VIETNAM NATIONAL UNIVERSITY HO CHI MINH CITY
HO CHI MINH CITY UNIVERSITY OF TECHNOLOGY
FALCUTY OF COMPUTER SCIENCE AND ENGINEERING

DISTRIBUTED SYSTEM ASSIGNMENT
FINAL REPORT

Topic

Federated learning in heart disease detection

Instructor: ThS. Diệp Thanh Đăng

Student group:	Trần Nguyễn Thái Bình	- 2110051
	Trương Hoàng Nguyên Vũ	- 2112673
	Nguyễn Sinh Thành	- 2112302

Ho Chi Minh city, November 2024

Contents

1 Introduction **2**

2 Background knowledge **3**
2.1 Federated learning concept . . . 3
2.2 Federated learning process . . . 4
2.3 Some common model fusion functions in federated learning . . . 5
2.4 Federated learning system requirements . . . 5

3 Methodology **6**
3.1 Overview . . . 6
3.2 Architecture . . . 7
3.3 Dataset . . . 7

4 Implementation **10**
4.1 Implementation requirements . . . 10
4.2 Our federated leanring system's classes . . . 10
4.2.1 Server class . . . 10
4.2.2 Client class . . . 14
4.2.3 ListeningThread class . . . 17
4.2.4 TrainingThread class . . . 18
4.3 Our federated learning system's global model . . . 20

5 Result and discussion **22**
5.1 Result of averaging model fusion function . . . 22
5.2 Result of loss model fusion function . . . 23
5.3 Result of gradient descent model fusion function . . . 24
5.4 Discussion . . . 25

6 Conclusion **27**

References **28**

1 Introduction

Medical problems are always the main concerns in daily life. It involves in health, living quality and lifespan of a person; thus, it is extremely important. Today, there are many types of medical issues that affect human. They are caused mainly by changing in living environment, bad factors from industrialization and modernization or improvement in life standard leading to unhealthy lifestyle, . . .

Nowadays, one of the most dangerous and common health problems is heart operation malfunction. Heart diseases are a group of disorders of the heart and blood vessels which causes thousands of deaths every year. Our objective is to use the power of modern machine learning approaches to help detect or predict the existence of heart diseases of a patient by utilizing some of his provided health features.

However, there is an enormous difficulty here which is about the data are used to train such those machine learning models. Normally, the patients' data are confidential and kept privately among medical facilities. Therefore, if we want to use those data for training, that will cause the lack of variety as well as the balance of data in training processes because each model can only be trained on data from one homogeneous distribution. In other words, our trained models does not have enough generality to be applied to other hospitals whose data are in very strange distributions. That is the reason for appearance of federated learning which is considered as the most effective tool to overcome those obstacles.

Federated learning (FL) enables locally collaborative model training across multiple medical facilities and combines their works into one global model without compromising patient privacy. To be clearer, each institution performs local training on its private data and sends only the model updates to the central model aggregator. This ensures that sensitive patient information remains within the institution while still contributing to a global model for wide usage.

Therefore, we propose a method to take advantage of all the outstanding features of federated learning to apply to our heart disease detection problem. Our combining approach provides a more reliable, general and robust model which can be used across multiple hospitals without worrying about bias or difference among data distributions.

In this topic, we just consider the case of heart disease detection and how to apply federated learning on it. However, our proposed method can be used in varied fields of disease detection which gives the same effectiveness as our case.

This report is constructed as six sections which are Introduction, Background knowledge, Methodology, Implementation, Result and discussion and Conclusion.

2 Background knowledge

In this section we will slightly go through some basic concepts of federated learning which are *definition of federated learning*, *federated learning process*, *some common model fusion functions in federated learning* and *federated learning system requirements.*

2.1 Federated learning concept

Federated learning (FL) is a machine learning technique developed to address the challenges of training models with high-quality data while maintaining privacy. Traditional machine learning approaches often require data to be centrally collected and managed, which can be difficult and ineffective, especially when privacy considerations prevent data from being centralized. Federated learning enables the training of models on data distributed across various locations without needing to centralize it, thereby mitigating privacy risks.

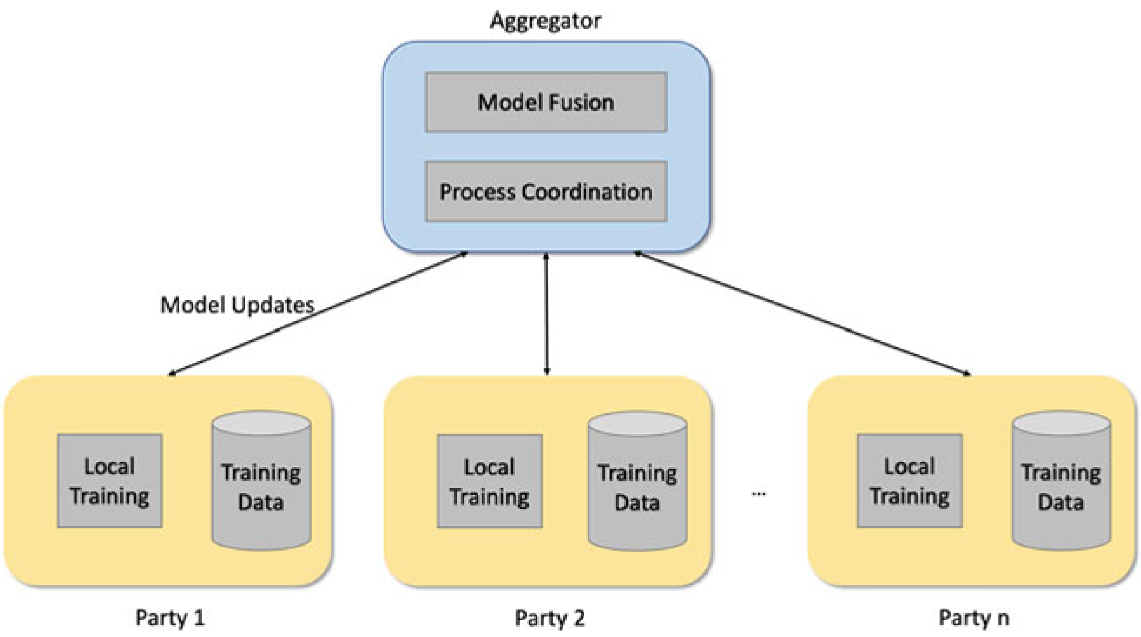


Figure 1: *Abstract overview of a federated learning system (Source: [Ludwig et al., 2020]).*

While the federated learning collaboration can be conducted in different ways, its most common form is outlined in Figure 1. An *aggregator*, also known as a server or coordinator, facilitates collaboration among different parties. Each *party* (or in our problem is *hospital*) conducts a local training process on its private data and sends the resulting model parameters, such as neural network weights, to the aggregator. The aggregator then performs *model fusion*, merging these updates into the global model.

Federated learning is distinct from distributed learning on clusters, as it does not centrally control data distribution and quantity, and all training data may remain private. This approach allows organizations to leverage valuable data for model training while respecting privacy and data ownership concerns, making it a crucial innovation in situations where data centralization is not feasible.

2.2 Federated learning process

The federated learning process begins with an aggregator using a function Q to generate a query q_t based on the previous model $\mathcal{M}_{t-1}$. This query is sent to participating parties, requesting information about their local models. Each party uses the query and their local dataset to produce a *model update* $r_{k,t}$. These updates are sent back to the aggregator, which collects all updates and applies a *model fusion function* $\mathcal{F}$ to generate the new global model $\mathcal{M}_t$. The federated learning process is described at abstract level in Figure 2.

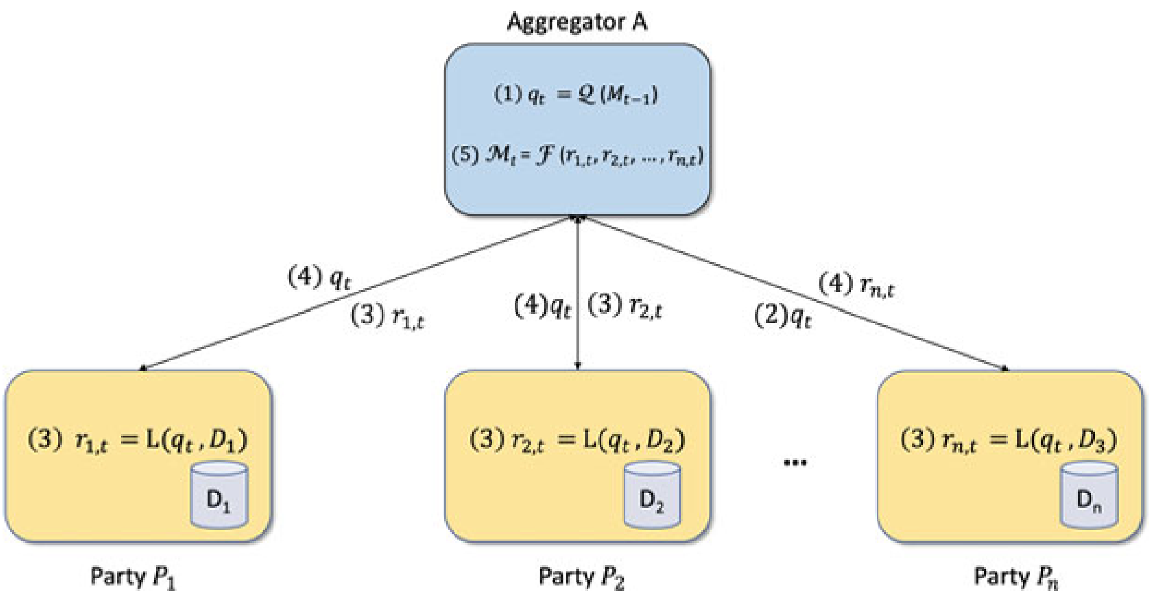

Aggregator A
(1) $q_t = \mathcal{Q}(\mathcal{M}_{t-1})$
(5) $\mathcal{M}_t = \mathcal{F}(r_{1,t}, r_{2,t}, \dots, r_{n,t})$
(4) q_t
(3) $r_{1,t}$
(4)q_t
(3) $r_{2,t}$
(4) $r_{n,t}$
(2)q_t
(3) $r_{1,t} = L(q_t, D_1)$
D_1
(3) $r_{2,t} = L(q_t, D_2)$
D_2
...
(3) $r_{n,t} = L(q_t, D_3)$
D_n
Party P_1
Party P_2
Party P_n

