# COM-402 Homework ML Solution



# Exercise 1

```
pip install phe==1.4.0
```

## Mathematical computation

Let's have a quick look at how things work. In homomorphic encryption,

$enc(m_1) * enc(m_2) = enc(m_1 + m_2)$

But if you go to the wikipedia page, you'll see a bunch of other fun properties (mildly simplified for ease of reading):

- $dec\Big(enc(m_1) \cdot g^{m_2}\Big) = m_1 + m_2$, a.k.a you can add another plaintext without need to encrypt it
- $dec\Big(enc(m_1)^{m_2}\Big) = m_1 \cdot m_2$, a.k.a same as above, but for multiplication

This is exactly what is used on the server. It holds plaintext weights and bias, that they want to multiply/add to your encrypted vector. To add and multiply by integers, they can do so only using the public key.

## `phe` implementation of precision

As mentioned in the handout, you need to perform the computations on integers. The library `phe` deals with it in the following manner: multiply it by a constant in base 16 (the size of which will affect the precision and runtime), then do your computation, and then divide again by a constant in base 16. *The choice of these constants are the core of this exercise, and will be explained later.*

## Outline

The big idea here is not too hard:

1. Generate your public/secret keys
2. Using the public key, encrypt your vector (as per the desired precision)
3. Send that
4. Receive your encrypted result
5. Decrypt it

As a true software engineer, you would want to add tons of checks and features, but we are kinda lazy, and thus will focus on getting things to work.

## The code

Everything will happen in a function called `query_pred`, with the following signature:

```
def query_pred(vector: list, keys: tuple = None):
```

The vector is obvious. The keys is either a tuple `(public_key, private_key)` if already generated, or `None`, in which case we generate our own keys. The form situation will prove useful in exercise 2.

We also do some imports: `phe` for Paillier, `requests` to query the server.

```
from phe import paillier
import requests
```

And our final goal is to compare the obtained value to our reference:

```
refx = [0.48555949, 0.29289251, 0.63463107, 0.41933057, 0.78672205,
        0.58910837, 0.00739207, 0.31390802, 0.37037496, 0.3375726 ]
refy = 0.44812144746653826
assert 2**(-16) > abs(query_pred(refx) - refy)
```

## Part 1: Generating the keys

As mentioned above, we only generate keys if they are not already provided.

```
if keys is None:
        pubkey, privkey = paillier.generate_paillier_keypair()
    else:
        pubkey, privkey = keys
```

We could specify here the length of 2048 bits for the key, but it is already the default, and so is `g`.

## Part 2: Encrypt your vector

We cannot encrypt a vector at once, we must encrypt each value separately.

```
PRECISION = 2**(-16) # as indicated
enc_vector = [pubkey.encrypt(x, precision=PRECISION)
              .ciphertext()
              for x in vector]
```

## Part 3: Send that

A simple `requests` POST:

```
r = requests.post("http://localhost:8000/prediction",
                  json={"pub_key_n": pubkey.n,
                        "enc_feature_vector": enc_vector})
```

Note that we follow the specifications provided in the handout, for the form of the JSON.

## Part 4: Receive your encrypted result

If everything is OK, we received a JSON with only one field, `enc_prediction`:

```
f r.status_code != 200:
        print("Error while requesting_ {}".format(r.status_code))
        return None
y_enc = r.json()['enc_prediction']
```

## Part 5: Decrypt it

Now, to the fun part. `y_enc` is just an integer. We need to indicate to `phe` how to interpret it. Then, we decrypt it using our private key

```
encrypted = paillier.EncryptedNumber(pubkey, y_enc, EXPONENT)
y = privkey.decrypt(encrypted)
return y
```

# Is it over?

Nope. If you followed closely, we omitted the core of the exercise: the constant `EXPONENT`. This exponent, a power to a base 16, indicates with which precision the number was encoded/encrypted. You may be tempted to say "*Well, we wanted a precision of* $2^{-16}$*, which is equal to* $16^{-4}$ *so we select EXPONENT=-4.*

## Down the ~~Paillier~~ rabbit hole

This is very naive of you. Things are not so simple. See, what is supposed to be computed on the server is the following:

$$x_1 w_1 + x_2 w_2 + \ldots + x_n w_n + b$$

But the $x_i$ are encrypted. So it works "*in the Paillier domain*". Thanks to properties seen earlier:

$$enc(x_1)^{w_1} \cdot enc(x_2)^{w_2} \cdot \ldots \cdot enc(x_n)^{w_n} \cdot b$$

Now, suppose each $x_i$, $w_i$ and $b$ are floats. To work with them in a fixed precision, we multiply each of them by a constant, let's call it $k$. It is defined using our chosen precision (but we don't really care about its actual value). What happens when we do

$$enc(x_1) \cdot enc(x_2) \sim enc(kx_1) \cdot enc(kx_2)$$

this is equivalent to

$$enc(kx_1 + kx_2) = enc(k(x_1 + x_2))$$

Nothing too fancy here. If we simply added the $x_i$, we could apply your suggestion, and keep the same exponent.

