Data-Privacy Lab1 Report

Name	ID
王昱	PB21030814

一些说明

本实验所有数据的测试均在linux服务器上完成

文件结构如下：

Privacy Preserving Logistics Regression(Version1)

> Attention：这部分是在看过助教群公告前写的，当时采用的是Composition Theorems(Sequential)，即每步的ϵ、δ用总的除以迭代轮数。

```
epsilon_u, delta_u = epsilon / self.num_iterations, delta / self.num_iterations
```

验证不同差分隐私预算下对模型效果的影响

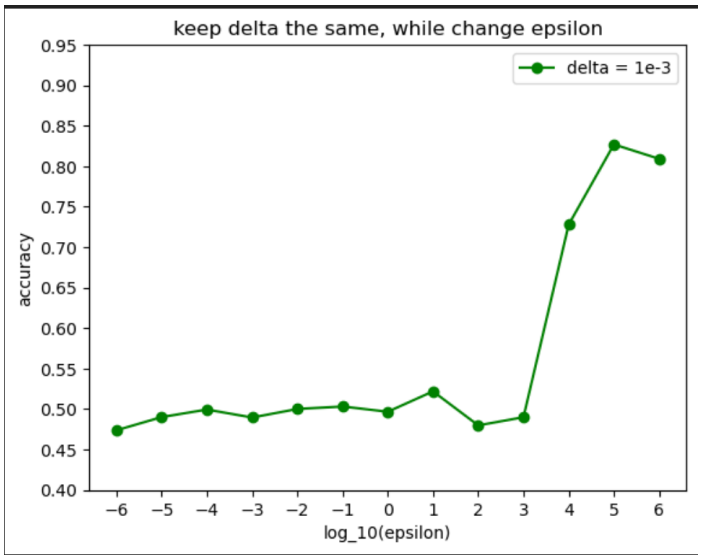
首先我固定了δ的值不变，更改ϵ的值观察模型效果的变化。

为了得到充足数量的测试数据，我写了一个 shell 脚本来运行 dp-sgd.py 文件。脚本如下：

```
E=0.000001
D=0.001
I=1000

while (( $(bc <<< "$E <= 1000000") ))
do
   i=0
   while (( i < 100 ))
   do
      python3 dp-sgd.py --epsilon $E --delta $D --iter $I
      (( i = i + 1 ))
   done
   E=$(bc <<< "$E * 10")
done
```

保持$\delta = 1e-3$，ϵ的取值为$1e-6, 1e-5, 1e-4, 1e-3, 1e-2, 1e-1, 1, 1e1, 1e2, 1e3, 1e4, 1e5, 1e6$，迭代次数设为1000(这里的$\delta$和$\epsilon$均为原始的，而不是$\epsilon_u$和$\delta_u$)。算出的隐私预算$\epsilon_u$为$1e-9, 1e-8, 1e-7, 1e-6, 1e-5, 1e-4, 1e-3, 1e-2, 1e-1, 1, 1e1, 1e2, 1e3$。每组数据测试100次，然后取均值。作为对比，不加噪的模型预测accuracy=0.8859649122807017

测试结果可视化如下：



- 隐私预算越小，加的噪声越大，隐私保护得越好，体现在上图中的accuracy在0.5附近波动。这说明了模型不能很好地作分类任务，结果跟 **“瞎猜”** 的结果差不多。
- 隐私预算越大，加的噪声越小，隐私保护得越差，从而模型受噪声的影响就越小，可以看到上图在$\epsilon = 1e5, 1e6$的时候模型的$accuracy$与不加噪的模型$accuracy$相近。

然后我固定了ϵ的值不变，更改δ的值观察模型效果的变化。

Attention：这部分看群公告助教没要求，就不详细分析了。

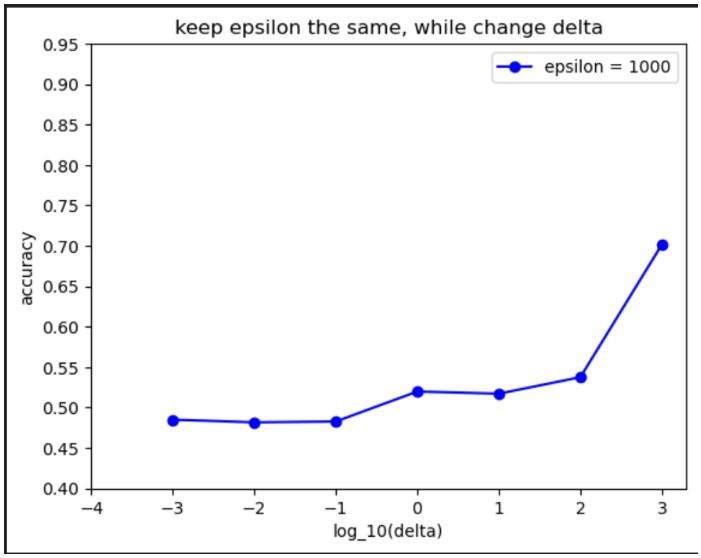
为了得到充足数量的测试数据，我写了一个 shell 脚本来运行 dp-sgd.py 文件。脚本如下：

```
E=1000
D=0.001
I=1000

while (( $(bc <<< "$D <= 1000") ))
do
   i=0
   while (( i < 100 ))
```

```
    do
        python3 dp-sgd.py --epsilon $E --delta $D --iter $I
        (( i = i + 1 ))
    done
    D=$(bc <<< "$D * 10")
done
```

保持$\epsilon = 1000$，δ的取值为$1e-3, 1e-2, 1e-1, 1, 1e1, 1e2, 1e3$，迭代次数设为1000。每组数据测试100次，然后取均值。