Figure 2: *Illustration of federated learning process in which the steps* (1), (2), (3), (4), (5) *are conducted in order. In this figure,* q_t *is the query and* $\mathcal{M}_t$ *is the global model at round* t. *For local dataset* D_k *in one party,* $r_{t,k}$ *is the model update which is sent back to aggregator after finishing local training.* $\mathcal{F}$ *is the model fusion function whose inputs are model updates* $r_{t,k}$. $\mathcal{Q}$ *is the query function at aggregator whose input is the global model* $\mathcal{M}_t$. $\mathcal{L}$ *is the training function whose input is query* q_t *and local dataset* D_k *as well as output is model update* $r_{t,k}$ *(Source: [Ludwig et al., 2020]).*

All the steps in federated learning process with their detailed activities are described below

(1) *Initialization:* The process starts at the aggregator. To train the model, the aggregator uses a function Q that takes as input the model of the previous round $\mathcal{M}_{t-1}$ and generates a query q_t for the current round. Initially, $\mathcal{M}_0$ may be empty or randomly seeded.

(2) *Query distribution:* The query q_t is sent to the parties, requesting information about their local models or datasets.

(3) *Local training:* Upon receiving q_t, each party performs local training using function $\mathcal{L}$ with the query q_t and local dataset D_k, producing a model update $r_{k,t}$.

(4) *Model update collection:* Once local training is completed, $r_{k,t}$ is sent back to the aggregator, which collects all updates $r_{k,t}$ from the parties.

(5) *Model fusion:* The aggregator processes the received updates using a fusion function $\mathcal{F}$ to form the new global model $\mathcal{M}_t$.

This process can be executed over multiple rounds and continues until a termination criterion is met.

2.3 Some common model fusion functions in federated learning

The mechanisms of model fusion functions depend on the model update value $r_{t,k}$ which is illustrated in Figure 2. There are three common model fusion functions corresponding to three different types of model update values which are

(1) *Gradient descent model fusion function:* The inputs $r_{t,k}$ of the function $\mathcal{F}$ are local weight gradient values of local models. The model fusion function $\mathcal{F}$ will use those inputs to perform gradient descent process on global model and form a new one.

(2) *Loss model fusion function:* The inputs $r_{t,k}$ of the function $\mathcal{F}$ are local loss values of local models. The model fusion function $\mathcal{F}$ will use those inputs to perform gradient descent process on global model and form a new one.

(3) *Averaging model fusion function:* The inputs $r_{t,k}$ of the function $\mathcal{F}$ are local weights of local models. The model fusion function $\mathcal{F}$ will use those inputs to perform weights averaging process on global model and form a new one.

In our system, we will implement and conduct experiments on these three common types of model fusion functions respectively to evaluate and discuss.

2.4 Federated learning system requirements

Federated learning system operates on a distributed system involving parties and an aggregator. This system must meet the computational, memory, and networking requirements of both parties and the aggregator, as well as their communication.

Local model training occurs where the data resides, highlighting the importance of the available resources at each party's location. Aggregators, typically housed in data centers, must be equipped to scale efficiently, especially when interacting with numerous parties.

Network connectivity and bandwidth requirements can vary based on factors such as model size, update content, frequency, and cryptographic protocols. Consequently, the system requirements for federated learning differ significantly from those of centralized learning, demanding careful consideration of these aspects.

3 Methodology

3.1 Overview

In this assignment we will construct a small-scale federated learning system with an aggregator and some parties. All those components will be undertaken by personal computers to perform a simulation. We will implement and conduct experiments on three common types of model fusion functions respectively which are mentioned above to evaluation and discussion.

About data, we will use a given public dataset on popular dataset providing platforms such as Kaggle or PapersWithCode. After data preprocessing, We will divide it into multiple batches. There are different batches of data in some personal computers which have roles as private patients' data in hospitals. Another batch of data is used for testing in the aggregator which is undertaken by a different personal computer.

About evaluation, because our detection problem only deals with two values which are 1 (if yes) and 0 (if no), we will use the *binary cross-entropy* as our metric. This metric is good at representing the performance of our models when outputs are in binary format.

About communication, we will use TCP/IP with Python socket programming. All the messages among all the system components are transferred through network. The federated learning process will be executed multiple rounds in a regular routine which is decided by *notifying messages*.

About implementation, we will use Python as our main programming language for this assignment.

The configuration of our federated system is given by the table below

$\mathcal{Q}$	The query function returns its input which is $Q(\mathcal{M}_{t-1}) = \mathcal{M}_{t-1}$ in which $\mathcal{M}_{t-1}$ is the global model at round $t-1$
q_t	The query is the global model at round $t-1$
$r_{t,k}$	The model update can be local gradient values, local loss value or local weight values*
$\mathcal{L}$	This function has outputs which can be local gradient values, local loss value or local weight values*
$\mathcal{F}$	This function has inputs which can be gradient values, loss value or weight values*

Table 1: *Our federated learning system configuration.*

*Note that in this federated learning system, different model fusion functions are used to compare with each other, evaluate and give results along with discussions.

3.2 Architecture

Our federated learning system is constructed as client-server architecture. In this architecture, server has the role as aggregator to perform model fusion and clients which are hospitals have the roles as parties to conduct local model trainings. It is illustrated in Figure 3.

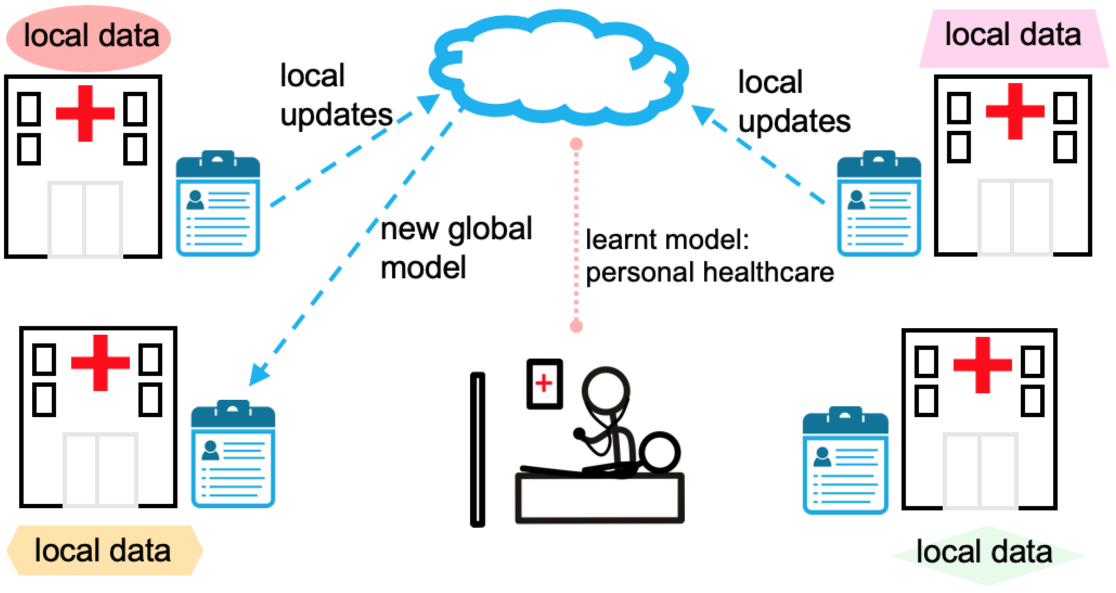


Figure 3: *Illustration of our architecture in which the cloud figure is considered as server and hospital figures are considered as clients. In our simulation, the server will be undertaken by one personal computer along with other personal computers which have the roles as clients in the system (Source: [ML-CMU, 2019]).*

In our simulation, there is one personal computer undertakes the server role in which it will distribute the global model to two or three other personal computers which have the roles as clients.

3.3 Dataset

In this assignment, we will use Cardiovascular Disease dataset which contains 70000 records in *csv* format on Kaggle. A few first records of the dataset is showed at Figure 4. This dataset has 13 fields which are described as

(1) *id*: Patient's ID which is integer number.

(2) *age*: Patient's age which is integer number representing age in number of days.

(3) *gender*: Patient's gender which is integer number as categories 1 is female, 2 is male.

(4) *height*: Patient's height which is integer number representing height in centimeter.

(5) *weight*: Patient's weight which is double number representing weight in kilogram.

(6) *ap_hi*: Patient's high blood pressure which is integer number presenting pressure in mmHg.

(7) *ap_ lo*: Patient's low blood pressure which is integer number presenting pressure in mmHg.

(8) *cholesterol*: Patient's cholesterol level which is integer number as categories 1 is normal, 2 is above normal and 3 is well above normal.

