Logistic Regression Dev Spec

Author: *dev-goodata*

Summary

In the main net, logistic regression is one part of vertical learning. We need to implement secure logistic regression when features of a dataset are scattered among multiple data owners.

Background

Logistic regression

With logistic regression we learn a linear model $\theta \in R^d$ to a binary label y ∈ {-1, 1}. The learning sample S is a set of n example-label pairs $\{x_i, y_i\}$ from = 1, …, n. The average logistic loss computed on the training set is

$$l_S(\theta) = \frac{1}{n}\sum_{i \in S} \quad log(1 + exp(-y_i \theta^\top x_i)) \qquad (1)$$

In turn, the stochastic gradients computed on a mini-batch $S` \subseteq S$ of size $s`$ are

$$\nabla l_{s'}(\theta) = \frac{1}{s'}\sum_{i \in s'}\left(\frac{1}{1 + exp(-y\theta^T x)} - 1\right) y_i x_i \qquad (2)$$

we need to adapt equations (1) and (2) to accommodate the encrypted mask. Learning requires the computation of gradients only, not of the loss itself.

Split

We assume that the entity resolution protocol has been run, thus A and B have permuted their datasets accordingly, which now have the same number of rows n. The complete dataset is a matrix $X \in R^{n*d}$. This matrix does not exist in one place but is composed of the columns of the datasets A and B held respectively by A and B; this gives the vertical partition:

$$X = [X_A | X_B] \ ,$$

A also holds the label vector y. Let x be a row of X. Define x_A to be the restriction of x to the columns of A, and similarly for θ in place of x and B in place of A. Then we can decompose $\theta^\top x$ as:

$$\theta^\top x = \theta_A^\top x_A + \theta_B^\top x_B,$$

so we can use $\theta_A^\top x_A$,$(\theta_A^\top x_A)^2$, $\theta_B^\top x_B$ and $(\theta_B^\top x_B)^2$ to compute $\theta^\top x$ and $(\theta^\top x)^2$.

Taylor loss

In order to operate under the constraints imposed by an additively homomorphic encryption scheme, we need to consider approximations to the logistic loss and the gradient. To achieve this, we take a Taylor series expansion of $log(1 + exp(-z)$ around $z = 0$:

$$log(1 + exp(-z)) = log2 - \frac{1}{2}z + \frac{1}{8}z^2 - \frac{1}{192}z^4 + O(z^6) \quad (3)$$

The second order approximation of (1) evaluated on H is:

$$l_H(\theta) \approx \frac{1}{h}\sum_{i\in H} \quad log2 - \frac{1}{2}y_i\theta^\top x_i + \frac{1}{8}(\theta^\top x_i)^2 \quad (4)$$

We Write the gradient for a mini-batch S` as:

$$\nabla l_{S'}(\theta) \approx \frac{1}{S'}\sum\nolimits_{i\epsilon s'} \left(\frac{1}{4}y_i\theta^\top x_i - \frac{1}{2}\right) y_i x_i \quad (5)$$

Where we have used the fact that $y_i^2 = 1, \forall i \in R$(y ∈ {-1, 1}). We call this function the Taylor loss.

By differentiation, we write the gradient for a mini-batch S`as:

$$\nabla l_{S'}(\theta) \approx \frac{1}{S'}\sum\nolimits_{i\in s'} \left(\frac{1}{4}\theta^\top x_i - \frac{1}{2}y_i\right) x_i \quad (6)$$

Additively homomorphic encryption

As a public key system, any party can encrypt their data with a known public key and perform computations with data encrypted by others with the same public key. To extract the plaintext, the result needs to be sent to the holder of the private key.

An additively homomorphic encryption scheme provides an operation that produces the encryption of the sum of two numbers, given only the encryptions of the numbers. Let the encryption of a number u be [[u]]. For simplicity we overload the notation and denote the operator with '+' as well. For any plaintexts u and v we have:

$$[[u]] + [[v]] = [[u + v]],$$

Hence we can also multiply a ciphertext and a plaintext together by repeated addition:

$$v * [[u]] = [[vu]],$$

These operations can be extended to work with vectors and matrix component-wise. Summation and matrix operations work similarly. Hense, using an additively homomorphic encryption scheme we can implement useful linear algebra primitives for machine learning.

