数据隐私实验一报告

傅申 PB20000051

目录

1. Privacy Preserving Logistics Regression 2
   1.1. 代码实现 2
      1.1.1. Clip 2
      1.1.2. 计算每一次迭代的 ε_u 和 δ_u 2
      1.1.3. 添加噪声 2
      1.1.4. 对框架的其他修改 3
   1.2. 实验结果 3
2. ElGamal Encryption 4
   2.1. 代码实现 4
      2.1.1. `mod_exp(base, exponent, modulus)` 4
      2.1.2. 密钥生成 `elgamal_key_generation(key_size)` 4
      2.1.3. 加密 `elgamal_encrypt(public_key, plaintext)` 5
      2.1.4. 解密 `elgamal_decrypt(private_key, private_key, ciphertext)` 5
   2.2. 大数据量场景下的优化 5
   2.3. 测试 6
      2.3.1. 基础部分 6
         2.3.1.1. 正确性测试 6
         2.3.1.2. 性能测试 6
      2.3.2. 性质验证 7
         2.3.2.1. 随机性 7
         2.3.2.2. 乘法同态性 7

1. Privacy Preserving Logistics Regression

1.1. 代码实现

由于实验框架的某些限制, 这里的代码实现可能与 PPT 中介绍的有所差异.

1.1.1. Clip

Clip 需要按照下式更新梯度

$$\bar{g}_t(x_i) \leftarrow \frac{g_t(x_i)}{\max\left(1, \frac{\|g_t(x_i)\|_2}{C}\right)} \tag{1}$$

考虑输入梯度 g 的维度, 如果 g 为 1 维, 则 $\bar{g}_i = \min(C, g_i)$, 否则按 公式 1 进行裁剪.

```
def clip_gradients(gradients, C):
    # TODO: Clip gradients.
    if gradients.ndim == 1:
        clip_gradients = np.minimum(gradients, C)
    else:
        gradients_norm = np.linalg.norm(gradients, ord=2, axis=1)
        clip_base = np.maximum(gradients_norm / C, 1)
        clip_gradients = gradients / clip_base[:, np.newaxis]
    return clip_gradients
```

1.1.2. 计算每一次迭代的 ε_u 和 δ_u

在论文的第 3.1 节, 提到如果每一次梯度下降都是 $(\varepsilon_u, \delta_u)$-DP 的话, 整个 DP-SGD 过程就是 $\left(\mathcal{O}\left(q\varepsilon_u\sqrt{T}\right), \delta_u\right)$-DP 的. 在本实验中, $q = 1$. 因此, 根据给定的 ε, δ, 可以计算 ε_u, δ_u 如下:

```
# Calculate epsilon_u, delta_u based epsilon, delta and epochs here.
epsilon_u, delta_u = (epsilon / np.sqrt(self.num_iterations), delta)
```

1.1.3. 添加噪声

因为本实验是对训练过程中的 `dz` 进行添加噪声, 然后再计算梯度, 所以在输入为 1 维时, 只需要添加高斯噪声即可, 不需要求加噪后均值. 在输入为更高维梯度时, 需要按照下式计算:

$$\tilde{g} = \frac{1}{L}\left(\sum_i \bar{g}_i + \mathcal{N}\left(0, \sigma^2 C^2 \boldsymbol{I}\right)\right) \tag{2}$$

其中, σ 与单次迭代的 ε_u 和 δ_u 有关

$$\sigma = \frac{\sqrt{2\log\left(\frac{1.25}{\delta_u}\right)}}{\varepsilon_u} \tag{3}$$

代码实现如下:

```
def add_gaussian_noise_to_gradients(gradients, epsilon, delta, C):
    # TODO: add gaussian noise to gradients.
    num_samples = gradients.shape[0]
    sigma = C * np.sqrt(2 * np.log(1.25 / delta)) / epsilon
    if gradients.ndim == 1:
        noisy_gradients = gradients + np.random.normal(0, sigma, gradients.shape)
```

```python
    else:
        sum_gradients = np.sum(gradients, axis=0)
        noise = np.random.normal(0, sigma, sum_gradients.shape)
        noisy_gradients = (sum_gradients + noise) / num_samples
    return noisy_gradients
```

