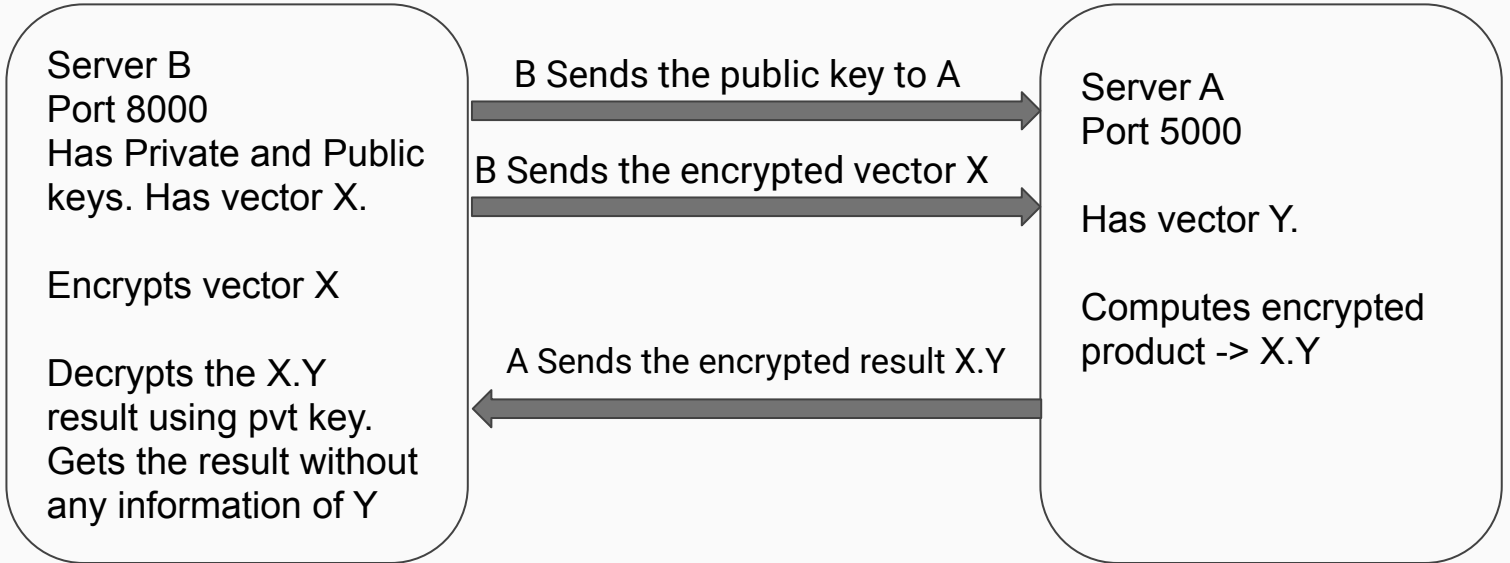
1. Compute the plaintext $m = L(c^\lambda \mod n^2) \cdot \mu \mod n$.

Homomorphic Properties

- addition of two ciphertexts
- multiplication of a ciphertext by a plaintext number

$$D_{priv}(E_{pub}(m_1) \cdot E_{pub}(m_2) \mod n^2) = m_1 + m_2 \mod n$$
$$D_{priv}(E_{pub}(m_1)^{m_2} \mod n^2) = m_1 \cdot m_2 \mod n$$

Dot Product



Argmax Protocol

- Party A has encrypted vector
- B has secret key
- A wants to deliver the argmax to B
- Neither should know the ordering
- A permutes
- B “Refreshes” encryption and reduces complexity.
- A must add “Noise” before sending !