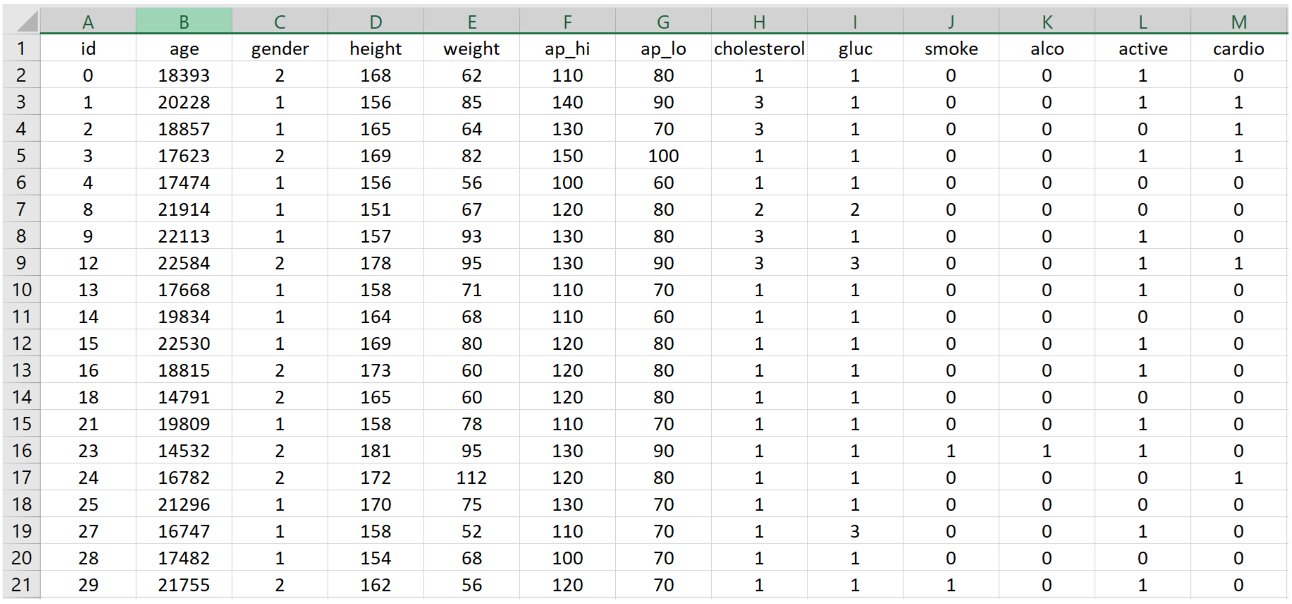
(9) *gluc*: Patient's glucose level which is integer number as categories 1 is normal, 2 is above normal and 3 is well above normal.

(10) *smoke*: Patient's smoking status which is integer number as categories 0 is no and 1 is yes.

(11) *alco*: Patient's alcohol consuming status which is integer number as categories 0 is no and 1 is yes.

(12) *active*: Patient's exercising status which is integer number as categories 0 is no and 1 is yes.

(13) *cardio*: Target field, patient's heart disease status which is integer number as categories 0 is no and 1 is yes.



	A	B	C	D	E	F	G	H	I	J	K	L	M
1	id	age	gender	height	weight	ap_hi	ap_lo	cholesterol	gluc	smoke	alco	active	cardio
2	0	18393	2	168	62	110	80	1	1	0	0	1	0
3	1	20228	1	156	85	140	90	3	1	0	0	1	1
4	2	18857	1	165	64	130	70	3	1	0	0	0	1
5	3	17623	2	169	82	150	100	1	1	0	0	1	1
6	4	17474	1	156	56	100	60	1	1	0	0	0	0
7	8	21914	1	151	67	120	80	2	2	0	0	0	0
8	9	22113	1	157	93	130	80	3	1	0	0	1	0
9	12	22584	2	178	95	130	90	3	3	0	0	1	1
10	13	17668	1	158	71	110	70	1	1	0	0	1	0
11	14	19834	1	164	68	110	60	1	1	0	0	0	0
12	15	22530	1	169	80	120	80	1	1	0	0	1	0
13	16	18815	2	173	60	120	80	1	1	0	0	1	0
14	18	14791	2	165	60	120	80	1	1	0	0	0	0
15	21	19809	1	158	78	110	70	1	1	0	0	1	0
16	23	14532	2	181	95	130	90	1	1	1	1	1	0
17	24	16782	2	172	112	120	80	1	1	0	0	0	1
18	25	21296	1	170	75	130	70	1	1	0	0	0	0
19	27	16747	1	158	52	110	70	1	3	0	0	1	0
20	28	17482	1	154	68	100	70	1	1	0	0	0	0
21	29	21755	2	162	56	120	70	1	1	1	0	1	0

Figure 4: *The first 21 records of the original cardiovascular disease dataset.*

We also apply several preprocessing methods on this dataset. Firstly, we eliminate all the records having any field value is null. After eliminating, we obtain a new dataset whose 68640 records. Then, we apply normalizing method for *age, height, weight, ap_ hi* and *ap_ lo* fields. Moreover, we apply one-hot encoding for non-binary category fields which are *gender, cholesterol* and *gluc*. A few first records of the dataset is showed at Figure 5. After preprocessing, the dataset has 17 fields which are described as:

(1) *age*: Age of the patient which is double number normalized with the number of days in 100 years.

(2) *height*: Height of the patient which is double number normalized with 100.

(3) *weight*: Weight of the patient which is double number normalized with 100.

(4) *ap_ hi*: The high blood pressure of the patient which is normalized with 200.

(5) *ap_ lo*: The low blood pressure of the patient which is normalized with 200.

(6) *smoke*: category integer number whether the patient smokes or not.

(7) *alco*: category integer number whether the patient drinks or not.

(8) *active*: category integer number whether the patient usually exercises or not.

(9) *female*: category integer number whether the patient is female or not.

(10) *male*: category integer number whether the patient is male or not.

(11) *cho_ high*: category integer number whether the cholesterol level is high or not.

(12) *cho_ medium*: category integer number whether the cholesterol level is medium or not.

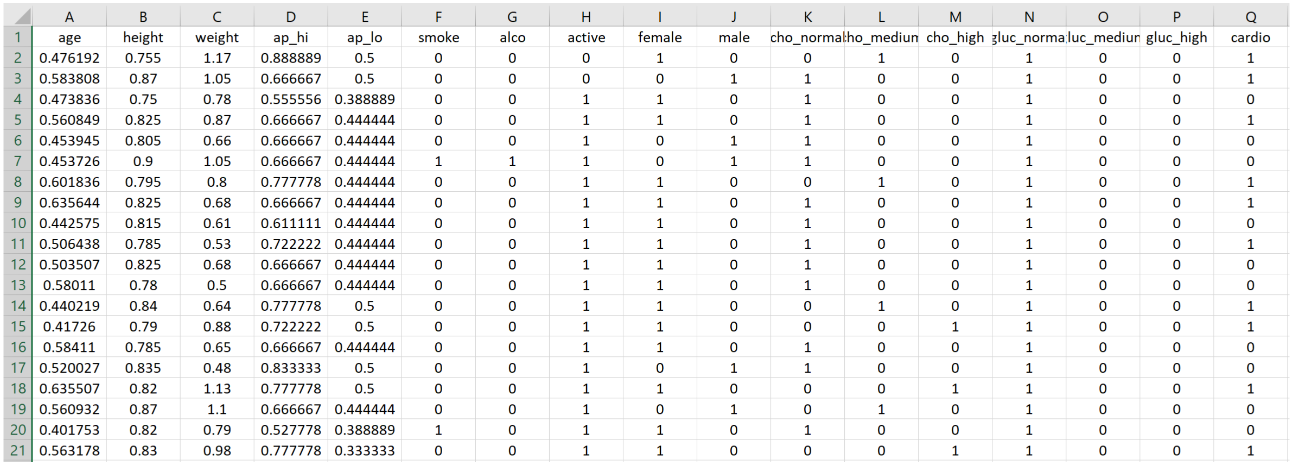
(13) *cho_ normal*: category integer number whether the cholesterol level is normal or not.

(14) *gluc_ high*: Category integer number whether the glucose level is high or not.

(15) *gluc_ medium*: Category integer number whether the glucose level is medium or not.

(16) *gluc_ low*: Category integer number whether the glucose level is low or not.

(17) *cardio*: The target value, category integer number whether the patient has heart disease or not.



