PEOPLE'S DEMOCRATIC REPUBLIC OF ALGERIA

MINISTRY OF HIGHER EDUCATION AND SCIENTIFIC RESEARCH

SAAD DAHLEB BLIDA 01 UNIVERSITY

DEPARTMENT OF COMPUTER SCIENCE

MASTER'S INTELLIGENT SYSTEMS ENGINEERING

COMPUTER SYSTEMS SECURITY

REPORT

Implementation of Horizontal & Vertical Federated Learning Models for Diabetes Dataset with a Secured Connection using Solidity

Implemented and written by Abdelatif Mekri

Academic year : 2024-2025

Introduction

In an era where data privacy and security are paramount, Federated Learning (FL) has emerged as a powerful solution to enable collaborative model training without compromising sensitive information. Traditional machine learning approaches require centralized data collection, raising concerns about data breaches and regulatory compliance, particularly in domains like healthcare and finance. Federated Learning addresses these challenges by allowing multiple entities to train a shared model locally on their devices while only exchanging model updates rather than raw data.

The integration of blockchain technology further strengthens the security and trustworthiness of federated learning systems. Smart contracts, implemented using Solidity, provide a decentralized and tamper-proof mechanism to manage client-server communication, ensuring data integrity and resistance to adversarial attacks.

This report explores the implementation of Federated Learning, focusing on both Horizontal and Vertical Federated Learning paradigms. Additionally, I integrate Solidity-based smart contracts to enhance the security of communication between the server and participating clients. The upcoming sections will provide a detailed overview of the problem statement, dataset, methodology, experimental setup, and results.

Problem Statement

Health data is highly sensitive and must be protected to comply with strict regulations and to maintain patient confidentiality. However, valuable insights can be drawn from medical data to improve diagnoses, treatment plans, and overall healthcare efficiency. Traditional centralized machine learning approaches pose significant privacy risks by requiring data to be stored in a single location, increasing exposure to breaches.

Federated Learning provides a solution by allowing healthcare institutions to collaborate on model training without sharing raw data. To illustrate the applicability of this approach, I focus on diabetes detection as a use case. While the choice of diabetes is specific, the principles and methodologies demonstrated here can be extended to other medical conditions and industries where privacy-preserving AI is crucial.

This report explores the implementation of Federated Learning, focusing on both Horizontal and Vertical Federated Learning paradigms. Additionally, I integrate Solidity-based smart contracts to enhance the security of communication between the server and participating clients. The upcoming sections will provide a detailed overview of the problem statement, dataset, methodology, experimental setup, and results.

Fundamentals

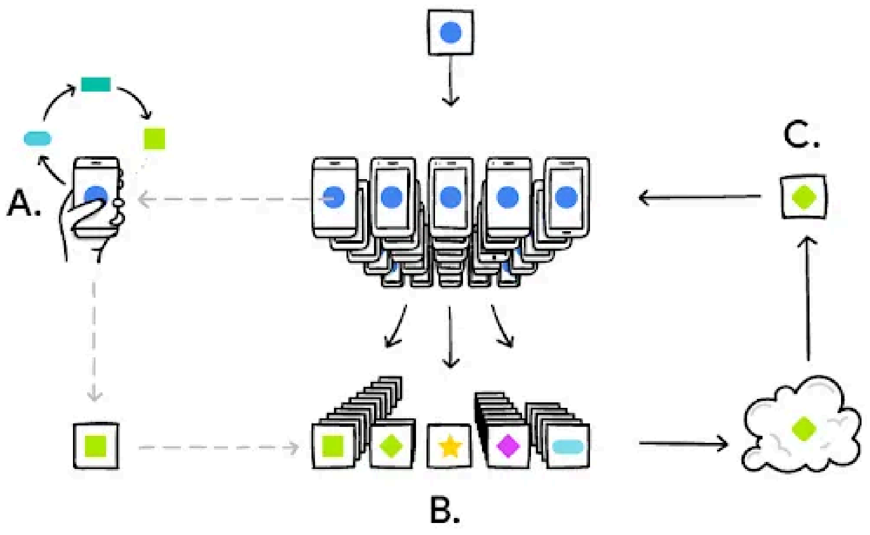
Federated Learning

What is Federated Learning?

Federated Learning (FL) is a decentralized machine learning approach that enables multiple entities to collaboratively train a model without sharing raw data. Unlike traditional centralized learning methods, where data is aggregated in a single location, FL allows training to occur locally on edge devices or servers. Instead of transferring data, only model updates ,such as gradients or parameters are exchanged, ensuring privacy and reducing the risk of data breaches.

Key Components of Federated Learning

- **Local Clients/Devices**: These are the nodes (such as hospitals, smartphones, or banks) that generate and/or store data. Each client trains a local model using its own dataset.
- **Central Server**: The central entity coordinates training by collecting and aggregating model updates from clients before distributing an improved global model.
- **Communication Protocol**: Defines how model updates are securely transmitted between clients and the server, ensuring efficiency and privacy protection.



[1]

Figure 1. Your phone personalizes the model locally, based on your usage A). Many users' updates are aggregated B) to form a consensus change C) to the shared model, after which the procedure is repeated.

Types of Federated Learning

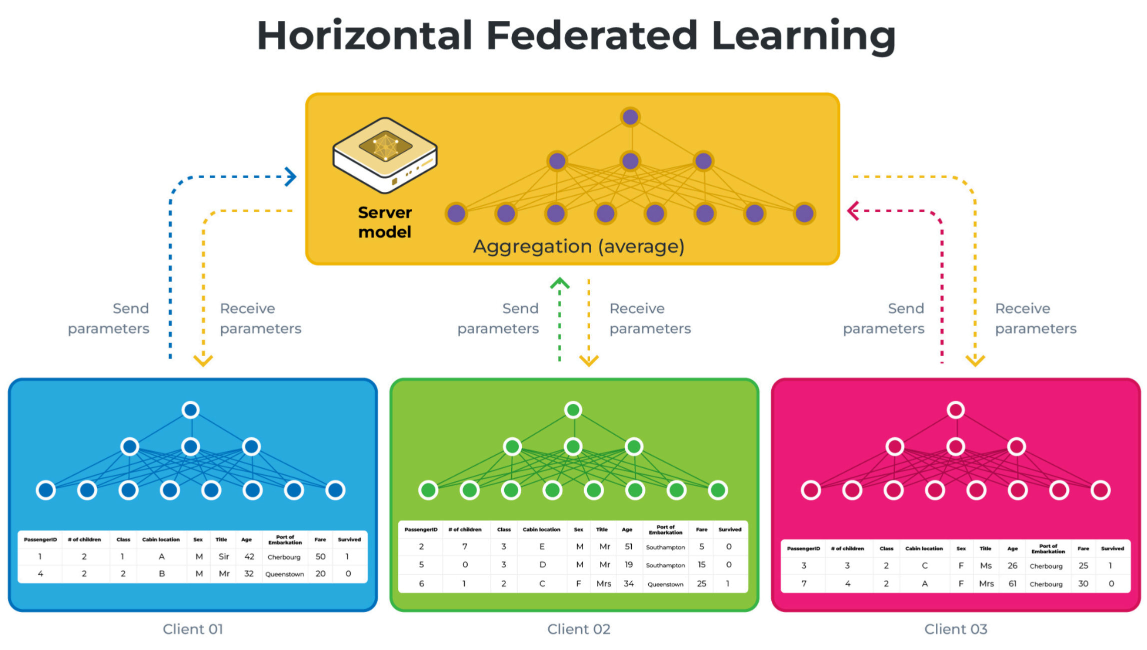
[1] McMahan, B., & Ramage, D. (2017, April 6). *Federated learning: Collaborative machine learning without centralized training data*. Google Research Blog. https://research.google/blog/federated-learning-collaborative-machine-learning-without-centralized-training-data/

Centralized vs. Decentralized Federated Learning

- **Centralized Federated Learning**: A central server manages communication and model aggregation. Clients train local models and send updates to the server, which refines the global model before redistributing it.
    - **Example**: Hospitals collaboratively training a disease prediction model while preserving patient confidentiality.
- **Decentralized Federated Learning**: Clients communicate directly with each other, eliminating the need for a central server. This enhances privacy, reduces dependency on a single point of failure, and ensures robustness.
    - **Example**: A network of mobile devices training a next-word prediction model without relying on a central authority.

Horizontal vs. Vertical Federated Learning

- **Horizontal Federated Learning (HFL)**: Clients have datasets with the same feature space but different user samples. This setup is common when organizations operate in the same industry and collect similar types of data.
    - **Example**: Multiple banks training a fraud detection model using their own transaction data.



PassengerID	# of children	Class	Cabin location	Sex	Title	Age	Port of Embarkation	Fare	Survived
1	2	1	A	M	Sir	42	Cherbourg	50	1
4	2	2	B	M	Mr	32	Queenstown	20	0

PassengerID	# of children	Class	Cabin location	Sex	Title	Age	Port of Embarkation	Fare	Survived
2	7	3	E	M	Mr	51	Southampton	5	0
5	0	3	D	M	Mr	19	Southampton	15	0
6	1	2	C	F	Mrs	34	Queenstown	25	1

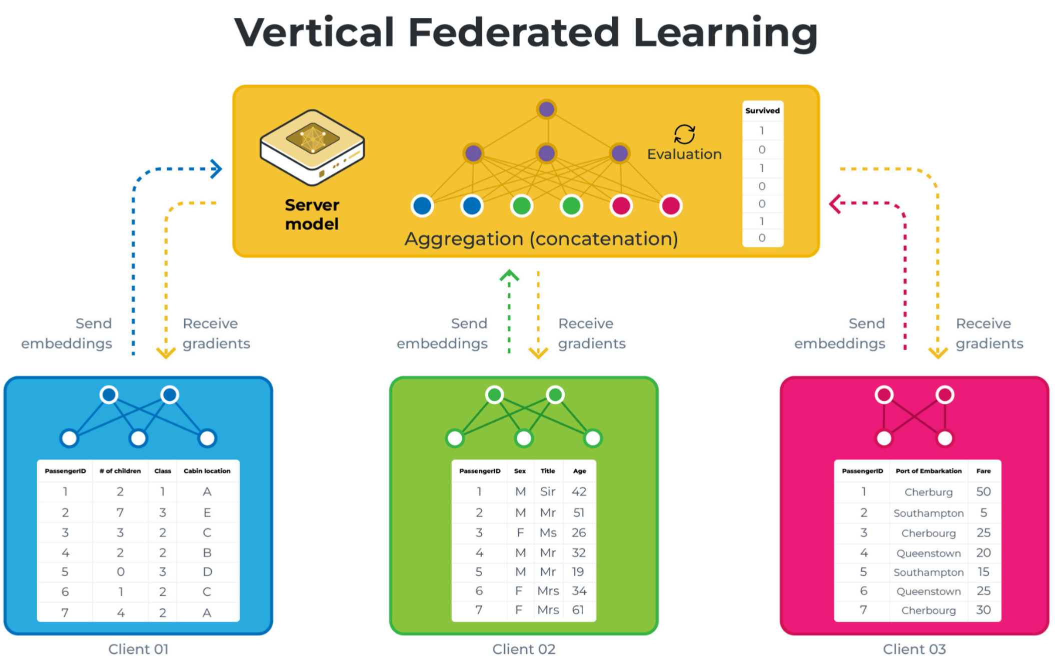
PassengerID	# of children	Class	Cabin location	Sex	Title	Age	Port of Embarkation	Fare	Survived
3	3	2	C	F	Ms	26	Cherbourg	25	1
7	4	2	A	F	Mrs	61	Cherbourg	30	0

[2]

- **Vertical Federated Learning (VFL)**: Clients share common user samples but have different feature spaces. This setup is useful when organizations hold complementary data about the same users.