测试结果可视化如下：



相同总隐私消耗量下，不同迭代轮数对模型效果的影响

为了得到充足数量的测试数据，我写了一个 shell 脚本来运行 dp-sgd.py 文件。脚本如下：

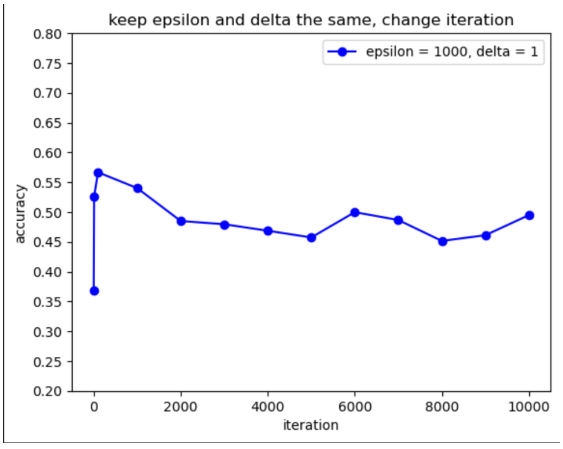
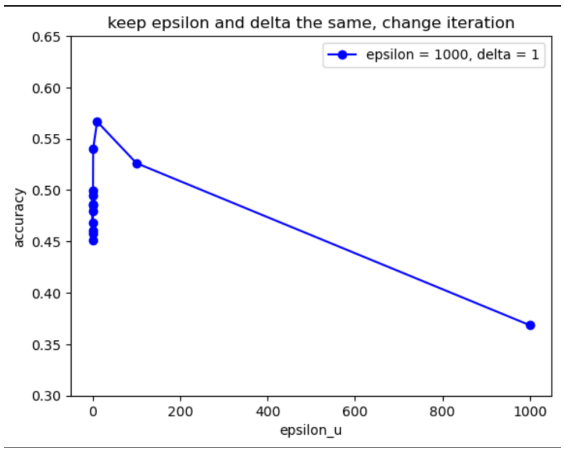
```
E=1000
D=1
I=1

while (( $(bc <<< "$I <= 1000") ))
do
    i=0
    while (( i < 100 ))
    do
        python3 dp-sgd.py --epsilon $E --delta $D --iter $I
        (( i = i + 1 ))
    done
    I=$(bc <<< "$I * 10")
done

I=2000

while (( $(bc <<< "$I <= 10000") ))
do
    i=0
    while (( i < 100 ))
    do
        python3 dp-sgd.py --epsilon $E --delta $D --iter $I
        (( i = i + 1 ))
    done
    I=$(bc <<< "$I + 1000")
done
```

保持$\delta = 1$，$\epsilon = 1000$，迭代次数的取值为$1, 10, 100, 1000, 2000, 3000, 4000, 5000, 6000, 7000, 8000, 9000, 10000$。每组数据测试100次，然后取均值。

测试原始模型的accuracy如下：

iteration	accuracy
1	0.3684210526315789
10	0.3684210526315789
100	0.37719298245614036
1000	0.8859649122807017
2000	0.9385964912280702
3000	0.9298245614035088
4000	0.9473684210526315
5000	0.9385964912280702
6000	0.6929824561403509
7000	0.8947368421052632
8000	0.8947368421052632
9000	0.8947368421052632
10000	0.8947368421052632

测试结果可视化如下：

横坐标是$iteration$	横坐标是ϵ_u
	

> 从上图中可以看出：
>
> - 当ϵ_u非常小(迭代轮数比较大)时，$accuracy$在0.5附近，这说明模型不能很好地作分类任务，结果跟 “瞎猜” 的结果差不多。这是因为当ϵ_u非常小时，隐私预算很小，加的噪声很大，隐私保护得很好。
> - 当ϵ_u非常大(迭代轮数比较小)时，这时隐私预算很大，加的噪声很小，隐私保护得很差。但是当iteration很小的时候，原始模型的accuracy也很小，如上表中iteration=1时accuracy=0.3684210526315789。因此这时加噪后的模型accuracy也很差。
> - 当ϵ_u适中(迭代轮数适中)时，accuracy会取到较大的值。

Privacy Preserving Logistics Regression(Version2)

> Attention：这部分是在看过助教群公告后写的，采用的是Advanced Composition。这里让ppt公式中的$\delta' = \delta$ 从而得到的公式为$\delta_u = \frac{\delta_总}{k+1}, \epsilon_u = \frac{\epsilon_总}{2\sqrt{2k\ln(\frac{1}{\delta_u})}}$

```
delta_u = delta / (self.num_iterations + 1)
epsilon_u = epsilon / (2 * math.sqrt(2 * self.num_iterations * math.log(1 / delta_u)))
```

验证不同差分隐私预算下对模型效果的影响

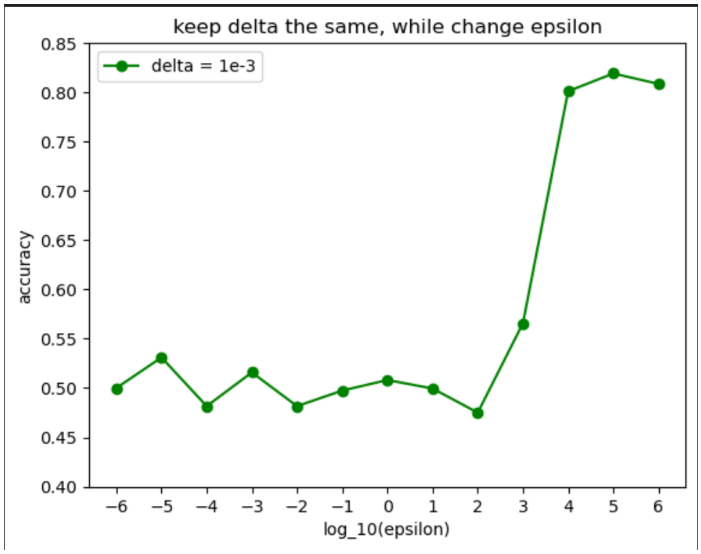
首先我固定了δ的值不变，更改ϵ的值观察模型效果的变化。

为了得到充足数量的测试数据，我写了一个 `shell` 脚本来运行 `dp-sgd.py` 文件。脚本如下

```
E=0.000001
D=0.001
I=1000

while (( $(bc <<< "$E <= 1000000") ))
do
    i=0
    while (( i < 100 ))
    do
        python3 dp-sgd.py --epsilon $E --delta $D --iter $I
        (( i = i + 1 ))
    done
    E=$(bc <<< "$E * 10")
done
```