1.1.4. 对框架的其他修改

在梯度下降的过程中, `dz` 的计算有误:

$$\ell(x, y) = -(y \log(\sigma(z)) + (1-y)\log(1-\sigma(z))) \Rightarrow \frac{\partial \ell}{\partial z} = -\left(\frac{y}{\sigma(z)} - \frac{1-y}{1-\sigma(z)}\right) \cdot \sigma(z) \cdot (1-\sigma(z)) \quad (4)$$

因此相应的代码修改为:

```python
# Compute predictions of the model
linear_model = np.dot(X, self.weights) + self.bias
predictions = self.sigmoid(linear_model)

# Compute loss and gradients
loss = -np.mean(
    y * np.log(predictions + self.tau)
    + (1 - y) * np.log(1 - predictions + self.tau)
)
d_prediction = -(
    y / (predictions + self.tau) - (1 - y) / (1 - predictions + self.tau)
)
dz = d_prediction * (predictions * (1 - predictions))
```

同时, 未对数据集进行归一化, 导致模型性能较差, 因此在 `get_train_data()` 中加入了如下代码

```python
# Normalize the data
X = (X - X.mean(axis=0)) / X.std(axis=0)
```

1.2. 实验结果

下面是朴素的梯度下降法和不同 (ε, δ)-DP-SGD 的 Loss 曲线:

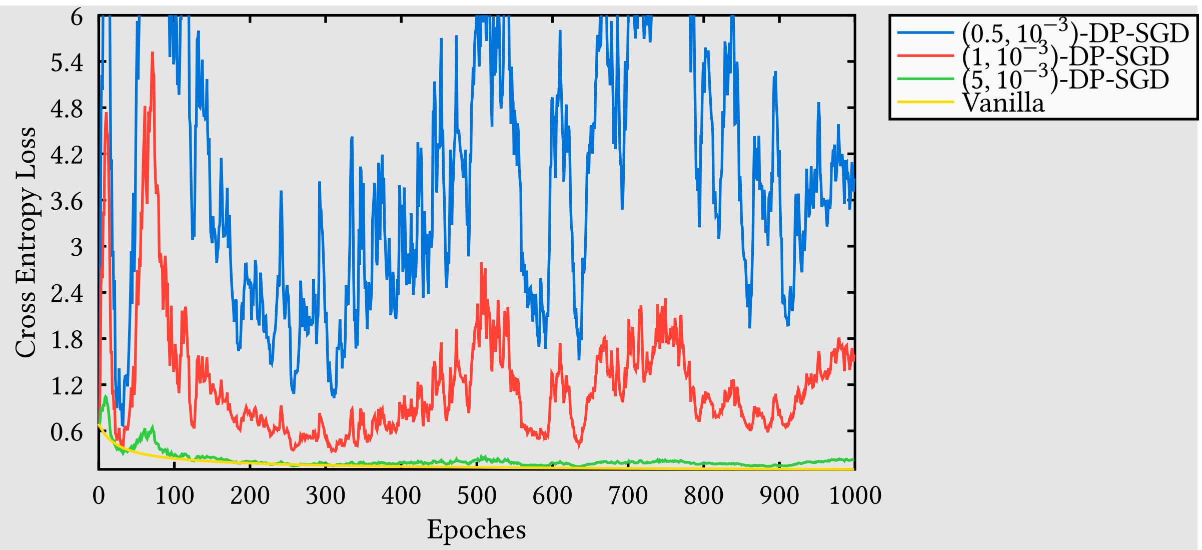

图 1 梯度下降过程中的 Loss 曲线

在最终的测试集上, 朴素的梯度下降法的准确率为 96.49%, $(0.5, 10^{-3})$-DP-SGD 的准确率为 62.68%, $(1, 10^{-3})$-DP-SGD 的准确率为 73.68%, $(5, 10^{-3})$-DP-SGD 的准确率为 88.60%. 从上述结果可以看出, DP-SGD 在一定程度上影响了模型的部分性能, 且该影响随着隐私预算的提高而降低.