[2] Flower. (n.d.). *Vertical Federated Learning with Flower*. Flower. https://flower.ai/docs/examples/vertical-fl.html

- **Example**: A retail company and a bank combining purchase history and financial data to improve credit risk assessments.[3]



Survived
1
0
1
0
0
1
0

PassengerID	# of children	Class	Cabin location
1	2	1	A
2	7	3	E
3	3	2	C
4	2	2	B
5	0	3	D
6	1	2	C
7	4	2	A

PassengerID	Sex	Title	Age
1	M	Sir	42
2	M	Mr	51
3	F	Ms	26
4	M	Mr	32
5	M	Mr	19
6	F	Mrs	34
7	F	Mrs	61

PassengerID	Port of Embarkation	Fare
1	Cherburg	50
2	Southampton	5
3	Cherbourg	25
4	Queenstown	20
5	Southampton	15
6	Queenstown	25
7	Cherbourg	30

[4]

Advantages of Federated learning (FL)

Federated learning (FL) offers several distinct advantages, particularly in scenarios where data privacy, decentralization, and efficiency are critical. Here's a structured overview of its key benefits:

1. Privacy Preservation
    - Data remains on local devices, avoiding direct sharing of raw sensitive information. Only model updates (e.g., gradients) are shared, reducing exposure to breaches and aligning with privacy-first principles.
2. Regulatory Compliance
    - Facilitates adherence to strict data protection laws (e.g., GDPR, HIPAA) by minimizing data transfer and central storage, which is crucial in healthcare, finance, and other regulated industries.
3. Bandwidth Efficiency
    - Reduces communication costs by transmitting only compact model updates instead of large raw datasets, ideal for environments with limited connectivity or costly data plans.

[3] GeeksforGeeks. (2024, May 27). *Types of Federated Learning in Machine Learning*. GeeksforGeeks. https://www.geeksforgeeks.org/types-of-federated-learning-in-machine-learning/
[4] Flower. (n.d.). *Vertical Federated Learning with Flower*. Flower. https://flower.ai/docs/examples/vertical-fl.html

4. Scalability
    - Enables training across thousands of devices (e.g., smartphones, IoT sensors) without centralized infrastructure bottlenecks, supporting large-scale distributed systems.
5. Enhanced Security
    - Decentralized data storage limits attack surfaces. Techniques like secure aggregation and differential privacy can further protect model updates.
6. Robustness via Data Diversity
    - Leverages heterogeneous data from diverse sources/environments, improving model generalization and reducing bias compared to centralized datasets.
7. Fault Tolerance
    - Resilient to device dropouts or network issues, as training continues with available participants, avoiding single points of failure.
8. Real-Time Personalization
    - Models can adapt locally to user-specific patterns (e.g., keyboard predictions, wearable health monitoring) without compromising privacy.
9. Collaboration Without Data Sharing
    - Organizations/institutions can jointly improve models without sharing proprietary data, fostering innovation in competitive or siloed sectors.
10. Edge Computing Synergy
    - Aligns with edge paradigms by processing data locally, reducing latency and enabling offline training, critical for applications like autonomous vehicles or smart factories.
11. User Trust and Participation
    - Increased user willingness to contribute to model training when privacy is assured, enhancing dataset diversity and model quality.
12. Reduced Centralized Storage Costs
    - Eliminates the need for large-scale centralized data storage infrastructure, distributing storage responsibilities across devices.
13. Continuous Learning
    - Models dynamically update as new data is generated on devices, enabling rapid adaptation to evolving trends (e.g., pandemic prediction, fraud detection).

By addressing privacy, efficiency, and scalability challenges, federated learning is particularly advantageous for applications in healthcare, IoT, mobile services, and industries requiring secure collaboration.

Solidity

What is solidity ?

Solidity is a high-level, statically-typed programming language designed for writing smart contracts on Ethereum-based blockchains. It is influenced by JavaScript, Python, and C++, making it accessible to developers familiar with these languages. Solidity is primarily used to create decentralized applications (dApps) and to implement smart contracts that run on the Ethereum Virtual Machine (EVM).

Advantages of using solidity

1. **EVM Compatibility**: Solidity is specifically designed for the Ethereum Virtual Machine (EVM), making it the ideal language for developing decentralized applications (dApps) on the Ethereum blockchain. This compatibility ensures that Solidity contracts can easily interact with the Ethereum network.
2. **Security Features**: Solidity provides robust security features, such as access control mechanisms (like require, assert, and revert), which help prevent unauthorized access and ensure that contracts operate as intended. This is crucial in the development of secure smart contracts.
3. **Strong Ecosystem and Community**: As the most widely used smart contract language, Solidity benefits from a large, active community and extensive documentation. There are numerous libraries, frameworks, and tools available to streamline development.
4. **Rich Data Structures**: Solidity supports a variety of complex data structures, including mappings, arrays, and structs, which allow developers to implement sophisticated logic and store data efficiently within contracts.
5. **Inheritance and Modular Development**: Solidity supports inheritance, enabling the reuse of code through base contracts. This allows for modular development and easy upgrades or extensions of smart contracts, promoting cleaner and more maintainable code.
6. **Interoperability with Ethereum-based Platforms**: Solidity is designed to work seamlessly with Ethereum and other EVM-compatible blockchains, ensuring that contracts can interact across different decentralized networks.
7. **Gas Optimization**: Solidity allows for fine-grained control over transaction costs (gas fees), enabling developers to optimize the execution of smart contracts for efficiency and reduced costs.
8. **Widely Supported by Tools and Frameworks**: Solidity has excellent support from development tools like Truffle, Hardhat, and Remix, which simplify testing, debugging, and deploying smart contracts on Ethereum networks.

These advantages make Solidity a powerful tool for building secure, efficient, and scalable smart contracts on blockchain platforms.

Dataset

The *Diabetes Prediction in America* dataset found on kaggle provides a rich, structured set of features that can be used to develop predictive models for diabetes diagnosis. It includes key variables that reflect a comprehensive range of health-related factors, making it a valuable resource for understanding the risk factors associated with diabetes in a U.S. population.

Dataset Overview:

- **Pregnancies**: This column represents the number of times a woman has been pregnant, which is a significant factor in determining the risk of gestational diabetes and other diabetes-related complications.
- **Glucose**: Measures the plasma glucose concentration after a 2-hour oral glucose tolerance test. Elevated glucose levels are one of the primary indicators of diabetes and prediabetes.
- **Blood Pressure**: Indicates the diastolic blood pressure in mmHg. Hypertension is a known risk factor for diabetes, and high blood pressure can worsen the effects of the disease.
- **Skin Thickness**: The thickness of the skin folds at the triceps, which is often used to estimate body fat percentage. High skin thickness can be an indicator of higher body fat, which is correlated with insulin resistance.
- **Insulin**: Represents the insulin level in the blood after a glucose load. High insulin levels are a typical sign of insulin resistance, which is a precursor to type 2 diabetes.
- **BMI (Body Mass Index)**: A critical measure of body fat based on height and weight. A high BMI is a significant risk factor for the development of diabetes, particularly type 2 diabetes.
- **Diabetes Pedigree Function**: A function that represents the genetic predisposition to diabetes. It provides a score based on family history and other genetic factors, allowing for a more nuanced risk assessment.
- **Age**: The age of the individual. Age is an important factor, as the risk of developing diabetes increases with age, particularly after 45.
- **Diabetes Diagnosis**: This is the target column, indicating whether the individual has diabetes (1) or not (0). It serves as the label for classification tasks, making the dataset suitable for binary classification models.

Why This Data is Valuable: This dataset provides a mix of both clinical measurements (e.g., glucose, insulin, BMI) and demographic factors (e.g., age), which are essential for understanding the complex interplay of genetics, lifestyle, and environment in diabetes prediction. It is particularly useful for building machine learning models for classification tasks, where the goal is to predict the likelihood of an individual having diabetes based on the provided attributes.

The combination of easily interpretable clinical features with the binary outcome label allows for efficient model training and evaluation. The dataset’s well-structured nature, with clear, consistent columns, also makes it straightforward to preprocess and use in various machine learning frameworks.

