CourseWork 2 – Anmol Arora

Linear Regression

It is a statistical method used to model relationship between a dependent variable and one or more independent variables. It assumes a linear relation between the variables.

Let's take example of House prices.

The cost of a house be

y = f(x) and y can depend on

i) Square footage
ii) How old the house is
iii) Closeness to city

Assuming all these variables are linearly related to y then we can say

y = b0 + b1.x1 + b2.x2 + b3.x3 + c

where

b0 = baseline price

b1, b2, b3 = coefficient that shows how the square footage, age of house and closeness to city effect the price

c = error in estimate

Error during model training is calculated by Sum of squared errors (SSE)

$$SSE = \sum(yactual - ypredicted)2$$

Mean squared error (MSE) is used to assess how well the model performed by providing a normalized measure of error. This is done post training.

$$MSE = \frac{SSE}{n}$$

Some more examples where linear regression works best are -

- Salary estimation
- Sales forecasting
- Stock market trends
- Energy consumption

Logistic regression

It is a statistical model used for binary classification, where the target variable has two possible outcomes (eg yes/no or 0/1). Unlike linear regression, it predicts the probability of an outcome instead of a continuous value.

Like linear regression where we assume a line equation for linear relation, logistic regression has a logistic function. Instead of a straight line it uses sigmoid function to map predictions to probabilities between 0 and 1.

$$P(y = 1) = \frac{1}{1 + e^{-(\beta 0 + \beta 1 x1 + \beta 2 x2 + \cdots)}}$$

- P(y = 1) : Probability of the positive class
- e : Exponential function
- β : Coefficients learned during training

If $P(y = 1) \geq 0.5$, classify as 1 (positive class)

Otherwise classify as 0 (negative class)

Let's take example of checking if an email is spam or not –

Features that classify email as spam (1) or not spam (0) can be

- Number of suspicious words (x1) = 10 (assuming values)
- Presence of hyperlink (x2, 0 or 1) = 1 (contains hyperlink)
- Email length (x3) = 200 (200 words)

And let $\beta 0, \beta 1, \beta 2, \beta 3 = -1, 0.3, 2, -0.01$ respectively then,

The logistic regression model would be –

$$P(y = 1) = \frac{1}{1 + e^{-(-1 + 0.3(10) + 2(1) - 0.01(200)}} = 0.73$$

The model predicts a 73% chance that the email is spam and since P > 0.5, classify it as spam (1).

Use cases of Logistic regression are –

- Predicting binary outcomes, spam/not spam, fraud/not fraud
- Medical diagnosis, if a patient has disease
- Credit scoring, if loan applicant has enough score or not

The paper Efficient Logistic Regression on Large Encrypted Data proposes a method for efficiently performing logistic regression on encrypted data using Homomorphic encryption. This allows us to securely compute logistic regression on encrypted data while preserving privacy.

Key ideas discussed in the paper include –

- Homomorphic encryption – This allows computations on encrypted data, while preserving privacy
- Logistic regression – Paper discusses applications of HE to securely compute LR for large data sets
- Optimization techniques – The authors optimize HE scheme to reduce computation time and memory overhead.
- Practical Evaluation – The approach is evaluated on real-world datasets to show feasibility.

TODO List for implementation

- Environment setup, install OpenFHE, pip libraries mnist, numpy, scikit-learn
- Download MNIST data set
- Extract it and place in samples dir
- Make python script which can read and sort the data
- Install openfhe-python wrapper (took a lot of time)
- Setup vanilla logistic regression model
- Setup Homomorphic encryption parameters
- Setup prediction model

Code and Env setup

I ran everything in a Ubuntu-24 VM, the same one I used for tutorial 10

I also used python3 venv.