保持$\delta = 1e-3$，ϵ的取值为$1e-6, 1e-5, 1e-4, 1e-3, 1e-2, 1e-1, 1, 1e1, 1e2, 1e3, 1e4, 1e5, 1e6$，迭代次数设为1000(这里的$\delta$和$\epsilon$均为原始的，而不是$\epsilon_u$和$\delta_u$)。算出的隐私预算$\epsilon_u$(保留三位有效数字)为$3.01\times10^{-9}, 3.01\times10^{-8}, 3.01\times10^{-7}, 3.01\times10^{-6}, 3.01\times10^{-5}, 3.01\times10^{-4}, 3.01\times10^{-3}, 3.01\times10^{-2}, 3.01\times10^{-1}, 3.01, 30.1, 301, 3010$。每组数据测试100次，然后取均值。作为对比，不加噪的模型预测accuracy=0.8859649122807017

测试结果可视化如下：



> 从上图可以看出：
>
> - 隐私预算越小，加的噪声越大，隐私保护得越好，体现在上图中的accuracy在0.5附近波动。这说明了模型不能很好地作分类任务，结果跟 **“瞎猜”** 的结果差不多。
> - 隐私预算越大，加的噪声越小，隐私保护得越差，从而模型受噪声的影响就越小，可以看到上图在$\epsilon = 1e4, 1e5, 1e6$的时候模型的$accuracy$与不加噪的模型$accuracy$相近。

然后我固定了ϵ的值不变，更改δ的值观察模型效果的变化。

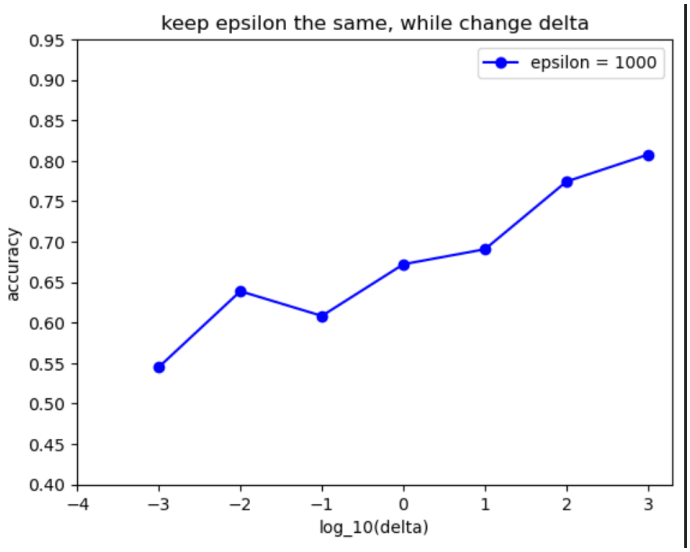
Attention：这部分看群公告助教没要求，就不详细分析了。

为了得到充足数量的测试数据，我写了一个 shell 脚本来运行 dp-sgd.py 文件。脚本如下：

```
E=1000
D=0.001
I=1000

while (( $(bc <<< "$D <= 1000") ))
do
    i=0
    while (( i < 100 ))
    do
        python3 dp-sgd.py --epsilon $E --delta $D --iter $I
        (( i = i + 1 ))
    done
    D=$(bc <<< "$D * 10")
done
```

保持$\epsilon = 1000$，δ的取值为$1e-3, 1e-2, 1e-1, 1, 1e1, 1e2, 1e3$，迭代次数设为1000。每组数据测试100次，然后取均值。

测试结果可视化如下：

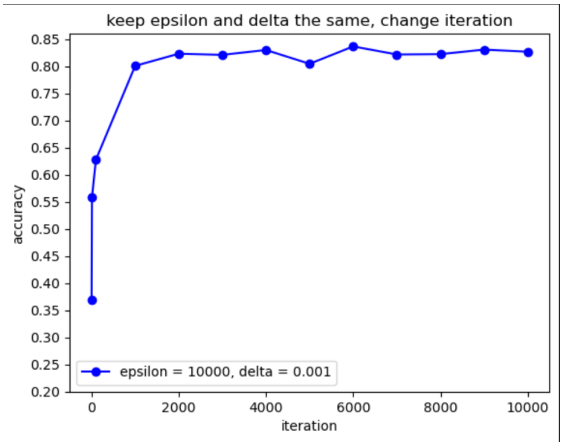
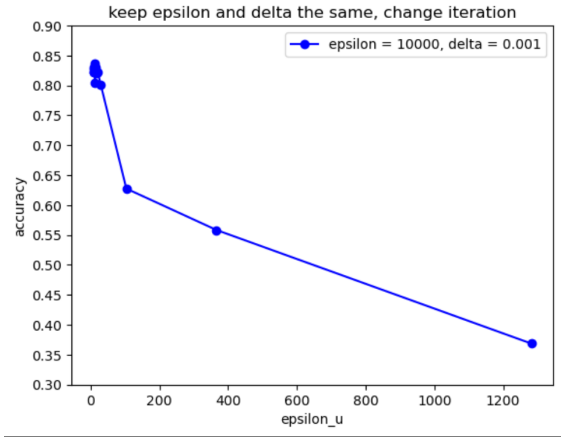