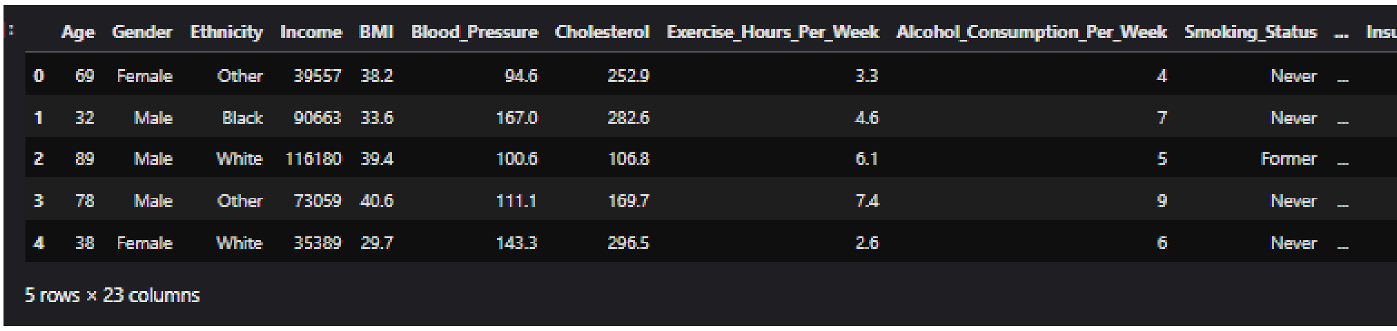
Given these attributes, the *Diabetes Prediction in America* dataset serves as an excellent foundation for predictive analytics in healthcare and can contribute to more accurate early diagnosis models for diabetes.

Data exploration

In this project, I employed a variety of data exploration techniques to analyze the diabetes dataset. These techniques helped understand the dataset's structure, identify patterns, and determine the relationships between features and the target variable. Below is a detailed explanation of the techniques used:

1. Data Summary Statistics

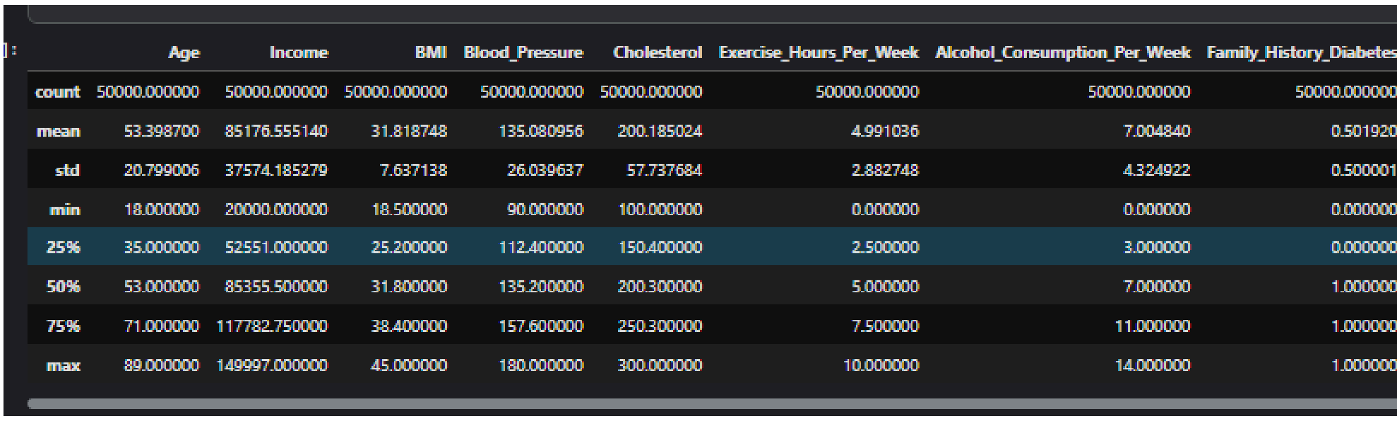
- df.head(): This function was used to display the first few rows of the dataset, providing a quick overview of the data structure and the types of features present.



	Age	Gender	Ethnicity	Income	BMI	Blood_Pressure	Cholesterol	Exercise_Hours_Per_Week	Alcohol_Consumption_Per_Week	Smoking_Status	...	Insu
0	69	Female	Other	39557	38.2	94.6	252.9	3.3	4	Never	...	
1	32	Male	Black	90663	33.6	167.0	282.6	4.6	7	Never	...	
2	89	Male	White	116180	39.4	100.6	106.8	6.1	5	Former	...	
3	78	Male	Other	73059	40.6	111.1	169.7	7.4	9	Never	...	
4	38	Female	White	35389	29.7	143.3	296.5	2.6	6	Never	...	

5 rows × 23 columns

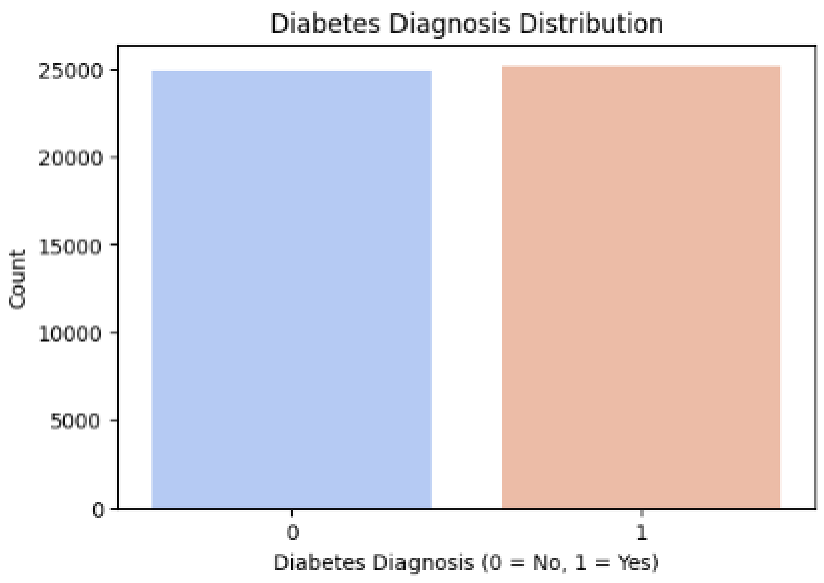
- df.describe(): This function provided summary statistics for the numerical columns, including measures such as mean, standard deviation, minimum, maximum, and quartiles. This helped us understand the distribution and range of numerical features.



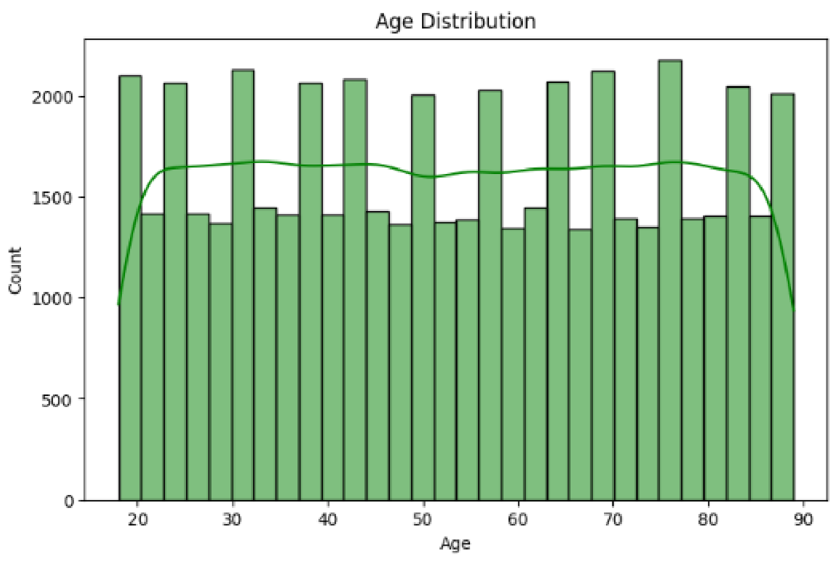
	Age	Income	BMI	Blood_Pressure	Cholesterol	Exercise_Hours_Per_Week	Alcohol_Consumption_Per_Week	Family_History_Diabetes
count	50000.000000	50000.000000	50000.000000	50000.000000	50000.000000	50000.000000	50000.000000	50000.00000
mean	53.398700	85176.555140	31.818748	135.080956	200.185024	4.991036	7.004840	0.501920
std	20.799006	37574.185279	7.637138	26.039637	57.737684	2.882748	4.324922	0.500001
min	18.000000	20000.000000	18.500000	90.000000	100.000000	0.000000	0.000000	0.00000
25%	35.000000	52551.000000	25.200000	112.400000	150.400000	2.500000	3.000000	0.00000
50%	53.000000	85355.500000	31.800000	135.200000	200.300000	5.000000	7.000000	1.00000
75%	71.000000	117782.750000	38.400000	157.600000	250.300000	7.500000	11.000000	1.00000
max	89.000000	149997.000000	45.000000	180.000000	300.000000	10.000000	14.000000	1.00000

2. Distribution Analysis

- Count Plot:
    - I used sns.countplot(x='Diabetes_Diagnosis', data=df, palette="coolwarm") to visualize the distribution of the target variable (Diabetes_Diagnosis). This plot showed the count of individuals diagnosed with diabetes (1) versus those without (0).