	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q
1	age	height	weight	ap_hi	ap_lo	smoke	alco	active	female	male	cho_normal	cho_medium	cho_high	gluc_normal	gluc_medium	gluc_high	cardio
2	0.476192	0.755	1.17	0.888889	0.5	0	0	0	1	0	0	1	0	1	0	0	1
3	0.583808	0.87	1.05	0.666667	0.5	0	0	0	0	1	1	0	0	1	0	0	1
4	0.473836	0.75	0.78	0.555556	0.388889	0	0	1	1	0	1	0	0	1	0	0	0
5	0.560849	0.825	0.87	0.666667	0.444444	0	0	1	1	0	1	0	0	1	0	0	1
6	0.453945	0.805	0.66	0.666667	0.444444	0	0	1	0	1	1	0	0	1	0	0	0
7	0.453726	0.9	1.05	0.666667	0.444444	1	1	1	0	1	1	0	0	1	0	0	0
8	0.601836	0.795	0.8	0.777778	0.444444	0	0	1	1	0	0	1	0	1	0	0	1
9	0.635644	0.825	0.68	0.666667	0.444444	0	0	1	1	0	1	0	0	1	0	0	1
10	0.442575	0.815	0.61	0.611111	0.444444	0	0	1	1	0	1	0	0	1	0	0	0
11	0.506438	0.785	0.53	0.722222	0.444444	0	0	1	1	0	1	0	0	1	0	0	1
12	0.503507	0.825	0.68	0.666667	0.444444	0	0	1	1	0	1	0	0	1	0	0	0
13	0.58011	0.78	0.5	0.666667	0.444444	0	0	1	1	0	1	0	0	1	0	0	0
14	0.440219	0.84	0.64	0.777778	0.5	0	0	1	1	0	0	1	0	1	0	0	1
15	0.41726	0.79	0.88	0.722222	0.5	0	0	1	1	0	0	0	1	1	0	0	1
16	0.58411	0.785	0.65	0.666667	0.444444	0	0	1	1	0	1	0	0	1	0	0	0
17	0.520027	0.835	0.48	0.833333	0.5	0	0	1	0	1	1	0	0	1	0	0	0
18	0.635507	0.82	1.13	0.777778	0.5	0	0	1	1	0	0	0	1	1	0	0	1
19	0.560932	0.87	1.1	0.666667	0.444444	0	0	1	0	1	0	1	0	1	0	0	0
20	0.401753	0.82	0.79	0.527778	0.388889	1	0	1	1	0	1	0	0	1	0	0	0
21	0.563178	0.83	0.98	0.777778	0.333333	0	0	1	1	0	0	0	1	1	0	0	1

Figure 5: *The first 21 records of the preprocessed cardiovascular disease dataset.*

This preprocessed dataset will be used in our system. We will distribute 90% dataset for clients (hospitals) for training local models and remaining 10% for server for testing.

4 Implementation

4.1 Implementation requirements

In this assignment, we use Python as our main programming language. Moreover, to build up the federated system, we also use several necessary libraries/frameworks which are

(1) *Kivy*: This framework is used to create GUI app for server and clients in our system.

(2) *Socket*: This library is used to conduct socket programming for network communication for our system.

(3) *Tensorflow, Keras*: These frameworks are used to create and train models for heart disease detection problem.

(4) *Numpy, Pandas*: These libraries are used to read, write and manipulate data in the system.

(5) *Threading*: This library is used to create and manipulate running threads in our system.

(6) *Socket*: This library is used for socket programming.

(7) *Pickle*: This library is used to serialize data for network transferring in our system.

(8) *Matploblib*: This library is used to plot graphs for visualizing in server GUI.

4.2 Our federated leanring system's classes

4.2.1 Server class

Server class is implemented for GUI Kivy server app which is inherited from kivy.app.App class. Our server GUI will contain one button for starting and finishing the server, one console for logging and two plot windows for plotting loss values and accuracy values during training processes. Here is the source code of Server class.

```python
class Server(kivy.app.App):
    def __init__(self):
        super().__init__()
        Window.size = (800, 600)
        self.server_layout = kivy.uix.boxlayout.BoxLayout(orientation="horizontal",
                                                          size_hint=(1, 1))

        self.log_layout = kivy.uix.boxlayout.BoxLayout(orientation="vertical",
                                                       size_hint=(None, 1),
   width=500)
        self.info_layout = kivy.uix.boxlayout.BoxLayout(orientation="vertical",
                                                        size_hint=(1, 1))

        self.start_button = kivy.uix.button.Button(text="START", disabled=False,
                                                   font_size="20pt",
   font_name="Roboto", bold=True,
                                                   size_hint=(1, None), height=100,
                                                   background_color=(1, 0, 0, 1))
        self.log_console = kivy.uix.label.Label(text=">> Welcome to our system",
halign="left", valign="top",
                                                text_size=(300, None), padding=(5,
5),
                                                font_size="10pt",
font_name="Roboto")
        self.log_console.bind(size=self.log_console.setter('text_size'))

        self.fig1, self.ax1 = plt.subplots()
        self.plot1, = self.ax1.plot([1.0])
        self.ax1.set_title("Loss values")

        self.fig2, self.ax2 = plt.subplots()
        self.plot2, = self.ax2.plot([0])
        self.ax2.set_title("Accuracy values")

        self.loss_plot = FigureCanvasKivyAgg(self.fig1)
        self.accuracy_plot = FigureCanvasKivyAgg(self.fig2)

        self.socket = socket.socket(family=socket.AF_INET, type=socket.SOCK_STREAM)
        self.listening_thread = ListeningThread(app=self, ip="localhost",
port=10000)

        self.stop_listening = False
        self.logs = ["Welcome to our system"]
        self.losses = [1.0]
        self.accuracies = [0]

        self.data_df = None
        self.model = None

    def do_action(self,  *args):
        if self.start_button.text == "START":
            self.stop_listening = False
            self.listening_thread = ListeningThread(app=self, ip="localhost",
port=10000)
            self.listening_thread.start()
            self.start_button.text = "FINISH"
        else:
            self.start_button.text = "START"
            try:
                self.stop_listening = True
                self.listening_thread.socket.close()
                pass
            except BaseException as e:
                print("Server error: " + str(e))
            pass

    def build(self):
        self.start_button.bind(on_press=self.do_action)

        self.log_layout.add_widget(self.start_button)
        self.log_layout.add_widget(self.log_console)
        self.info_layout.add_widget(self.loss_plot)
        self.info_layout.add_widget(self.accuracy_plot)

        self.server_layout.add_widget(self.log_layout)
        self.server_layout.add_widget(self.info_layout)

        return self.server_layout

    def start(self):
        Clock.schedule_interval(self.update_plots, 1)
        Clock.schedule_interval(self.update_logs, 1)

        df = pd.read_csv("./data/data.csv")
        self.data_df = df.to_numpy()

        self.model = Sequential()
        self.model.add(Input(shape=(16,)))
        self.model.add(Dense(32, activation="relu"))
        self.model.add(Dense(16, activation="relu"))
        self.model.add(Dense(10, activation="relu"))
        self.model.add(Dense(6, activation="relu"))
        self.model.add(Dense(3, activation="relu"))
        self.model.add(Dense(1, activation="sigmoid"))
        self.model.compile(optimizer='adam',
                           loss='binary_crossentropy',
                           metrics=[keras.metrics.BinaryAccuracy()])

        self.run()

    def append_logs(self, log):
        if len(self.logs) > 10:
            self.logs = self.logs[-10:-1]
        else:
            self.logs.append(log)

    def append_losses(self, loss):
        self.losses.append(loss)

    def append_accuracies(self, accuracy):
        self.accuracies.append(accuracy)

    def update_logs(self, dt):
        log_str = ""
        for e in self.logs:
            log_str += ">> " + e + "\n"
        self.log_console.text = log_str

    def update_plots(self, dt):
        self.plot1.set_data(x1, self.losses)
        self.ax1.relim()
        self.ax1.autoscale_view()

        self.plot2.set_data(x2, self.accuracies)
        self.ax2.relim()
        self.ax2.autoscale_view()

        self.fig1.canvas.draw_idle()
        self.fig2.canvas.draw_idle()

    def do_model_fusion(self, value, kind="model"):
        if kind == "model":
            current_weights = self.model.get_weights()
            new_weights = value.get_weights()

            saved_weights = [current_weights, new_weights]
            mean_weights = list()
            for weights_list_tuple in zip(*saved_weights):
                mean_weights.append(
                    np.array([np.array(w).mean(axis=0) for w in
   zip(*weights_list_tuple)]))
            self.model.set_weights(mean_weights)
        elif kind == "loss":
            self.model = value
        elif kind == "gradient":
            optimizer = Adam()
            optimizer.apply_gradients(zip(value, self.model.trainable_variables))
        else:
            pass
        return self.model
```

The Server class has 9 methods which are

(1) *do_ action*: A callback for click-to-button event.

(2) *build*: This is default method in kivy.app.App class which is used to build server app.

(3) *start*: This method is used to start the server app.

(4) *append_ logs*: Appending new logs into logging console of the server app.

(5) *append_ losses*: Appending new loss value into loss plot of the server app.