But behold: what happens for

$$enc(x_i)^{w_i} \sim enc(kx_i)^{kw_i}$$

Some more math:

$$enc(kx_i)^{kw_i} \equiv \left((1+n)^{kx_i} r_i^n\right)^{kw_i} \mod n^2 \equiv (1+n)^{kx_i \cdot kw_i} r_i^{n \cdot kw_i} \mod n^2$$

We have 2 parts here. On the right

$$r_i^{n \cdot kw_i} \mod n^2 \simeq r_j^n$$

Because it will simply create another random value, which does not change the scheme. And on the left:

$$(1+n)^{kx_i \cdot kw_i} = (1+n)^{k^2 x_i w_i}$$

This represents a new number, that is now scaled by a constant $k^2$. Hence, we need to provide an exponent that is the *square* of the previous one, in order for things to scale accordingly. Remember, also, we provided the *precision* earlier as a floating value ($2^{-16}$), but now we need to provide an *exponent* of a base 16. Which means, we need to find $x$ such that $(2^{-16})^2 = 16^x$. Some quick maths later, we find $x = -8$.

# Exercise 2: [attack] Prediction-as-a-Service Model Stealing

We have (at least) two ways of attacking the model. Either by going weight by weight, or solving a linear equation.

## How to

### Weight-by-weight approach

The idea is simple. If we discard the concept of encryption, the model simply is

$$y = x_1 \cdot w_1 + x_2 \cdot w_2 + \ldots + x_n \cdot w_n + b$$

If we suppose there is no "noise" (voluntary or not), we can simply do the following:

- Provide the vector $[0, 0, 0, \ldots, 0]$. We will receive $y = 0 \cdot w_1 + \ldots + 0 \cdot w_n + b = b$. Cool, we have the bias
- Then, once we have it, we send unitary vectors: $[1, 0, 0, 0, .., 0]$, then $[0, 1, 0, 0, \ldots, 0]$, then $[0, 0, 1, 0, 0, .., 0]$. You get the idea.
- We will receive, in return, the value $1 \cdot w_1 + 0 \cdot w_2 + \ldots + 0 \cdot w_n + b = w_1 + b$
- From that, we simply deduct the bias we obtained earlier, which yields the value $w_1$ (or others, according to which vector we sent).

So we only need to send one query for the vector, then one query by weight. Cool!

The method above is easy, but *could* be detected. In fact, the *very basic protection* the handout speaks about, ensure there are at least 3 different ciphers. So you couldn't encrypt a 1, encrypt a 0, and send the cipher for the 0 multiple times. But lucky us, Paillier helps us here. Because encrypting includes a random value `r`, by encrypting the same value multiple times, we won't encounter (a lot of) duplicates. So we're safe here.

## Linear equation

Another cool method is by solving a linear equation. After all, this is what is used to compute the result, so why not use it to decompose the model?

We have, in vectors form,

$$\mathbf{x} \cdot \mathbf{w}^\top + b = y \simeq \tilde{\mathbf{x}} \cdot \tilde{\mathbf{w}}^\top = y$$

Where $\tilde{\mathbf{x}}$ is $\mathbf{x}$ with an additional $1$ and $\tilde{\mathbf{w}}$ is the weight vector, with one more component being the bias.

If we did one query, and had one result, we have an equation with 11 unknown. But as $\tilde{\mathbf{w}}$ is fixed, we can simply make 11 different queries, and solve it!

That's basically it. We do 11 queries, get 11 values in return. Then we have a matrix 11x11 (remember the additional column of $1$s for the bias), so we can solve it easily with any math library.

# Let's code!

In both solutions below, we will use the method `query_pred` we coded earlier. We input a vector, a pair of keys, and we get a deciphered $y$ in return.

## Weight-by-weight

The code is relatively straightforward.

```
# only create keys once
keys = paillier.generate_paillier_keypair()

print("Attacking bias") # pass only 0s
bias = query_pred([0]*10, keys)

weights = []
for i in range(10):
    print("Attacking weight {}".format(i), end="\r")
    # create vector [0,0,..,1,..,0,0]
    vector = np.zeros(10)
    vector[i] = 1
    # receive w_1 + b, deduce bias
    y = query_pred(vector, keys) - bias
    weights.append(y) # save the bias
# tada, you have the bias and weights!
print("\nWeights: {}\nbias: {}".format(weights, bias))
```

## Linear equation

This one needs to use numpy:

```
keys = paillier.generate_paillier_keypair()
X, y = [], []
```

```
for i in range(11):
    print("Iteration", i)
    # random vector of size 10
    vector = np.random.random_sample(10)
    y.append(query_pred(vector, keys))
    # save it as [x1, x2,x3,...,xn, 1], for the bias
    X.append(np.append(vector, 1))
#numpy-ize the results
X = np.array(X)
y = np.array(y)
#solve it
sol = np.linalg.solve(X, y)
#weights are the first N terms, bias is the last one
weights = sol[:-1], bias = sol[-1]
print("\nWeights: {}\nbias: {}".format(weights, bias))
```