NYU

Secure Outsourcing of XGB Classification using Homomorphic Encryption (Project Type 4)

By Team ENIGMA

Andy Chen, Avinash Narasimhan, Roman Negri
Applied Cryptography CS-GY 6903

.

Project Overview

- In this project, we aim to design and implement a secure solution that allows a server to perform machine learning classification tasks on encrypted data supplied by a client
- The key challenge is to ensure the confidentiality of the data owner's input data while still enabling the server to compute the desired classification results
- To address this challenge, we will leverage the power of homomorphic encryption, which allows computations to be performed directly on encrypted data without decryption
- This enables the server to apply a machine learning classifier on the data owner's encrypted inputs and return the encrypted classification results, which they can then decrypt to obtain the final output

Chosen ML Classifier

- For this project, we have opted to implement an XGBoost classifier as the machine learning model. XGBoost, an advanced implementation of gradient boosting having exceptional performance in various machine learning competitions and real-world applications

- XGBoost can handle a broader range of problems, including binary and multiclass classification, tasks. XGBoost constructs an ensemble of weak learners sequentially, correcting the errors made by the previous ones. This allows XGBoost to capture complex relationships within the data

- XGBoost offers various techniques for feature importance analysis and model interpretation, enabling insights into the underlying patterns learned by the model. From a technical standpoint, XGBoost's operations, including tree construction and optimization, are highly optimized, making it computationally efficient and scalable to large datasets

- XGBoost provides parameters for regularization and control over model complexity, allowing fine-tuning to prevent overfitting. In the context of this project, XGBoost's flexibility, robustness, and superior performance make it a compelling choice for handling the classification tasks

NYU

Homomorphic Encryption Scheme

- To facilitate the secure outsourcing of the XGBoost classifier, we will utilize the TFHE (Torus Fully Homomorphic Encryption) scheme as the underlying homomorphic encryption mechanism

- TFHE is capable of supporting both boolean and arithmetic circuit computations. This versatility aligns well with the requirements of implementing complex machine learning algorithms like XGBoost, which involve a mixture of boolean and arithmetic operations

- TFHE's reputation for computational efficiency is particularly valuable for practical applications, especially those involving machine learning tasks. The scheme's ability to handle both boolean and arithmetic circuits ensures its applicability to a wide spectrum of machine learning models

- Based on the learning with errors (LWE) problem, TFHE offers robust cryptographic security guarantees. Leveraging TFHE for homomorphic encryption allows cloud service providers to execute XGBoost classification on encrypted inputs while maintaining the confidentiality of the data

- With TFHE as the foundation for homomorphic encryption, we can confidently enable secure outsourcing of XGBoost classification tasks, unlocking the potential for privacy-preserving machine learning in cloud-based environments.

NYU

Chosen Dataset

- For this project, we will be using the Breast Cancer Wisconsin (Diagnostic) dataset originated from research conducted at the University of Wisconsin. This dataset is commonly referred to as the "Wisconsin Breast Cancer Dataset" or simply the "Breast Cancer Dataset."

- This dataset contains features computed from digitized images of fine needle aspirates (FNA) of breast masses. These features describe various characteristics of cell nuclei present in the images, such as radius, texture, perimeter, area, smoothness, compactness, concavity, symmetry, and fractal dimension

- The dataset includes measurements from images of breast cancer cells, classified as either malignant or benign. Each instance in the dataset represents a single FNA sample, with features describing the cell nuclei and a corresponding label indicating whether the sample is malignant or benign

- The dataset contains a total of 569 instances, with 212 malignant samples and 357 benign samples. The primary objective of using this dataset in machine learning is to develop predictive models that can accurately classify breast masses as either malignant or benign based on their features

- The dataset's relatively small size and high-quality features make it accessible for experimentation and analysis by students, researchers, and data scientists alike.

Implementation

- We used the Concrete-ML library in order to compile our machine learning model to a FHE model. The code begins by importing necessary libraries and modules, including FHEModelClient, FHEModelDev, and FHEModelServer from concrete.ml.deployment, as well as other libraries
- The Network class represents a network for transferring files between the development environment, client, and server. It contains methods for sending parameters to the client, sending the model to the server, and sending public keys to the server
- The generate_data() function loads the Breast Cancer Wisconsin dataset (load_breast_cancer) and splits it into training data (X, y) and client data (client_X, client_y)
- The create_and_train_model() function trains an XGBoost classifier (XGBClassifier) on the training data and compiles it. The compiled model and a deep copy of the model are returned
- The script sets up directories for the development environment, client, and server, ensuring clean environments for each