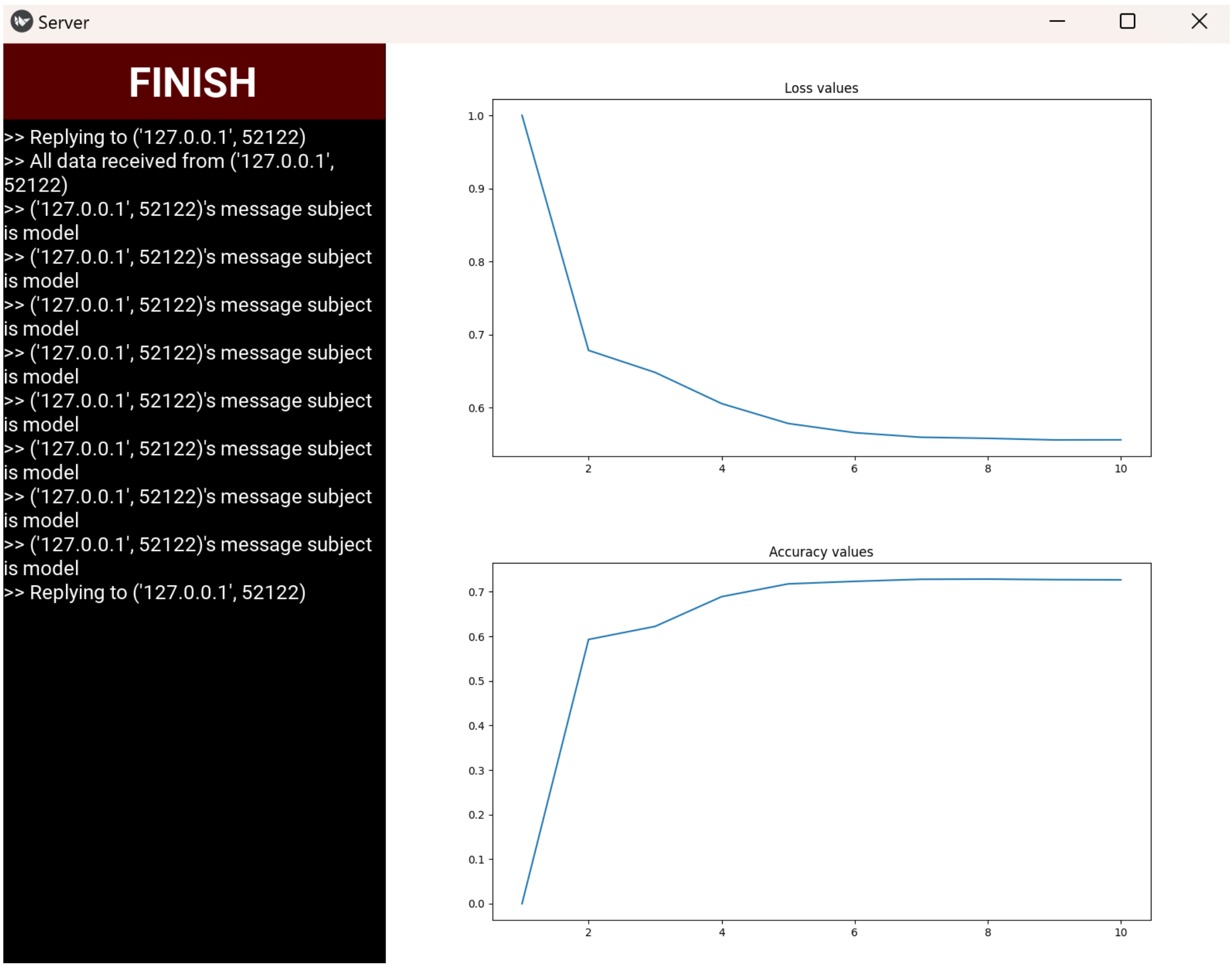
(6) *append_ accuracies*: Appending new accuracy value into accuracy plot of the server app.

(7) *update_ logs*: This method is called regularly for every 1 second to update displaying of log console.

(8) *update_ plots*: This method is called regularly for every 1 second to update displaying of loss plot and windows.

(9) *do_ model_ fusion*: This method is used to fuse the models in each federated learning process. There are three kinds of model fusion which are “model”, “loss” and “gradient”.

Here is a demonstration of our server app.



```
>> Replying to ('127.0.0.1', 52122)
>> All data received from ('127.0.0.1',
52122)
>> ('127.0.0.1', 52122)'s message subject
is model
>> ('127.0.0.1', 52122)'s message subject
is model
>> ('127.0.0.1', 52122)'s message subject
is model
>> ('127.0.0.1', 52122)'s message subject
is model
>> ('127.0.0.1', 52122)'s message subject
is model
>> ('127.0.0.1', 52122)'s message subject
is model
>> ('127.0.0.1', 52122)'s message subject
is model
>> ('127.0.0.1', 52122)'s message subject
is model
>> Replying to ('127.0.0.1', 52122)
```

Figure 6: *Our server app demonstration.*

4.2.2 Client class

Like Server class, Client class is implemented for GUI Kivy client app which is inherited from kivy.app.App class. Our client GUI will contain many buttons for many tasks for that client app. Here is the source code of Client class.

```
class Client(kivy.app.App):

    def __init__(self):
        super().__init__()
        self.training_thread = None
        self.data_df = None

    def create_socket(self, *args):
        self.soc = socket.socket(family=socket.AF_INET, type=socket.SOCK_STREAM)
        self.label.text = "Socket Created"

        self.create_socket_btn.disabled = True
        self.connect_btn.disabled = False
        self.close_socket_btn.disabled = False
```

```
 def connect(self, *args):
     try:
         self.soc.connect((self.server_ip.text, int(self.server_port.text)))
         self.label.text = "Successful connecting to the server"

         self.connect_btn.disabled = True
         self.recv_train_model_btn.disabled = False

     except BaseException as e:
         self.label.text = "Client error: " + str(e)

         self.connect_btn.disabled = False
         self.recv_train_model_btn.disabled = True

 def recv_train_model(self, *args):
     self.recv_train_model_btn.disabled = True
     self.training_thread = TrainingThread(app=self, buffer_size=1024,
recv_timeout=10, kind="model")
     self.training_thread.start()

 def close_socket(self, *args):
     self.soc.close()
     self.label.text = "Socket Closed"

     self.create_socket_btn.disabled = False
     self.connect_btn.disabled = True
     self.recv_train_model_btn.disabled = True
     self.close_socket_btn.disabled = True

 def build(self):
     self.create_socket_btn = kivy.uix.button.Button(text="Create Socket")
     self.create_socket_btn.bind(on_press=self.create_socket)

     self.server_ip = kivy.uix.textinput.TextInput(hint_text="Server IPv4
Address", text="localhost")
     self.server_port = kivy.uix.textinput.TextInput(hint_text="Server Port
Number", text="10000")

     self.server_info_boxlayout =
kivy.uix.boxlayout.BoxLayout(orientation="horizontal")
     self.server_info_boxlayout.add_widget(self.server_ip)
     self.server_info_boxlayout.add_widget(self.server_port)

     self.connect_btn = kivy.uix.button.Button(text="Connect to Server",
disabled=True)
     self.connect_btn.bind(on_press=self.connect)

     self.recv_train_model_btn = kivy.uix.button.Button(text="Receive & Train
Model", disabled=True)
     self.recv_train_model_btn.bind(on_press=self.recv_train_model)

     self.close_socket_btn = kivy.uix.button.Button(text="Close Socket",
disabled=True)
```

```
        self.close_socket_btn.bind(on_press=self.close_socket)

        self.label = kivy.uix.label.Label(text="Socket Status")

        self.box_layout = kivy.uix.boxlayout.BoxLayout(orientation="vertical")
        self.box_layout.add_widget(self.create_socket_btn)
        self.box_layout.add_widget(self.server_info_boxlayout)
        self.box_layout.add_widget(self.connect_btn)
        self.box_layout.add_widget(self.recv_train_model_btn)
        self.box_layout.add_widget(self.close_socket_btn)
        self.box_layout.add_widget(self.label)

        return self.box_layout

    def start(self):
        df = pd.read_csv("./data/data.csv")
        self.data_df = df.to_numpy()
        self.run()
```

The Client class has 6 methods which are

(1) *create_socket*: A callback for click-to-button event which creates a socket for connection.

(2) *connect*: A callback for click-to-button event which connects client to server.

(3) *recv_train_model*: A callback for click-to-button event which starts the federated learning processes.

(4) *close_socket*: A callback for click-to-button event which closes the existing socket.

(5) *build*: This is default method in kivy.app.App class which is used to build client app.

(6) *start*: This method is used to start the client app.