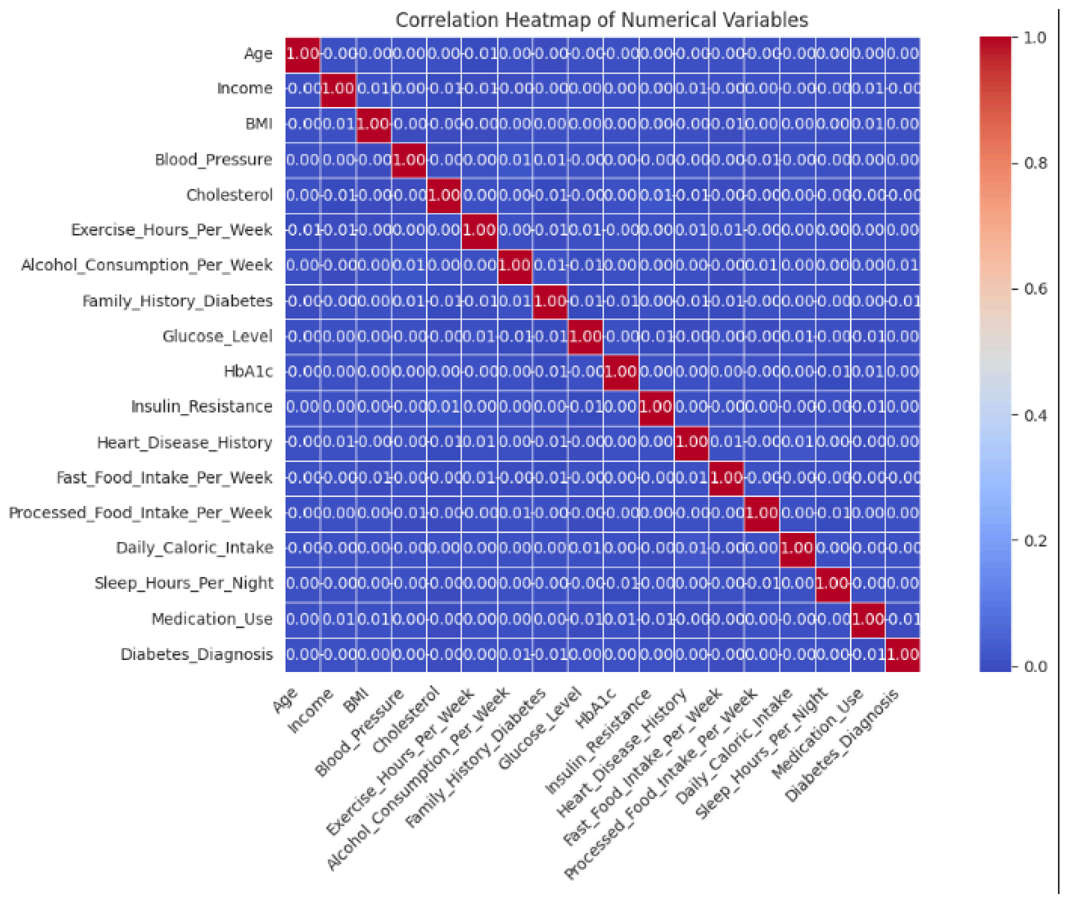
- Histogram:
    - A histogram was created using sns.histplot(df["Age"], bins=30, kde=True, color="green") to visualize the distribution of the "Age" feature. The Kernel Density Estimate (KDE) curve was added to smooth the distribution.



3. Correlation Analysis

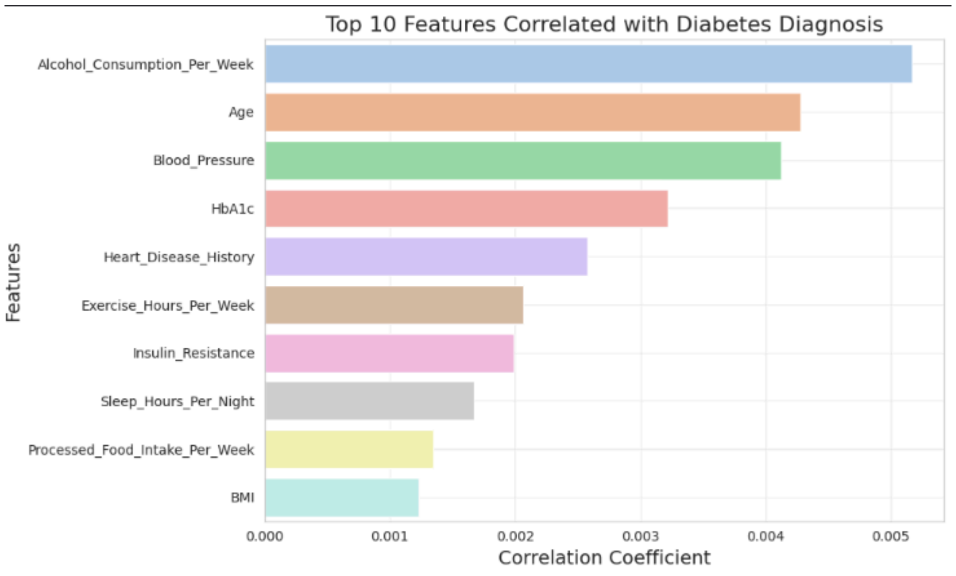
Correlation Heatmap:

- I generated a correlation heatmap using sns.heatmap. This heatmap visualized the correlation matrix of all numerical variables, helping us identify strong positive or negative correlations between features.



Top Correlated Features:

    ○ The code calculated the correlation of each numerical feature with the target variable (Diabetes_Diagnosis) and printed the top correlated features. This helped us identify which features were most strongly associated with diabetes diagnosis.



4. Categorical Feature Encoding

- Categorical features such as Gender, Ethnicity, Smoking_Status, and others were converted into numeric codes using astype('category').cat.codes. This step was essential to prepare the data for machine learning models, as most algorithms require numerical input.

Testing Local Models

1. Random Forest Classifier

Model Overview

Random Forest is an ensemble learning method that constructs multiple decision trees during training and aggregates their predictions to improve accuracy and reduce overfitting. It is particularly effective for handling high-dimensional data and capturing complex relationships between features.

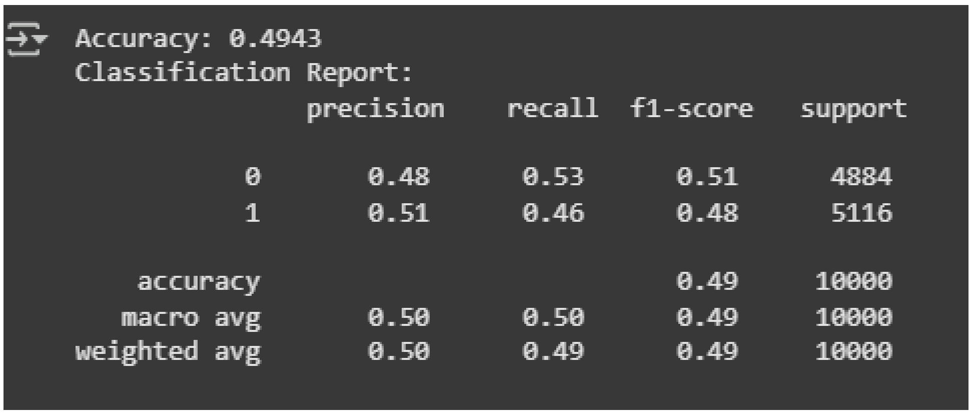
Implementation

- Algorithm: I used the RandomForestClassifier from the sklearn.ensemble module.
- Hyperparameters:
    - Number of trees (n_estimators): 100
    - Random seed (random_state): 42 (for reproducibility)

- Data Splitting: The dataset was split into training and testing sets using an 80-20 split (train_test_split).
- Training: The model was trained on the training data using rf_classifier.fit(X_train, y_train).
- Prediction: Predictions were made on the test set using rf_classifier.predict(X_test).

Evaluation

- Metrics:
    - Accuracy: The proportion of correctly classified instances.
    - Classification Report: Includes precision, recall, and F1-score for each class.



```
Accuracy: 0.4943
Classification Report:
              precision    recall  f1-score   support

           0       0.48      0.53      0.51      4884
           1       0.51      0.46      0.48      5116

    accuracy                           0.49     10000
   macro avg       0.50      0.50      0.49     10000
weighted avg       0.50      0.49      0.49     10000
```

2. Gradient Boosting Classifier

Model Overview

Gradient Boosting is another ensemble technique that builds trees sequentially, with each tree correcting the errors of the previous one. It is known for its high predictive accuracy and ability to handle complex datasets.

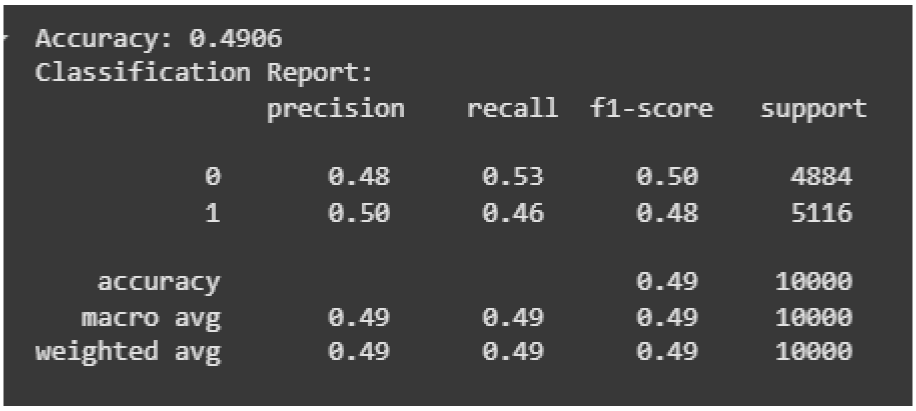
Implementation

- Algorithm: I used the GradientBoostingClassifier from the sklearn.ensemble module.
- Hyperparameters:
    - Number of trees (n_estimators): 100
    - Learning rate (learning_rate): 0.1
    - Maximum tree depth (max_depth): 3
    - Random seed (random_state): 42 (for reproducibility)
- Data Splitting: The dataset was split into training and testing sets using an 80-20 split (train_test_split).
- Training: The model was trained on the training data using gb_classifier.fit(X_train, y_train).

- Prediction: Predictions were made on the test set using gb_classifier.predict(X_test).

Evaluation

- Metrics:
  - Accuracy: The proportion of correctly classified instances.
  - Classification Report: Includes precision, recall, and F1-score for each class.



```
Accuracy: 0.4906
Classification Report:
              precision    recall  f1-score   support

           0       0.48      0.53      0.50      4884
           1       0.50      0.46      0.48      5116

    accuracy                           0.49     10000
   macro avg       0.49      0.49      0.49     10000
weighted avg       0.49      0.49      0.49     10000
```

3. Meta-Model: Stacking Ensemble

Overview of Stacking Ensemble

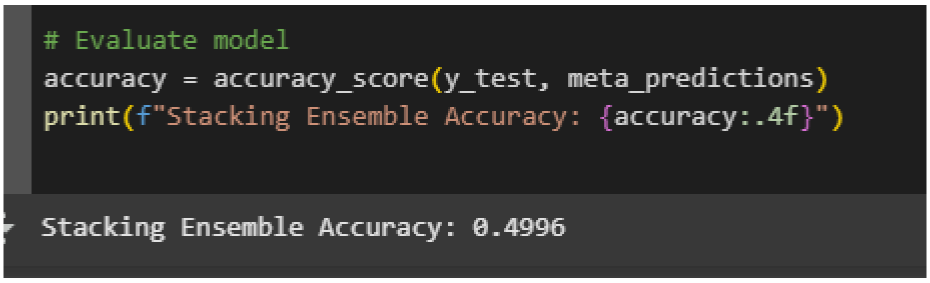
Stacking is an advanced ensemble technique that combines the predictions of multiple base models (e.g., Random Forest, Gradient Boosting) and uses a meta-model to make the final prediction. This approach leverages the strengths of individual models while mitigating their weaknesses.