相同总隐私消耗量下，不同迭代轮数对模型效果的影响

测试原始模型的accuracy如下：

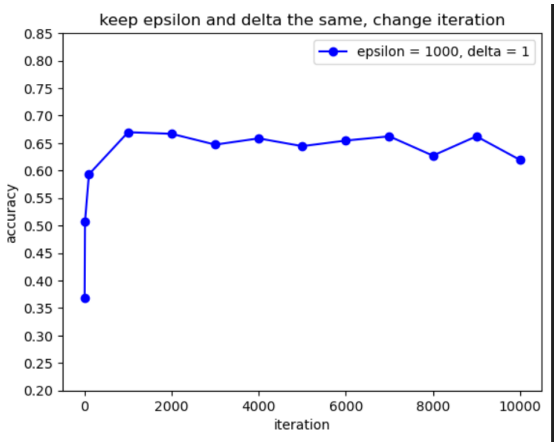
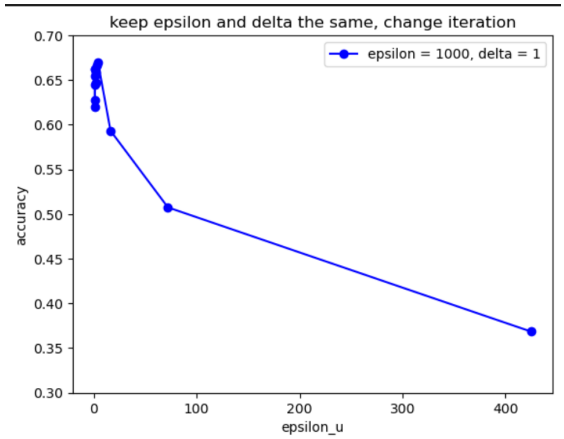
iteration	accuracy
1	0.3684210526315789
10	0.3684210526315789
100	0.37719298245614036
1000	0.8859649122807017
2000	0.9385964912280702
3000	0.9298245614035088
4000	0.9473684210526315
5000	0.9385964912280702
6000	0.6929824561403509
7000	0.8947368421052632
8000	0.8947368421052632
9000	0.8947368421052632
10000	0.8947368421052632

将测试结果可视化如下：

横坐标是$iteration$	横坐标是ϵ_u
	

保持$\delta = 1$，$\epsilon = 1000$，迭代次数的取值为$1, 10, 100, 1000, 2000, 3000, 4000, 5000, 6000, 7000, 8000, 9000, 10000$。每组数据测试100次，然后取均值。

将测试结果可视化如下：

横坐标是$iteration$	横坐标是ϵ_u
	

从上面两个测试结果可以看出：

- 当ϵ_u较小(迭代轮数比较大)的时候，随着ϵ_u的增加，隐私预算增大，隐私保护得不好，模型受噪声得影响较小，从而accuracy增加/在较大值附近波动。
- 当ϵ_u太大(迭代轮数很小)时，噪声几乎没有影响。但是此时由于迭代次数很小，原始模型的accuracy很小，因此上面的右图中最后呈现的是递减趋势。

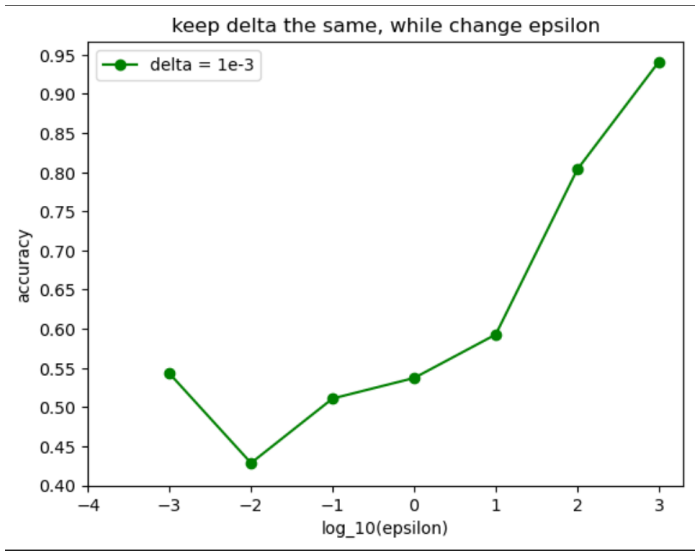
Privacy Preserving Logistics Regression(Version3)

Attention：这部分是看群友分享的代码框架之后写的

验证不同差分隐私预算下对模型效果的影响

保持$\delta = 1e-3$，ϵ的取值为$1e-3, 1e-2, 1e-1, 1, 1e1, 1e2, 1e3$，迭代次数设为1000(这里的$\delta$和$\epsilon$均为原始的，而不是$\epsilon_u$和$\delta_u$)。算出的隐私预算$\epsilon_u$(保留三位有效数字)为$3.01 \times 10^{-6}, 3.01 \times 10^{-5}, 3.01 \times 10^{-4}, 3.01 \times 10^{-3}, 3.01 \times 10^{-2}, 3.01 \times 10^{-1}, 3.01$每组数据测试100次，然后取均值。作为对比，不加噪的模型预测accuracy=0.9649122807017544

测试结果可视化如下：



从上图可以看出：

- 隐私预算越小，加的噪声越大，隐私保护得越好，体现在上图中的accuracy在0.5附近波动。这说明了模型不能很好地作分类任务，结果跟 **“瞎猜”** 的结果差不多。

- 隐私预算越大，加的噪声越小，隐私保护得越差，从而模型受噪声的影响就越小，可以看到上图在$\epsilon = 1e2, 1e3$的时候模型的$accuracy$与不加噪的模型$accuracy$相近。