Here is a demonstration of our client app

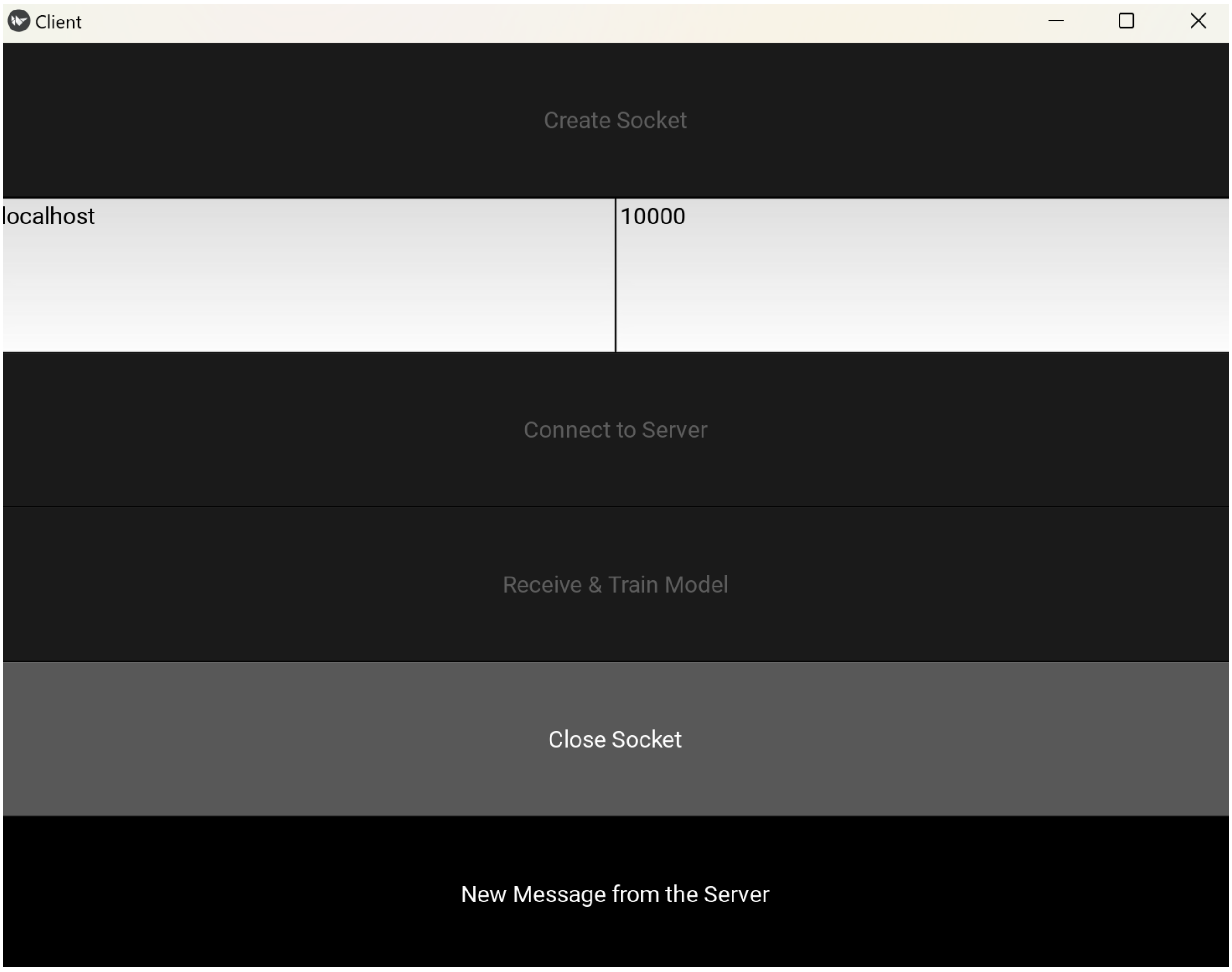


Figure 7: *Our client app demonstration.*

4.2.3 ListeningThread class

ListeningThread is inherited from threading.Thread class which is used to listen to upcoming connections from clients to server. This class is also used to manage and manipulate training threads from clients which helps our system running smoothly even if there are many connections from clients. Here is the source code of ListeningThread class.

```python
class ListeningThread(threading.Thread):
    def __init__(self, app, ip="localhost", port=10000):
        threading.Thread.__init__(self)
        self.ip = ip
        self.port = port
        self.app = app
        self.socket = None

    def run(self):
        self.socket = socket.socket(family=socket.AF_INET, type=socket.SOCK_STREAM)
        self.socket.bind((self.ip, self.port))
        self.socket.listen(1)
        while not self.app.stop_listening:
            try:
                connection, client_info = self.socket.accept()
                self.app.update_logs("New connection from
    {client_info}".format(client_info=client_info))
                socket_thread = TrainingThread(connection=connection,
                                               client_info=client_info,
                                               app=self.app,
                                               buffer_size=1024,
                                               recv_timeout=10)
                socket_thread.start()
            except BaseException as e:
                self.socket.close()
                self.app.update_logs("Server error: " + str(e))
                break
```

The ListeningThread class has 1 method which is

(1) *run*: This method will always on to listen to upcoming connection from clients to server. When a connection is accepted, a new training thread will be created and managed by this listening thread.

4.2.4 TrainingThread class

TrainingThread is inherited from threading.Thread class which is used to perform the training processes. Here is the source code of TrainingThread class.

```
class TrainingThread(threading.Thread):

    def __init__(self, app, buffer_size, recv_timeout, kind="model"):
        threading.Thread.__init__(self)
        self.app = app
        self.buffer_size = buffer_size
        self.recv_timeout = recv_timeout
        self.kind = kind

    def recv(self):
        received_data = b""
        while True:
            try:
                self.app.soc.settimeout(self.recv_timeout)
                received_data += self.app.soc.recv(self.buffer_size)

                try:
                    pickle.loads(received_data)
                    self.app.label.text = "All data is received from the server."
                    print("All data is received from the server.")
                    break
                except BaseException as e:
                    pass

            except socket.timeout:
                self.app.label.text = "{recv_timeout} Seconds of Inactivity.
    socket.timeout Exception Occurred"\
                    .format(recv_timeout=self.recv_timeout)
                return None, 0
            except BaseException as e:
                self.app.label.text = "Client error: " + str(e)
                return None, 0

        try:
            received_data = pickle.loads(received_data)
        except BaseException as e:
            print("Error Decoding the Data: {msg}.\n".format(msg=e))
            self.app.label.text = "Error Decoding the Client’s Data"
            return None, 0

        return received_data, 1

    def run(self):
        subject = "echo"
        value = None
        data_df = self.app.data_df

        while True:
            id_lst = list(range(data_df.shape[0]))
            random.shuffle(id_lst)
            data_inputs = data_df[id_lst[:1000], :-1].copy()
            data_outputs = data_df[id_lst[:1000], -1].copy()

            data = {"subject": subject, "data": value}
            data_byte = pickle.dumps(data)

            self.app.label.text = "Sending a message of type {subject} to the
    server".format(subject=subject)
            try:
                self.app.soc.sendall(data_byte)
            except BaseException as e:
                self.app.label.text = "Client error: " + str(e)
                break

            self.app.label.text = "Receiving Reply from the Server"
            received_data, status = self.recv()
            if status == 0:
                self.app.label.text = "Nothing Received from the Server"
                break
            else:
                self.app.label.text = "New Message from the Server"

            subject = received_data["subject"]
            if subject == "model":
                model = received_data["data"]
            elif subject == "done":
                self.app.label.text = "Model is trained"
                break
            else:
                self.app.label.text = "Unrecognized message type:
    {subject}".format(subject=subject)
```

```
                break

            if self.kind == "model":
                value = model
                value.fit(data_inputs, data_outputs, epochs=20, batch_size=128)
                subject = "model"
            elif self.kind == "loss":
                optimizer = Adam()
                loss_fn = tf.keras.losses.BinaryCrossentropy()
                with tf.GradientTape() as tape:
                    predictions = model(data_inputs, training=True)
                    loss = loss_fn(data_outputs.reshape(-1, 1), predictions)
                gradients = tape.gradient(loss, model.trainable_variables)
                optimizer.apply_gradients(zip(gradients, model.trainable_variables))
                value = model
                subject = "loss"
            elif self.kind == "gradient":
                loss_fn = tf.keras.losses.BinaryCrossentropy()
                with tf.GradientTape() as tape:
                    predictions = model(data_inputs, training=True)
                    loss = loss_fn(data_outputs.reshape(-1, 1), predictions)
                gradients = tape.gradient(loss, model.trainable_variables)
                value = gradients
                subject = "gradient"
```

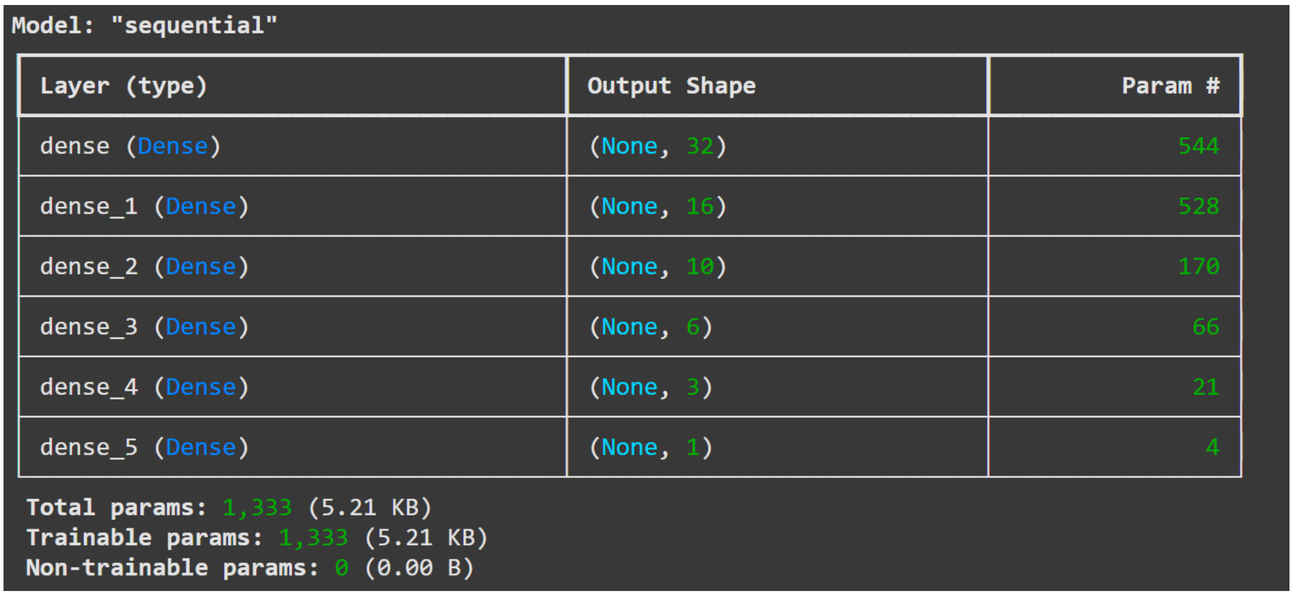
The TrainingThread class has 2 methods which are

1. *recv*: This method is in charge of receiving data from server for clients to perform training process.
2. *run*: This method is always on to receive, do training process and reply the server with model update values.

4.3 Our federated learning system's global model

The global model is significant for a federated learning system. In this implementation, we will use a simple neural network model whose the dimension of input is 16 (corresponding to 16 fields in preprocessed dataset) and the output activation function is sigmoid for binary value.