Base Models

- Random Forest: Provides robust predictions by aggregating multiple decision trees.
- Gradient Boosting: Sequentially corrects errors, often achieving high accuracy.
- Support Vector Machine (SVM): Adds diversity to the ensemble by using a different learning approach.

Meta-Model

- Algorithm: Logistic Regression was used as the meta-model.
- Role: The meta-model learns to optimally combine the predictions of the base models to improve overall performance.



```
# Evaluate model
accuracy = accuracy_score(y_test, meta_predictions)
print(f"Stacking Ensemble Accuracy: {accuracy:.4f}")
```

```
Stacking Ensemble Accuracy: 0.4996
```

Implementation

- Base Model Training:
    - Each base model was trained on the training data, and their predicted probabilities were stored as meta-features.
    - For example, meta_features_train[:, i] = model.predict_proba(X_train)[:, 1] was used to store the probabilities for each base model.
- Meta-Model Training:
    - The meta-model (LogisticRegression) was trained on the meta-features generated by the base models.
    - The meta-model learned to combine the predictions of the base models to improve overall accuracy.

Evaluation

- Metrics:
    - Accuracy: The proportion of correctly classified instances.
- Results:]
    - Comparison: The stacking ensemble model typically outperforms individual base models by leveraging their strengths and compensating for their weaknesses.

Federated Learning Models: Horizontal and Vertical Federated Learning

In this project, I implemented Federated Learning (FL) to train machine learning models in a distributed manner while preserving data privacy. Federated Learning allows multiple clients (e.g., hospitals, organizations) to collaboratively train a model without sharing their raw data. I explored two types of federated learning: Horizontal Federated Learning (HFL) and Vertical Federated Learning (VFL). Below, I provide a detailed explanation of these approaches, their implementation, and the rationale behind using PyTorch instead of Keras.

Horizontal Federated Learning (HFL)

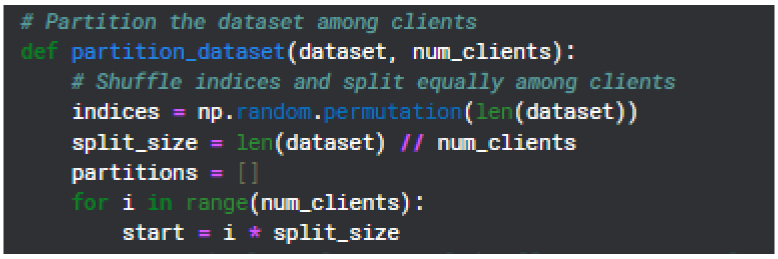
Overview of HFL

- Definition: In Horizontal Federated Learning, multiple clients share the same feature space but have different data samples. For example, different hospitals may collect the same set of features (e.g., age, BMI, glucose levels) for their patients, but the patients themselves are different.
- Use Case: HFL is ideal for scenarios where data is distributed across multiple organizations, but the **features are consistent across all datasets.**

Implementation

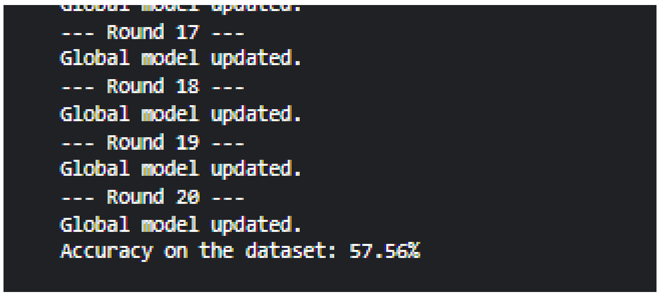
- Framework: used PyTorch to implement the HFL model.
- Model Architecture: A simple neural network (SimpleTabularNN) was designed for tabular data. The model consists of:
    - Input Layer: Number of input features.
    - Hidden Layer: 64 neurons with ReLU activation.
    - Output Layer: 2 neurons (for binary classification).
- Splitting the data among the clients

The dataset was split into non-overlapping subsets for multiple clients using the *partition_dataset* function:



```
# Partition the dataset among clients
def partition_dataset(dataset, num_clients):
    # Shuffle indices and split equally among clients
    indices = np.random.permutation(len(dataset))
    split_size = len(dataset) // num_clients
    partitions = []
    for i in range(num_clients):
        start = i * split_size
```

- Federated Training :
    - The dataset was partitioned among multiple clients (run on 3 clients).
    - Each client trained the model locally on its own data for a fixed number of epochs.
    - The global model was updated by aggregating the weights from all clients using Federated Averaging.
- Evaluation:
    - The global model was evaluated on the entire dataset to measure its accuracy.



```
--- Round 17 ---
Global model updated.
--- Round 18 ---
Global model updated.
--- Round 19 ---
Global model updated.
--- Round 20 ---
Global model updated.
Accuracy on the dataset: 57.56%
```

Vertical Federated Learning (VFL)

Overview of VFL

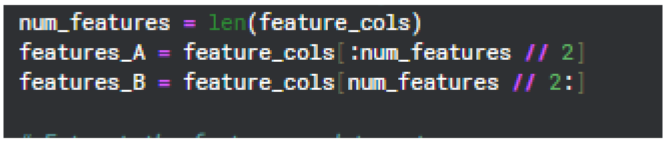
- Definition: In Vertical Federated Learning, different clients hold different features of the same data samples. For example, one organization may have medical records (e.g., age, BMI), while another organization has financial data (e.g., income, spending habits) for the same individuals.
- Use Case: VFL is ideal for scenarios where different organizations hold complementary features for the same individuals, and collaboration is required to build a comprehensive model.

Implementation

- Framework: I used PyTorch to implement the VFL model.

- Model Architecture:
  - Client Models: Each client (e.g., Party A and Party B) had its own local model (ClientModel) to process its unique features.
  - Server Model: A server model (ServerModel) was used to combine the embeddings from both clients and make the final prediction.
- Splitting the data :

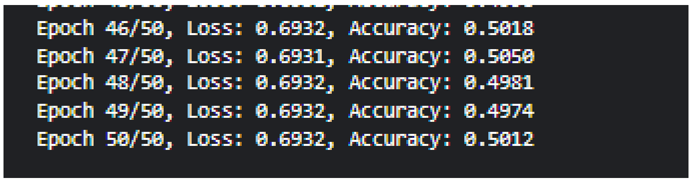
Features were split between two parties (Party A and Party B):



```
num_features = len(feature_cols)
features_A = feature_cols[:num_features // 2]
features_B = feature_cols[num_features // 2:]
```

- Federated Training:
  - Each client trained its local model on its own features.
  - The server model combined the embeddings from both clients and trained on the combined data.
  - The training process involved multiple epochs, with gradients flowing back to both client models and the server model.
- Evaluation:

- The combined model was evaluated on the entire dataset to measure its accuracy.



```
Epoch 46/50, Loss: 0.6932, Accuracy: 0.5018
Epoch 47/50, Loss: 0.6931, Accuracy: 0.5050
Epoch 48/50, Loss: 0.6932, Accuracy: 0.4981
Epoch 49/50, Loss: 0.6932, Accuracy: 0.4974
Epoch 50/50, Loss: 0.6932, Accuracy: 0.5012
```

Why PyTorch Instead of Keras?

Version Compatibility Issues

- During the development of this project, I encountered version compatibility issues with Keras and TensorFlow. These issues made it difficult to set up the environment and run the federated learning simulations smoothly.
- Specifically, the versions of Keras and TensorFlow available in the environment were not compatible with some of the libraries required for federated learning, such as Flower.

Introducing Smart Contract

1. Writing the contract on Solidity

The FLConsent smart contract is designed to handle participant consent in a federated learning system. It allows participants to give their consent, records it on the blockchain, and provides a way to check whether a specific participant has signed. Below is a breakdown of the contract's components and functionality.

```
// SPDX-License-Identifier: MIT
pragma solidity ^0.8.19;

contract FLConsent {
   mapping(address => bool) public consentGiven;

   event ConsentSigned(address participant);

   function signConsent() external {
      require(!consentGiven[msg.sender], "Already signed");
      consentGiven[msg.sender] = true;
```

```
        emit ConsentSigned(msg.sender);
    }

    function hasSignedConsent(address participant) external view returns (bool) {
        return consentGiven[participant];
    }
}
```

Contract Breakdown

License and Solidity Version

```
// SPDX-License-Identifier: MIT
pragma solidity ^0.8.19;
```

- **// SPDX-License-Identifier: MIT**: Specifies the contract's open-source license (MIT), allowing others to freely use and modify it.
- **pragma solidity ^0.8.19;**: Ensures the contract runs on Solidity version 0.8.19 or later, benefiting from the latest security updates and features.

State Variables

```
mapping(address => bool) public consentGiven;
```

- **mapping(address => bool) public consentGiven;**
  - This mapping stores whether an address (participant) has given consent.
  - The key is the Ethereum **address** of the participant.
  - The value is a **boolean (true or false)**, indicating whether consent has been given.
  - The public keyword allows anyone to check if a specific participant has signed.

Event Declaration

```
event ConsentSigned(address participant);
```

- **Events in Solidity**
  - Events allow logging of important contract activities on the blockchain.
  - The ConsentSigned event records when a participant signs the consent.
  - The address participant parameter stores the address of the participant who signed.

Consent Signing Function

```
function signConsent() external {
require(!consentGiven[msg.sender], "Already signed");
consentGiven[msg.sender] = true;
emit ConsentSigned(msg.sender);
}
```

- **signConsent()**
  - Allows a participant to sign their consent.
  - msg.sender: Represents the address calling the function.
  - **require(!consentGiven[msg.sender], "Already signed");**
    - Ensures that a participant can only sign once.
    - If they have already signed, the transaction fails with an error message: "Already signed".
  - **consentGiven[msg.sender] = true;**
    - Stores the participant's consent by setting their address to true in the mapping.
  - **emit ConsentSigned(msg.sender);**
    - Emits an event to record the signing action on the blockchain.

Checking Consent Status

```
function hasSignedConsent(address participant) external view returns (bool) {
return consentGiven[participant];
}
```

- **hasSignedConsent(address participant)**
  - Allows external users (e.g., other smart contracts or front-end applications) to check if a participant has signed.
  - **external view**:
    - external: Can be called from outside the contract but not internally.