Applying the encrypted mask

The encrypted mask can be incorporated into (1) by multiplying each term by [[m_i]]. Combined with the Taylor loss, the masked gradient for a mini-batch S` is

$$\left[\left[\nabla l_{S`}(\theta)\right]\right] \approx \frac{1}{S`}\sum_{i\in s`}[[m_i]]\left(\frac{1}{4}\theta^{\mathrm{T}}x_i - \frac{1}{2}y_i\right)x_i \quad (7),$$

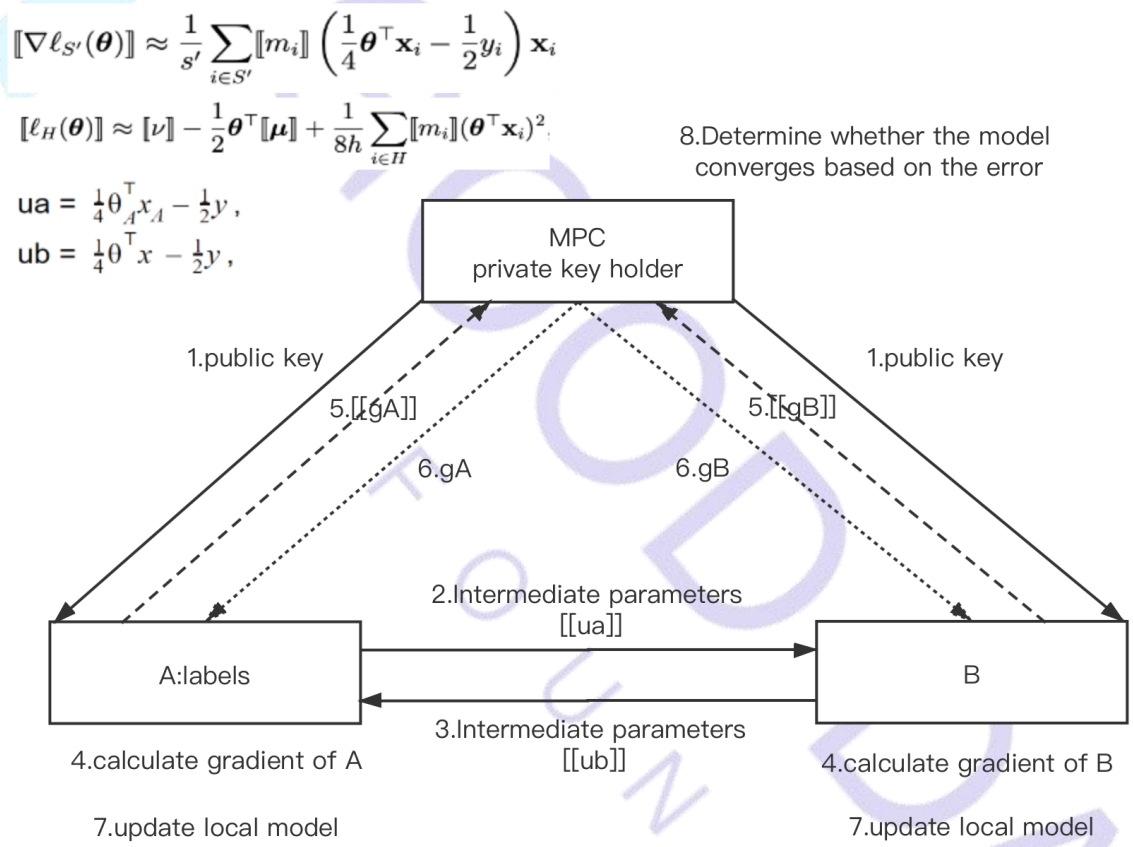
and the masked logistic loss on H is

$$[[l_H(\theta)]] \approx [[v]] - \frac{1}{2}\theta^{\top}[[\mu]] + \frac{1}{8h}\sum_{i\epsilon H} \quad [[m_i]](\theta^{\top}x_i)^2 \qquad (8),$$

where $[[v]] = ((log2)/h)\sum_{i\epsilon H}$ $[[m_i]]$ and $[[\mu]] = (1/h)\sum_{i\in H}$ $[[m_i]]y_i x_i$.

Secure federated logistic regression

Schematic diagram



1. MPC node generates key pairs and sends the public key to DOs.
2. A calculates intermediate parameters and sends the parameters encrypted to B.
3. B calculates the intermediate parameters and sends them to A.
4. A calculates the loss and the gradient of A, B calculates the gradient of B.
5. DOs send their data encrypted to the MPC node.
6. MPC decrypts DOs' gradients and sends them to DOs.
7. DOs update local models.
8. MPC node judges whether loss has converged.