NYU

Implementation Continued

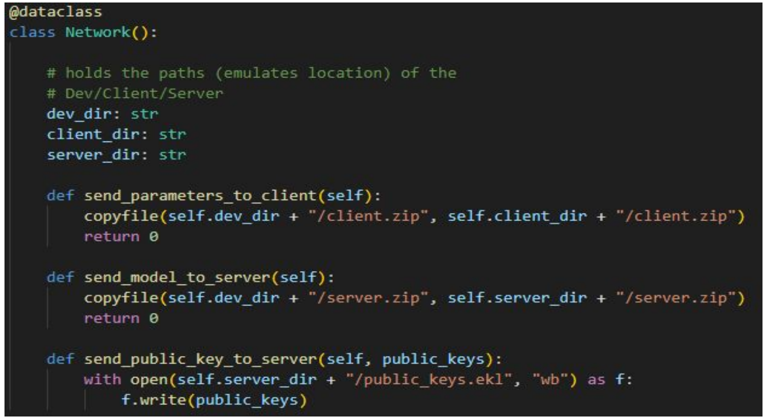
- The created network is used to send the necessary data and model specifications from the development environment to the client and server

- An instance of FHEModelClient is created with the client directory, and private and evaluation keys are generated for the client. An instance of FHEModelServer is created with the server directory

- Encrypted data from the client is sent to the server for classification using the FHEModelClient and FHEModelServer classes. The process involves quantizing, encrypting, serializing, running the model on the encrypted data, deserializing, decrypting, and dequantizing the results

- The script measures the time taken for predictions with and without FHE, compares the accuracy of predictions between FHE-enabled inference and the clear model, and prints the results

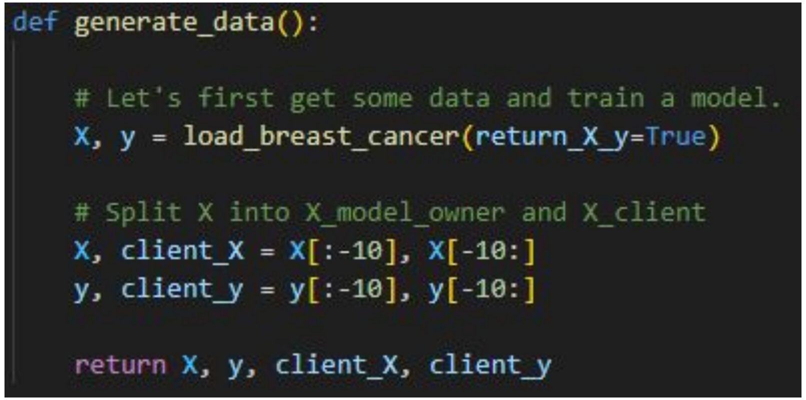
Discussion about specific Code



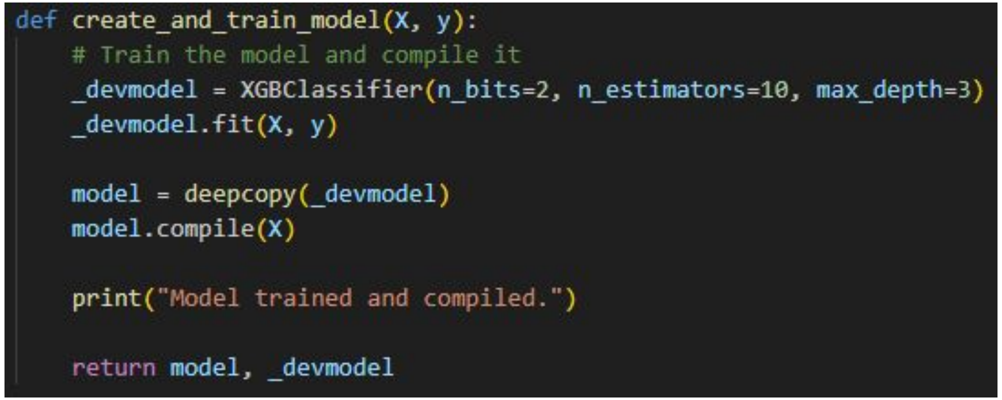
```
@dataclass
class Network():

    # holds the paths (emulates location) of the
    # Dev/Client/Server
    dev_dir: str
    client_dir: str
    server_dir: str

    def send_parameters_to_client(self):
        copyfile(self.dev_dir + "/client.zip", self.client_dir + "/client.zip")
        return 0

    def send_model_to_server(self):
        copyfile(self.dev_dir + "/server.zip", self.server_dir + "/server.zip")
        return 0

    def send_public_key_to_server(self, public_keys):
        with open(self.server_dir + "/public_keys.ekl", "wb") as f:
            f.write(public_keys)
```

The Network class will handle sending parameters to client and sending model and public key to server.