        - view: Indicates it does not modify the blockchain state.
    - **Returns true if the participant has signed and false otherwise.**

2.Summary of Functionality

Function Name	Description
signConsent()	Allows participants to give their consent. Prevents duplicate signings.
hasSignedConsent(address participant)	Checks if a given participant has signed.

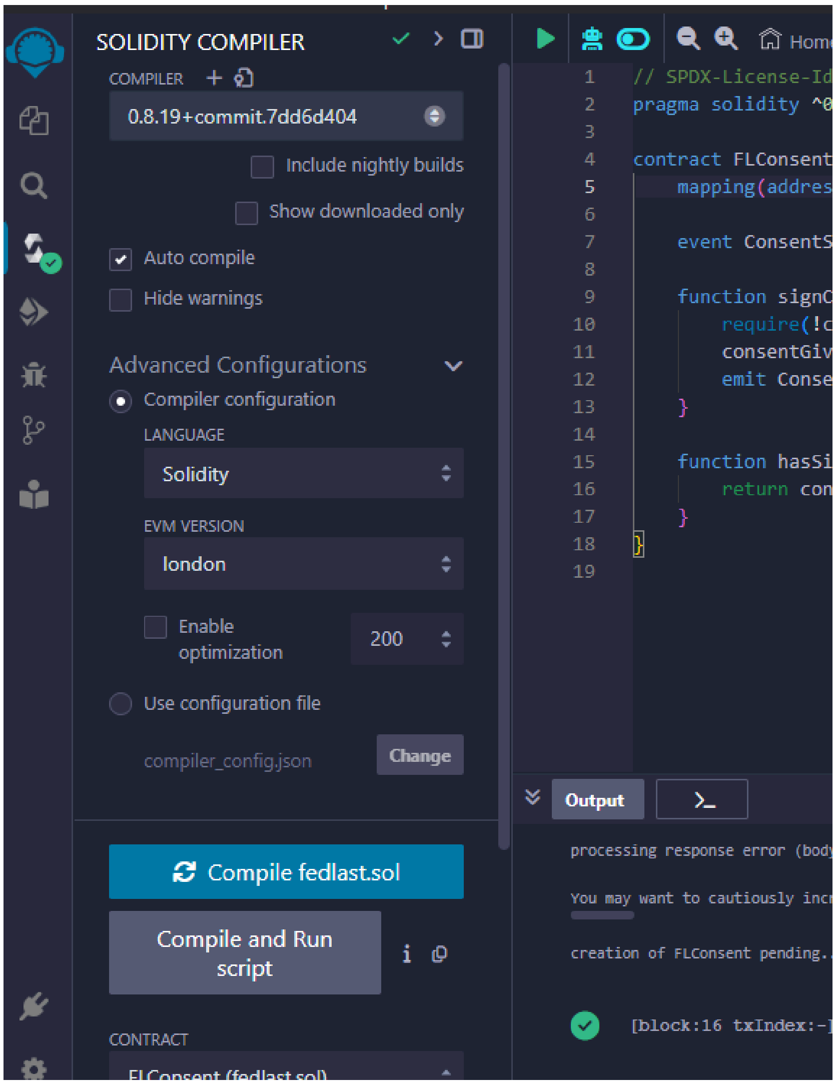
Key Features

Ensures one-time consent per participant
Records signing events on the blockchain
Provides transparency and easy verification
Efficient and gas-optimized design

2. Deploy and running

2.1.Compiling the contract

On Solidity
I began by compiling the contract using the Solidity compiler. To ensure compatibility, I selected Solidity version 0.8.19 in Remix and enabled **Auto Compile**. The compilation process successfully generated the **bytecode** and **ABI**, which are essential for deployment. No errors were encountered, and the contract was ready for deployment.



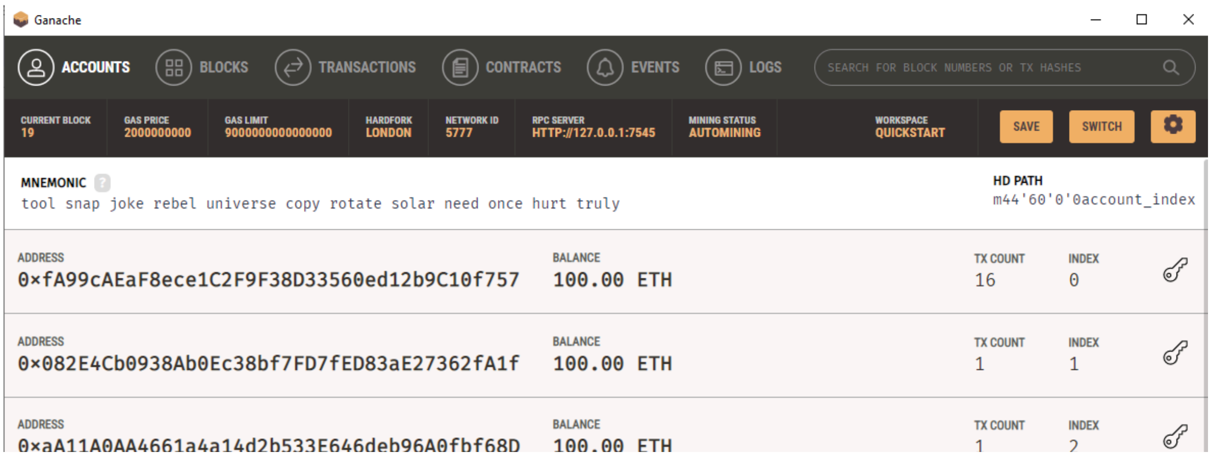
COMPILER
0.8.19+commit.7dd6d404
Include nightly builds
Show downloaded only
[✓] Auto compile
Hide warnings
Advanced Configurations
[● Compiler configuration]
LANGUAGE
Solidity
EVM VERSION
london
Enable optimization 200
Use configuration file
compiler_config.json
CONTRACT
FLConsent (fedlast.sol)

```
// SPDX-License-Id
pragma solidity ^0

contract FLConsent
    mapping(addres

    event ConsentS

    function signC
        require(!c
        consentGiv
        emit Conse
    }

    function hasSi
        return con
    }
}

```

```
processing response error (body

You may want to cautiously incr

creation of FLConsent pending...

[block:16 txIndex:-]
```

On Ganache

To test the contract in a local blockchain environment, I set up **Ganache** and configured it as the blockchain network. I ensured that the Solidity compiler (solc) matched the version required for deployment. Ganache provided a list of test accounts with preloaded ETH, which was useful for deploying and interacting with the contract.



CURRENT BLOCK	GAS PRICE	GAS LIMIT	HARDFORK	NETWORK ID	RPC SERVER	MINING STATUS	WORKSPACE
19	20000000000	9000000000000000	LONDON	5777	HTTP://127.0.0.1:7545	AUTOMINING	QUICKSTART

MNEMONIC
tool snap joke rebel universe copy rotate solar need once hurt truly

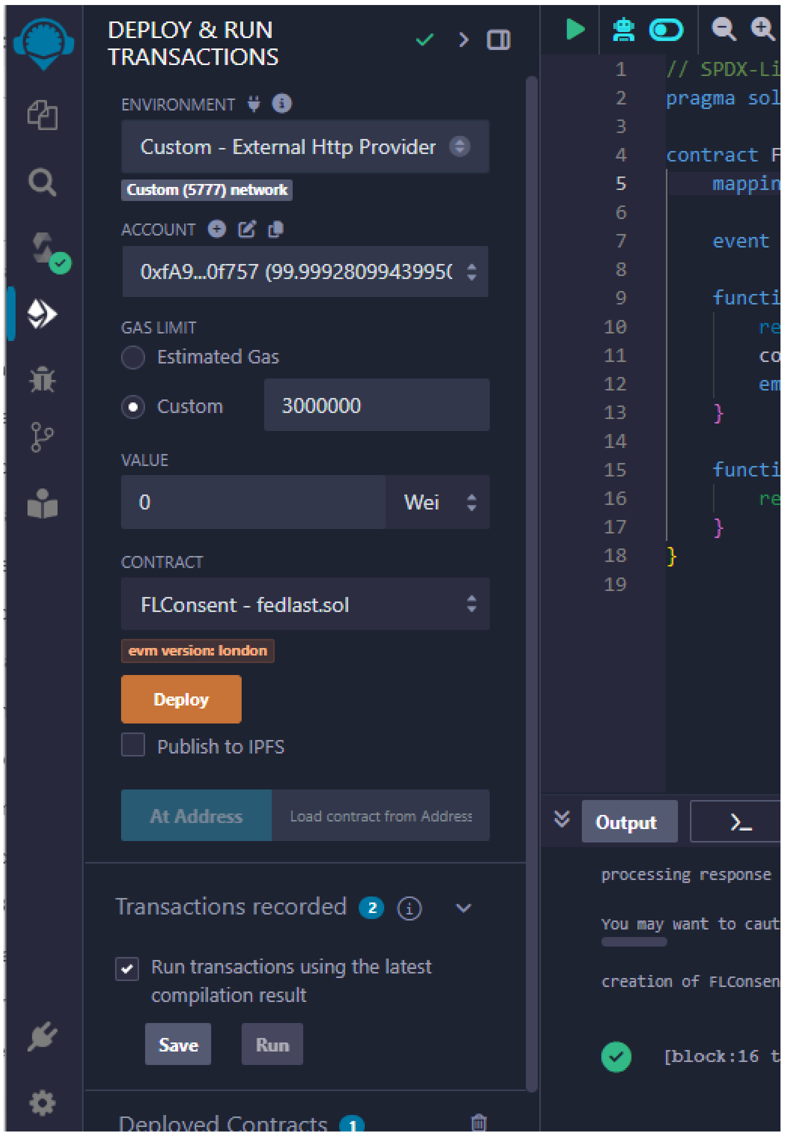
HD PATH
m44'60'0'0account_index

ADDRESS	BALANCE	TX COUNT	INDEX
0xfA99cAEaF8ece1C2F9F38D33560ed12b9C10f757	100.00 ETH	16	0
0x082E4Cb0938Ab0Ec38bf7FD7fED83aE27362fA1f	100.00 ETH	1	1
0xaA11A0AA4661a4a14d2b533E646deb96A0fbf68D	100.00 ETH	1	2

2.2.deploying contract

Once the contract was compiled, I proceeded with deployment.