Implementation

Algorithm

Algorithm 1: Secure logistic regression

Note: Algorithm 1 computes the secure logistic regression and it is executed by MPC.
Data: Mask m, learning rate lr, regularisation Γ, hold-out size h, batch size $s`$
Result: Model θ
Process:

Get info from PSI
Create an additively homomorphic encryption key pair
Send the public key to DOs
Encrypt m with the public key, send [[m]] to DOs

$$\theta \leftarrow 0, l_H(\theta) \leftarrow \infty$$

Repeat
for every mini-batch $S`$ do
get gradient from Algorithm 2
Update local model
$l_H(\theta)$ ←Algorithm 3
if $l_H(\theta)$ has not decreased for a while **then break**
Until max iterations
return 0

Algorithm 2: Secure gradient

Note: Algorithm 2 do the secure logistic regression and it is executed by MPC.
Data:Model θ, batch size $s`$
Result: gradient of an mini-batch $S`$
Process:

1. MPC→DO0
   MPC send θ to DO0
2. DO0→DO1
   DO0 select the next batch $S`$

$$u = \frac{1}{4} X_{AS`} \theta_A$$

$$[[u`]] = [[m]]_{S`} \circ (u - \frac{1}{2} y_{S`})$$

   Send θ, $S`$and [[$u`$]]to DO1
3. DO1→DO0

$$u = \frac{1}{4} X_{BS`} \theta_B$$

$$[[w]] = [[u`]] + [[m]]_{S`} \circ v$$

$$[[z]] = X_{BS`}[[w]]$$

   Send [[$z`$]]and[[z]]to MPC

4. MPC
   Obtain [[loss]] by concatenating $[[z`]]$ and $[[z]]$
   Obtain loss decrypting with the private key

Algorithm 3: Secure logistic loss

Note: Algorithm 3 calculates the secure logistic loss. It is called in Algorithm 1, once per iteration of the outer loop.
Data: Model θ, requires $[[\mu_H]]$ and H cached by PSI
Result: $l_H(\theta)$ of the (undisclosed) hold-out
Process:

1. MPC→DO0
   Send θ to DO0
2. DO0→DO1

$$u \leftarrow X_{AH}\theta_A$$
$$[[m_H \circ u]] \leftarrow [[m]]_H \circ u$$
$$[[u`]] \leftarrow \frac{1}{8h}(v \circ v)^\top [[m]]$$

   Send θ, $[[m_H \circ u]]$ and $[[u`]]$ to DO0
3. DO1→DO0

$$v = X_{BH}\theta_B$$
$$[[v`]] \leftarrow \frac{1}{8h}(v \circ v)^\top [[m]]_H$$
$$[[w]] \leftarrow [[u`]] + [[v`]] + \frac{1}{4h}v^\top [[m_H \circ u]]$$
$$[[l_H(\theta)]] \leftarrow [[w]] - \frac{1}{4h}\theta^\top [[\mu_H]]$$

   Send $[[l_H(\theta)]]$ to MPC
4. MPC
   Obtain $l_H(\theta)$ by decryption with the private key

Class

class HE

1. def key_generate() -> [private_key, public_key]
2. def encrypt(plaintext, public_key) -> ciphertext
3. def decrypt(ciphertext, private_key) -> plaintext

class HostDO

1. def receive_data_from_mpc(**args)->data
2. def receive_data_from_guest(**args)->data
3. def send_data_to_guest(**args)->bool
4. def send_data_to_mpc(**args)->bool
5. def update_local_model()->None

6. def calculate_loss(**args)->loss

class GuestDO

1. def send_data_to_host(**args)->bool
2. def send_data_to_mpc(**args)->bool
3. def receive_data_from_host(**args)->data
4. def receive_data_from_mpc(**args)->data
5. def updata_local_model()->None

class MPC

1. def loss_is_convergent()->bool
2. def receive_data_from_host(**args)->data
3. def receive_data_from_guest(**args)->data
4. def send_data_to_host(**args)->bool
5. def send_data_to_guest(**args)->bool

class DatabaseLoader

DOs' data is placed in the database, we need to interact with the database.This class is a base class for interacting with the database.

1. def get_dataset_from_database(**args)->np.array
2. def store_dataset(**args, dataset)->bool

class Dataset

Data preprocessing

class DataLoader

Access data in batches

Reference

1. Private federated learning on vertically partitioned data via entity resolution and additively homomorphic encryption(https://arxiv.org/abs/1711.10677)
2. Public-Key Cryptosystems Based on Composite Degree Residuosity Classes(https://link.springer.com/chapter/10.1007/3-540-48910-X_16)
3. Scalable and Secure Logistic Regression via Homomorphic Encryption(https://eprint.iacr.org/2016/111.pdf)