Discussion about specific Code



```python
def generate_data():

    # Let's first get some data and train a model.
    X, y = load_breast_cancer(return_X_y=True)

    # Split X into X_model_owner and X_client
    X, client_X = X[:-10], X[-10:]
    y, client_y = y[:-10], y[-10:]

    return X, y, client_X, client_y
```

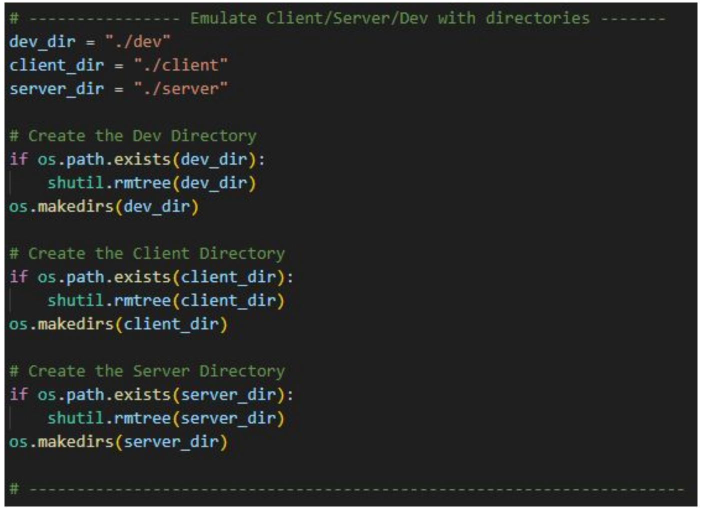


```python
def create_and_train_model(X, y):
    # Train the model and compile it
    _devmodel = XGBClassifier(n_bits=2, n_estimators=10, max_depth=3)
    _devmodel.fit(X, y)

    model = deepcopy(_devmodel)
    model.compile(X)

    print("Model trained and compiled.")

    return model, _devmodel
```

This helper function will prepare the data. For this project, we will be using the public breast cancer dataset.

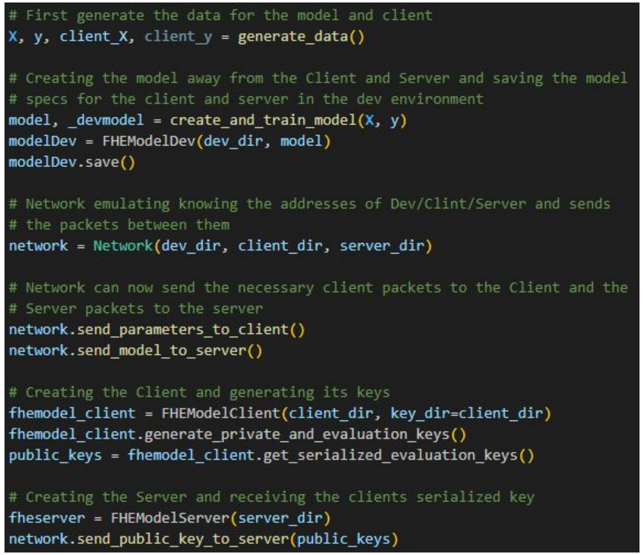
This helper function will create the XGBClassifier model and train the model on the given data.

Discussion about specific Code



```python
# ---------------- Emulate Client/Server/Dev with directories -------
dev_dir = "./dev"
client_dir = "./client"
server_dir = "./server"

# Create the Dev Directory
if os.path.exists(dev_dir):
    shutil.rmtree(dev_dir)
os.makedirs(dev_dir)

# Create the Client Directory
if os.path.exists(client_dir):
    shutil.rmtree(client_dir)
os.makedirs(client_dir)

# Create the Server Directory
if os.path.exists(server_dir):
    shutil.rmtree(server_dir)
os.makedirs(server_dir)

# ---------------------------------------------------------------------
```

Firstly, we want to define and create our directories to emulate client/server/dev.