- I used **Remix IDE** to deploy the contract, selecting **External http provider** and connecting it to my local **Ganache blockchain**.
- After ensuring that the deployment account had sufficient funds, I initiated the deployment transaction.
- The transaction was successfully mined, and the contract was deployed with a unique contract address.
- I verified the deployment by checking the transaction receipt and ensuring that the contract appeared on Ganache's blockchain explorer.



DEPLOY & RUN TRANSACTIONS

ENVIRONMENT: Custom - External Http Provider

Custom (5777) network

ACCOUNT: 0xfA9...0f757 (99.9992809943995(

GAS LIMIT: ( ) Estimated Gas [✓ Custom] 3000000

VALUE: 0 Wei

CONTRACT: FLConsent - fedlast.sol

evm version: london

Deploy

☐ Publish to IPFS

At Address | Load contract from Address

Transactions recorded 2

☑ Run transactions using the latest compilation result

Save | Run

Deployed Contracts 1

```
// SPDX-Li
pragma sol

contract F
    mappin

    event

    functi
        re
        co
        em
    }

    functi
        re
    }
}
```

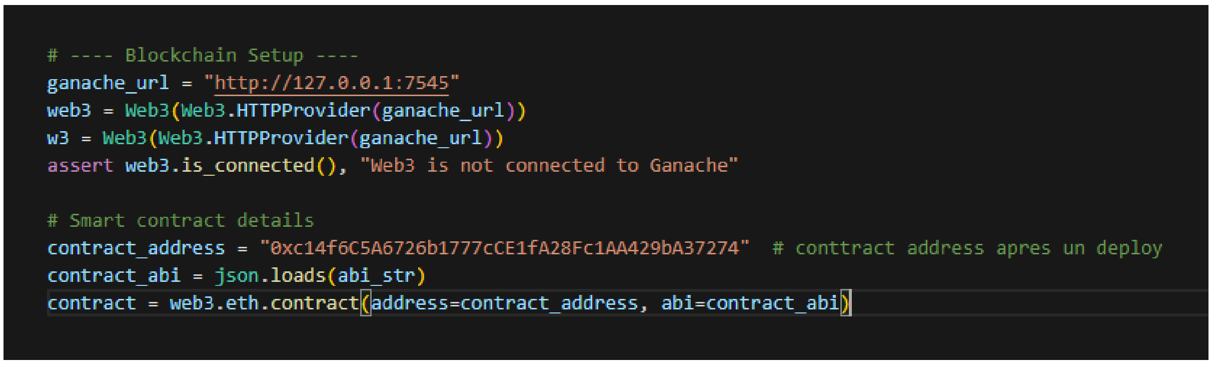
Output

```
processing response

You may want to caut

creation of FLConsen

[block:16 t
```

2.3. Updating the code with the contract address and ABI

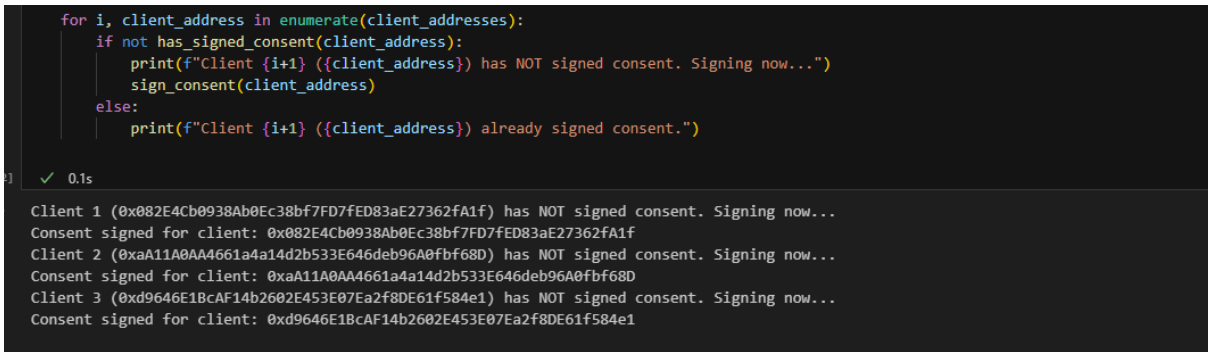
After deployment, I updated my project file to include the newly generated **contract address** and **ABI**.

- I copied the **contract address** and updated my code to interact with the deployed contract.
- The **ABI** json file was updated as well, allowing me to call contract functions .



```python
# ---- Blockchain Setup ----
ganache_url = "http://127.0.0.1:7545"
web3 = Web3(Web3.HTTPProvider(ganache_url))
w3 = Web3(Web3.HTTPProvider(ganache_url))
assert web3.is_connected(), "Web3 is not connected to Ganache"

# Smart contract details
contract_address = "0xc14f6C5A6726b1777cCE1fA28Fc1AA429bA37274"  # conttract address apres un deploy
contract_abi = json.loads(abi_str)
contract = web3.eth.contract(address=contract_address, abi=contract_abi)
```

- I tested the interaction by executing the signConsent function fr, confirming that the contract stored consent records correctly.



```python
for i, client_address in enumerate(client_addresses):
    if not has_signed_consent(client_address):
        print(f"Client {i+1} ({client_address}) has NOT signed consent. Signing now...")
        sign_consent(client_address)
    else:
        print(f"Client {i+1} ({client_address}) already signed consent.")
```

✓ 0.1s

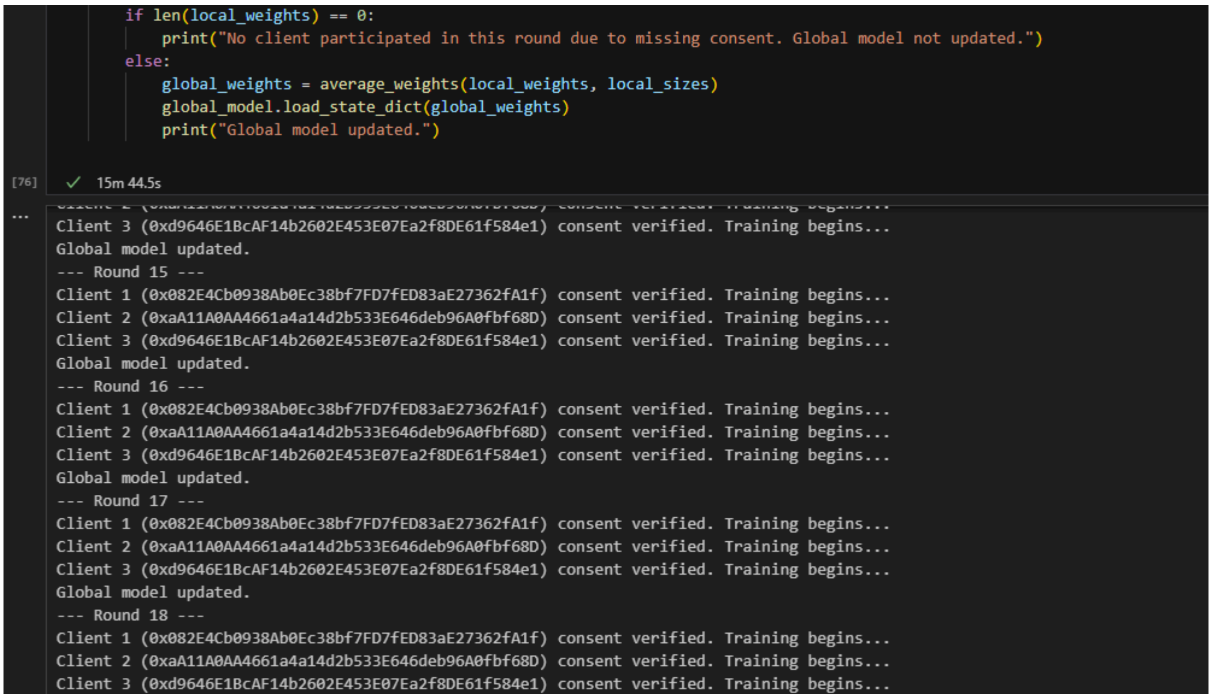
```
Client 1 (0x082E4Cb0938Ab0Ec38bf7FD7fED83aE27362fA1f) has NOT signed consent. Signing now...
Consent signed for client: 0x082E4Cb0938Ab0Ec38bf7FD7fED83aE27362fA1f
Client 2 (0xaA11A0AA4661a4a14d2b533E646deb96A0fbf68D) has NOT signed consent. Signing now...
Consent signed for client: 0xaA11A0AA4661a4a14d2b533E646deb96A0fbf68D
Client 3 (0xd9646E1BcAF14b2602E453E07Ea2f8DE61f584e1) has NOT signed consent. Signing now...
Consent signed for client: 0xd9646E1BcAF14b2602E453E07Ea2f8DE61f584e1
```

With the contract successfully compiled, deployed, and integrated, it was ready for real-world testing.