固定 (ε, δ), 考察不同的迭代次数对于模型的影响:

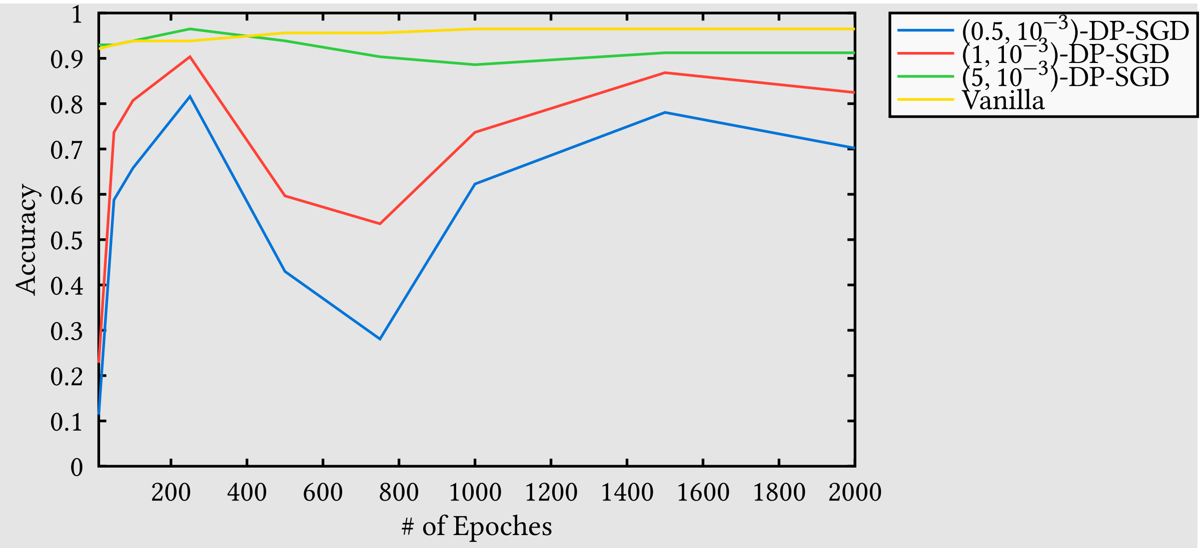

图 2 迭代次数与准确率的关系

可以看到, 在隐私预算较大时, 模型的性能受到迭代次数的影响较小. 而在隐私预算较小时, 模型的性能随着迭代次数的增加, 先降低, 后提高到一个稳定值, 且该值要小于模型的最佳性能.

2. ElGamal Encryption

2.1. 代码实现

2.1.1. `mod_exp(base, exponent, modulus)`

函数需要实现带模幂运算, 即 $b^e \bmod m$, 可以使用快速幂进行实现:

```python
def mod_exp(base, exponent, modulus):
    if exponent == 0:
        return 1
    if exponent == 1:
        return base % modulus
    if exponent % 2 == 0:
        return (mod_exp(base, exponent // 2, modulus) ** 2) % modulus
    if exponent % 2 == 1:
        return (mod_exp(base, exponent // 2, modulus) ** 2 * base) % modulus
```

但是, 由于 Python 的内建函数 `pow()` 可以实现完全一样的功能, 出于性能考虑, 代码实现中直接使用了 `pow()`, 即

```python
def mod_exp(base, exponent, modulus):
    return pow(base, exponent, modulus)
```

2.1.2. 密钥生成 `elgamal_key_generation(key_size)`

按照算法的描述, 密钥生成的实现如下:

```python
def elgamal_key_generation(key_size):
    """Generate the keys based on the key_size."""
    # generate a large prime number p and a primitive root g
    p, g = generate_p_and_g(key_size)

    # generate x and y here.
    x = random.randint(1, p - 2)
    y = mod_exp(g, x, p)

    return (p, g, y), x
```