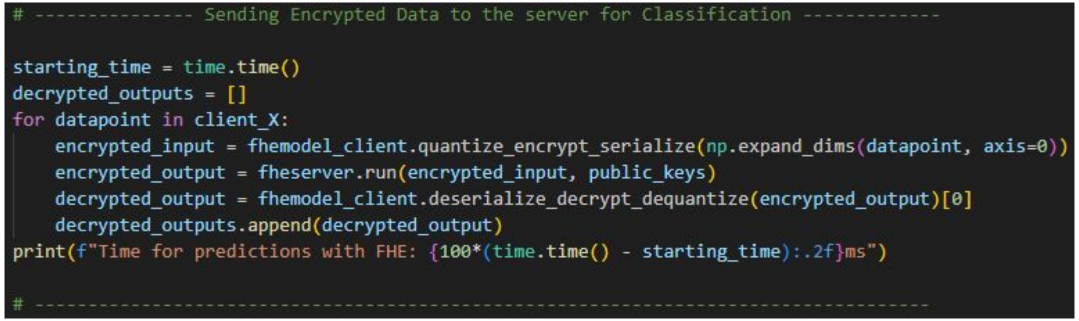


```python
# First generate the data for the model and client
X, y, client_X, client_y = generate_data()

# Creating the model away from the Client and Server and saving the model
# specs for the client and server in the dev environment
model, _devmodel = create_and_train_model(X, y)
modelDev = FHEModelDev(dev_dir, model)
modelDev.save()

# Network emulating knowing the addresses of Dev/Clint/Server and sends
# the packets between them
network = Network(dev_dir, client_dir, server_dir)

# Network can now send the necessary client packets to the Client and the
# Server packets to the server
network.send_parameters_to_client()
network.send_model_to_server()

# Creating the Client and generating its keys
fhemodel_client = FHEModelClient(client_dir, key_dir=client_dir)
fhemodel_client.generate_private_and_evaluation_keys()
public_keys = fhemodel_client.get_serialized_evaluation_keys()

# Creating the Server and receiving the clients serialized key
fheserver = FHEModelServer(server_dir)
network.send_public_key_to_server(public_keys)
```

Next, we utilize our helper functions to generate the data, create our training model, and create and send information to the client and server.

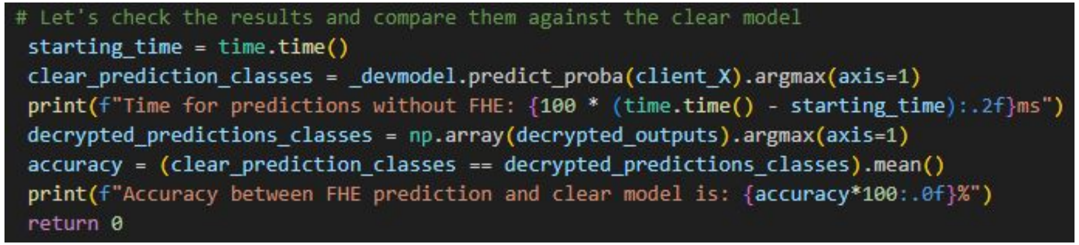
NYU

Discussion about specific Code



```
# -------------- Sending Encrypted Data to the server for Classification -------------

starting_time = time.time()
decrypted_outputs = []
for datapoint in client_X:
    encrypted_input = fhemodel_client.quantize_encrypt_serialize(np.expand_dims(datapoint, axis=0))
    encrypted_output = fheserver.run(encrypted_input, public_keys)
    decrypted_output = fhemodel_client.deserialize_decrypt_dequantize(encrypted_output)[0]
    decrypted_outputs.append(decrypted_output)
print(f"Time for predictions with FHE: {100*(time.time() - starting_time):.2f}ms")

# ---------------------------------------------------------------------------------------
```

We will send our encrypted data to the server for classification.



```
# Let's check the results and compare them against the clear model
 starting_time = time.time()
 clear_prediction_classes = _devmodel.predict_proba(client_X).argmax(axis=1)
 print(f"Time for predictions without FHE: {100 * (time.time() - starting_time):.2f}ms")
 decrypted_predictions_classes = np.array(decrypted_outputs).argmax(axis=1)
 accuracy = (clear_prediction_classes == decrypted_predictions_classes).mean()
 print(f"Accuracy between FHE prediction and clear model is: {accuracy*100:.0f}%")
 return 0
```

Finally, we check our results against the clear model and report the accuracy.

Performance of the algorithm with and without input encryption

Time Analysis:

We found that on average the FHE model ran around 10,000x slower than the model before being compiled to FHE.

Accuracy Analysis:

But on our sample dataset, since it is a relatively simple and low-dimensional dataset, our accuracy between the two models (FHE and not FHE compiled) was exactly the same.

NYU

Thank you!