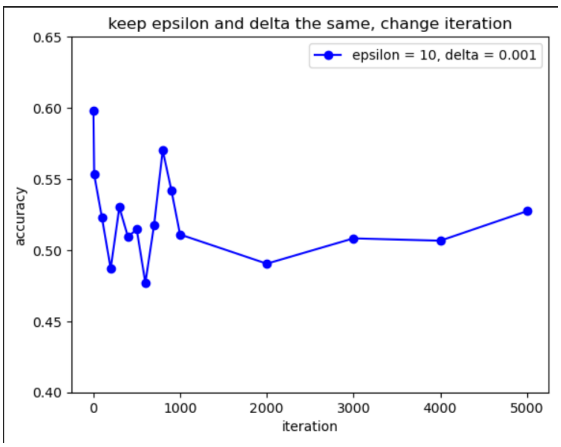
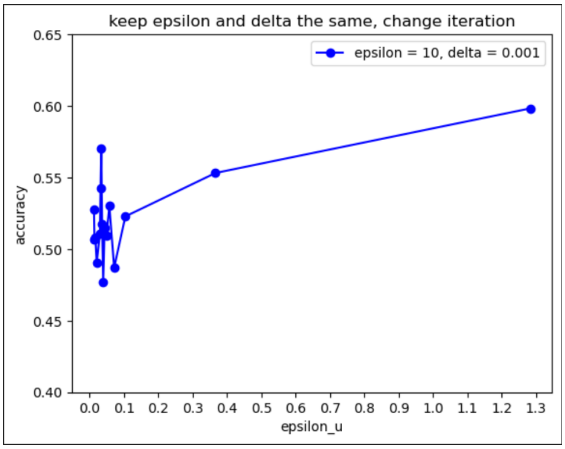
相同总隐私消耗量下，不同迭代轮数对模型效果的影响

测试原始模型的accuracy如下：

iteration	accuracy
1	0.9210526315789473
10	0.9210526315789473
100	0.9385964912280702
200	0.9385964912280702
300	0.9385964912280702
400	0.956140350877193
500	0.956140350877193
600	0.956140350877193
700	0.956140350877193
800	0.956140350877193
900	0.9649122807017544
1000	0.9649122807017544
2000	0.9649122807017544
3000	0.9649122807017544
4000	0.9736842105263158
5000	0.9736842105263158

保持$\delta = 0.001$，$\epsilon = 10$，迭代次数的取值为$1, 10, 100, 200, 300, 400, 500, 600, 700, 800, 900, 1000, 2000, 3000, 4000, 5000$。每组数据测试100次，然后取均值。

将测试结果可视化如下：

横坐标是$iteration$	横坐标是ϵ_u
	

- 当ϵ_u较小(迭代轮数较大)时，隐私预算小，对应的噪声大，体现在上图中的accuracy在0.5附近波动。这说明了模型不能很好地作分类任务，结果跟 **“瞎猜”** 的结果差不多。
- 随着ϵ_u的增大(迭代轮数减小)，噪声减小，隐私保护得差，模型受噪声影响小。由于原始模型在迭代次数较小时accuracy就很大，所以加噪后的模型accuracy也大(体现为右图的上升趋势)

ElGamal加密算法

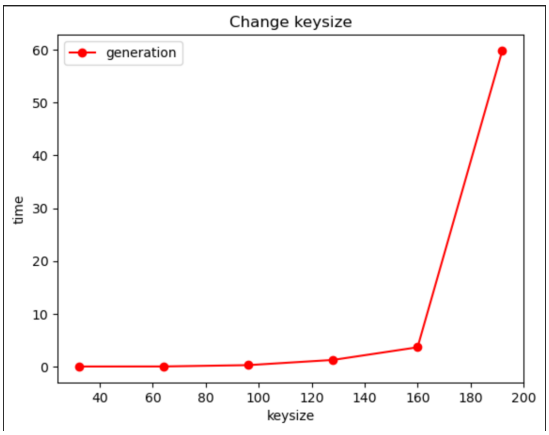
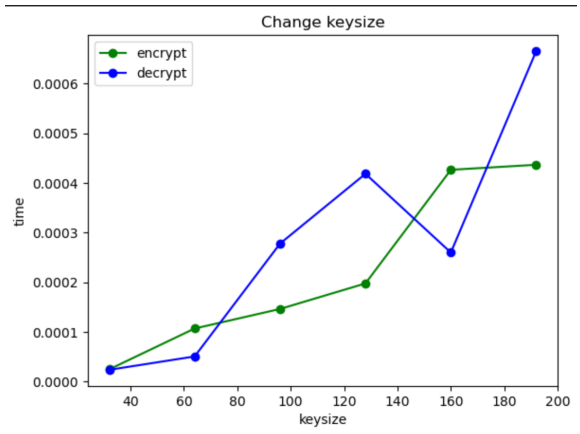
测试不同keysize下三个阶段时间开销

为了得到充足数量的测试数据，我写了一个 `shell` 脚本来运行 `elgamal.py` 文件。脚本如下：

```
keysize=32
plaintext=100
while (( $keysize <= 192 ))
do
    i=0
    while (( i < 100 ))
    do
        python3 elgamal.py --keysize $keysize --plaintext $plaintext
        (( i = i + 1 ))
    done
    (( keysize = keysize + 32 ))
done
```

将keysize和plaintext作为参数传入 `elgamal.py` 文件中，这里保持plaintext不变，改变keysize观察它的变化对三个阶段时间开销的影响。keysize设置的值为 `32、64、96、128、160、192` ，每个keysize重复100次，取时间的平均值。

测试结果可视化如下：

Left	Right
	

三个阶段： `generation(密钥生成)` 、 `encrypt(加密)` 、 `decrypt(解密)`

- 从左图可以看出 `generation(密钥生成)` 时间开销随着keysize的增大而增大，在keysize=128之前增加的并不显著，在keysize=128之后增加较为明显。
- 从右图可以看出 `encrypt(加密)` 、 `decrypt(解密)` 时间开销随着keysize的增大几乎不变(相较于 `generation(密钥生成)` 的时间开销可以忽略不计)。

说明随着keysize的增大， `generation(密钥生成)` 的时间开销受较大影响；而 `encrypt(加密)` 、 `decrypt(解密)` 的时间开销几乎不受影响。