Here is the architecture of our system's global model.



```
Model: "sequential"
┏━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━┳━━━━━━━━━━━━━━━━━━━━━━━━┳━━━━━━━━━━━━━━━┓
┃ Layer (type)                    ┃ Output Shape           ┃       Param # ┃
┡━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━╇━━━━━━━━━━━━━━━━━━━━━━━━╇━━━━━━━━━━━━━━━┩
│ dense (Dense)                   │ (None, 32)             │           544 │
├─────────────────────────────────┼────────────────────────┼───────────────┤
│ dense_1 (Dense)                 │ (None, 16)             │           528 │
├─────────────────────────────────┼────────────────────────┼───────────────┤
│ dense_2 (Dense)                 │ (None, 10)             │           170 │
├─────────────────────────────────┼────────────────────────┼───────────────┤
│ dense_3 (Dense)                 │ (None, 6)              │            66 │
├─────────────────────────────────┼────────────────────────┼───────────────┤
│ dense_4 (Dense)                 │ (None, 3)              │            21 │
├─────────────────────────────────┼────────────────────────┼───────────────┤
│ dense_5 (Dense)                 │ (None, 1)              │             4 │
└─────────────────────────────────┴────────────────────────┴───────────────┘
 Total params: 1,333 (5.21 KB)
 Trainable params: 1,333 (5.21 KB)
 Non-trainable params: 0 (0.00 B)
```

Figure 8: *Architecture of our system's global model.*

This model is configured and tuned by applying usual tuning methods during training process of traditionally central machine learning approach using the preprocessed dataset.

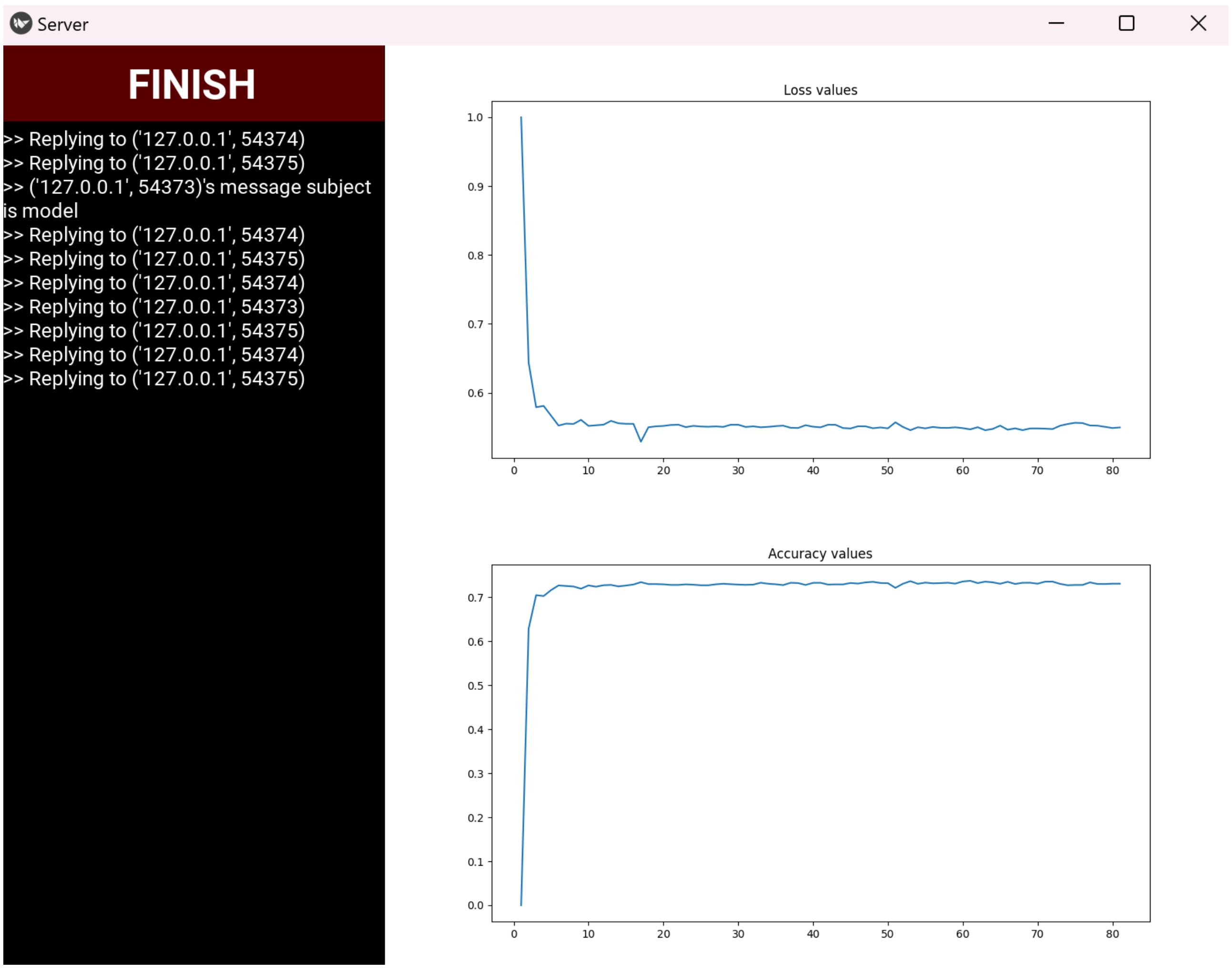
5 Result and discussion

In this section, we will run our system using three different model fusion functions respectively. Then we will give some discussions on results we obtained.

Our federated learning system for experiments will have one server and three clients. At server, 10% of preprocessed dataset is used for testing. At each client, 30% of preprocessed dataset is used for training. Therefore, every client has its own dataset which is different from each other. At each federated learning process, a random 4096 records will be chosen from dataset of each client for training which will simulate that the new data arrives regularly.

5.1 Result of averaging model fusion function

We will set our model fusion function to be an averaging model fusion function. In this setting, each training processing of each client will have 40 epochs with batch size is 256. The local model will be trained on 4096 records which is selected at the beginning of the federated learning process. Here is the figure of information on server app after running our system for a while.



```
>> Replying to ('127.0.0.1', 54374)
>> Replying to ('127.0.0.1', 54375)
>> ('127.0.0.1', 54373)'s message subject
is model
>> Replying to ('127.0.0.1', 54374)
>> Replying to ('127.0.0.1', 54375)
>> Replying to ('127.0.0.1', 54374)
>> Replying to ('127.0.0.1', 54373)
>> Replying to ('127.0.0.1', 54375)
>> Replying to ('127.0.0.1', 54374)
>> Replying to ('127.0.0.1', 54375)
```

Figure 9: *Information on server app after running our system for a while.*

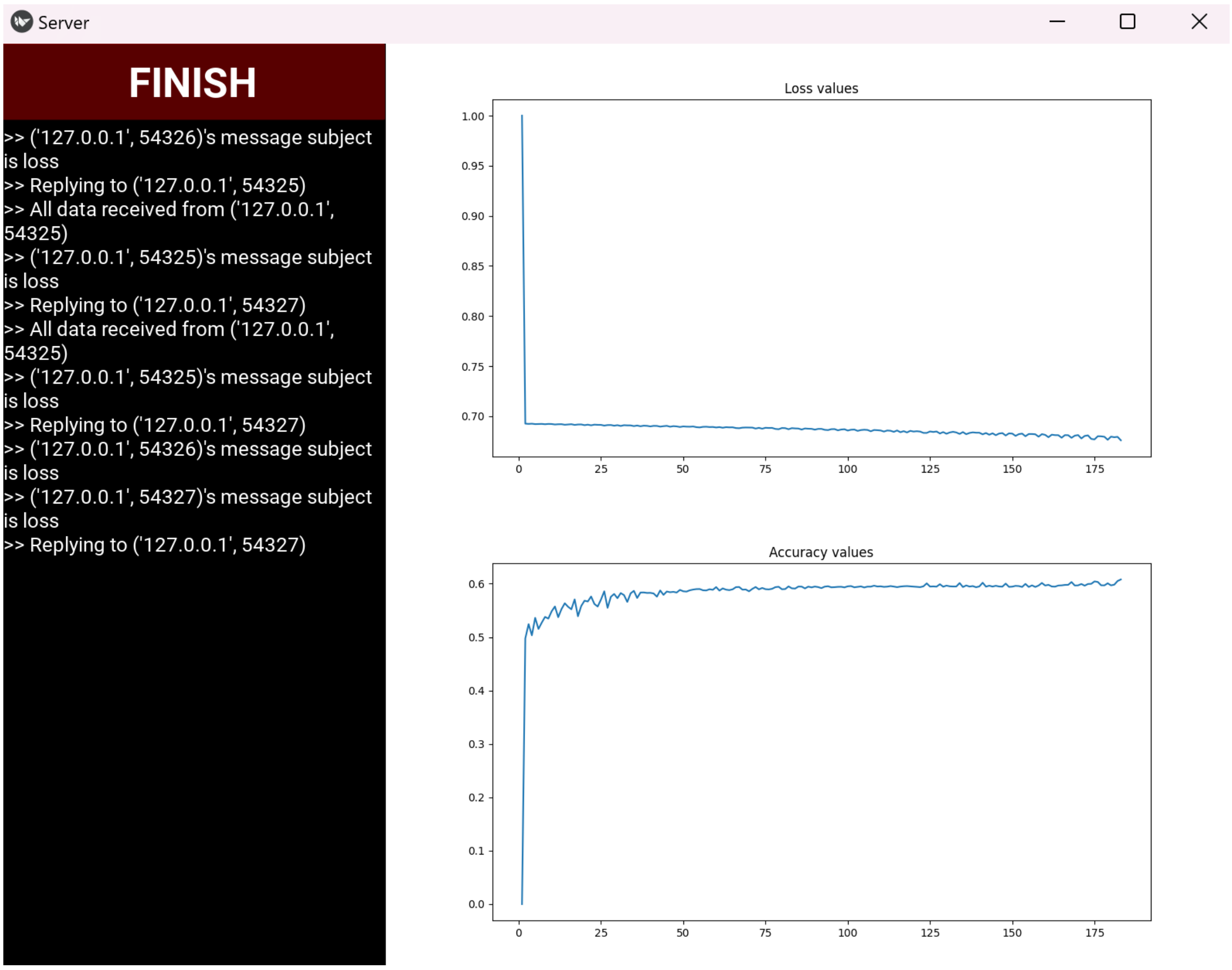
Here are the results of our system including loss values and accuracy values of all clients and server.