Running the federated model

After assuring the correct deployment of the contract , running the federated learning model and seeing the application of the smart contract in real time

On VSC

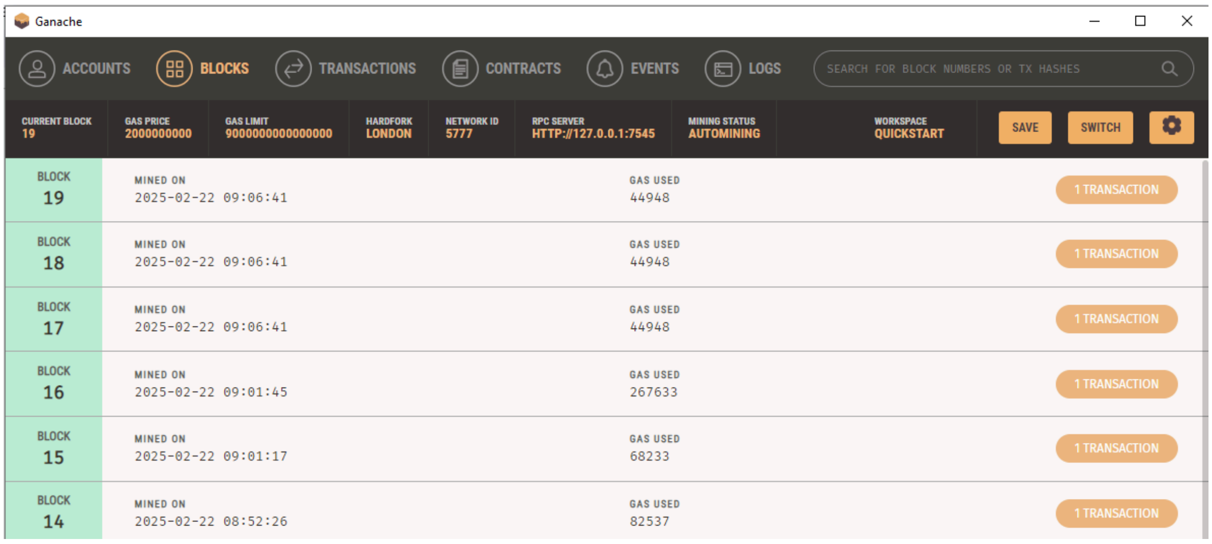


```
        if len(local_weights) == 0:
            print("No client participated in this round due to missing consent. Global model not updated.")
        else:
            global_weights = average_weights(local_weights, local_sizes)
            global_model.load_state_dict(global_weights)
            print("Global model updated.")
```

[76] ✓ 15m 44.5s

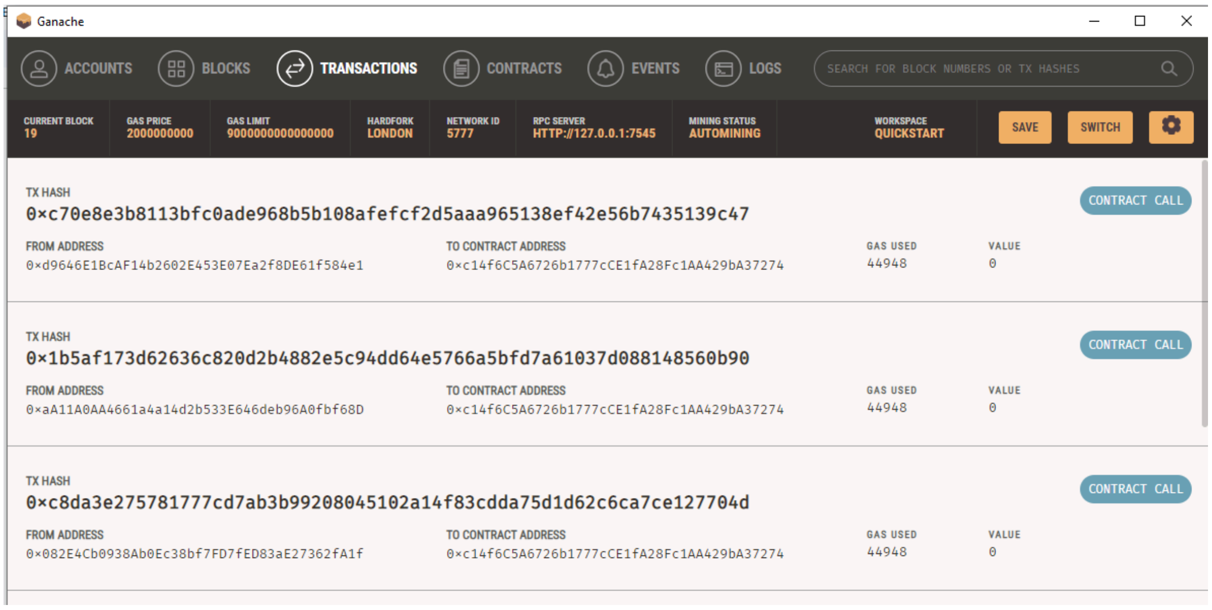
```
Client 2 (0xaA11A0AA4661a4a14d2b533E646deb96A0fbf68D) consent verified. Training begins...
Client 3 (0xd9646E1BcAF14b2602E453E07Ea2f8DE61f584e1) consent verified. Training begins...
Global model updated.
--- Round 15 ---
Client 1 (0x082E4Cb0938Ab0Ec38bf7FD7fED83aE27362fA1f) consent verified. Training begins...
Client 2 (0xaA11A0AA4661a4a14d2b533E646deb96A0fbf68D) consent verified. Training begins...
Client 3 (0xd9646E1BcAF14b2602E453E07Ea2f8DE61f584e1) consent verified. Training begins...
Global model updated.
--- Round 16 ---
Client 1 (0x082E4Cb0938Ab0Ec38bf7FD7fED83aE27362fA1f) consent verified. Training begins...
Client 2 (0xaA11A0AA4661a4a14d2b533E646deb96A0fbf68D) consent verified. Training begins...
Client 3 (0xd9646E1BcAF14b2602E453E07Ea2f8DE61f584e1) consent verified. Training begins...
Global model updated.
--- Round 17 ---
Client 1 (0x082E4Cb0938Ab0Ec38bf7FD7fED83aE27362fA1f) consent verified. Training begins...
Client 2 (0xaA11A0AA4661a4a14d2b533E646deb96A0fbf68D) consent verified. Training begins...
Client 3 (0xd9646E1BcAF14b2602E453E07Ea2f8DE61f584e1) consent verified. Training begins...
Global model updated.
--- Round 18 ---
Client 1 (0x082E4Cb0938Ab0Ec38bf7FD7fED83aE27362fA1f) consent verified. Training begins...
Client 2 (0xaA11A0AA4661a4a14d2b533E646deb96A0fbf68D) consent verified. Training begins...
Client 3 (0xd9646E1BcAF14b2602E453E07Ea2f8DE61f584e1) consent verified. Training begins...
```

On Ganache



CURRENT BLOCK	GAS PRICE	GAS LIMIT	HARDFORK	NETWORK ID	RPC SERVER	MINING STATUS	WORKSPACE
19	2000000000	9000000000000000	LONDON	5777	HTTP://127.0.0.1:7545	AUTOMINING	QUICKSTART

BLOCK	MINED ON	GAS USED	
19	2025-02-22 09:06:41	44948	1 TRANSACTION
18	2025-02-22 09:06:41	44948	1 TRANSACTION
17	2025-02-22 09:06:41	44948	1 TRANSACTION
16	2025-02-22 09:01:45	267633	1 TRANSACTION
15	2025-02-22 09:01:17	68233	1 TRANSACTION
14	2025-02-22 08:52:26	82537	1 TRANSACTION



```
CURRENT BLOCK  GAS PRICE   GAS LIMIT         HARDFORK  NETWORK ID  RPC SERVER             MINING STATUS  WORKSPACE
19             2000000000  9000000000000000  LONDON    5777        HTTP://127.0.0.1:7545  AUTOMINING     QUICKSTART

TX HASH                                                                                   CONTRACT CALL
0×c70e8e3b8113bfc0ade968b5b108afefcf2d5aaa965138ef42e56b7435139c47
FROM ADDRESS                                  TO CONTRACT ADDRESS                         GAS USED  VALUE
0×d9646E1BcAF14b2602E453E07Ea2f8DE61f584e1    0×c14f6C5A6726b1777cCE1fA28Fc1AA429bA37274  44948     0

TX HASH                                                                                   CONTRACT CALL
0×1b5af173d62636c820d2b4882e5c94dd64e5766a5bfd7a61037d088148560b90
FROM ADDRESS                                  TO CONTRACT ADDRESS                         GAS USED  VALUE
0×aA11A0AA4661a4a14d2b533E646deb96A0fbf68D    0×c14f6C5A6726b1777cCE1fA28Fc1AA429bA37274  44948     0

TX HASH                                                                                   CONTRACT CALL
0×c8da3e275781777cd7ab3b99208045102a14f83cdda75d1d62c6ca7ce127704d
FROM ADDRESS                                  TO CONTRACT ADDRESS                         GAS USED  VALUE
0×082E4Cb0938Ab0Ec38bf7FD7fED83aE27362fA1f    0×c14f6C5A6726b1777cCE1fA28Fc1AA429bA37274  44948     0
```

Conclusion and Discussion

This project successfully demonstrated the integration of **federated learning** for **diabetes prediction** with **blockchain-based security**, addressing critical challenges in medical data privacy and secure model training. Federated learning was particularly well-suited for this use case due to the **sensitive nature of healthcare data**, allowing multiple institutions to collaboratively train a predictive model without the need for centralized data sharing. This approach ensures compliance with **data protection regulations** while preserving patient confidentiality.

However, federated learning introduces security concerns, particularly regarding the **integrity of communications** between participating entities. To mitigate these risks, a **Solidity-based smart contract** was implemented, providing a **transparent, decentralized, and tamper-proof mechanism** for managing participant consent. By leveraging blockchain, this solution ensures that only authorized parties contribute to the training process, prevents data manipulation, and enhances trust through immutable transaction records.

The integration of federated learning with blockchain security highlights a **scalable and privacy-preserving approach** to AI-driven medical research. This methodology not only enables secure collaboration across multiple institutions but also sets a foundation for future advancements in **secure, decentralized AI applications**. Future work may explore **smart contract-based incentive mechanisms**, **automated model auditing**, and **expanding the framework to other medical conditions**, further enhancing the reliability and applicability of this approach in healthcare and beyond.

References

[1]	GeeksforGeeks. (2024, May 27). *Types of Federated Learning in Machine Learning*. GeeksforGeeks. https://www.geeksforgeeks.org/types-of-federated-learning-in-machine-learning/
[2]	Kelvin. (2020, October 18). *Introduction to Federated Learning and Challenges*. Towards Data Science. https://medium.com/towards-data-science/introduction-to-federated-learning-and-challenges-ea7e02f260ca
[3]	McMahan, B., & Ramage, D. (2017, April 6). *Federated learning: Collaborative machine learning without centralized training data*. Google Research Blog. https://research.google/blog/federated-learning-collaborative-machine-learning-without-centralized-training-data/
[4]	Flower. (n.d.). *Vertical Federated Learning with Flower*. Flower. https://flower.ai/docs/examples/vertical-fl.html
[5]	IBM Research. (n.d.). *What is federated learning?* IBM Research. https://research.ibm.com/blog/what-is-federated-learning
[6]	Solidity. (n.d.). *Solidity documentation*. Solidity. https://docs.soliditylang.org/en/latest/
[7]	