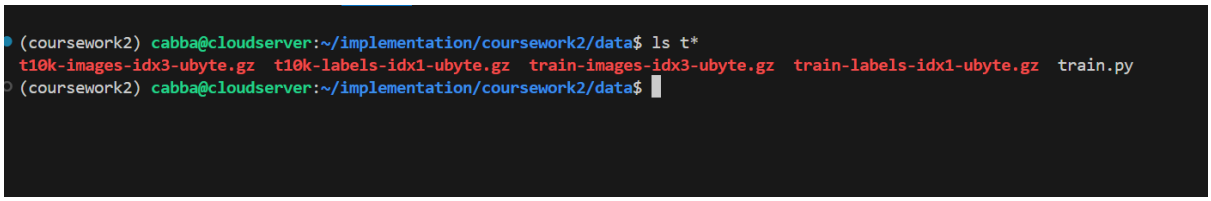
Part 1 – Vanilla model

Start by downloading dataset from https://github.com/fgnt/mnist

Just use wget to get all files

- https://raw.githubusercontent.com/fgnt/mnist/master/train-images-idx3-ubyte.gz

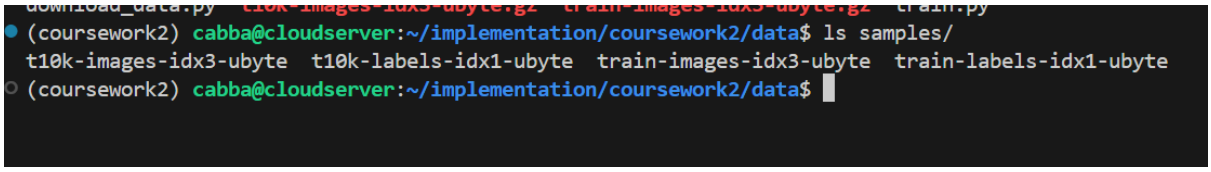
- https://raw.githubusercontent.com/fgnt/mnist/master/train-labels-idx1-ubyte.gz
- https://raw.githubusercontent.com/fgnt/mnist/master/t10k-images-idx3-ubyte.gz
- https://raw.githubusercontent.com/fgnt/mnist/master/t10k-labels-idx1-ubyte.gz



Extract them all using gzip -dk

(coursework2) cabba@cloudserver:~/implementation/coursework2/data$ gzip -dk train-images-idx3-ubyte.gz

And so on, make a samples directory



And move dataset there.

Python code

Import libraries

```
from mnist import MNIST
import numpy as np
from sklearn.model_selection import train_test_split
from sklearn.linear_model import LogisticRegression
from sklearn.metrics import accuracy_score
```

Main Function

```python
# Main function
def main():
    # Load MNIST data
    mndata = MNIST('samples')
    images, labels = mndata.load_training()

    # Preprocess and split data
    train_images, train_labels = preprocess_data(images, labels)
    train_images, val_images, train_labels, val_labels = train_test_split(
        train_images, train_labels, test_size=0.165, random_state=42
    )

    # Train model
    model = train_model(train_images, train_labels)

    # Validate the model
    val_predictions = model.predict(val_images)
    val_accuracy = accuracy_score(val_labels, val_predictions)
    val_precision = precision_score(val_labels, val_predictions)
    val_recall = recall_score(val_labels, val_predictions)
    val_f1 = f1_score(val_labels, val_predictions)

    print(f"Validation Metrics:")
    print(f"- Accuracy: {val_accuracy * 100:.2f}%")
    print(f"- Precision: {val_precision:.2f}")
    print(f"- Recall: {val_recall:.2f}")
    print(f"- F1 Score: {val_f1:.2f}")

    # Preprocess and test the model
    test_images, test_labels = mndata.load_testing()
    test_resized_images, test_filtered_labels = preprocess_data(test_images, test_labels)

    print("\nTesting the Model on the Test Dataset...")
    test_accuracy = test_model(model, test_resized_images, test_filtered_labels)
```

Train and test function

```python
# Logistic Regression functions
def train_model(train_images, train_labels):
    model = LogisticRegression(max_iter=1000)
    model.fit(train_images, train_labels)
    return model

def test_model(model, test_images, test_labels):
    test_predictions = model.predict(test_images)
    test_accuracy = accuracy_score(test_labels, test_predictions)
    print(f"Test Accuracy: {test_accuracy * 100:.2f}%")
    return test_accuracy
```

Pre-processing the images

```python
# Preprocessing functions
def resize_images(images):
    resized_images = []
    for image in images:
        image_14x14 = image.reshape(14, 2, 14, 2).mean(axis=(1, 3))
        resized_images.append(image_14x14)
    return np.array(resized_images)

def preprocess_data(images, labels):
    filtered_images, filtered_labels = [], []
    ## Positive class 1 if image is 3
    for i in range(len(labels)):
        if labels[i] == 3:
            filtered_images.append(images[i])
            filtered_labels.append(1)
    ## Negative class 0 if image is 8
        elif labels[i] == 8:
            filtered_images.append(images[i])
            filtered_labels.append(0)

    filtered_images = np.array(filtered_images)
    filtered_labels = np.array(filtered_labels)
    resized_images = resize_images(filtered_images).astype('float32') / 255.0
    resized_images = resized_images.reshape((resized_images.shape[0], -1))
    return resized_images, filtered_labels
```

From the paper we know that images must be resized to 14x14 from 28x28 pixels. Then I also sort from all images, images of 3 and 8. 3 Is positive class and 8 gets negative class. Labels are also sorted accordingly.