2.1.3. 加密 `elgamal_encrypt(public_key, plaintext)`

按照算法的描述, 函数首先检查明文 m 是否满足 $0 < m < p - 1$, 然后随机选择临时私钥 k, 最后计算临时公钥和临时密文:

```python
def elgamal_encrypt(public_key, plaintext):
    """Encrypt the plaintext with the public key."""
    # unpack public key
    p, g, y = public_key
    # check if plaintext is smaller than p
    if plaintext >= p:
        raise ValueError("plaintext should be smaller than p")
    # choose a temporary secret key k
    k = random.randint(1, p - 2)
    c1 = mod_exp(g, k, p)  # temporary public key
    c2 = plaintext * mod_exp(y, k, p) % p  # temporary ciphertext
    return c1, c2
```

2.1.4. 解密 `elgamal_decrypt(private_key, private_key, ciphertext)`

按照算法描述, 函数首先计算 c_1 的模反演 s, 然后解密出明文:

```python
def elgamal_decrypt(public_key, private_key, ciphertext):
    """Decrypt the ciphertext with the public key and the private key."""
    # unpack public key and ciphertext
    p, _, _ = public_key
    c1, c2 = ciphertext

    s = mod_exp(c1, private_key, p)
    s_inv = sympy.mod_inverse(s, p)  # modular inverse of s
    plaintext = c2 * s_inv % p
    return plaintext
```

2.2. 大数据量场景下的优化

在大数据的场景下, 我采用 Python 并行计算的方式, 来优化 ElGamal 算法加解密的时间开销. 如下代码所示:

```python
from multiprocessing.pool import ThreadPool

def elgamal_encrypt_batch(public_key, plaintexts):
    """Encrypt a batch of plaintexts."""
    # multiprocessing
    with ThreadPool() as pool:
        return pool.starmap(
            elgamal_encrypt,
```

```
            zip(repeat(public_key), plaintexts)
        )


def elgamal_decrypt_batch(public_key, private_key, ciphertexts):
    """Decrypt a batch of ciphertexts."""
    # multiprocessing
    with ThreadPool() as pool:
        return pool.starmap(
            elgamal_decrypt,
            zip(repeat(public_key), repeat(private_key), ciphertexts)
        )
```

在此方案中, 我使用了 `multiprocessing.pool` 模块中的线程池 `ThreadPool`, 将批量加解密任务分配给多个线程, 从而实现并行计算. 在加解密实现正确的前提下, 该方案显然是正确的.

2.3. 测试

为了实现要求的功能, 我对程序的运行方式作了一些修改. 程序接受一些命令行参数, 其中第一个参数为程序需要执行的功能, 可以为 `interact`, `profile` 和 `verify`. 三个功能的作用分别为:

- `interact`: 代码框架中提供的交互式加解密流程. (`python elgamal.py interact`)
- `profile`: 对加解密函数进行性能测试, 并输出测试结果. 有三种测试模式:
  - `simple`: 测试不同 key size 下三个阶段的时间开销.
  - `batch`: 测试不同 key size 下使用并行计算对多个明文/密文加解密的时间开销.
  - `homo`: 根据乘法同态性, 测试 `decrypt(a) * decrypt(b)` 和 `decrypt(a * b)` 的时间开销.

  三种模式都接收命令行参数 `--key-sizes` (`-k`) 和 `--repeat`(`-r`), `batch` 模式下还接收命令行参数 `--batch-size` (`-b`).
- `verify`: 对 ElGamal 加密的性质进行验证, 采用交互式的方式. 由两种验证模式:
  - `random`: 验证 ElGamal 加密的随机性. (`python elgamal.py verify random`)
  - `homo`: 验证 ElGamal 加密的乘法同态性. (`python elgamal.py verify homo`)

2.3.1. 基础部分

2.3.1.1. 正确性测试

运行 `python elgamal.py interact` 进行测试, 结果如下:

```
$ python elgamal.py interact
Please input the key size: 32
Public Key: (3256273589, 2, 1048833368)
Private Key: 615486090
Please input an integer m (0 < m < 3256273588): 923746237
Ciphertext: (1454222740, 217534487)
Decrypted Text: 923746237
```

可以看到, 程序对加密后的密文进行了解密, 得到了正确的明文.