Protocol 1 argmax over encrypted data

Input A: k encrypted integers $([\![a_1]\!], \ldots, [\![a_k]\!])$, the bit length l of the a_i, and public keys PK_{QR} and PK_P
Input B: Secret keys SK_P and SK_{QR}, the bit length l
Output A: $\mathrm{argmax}_i\, a_i$

1: A: chooses a random permutation π over $\{1, \ldots, k\}$
2: A: $[\![\mathsf{max}]\!] \leftarrow [\![a_{\pi(1)}]\!]$
3: B: $m \leftarrow 1$
4: **for** $i = 2$ **to** k **do**
5: Using the comparison protocol (Sec. 4.1.3), B gets the bit $b_i = (\mathsf{max} \leq a_{\pi(i)})$
6: A picks two random integers $r_i, s_i \leftarrow (0, 2^{\lambda+l}) \cap \mathbb{Z}$
7: A: $[\![m'_i]\!] \leftarrow [\![\mathsf{max}]\!] \cdot [\![r_i]\!]$ ▷ $m'_i = \mathsf{max} + r_i$
8: A: $[\![a'_i]\!] \leftarrow [\![a_{\pi(i)}]\!] \cdot [\![s_i]\!]$ ▷ $a'_i = a_{\pi(i)} + s_i$
9: A sends $[\![m'_i]\!]$ and $[\![a'_i]\!]$ to B
10: **if** b_i is true **then**
11: B: $m \leftarrow i$
12: B: $[\![v_i]\!] \leftarrow \mathsf{Refresh}[\![a'_i]\!]$ ▷ $v_i = a'_i$
13: **else**
14: B: $[\![v_i]\!] \leftarrow \mathsf{Refresh}[\![m'_i]\!]$ ▷ $v_i = m'_i$
15: **end if**
16: B sends to A $[\![v_i]\!]$
17: B sends to A $[\![b_i]\!]$
18: A: $[\![\mathsf{max}]\!] \leftarrow [\![v_i]\!] \cdot (g^{-1} \cdot [\![b_i]\!])^{r_i} \cdot [\![b_i]\!]^{-s_i}$
19: ▷ $\mathsf{max} = v_i + (b_i - 1) \cdot r_i - b_i \cdot t_i$
20: **end for**
21: B sends m to A
22: A outputs $\pi^{-1}(m)$

DEMO

Naive Bayes Classifier

1. The first of the multiple classifiers we aim to prepare using the primitives
2. Two party protocol
3. Server has the prior and conditional probabilities.
4. Client has the data , feature values.
5. Special pre-processing required.

1. Server has the private key and the logarithmic probability tables.
2. Server encrypts with paillier scheme.
3. Client has the data , but not the private key.
4. Client performs posterior probability calculations of all the classes .
5. Client-Server engage in the secure argmax protocol and find the best class .

Special Preprocessing For Bayes Classifier

- Paillier encryption scheme to encrypt probabilities
- Paillier works only on integers , probabilities doubles
- Use a large integer ‘K’ to multiply and convert doubles to integers
- Paillier has large message space , overflow avoided
- K determined by exploiting 52-bit precision representation of doubles in the IEEE-754 protocol .

$$p = m \cdot 2^e$$

where m binary representation is $(m)_2 = 1.d$ and d is a 52 bits integer. Hence we have $1 \leq m < 2$ and we can rewrite m as

$$m = \frac{m'}{2^{52}} \text{ with } m' \in \mathbb{N} \cap [2^{52}, 2^{53})$$

We are using this representation to find a constant K such that $K \cdot v_i \in \mathbb{N}$ for all i. As seen before, we can write the v_i's as

$$v_i = m_i' \cdot 2^{e_i - 52}$$

Let $e^* = \min_i e_i$, and $\delta_i = e_i - e^* \geq 0$. Then,

$$v_i = m_i' \cdot 2^{\delta_i} \cdot 2^{e^* - 52}$$

DEMO

Hyperplane Classifier

1. Two party protocol
2. Server has encrypted weights
3. Client has the data , feature values.
4. No special pre-processing required.

1. Server has private key and encrypted weights. We have a d*k weights matrix where d is the dimension and k are the number of classes

2. Server encrypts with paillier scheme.

3. Client sends the 1*d vector. It doesnt have the private key.

4. We first perform the dot product protocol with each class weights and have a list of encrypted values which are sent back to client.

5. We then perform argmax protocol to get the final class. All operations are performed on encrypted numbers.

DEMO

Possible Extensions

1. Modern Deep Learning components such as ReLU activation, matrix multiplication can be implemented using naive for loops and the comparison protocol for encrypted numbers.

2. Further Parallelization of Modules.