Run the model.

```
(coursework2) cabba@cloudserver:~/implementation/coursework2/data$ python3 dataset.py
Validation Metrics:
- Accuracy: 96.66%
- Precision: 0.97
- Recall: 0.96
- F1 Score: 0.97

Testing the Model on the Test Dataset...
Test Accuracy: 97.08%
```

Now let's implement HE,

In main function we add

```
    weights, bias = model.coef_.flatten(), model.intercept_[0]

    # # Encrypt and test one image
    print("Running homomorphic encryption on one test image...")
    index = random.randrange(0,len(test_resized_images))
    print("Real label of test image is: ", test_filtered_labels[index])
    ## If label = 1 then image is 3 and 8 if label = 0

    prediction = HE(weights, bias, test_resized_images[index])
```

We take weights and bias from the vanilla model and pass it to function HE, along with a random resized image.

Function HE

```
def HE(weights, bias, test_image):
    # Step 1: Initialize the CKKS context
    mult_depth, scale_mod_size, batch_size = 5, 41, 1024    # Values found by trial and error for scale_mod_size.

    parameters = CCParamsCKKSRNS()
    parameters.SetMultiplicativeDepth(mult_depth)
    parameters.SetScalingModSize(scale_mod_size)
    parameters.SetBatchSize(batch_size)

    cc = GenCryptoContext(parameters)
    cc.Enable(PKESchemeFeature.PKE)
    cc.Enable(PKESchemeFeature.KEYSWITCH)
    cc.Enable(PKESchemeFeature.LEVELEDSHE)
    cc.Enable(PKESchemeFeature.ADVANCEDSHE)

    print(f"The CKKS scheme is using ring dimension: {cc.GetRingDimension()}")

    # Generate keys
    keys= cc.KeyGen()
    cc.EvalMultKeyGen(keys.secretKey)
    cc.EvalSumKeyGen(keys.secretKey)
    cc.EvalRotateKeyGen(keys.secretKey, [1, -2])
```

First we setup parameters for the encryption. Help from this examples was taken - https://github.com/openfheorg/openfhe-python/tree/main?tab=readme-ov-file

Batch size is kept 1024 according to paper, and mult_depth is 5 because we atleast calculate a power of x^3 later so 5 should provide enough room for it.

Then we generate the keys which will be used in encryption and decryption.

Once we have all the params we move to step 2

```
# Step 2: Encrypt weights, bias, and test image
encrypted_weights = cc.Encrypt(keys.publicKey, cc.MakeCKKSPackedPlaintext(weights))
encrypted_bias = cc.Encrypt(keys.publicKey, cc.MakeCKKSPackedPlaintext([bias]))
encrypted_image = cc.Encrypt(keys.publicKey, cc.MakeCKKSPackedPlaintext(test_image))

# Step 3: Perform inference on encrypted data
encrypted_prediction = homomorphic_logistic_regression_predict(encrypted_weights, encrypted_image, encrypted_bias, cc, test_image)
```

Here we encrypt the weight, bias and the image and then pass it to our logistic model for HE.

```
# Logistic regression model algorithm
def homomorphic_logistic_regression_predict(encrypted_weights, encrypted_image, encrypted_bias, cc, test_image):
    # Compute dot product of encrypted image and encrypted weights (vectorized approach)
    encrypted_dot_product = cc.EvalInnerProduct(encrypted_weights, encrypted_image, 512)

    # Add the bias term to the dot product
    encrypted_sum_with_bias = cc.EvalAdd(encrypted_dot_product, encrypted_bias)

    # Apply the polynomial approximation for sigmoid: y = 0.5 - 0.25 * x - 0.0417 * x^3
    x = encrypted_sum_with_bias
    x2 = cc.EvalMult(x, x)  # x^2
    x3 = cc.EvalMult(x2, x)  # x^3

    # Apply the terms in the sigmoid approximation polynomial
    term1 = cc.EvalMult(x3, cc.MakeCKKSPackedPlaintext([-0.0417]))  # 0.-0.0417 * x^3
    term2 = cc.EvalMult(x, cc.MakeCKKSPackedPlaintext([0.25]))   # 0.25 * x
    constant_term = cc.MakeCKKSPackedPlaintext([0.5])  # Constant term 0.5

    sum_1 = cc.EvalAdd(term2, term1)
    sigmoid = cc.EvalAdd(sum_1, 0.5)

    return sigmoid
```

The algorithm works by calculating dot product of encrypted_weight and encrypted_image while maintaining batch_size 512.

After dot product the encrypted_bias is added.

Then we use this value as x in the sigmoid equation

$$P = 0.5 + 0.25x - 0.0417x^3$$

Terms are calculated and then value of sigmoid is returned.