2.3.1.2. 性能测试

运行 `python elgamal.py profile simple`, 对 key size 分别为 8, 16, 32, 64, 128 的情况进行测试, 每个测试重复 100 次, 结果如下表所示

Key Size	Key Generation	Encryption	Decryption
8	12.77 μs	1.22 μs	2.93 μs
16	23.63 μs	2.00 μs	3.62 μs
32	211.43 μs	9.57 μs	11.69 μs
64	3515.84 μs	25.65 μs	33.79 μs
128	185301.92 μs	65.04 μs	69.47 μs
256*	1.35 s	233.88 μs	233.78 μs

表 1 ElGamal 加解密性能测试

在 key size = 256 的情况下, ElGamal 加密的 key generation 有一定概率需要很长时间, 因此测试中只重复了 10 次. (使用命令 `python elgamal.py profile simple -k 256 -r 10`)

从测试结果可以看出, 在三个阶段中, key generation 的时间开销最大, 其增长速度也最快, 这可能是由于目前暂时没有有效的求原根的算法. 相比之下, 加解密阶段的时间开销较小, 且增长较为平缓.

2.3.2. 性质验证

2.3.2.1. 随机性

运行 `python elgamal.py verify random`, 结果如下:

```
$ python elgamal.py verify random
Please input the key size: 32
Public Key: (3619562537, 3, 284917819)
Private Key: 1543472710
Please input an integer m (0 < m < 3619562536): 1283724387
Ciphertext 1: (2938195427, 2737285665)
Ciphertext 2: (563872373, 3567224372)
Decrypted Text 1: 1283724387
Decrypted Text 2: 1283724387
```

可以看到, 对于相同的明文, ElGamal 加密算法会生成不同的密文, 并解密出相同的明文. 这验证了 ElGamal 加密算法的随机性.

2.3.2.2. 乘法同态性

运行 `python elgamal.py verify homo`, 结果如下:

```
$ python elgamal.py verify homo
Please input the key size: 32
Public Key: (2431040257, 5, 516263206)
Private Key: 810561174
Please input an integer m1 (0 < m1 < 2431040256): 1232412378
Please input an integer m2 (0 < m2 < 2431040256): 1297312612
m1 * m2 % p: 1011083189
Ciphertext 1: (2101000828, 1077557602)
Ciphertext 2: (350039967, 1431015853)
Ciphertext 1 * Ciphertext 2: (2341359810, 2328606162)
Decrypted Text 1: 1232412378
Decrypted Text 2: 1297312612
Decrypted Text of Multiplied Ciphertext: 1011083189
```

可以看到, 对于两个明文 m_1 和 m_2, 分别记它们的密文为 C_1 和 C_2, 则 $C_1 \cdot C_2$ 解密后的结果与 $m_1 \cdot m_2 \bmod p$ 的结果相同, 验证了 ElGamal 加密算法的乘法同态性.

同时, 运行 `python elgamal.py profile homo`, 测试 `decrypt(a) * decrypt(b)` 和 `decrypt(a * b)` 的时间开销, 结果如下表所示:

Key Size	**`time(dec(a) * dec(b))`**	**`time(dec(a * b))`**
8	3.92 μs	1.92 μs
16	6.95 μs	3.13 μs
32	21.68 μs	9.92 μs
64	59.79 μs	27.50 μs
128	129.76 μs	62.28 μs

表 2 `decrypt(a) * decrypt(b)` 和 `decrypt(a * b)` 的时间开销

可以看出, `decrypt(a) * decrypt(b)` 的时间开销要大于 `decrypt(a * b)`.