验证ElGamal算法的随机性

ElGamal 算法的随机性是指相同的明文在多次加密时产生不同的密文。基于此，我进行了三次连续的测试，结果如下：

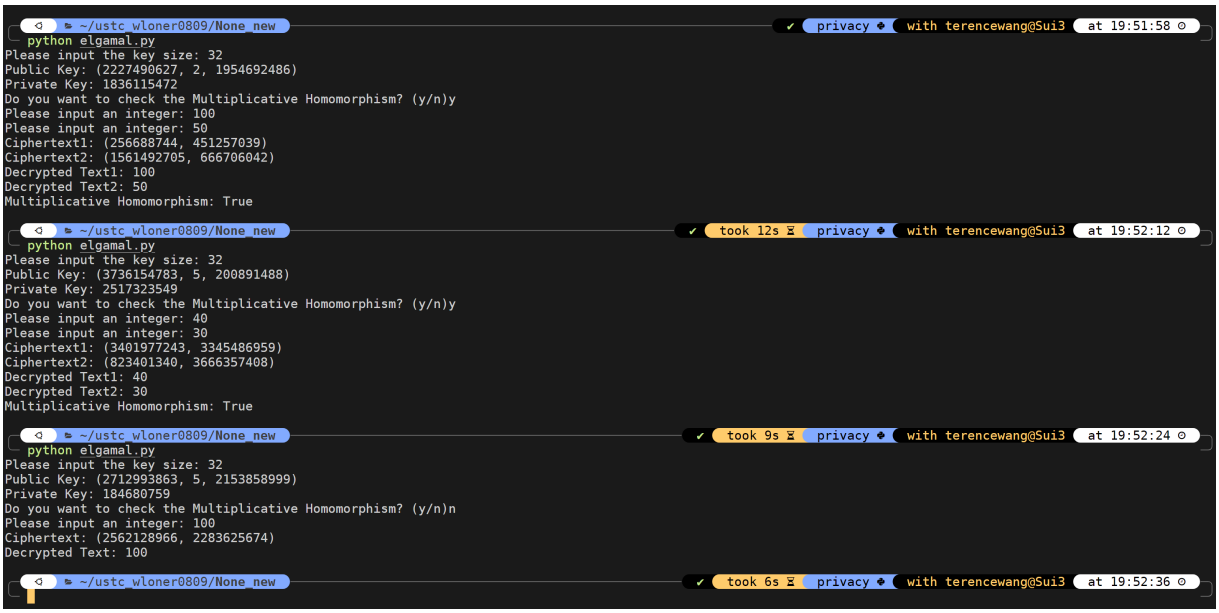
> 从上图可以看出：三次测试的keysize均为32，明文均为100。但是第一次密文为(603170769, 1061079230)，第二次密文为(1115038272, 50804480)，第三次密文为(2222498018, 868286256)。可以验证 ElGamal 算法的随机性。

验证ElGamal算法的乘法同态性质

ElGamal 算法的乘法同态性是指两个密文的乘积解密后等于对应明文的乘积。基于此，我写了一个函数用来验证乘法同态性，如下：

```python
def check_MultiplicativeHomomorphism(c1, c2, public_key, private_key, m1, m2):
    # 验证乘法同态性
    c11, c12 = c1
    c21, c22 = c2
    ciphertext1= c11 * c21
    ciphertext2 = c12 * c22
    m = elgamal_decrypt(public_key, private_key, (ciphertext1, ciphertext2))
    if m == m1 * m2:
        return True
    else:
        return False
```

测试的结果如下：



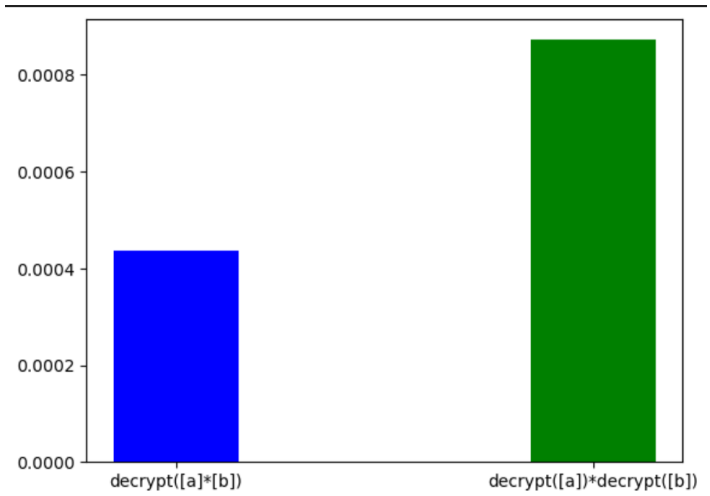
> 这里加了一个选择：**是否判断乘法同态性**。如果选择判断**乘法同态性**，则输入两个明文进行加密得到两个密文，然后查看两个密文的乘积解密后是否等于对应明文的乘积；如果选择不判断**乘法同态性**，则只输入一个明文。
>
> 通过上图的两个测试样例可以观察到：`ElGamal`算法满足乘法同态性。

对比乘法同态性质运算的时间开销

测试时间开销时，固定keysize=128，两个明文分别为100和50，一共循环500次。代码如下：

```
    # 对比乘法同态性质运算的时间开销
    key_size = 128

    # generate keys
    public_key, private_key = elgamal_key_generation(key_size)

    plaintext1 = 100
    plaintext2 = 50
    for i in range(500):
        ciphertext1 = elgamal_encrypt(public_key, plaintext1)
        ciphertext2 = elgamal_encrypt(public_key, plaintext2)
        # 测试先相乘后解密的时间开销
        start1 = time.perf_counter()
        c11, c12 = ciphertext1
        c21, c22 = ciphertext2
        ciphertext = c11 * c21, c12 * c22
        decrypted_text = elgamal_decrypt(public_key, private_key, ciphertext)
        end1 = time.perf_counter()

        # 测试先解密后相乘的时间开销
        start2 = time.perf_counter()
        m1 = elgamal_decrypt(public_key, private_key, ciphertext1)
        m2 = elgamal_decrypt(public_key, private_key, ciphertext2)
        result = m1 * m2
        end2 = time.perf_counter()
        print(end1-start1, end2-start2)
```