Back in HE function now we do step 4

```
    # Step 4: Decrypt prediction for verification
    decrypted_prediction = cc.Decrypt(encrypted_prediction, keys.secretKey)
    decrypted_prediction.SetLength(1)  # Only one value
    print("Decrypted prediction:", decrypted_prediction)
    prediction_value = decrypted_prediction.GetRealPackedValue()[0]

        # Now classify based on the value
    if prediction_value > 1:
        predicted_class = 0  # Positive class
    else:
        predicted_class = 1  # Negative class

    # Output the results
    print(f"Decrypted Prediction: {prediction_value}")
    print(f"Predicted Class: {predicted_class}")

    return predicted_class
```

We decrypt the prediction using the secret key and get the value out. If prediction value is positive it goes to positive class, else negative.

Back in main function

```
prediction = HE(weights, bias, test_resized_images[index])

if prediction == 1:
    print("Model predicts it is : 3")
else:
    print("Model predicts it is : 8")
```

Now we map prediction to 1 and 0 and get our final prediction.

Running the model with HE

```
(coursework2) cabba@cloudserver:~/implementation/coursework2/data$ python3 dataset.py
Validation Metrics:
- Accuracy: 96.66%
- Precision: 0.97
- Recall: 0.96
- F1 Score: 0.97

Testing the Model on the Test Dataset...
Test Accuracy: 97.08%
Running homomorphic encryption on one test image...
Real Class of test image is:  1
The CKKS scheme is using ring dimension: 32768
Decrypted prediction: (-1.53288,  ... ); Estimated precision: 30 bits

Decrypted Prediction: -1.5328823856985774
Predicted Class: 1
Model predicts it is : 3
(coursework2) cabba@cloudserver:~/implementation/coursework2/data$
```

We can see that the model correctly predicted the class of the image. The random image we sent was of class 1 ie it was a 3 and the model also predicted it is a 3.

Running the model more than once to see how it performs

```
    # Initialize a counter for correct predictions
    correct_predictions = 0

    # Loop through all test images
    for i in range(30):
        # Get the true label (1 for 3, 0 for 8)
        true_label = test_filtered_labels[i]
        if true_label == 1:
            true_label_str = "3"
        else:
            true_label_str = "8"

        # Run homomorphic encryption prediction for the current image
        prediction = HE(weights, bias, test_resized_images[i])

        # Print prediction
        if prediction == 1:
            predicted_class = "3"
        else:
            predicted_class = "8"

        # Print the result for the current test image
        print(f"True label: {true_label_str}, Model predicts: {predicted_class}")

        # Compare prediction with true label
        if prediction == true_label:
            correct_predictions += 1

    # Print the total number of correct predictions
    print(f"Correct predictions: {correct_predictions} out of 30")
```

Output

```
Decrypted Prediction: -18.846052275172802
Predicted Class: 1
True label: 3, Model predicts: 3
The CKKS scheme is using ring dimension: 32768
Decrypted prediction: (-6.58477,  ... ); Estimated precision: 30 bits

Decrypted Prediction: -6.5847710597620726
Predicted Class: 1
True label: 3, Model predicts: 3
The CKKS scheme is using ring dimension: 32768
Decrypted prediction: (-1.81948,  ... ); Estimated precision: 30 bits

Decrypted Prediction: -1.8194771995288432
Predicted Class: 1
True label: 3, Model predicts: 3
The CKKS scheme is using ring dimension: 32768
Decrypted prediction: (-21.1741,  ... ); Estimated precision: 30 bits

Decrypted Prediction: -21.17414248641719
Predicted Class: 1
True label: 3, Model predicts: 3
Correct predictions: 29 out of 30
(coursework2) cabba@cloudserver:~/implementation/coursework2/data$
```

So pretty good accuracy 😊

My thoughts –

I found the paper very challenging and not at all intuitive, it would have been so much easier if some instructions on setting up openfhe-python were provided though it was my choice to use python i feel python is the way to go for such tasks. If i used c++ it would've only made things much worse.

I struggled a lot to setup openfhe and when i did the HE function did not come naturally to me, to go through all those examples in the repo and the Algorithm 5 being so elaborate and complex, expecting students who just learnt about HE from 3-5 videos is a long shot imo.

Still I have learnt something and i hope its enough for this report and coursework.

Steps to run the code –

1. Extract data from samples folder

   tar -zxvf samples.tar.gz

2. Install python requirements if any

   pip install <missing package>

3. Setup openfhe-python - https://github.com/openfheorg/openfhe-python/tree/main

   The instructions in readme should work fine if openfhe-development is working.
   Remember to use sudo make install so its accessible globally, that cause a lot of issues.
   Also check after openfhe-python is installed that python has it in pythonpath.
   My openfhe-python installed in /usr/local so i had to add
   Export PYTHONPATH=/usr/local:$PYTHONPATH to my terminal env to make it run

4. Run the script. python3 dataset.py