测试结果可视化如下：



> 根据上图可以看到，decrypt([a]*[b])的时间大概是decrypt([a])*decrypt([b])的一半。这说明与乘法操作相比，decrypt()函数的时间开销更大，乘法的开销可以忽略不计。

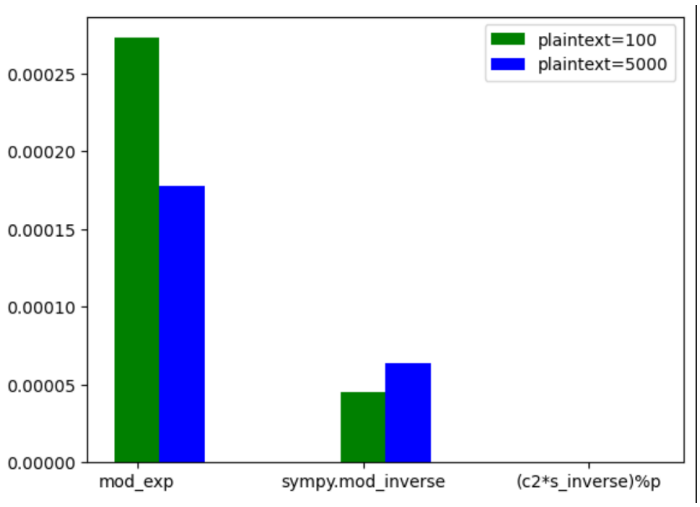
为了进一步探究，我将原来的代码改成如下的形式：

```
def elgamal_decrypt(public_key, private_key, ciphertext):
    """TODO: decrypt the ciphertext with the public key and the private key.
    """
```

```
    p, g, y = public_key
    x = private_key
    c1, c2 = ciphertext
    # 计算c1的模反演
    s1 = time.perf_counter()
    s = mod_exp(c1, x, p)
    e1 = time.perf_counter()
    # 计算s的模逆元
    s2 = time.perf_counter()
    s_inverse = sympy.mod_inverse(s, p)
    e2 = time.perf_counter()
    # 计算明文消息
    s3 = time.perf_counter()
    m = (c2 * s_inverse) % p
    e3 = time.perf_counter()
    return m , e1-s1, e2-s2, e3-s3
```

> 该函数测试了 `s = mod_exp(c1, x, p)`、`s_inverse = sympy.mod_inverse(s, p)`、`m = (c2 * s_inverse) % p` 三条语句的时间开销。

固定keysize=128，明文为100和5000，一共循环500次，测试结果可视化如下：



> 可以看到decrypt([a]*[b])和decrypt([a])主要时间开销在mod_exp()函数上，sympy.mod_inverse()函数次之。

总结：每次调用decrypt()函数 `s = mod_exp(c1, x, p)`、`s_inverse = sympy.mod_inverse(s, p)` 占据了大部分时间开销，time(decrypt([a]*[b]))只调用了一次decrypt()函数而time(decrypt([a])*decrypt([b]))调用了两次decrypt()函数，所以出现了decrypt([a]*[b])的时间大概是decrypt([a])*decrypt([b])的一半的现象。

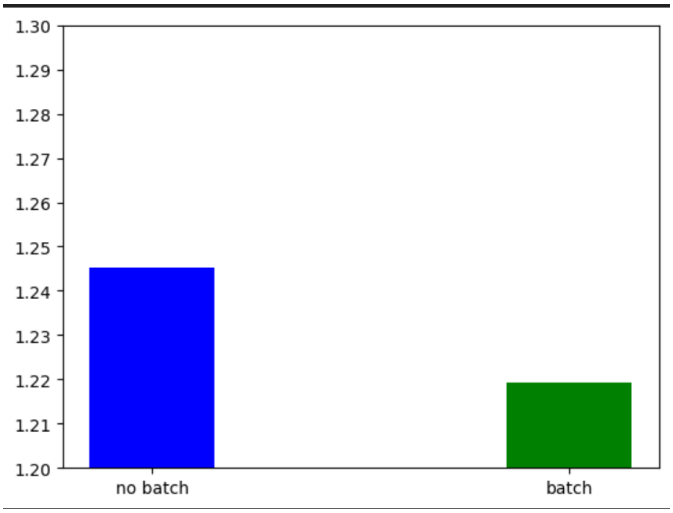
优化尝试1：批量加密和解密

这部分主要做出的尝试是：将多个明文的加密和解密通过一次调用完成，在 `elgamal_batch.py` 文件中对加解密函数进行修改如下：

```
def elgamal_encrypt_batch(public_key, plaintext_batch):
    p, g, y = public_key
    k_values = [random.randint(1, p - 1) for _ in range(len(plaintext_batch))]
    c1_values = [mod_exp(g, k, p) for k in k_values]
    c2_values = [(plaintext * mod_exp(y, k, p)) % p for plaintext, k in zip(plaintext_batch, k_values)]
    return list(zip(c1_values, c2_values))

def elgamal_decrypt_batch(public_key, private_key, ciphertext_batch):
    p, _, _ = public_key
    x = private_key
    s_values = [mod_exp(c1, x, p) for c1, _ in ciphertext_batch]
    s_inv_values = [sympy.mod_inverse(s, p) for s in s_values]
```

```
    plaintext_values = [(c2 * s_inv) % p for (_, c2), s_inv in zip(ciphertext_batch, s_inv_values)]
    return plaintext_values
```

测试时，将keysize设置为128，明文为[100, 101, 102, 103, 104]。作为对比，elgamal.py文件进行相同的设置，只是每次处理一个明文都要调用一次加密、解密函数。重复测试16000次，将测试结果可视化如下：



> 可以从上述结果看出：批量加密和解密可以优化ElGamal算法的时间开销。
>
> 有效性证明：通过批量加密和解密，可以减少调用函数的次数。在大数据量的场景下，一次性执行多个操作要优于分别执行操作，因此可以有效优化算法的时间开销。

优化尝试2：python并行计算

这部分做出的一个尝试是：调用 from concurrent.futures import ThreadPoolExecutor 中的 ThreadPoolExecutor 类实现多线程。在 elgamal_thread.py 文件中对加解密进行的修改如下：

```
def encrypt_wrapper(args):
    public_key, plaintext = args
    return elgamal_encrypt(public_key, plaintext)

def decrypt_wrapper(args):
    public_key, private_key, ciphertext = args
    return elgamal_decrypt(public_key, private_key, ciphertext)

def parallel_encrypt(public_key, plaintext_batch):
    args_list = [(public_key, plaintext) for plaintext in plaintext_batch]
    with ThreadPoolExecutor() as executor:
        encrypted_batch = list(executor.map(encrypt_wrapper, args_list))
    return encrypted_batch

def parallel_decrypt(public_key, private_key, ciphertext_batch):
    args_list = [(public_key, private_key, ciphertext) for ciphertext in ciphertext_batch]
    with ThreadPoolExecutor() as executor:
        decrypted_batch = list(executor.map(decrypt_wrapper, args_list))
    return decrypted_batch
```

> 这其中 elgamal_encrypt() 函数与 elgamal_decrypt() 函数保持不变，而 parallel_encrypt() 函数实现多线程加密， parallel_decrypt() 函数实现多线程解密。

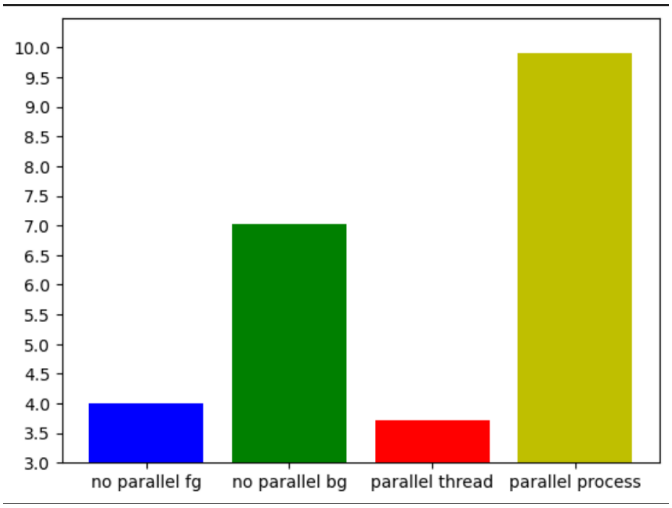
这部分做出的另一个尝试是：调用 from concurrent.futures import ProcessPoolExecutor 中的 ProcessPoolExecutor 类实现多进程。在 elgamal_process.py 文件中对加解密进行的修改如下：

```
def encrypt_wrapper(args):
    public_key, plaintext = args
    return elgamal_encrypt(public_key, plaintext)

```

```python
def decrypt_wrapper(args):
    public_key, private_key, ciphertext = args
    return elgamal_decrypt(public_key, private_key, ciphertext)

def parallel_encrypt(public_key, plaintext_batch):
    args_list = [(public_key, plaintext) for plaintext in plaintext_batch]
    with ProcessPoolExecutor() as executor:
        encrypted_batch = list(executor.map(encrypt_wrapper, args_list))
    return encrypted_batch

def parallel_decrypt(public_key, private_key, ciphertext_batch):
    args_list = [(public_key, private_key, ciphertext) for ciphertext in ciphertext_batch]
    with ProcessPoolExecutor() as executor:
        decrypted_batch = list(executor.map(decrypt_wrapper, args_list))
    return decrypted_batch
```

> 与上面类似，只不过调用的是多进程。

测试时，将keysize设为128，明文为 `plaintext_batch = [plaintext for plaintext in range(10000)]` 。作为对比，`elgamal.py` 文件进行相同的设置，只是每次处理一个明文都要调用一次加密、解密函数。重复测试50次，将测试结果可视化如下：



> 首先解释这四列的含义：`no parallel fg` 是指直接在终端运行 `shell` 脚本从而运行 `elgamal.py` 文件；`no parallel bg` 是指把 `shell` 脚本放在后台运行从而在后台运行 `elgamal.py` 文件；`parallel thread` 是指直接在终端运行 `shell` 脚本从而运行 `elgamal_thread.py` 文件；`parallel process` 是指直接在终端运行 `shell` 脚本从而运行 `elgamal_process.py` 文件。
>
> 从结果我们可以看出：
>
> - 多线程相比较原始代码时间开销减小
> - 多进程相比较原始代码时间开销增大
> - 原始代码放在后台运行测出的时间开销比放在前台测出的时间开销要显著增加
>
> 逐个推测出现上述现象的原因：
>
> - 多线程使得任务可以在多个线程中执行，减小时间开销，符合预期。但是优化效果可能还受GIL(全局解释器锁)的影响。
> - 多进程的时间开销很大，可能的原因如下：
>   - 可能选取的明文数据不够多/复杂，没有满足“大数据量”的场景，以及keysize的选取等。
>   - **资源开销**：在创建和管理多个进程时，涉及到更多的资源分配和切换，导致额外的开销。
>   - **切换开销**：进程切换通常比线程切换的开销更大。进程切换涉及到上下文切换、页表切换等操作，而线程切换仅涉及到寄存器的切换。在某些情况下，频繁的进程切换可能导致性能下降。
>   - **通信开销**：多进程间的通信比较复杂，可能引入额外的开销。
> - 后台运行的代码比前台运行的相同代码时间开销更大，可能的原因如下：
>   - 可能与操作系统调整前后台优先级有关，前台的优先级更高而后台的优先级更低。

- 操作系统可能会对后台进程施加一些资源限制，如CPU时间片的分配或内存使用的限制。可能导致后台进程在执行时受到更多的约束，执行速度相对较慢。
- 与操作系统的调度策略可能有关。