	Loss value	Binary accuracy values
Client 1	0.5049	0.7571
Client 2	0.5113	0.7680
Client 3	0.5120	0.7550
Server	0.5288	0.7466

Table 2: *Result of our federated learning system using averaging model fusion function.*

5.2 Result of loss model fusion function

We will set our model fusion function to be a loss model fusion function. In this setting, in each federated learning process, the model update values which are loss values of the whole chosen dataset in term of the global model will be returned. Those values will be used to train the global model at server. The dataset is the collection of 4096 records which is randomly selected at the beginning of the federated learning process. Here is the figure of information on server app after running our system for a while.



```
>> ('127.0.0.1', 54326)'s message subject is loss
>> Replying to ('127.0.0.1', 54325)
>> All data received from ('127.0.0.1', 54325)
>> ('127.0.0.1', 54325)'s message subject is loss
>> Replying to ('127.0.0.1', 54327)
>> All data received from ('127.0.0.1', 54325)
>> ('127.0.0.1', 54325)'s message subject is loss
>> Replying to ('127.0.0.1', 54327)
>> ('127.0.0.1', 54326)'s message subject is loss
>> ('127.0.0.1', 54327)'s message subject is loss
>> Replying to ('127.0.0.1', 54327)
```

Figure 10: *Information on server app after running our system for a while.*

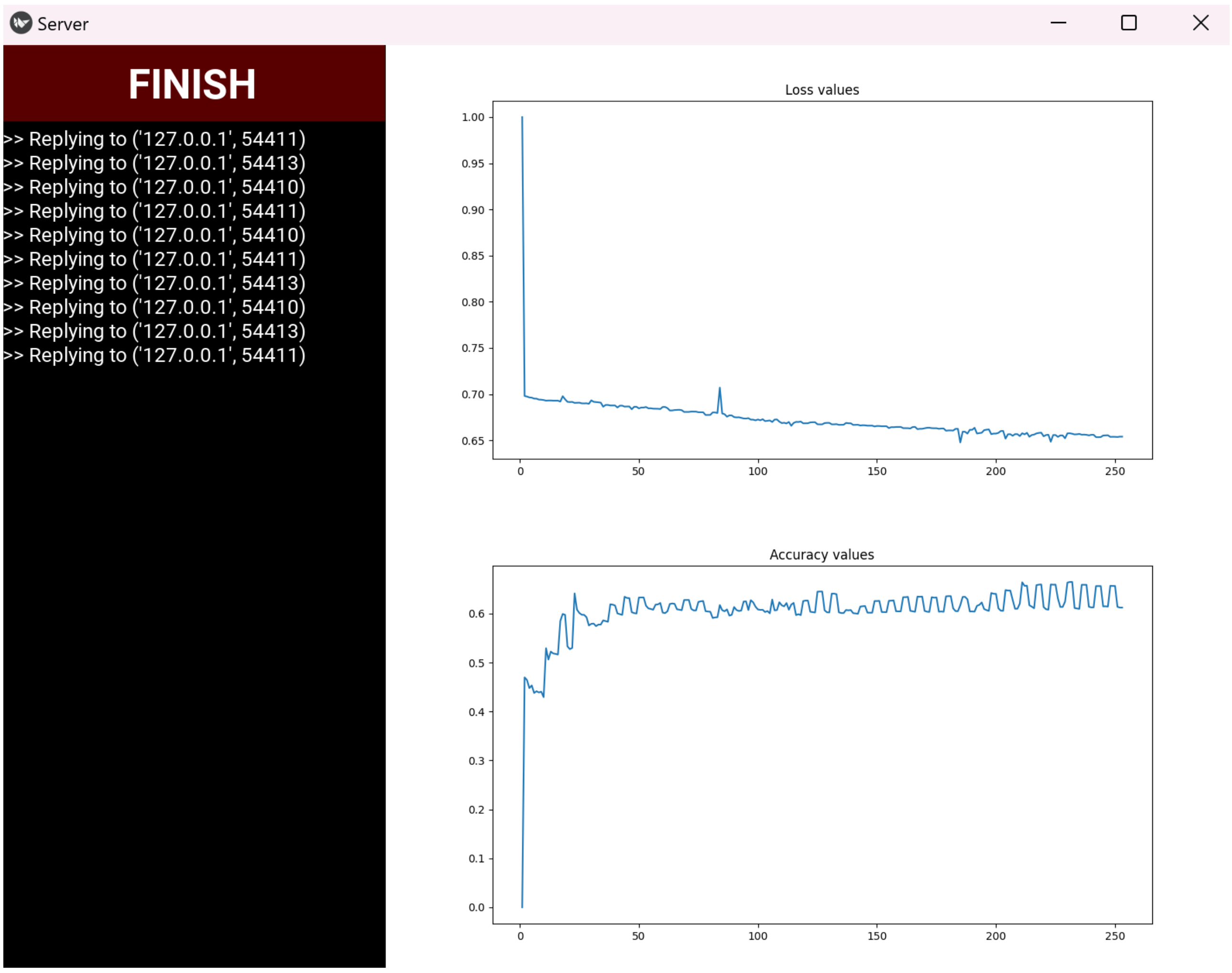
Here is the result of our system including loss value and accuracy value of server. In this setting, the reason why the loss values and accuracy values of all clients are not estimated is that when we use loss model fusion function, the local model is not trained. Hence, those values does not exist.

	Loss value	Binary accuracy values
Server	0.6558	0.6514

Table 3: *Result of our federated learning system using loss model fusion function.*

5.3 Result of gradient descent model fusion function

We will set our model fusion function to be a gradient descent model fusion function. In this setting, in each federated learning process, the model update values which are gradients values of the whole chosen dataset in term of the global model will be returned. Those values will be used to train the global model at server. The dataset is the collection of 4096 records which is randomly selected at the beginning of the federated learning process. Here is the figure of information on server app after running our system for a while.



```
>> Replying to ('127.0.0.1', 54411)
>> Replying to ('127.0.0.1', 54413)
>> Replying to ('127.0.0.1', 54410)
>> Replying to ('127.0.0.1', 54411)
>> Replying to ('127.0.0.1', 54410)
>> Replying to ('127.0.0.1', 54411)
>> Replying to ('127.0.0.1', 54413)
>> Replying to ('127.0.0.1', 54410)
>> Replying to ('127.0.0.1', 54413)
>> Replying to ('127.0.0.1', 54411)
```

Figure 11: *Information on server app after running our system for a while.*

Here is the result of our system including loss value and accuracy value of server. In this setting, the reason why the loss values and accuracy values of all clients are not estimated is that when we use gradient descent model fusion function, the local model is not trained. Hence, those values does not exist.

	Loss value	Binary accuracy values
Server	0.6464	0.6680

Table 4: *Result of our federated learning system using gradient descent model fusion function.*

5.4 Discussion

In the first federated learning system when we use averaging model fusion function. We can look at two plots on the server app (Figure 9) and know that our global model converges very quickly. After about 40-80 federated learning processes, our model has converged and reached the loss value as 0.5288 and the accuracy value as 0.7466 (Table 2). The loss value and accuracy value between server and all the clients does not have so large gap. Comparing those two values with corresponding values of all the clients, we see that it reflect enough generally all the data in each client. Moreover, the loss value and accuracy value lines of this system is quite smooth, does not fluctuate suddenly.

Coming to the second federated learning system when we use loss model fusion function. We can look at two plots on the server app (Figure 10) and see that the accuracy line fluctuates so much at the beginning. This is caused by at very first steps of training process, our global model is still finding stable paths for the gradient descent algorithm using only loss values of separating datasets. As a result, when a stable path is found, the accuracy line is stable. Moreover, our global model converges very slowly and obtains its best accuracy value as 0.6514 and loss value as 0.6558 after about 800-1000 federated learning processes (Table 3). Comparing those results with ones from the first system, we see that they are very poor. That means this model fusion function is not suitable for our system as well as our problem.

Finally, we consider the third federated learning system when we use gradient descent model fusion function. We can look at two plots on the server app (Figure 11) and see that the accuracy line fluctuates so much throughout the training process. This is caused by the fact that when we use this model fusion function, our training process gradient descent algorithm turns into a stochastic gradient descent algorithm. The reason is that at each federated learning process, we use separating gradient values from separating datasets. However, our global model converges very quickly and obtains its best accuracy value as 0.6680 and loss value as 0.6464 after about 100-200 federated learning processes (Table 4). These values are better than ones of the second system. However, comparing those results with ones from the first system, we see that they are very poor. That means this model fusion function is also not suitable for our system as well as our problem.

The first federated learning system, although it has the best performance, it still has some disadvantages. One of them is that it is time-consuming. Because at each federated learning process, instead of returning loss values or gradient values as loss and gradient descent model fusion functions do, each client must train its own local models and return those models to server. Another disadvantage is that the volume of local models which are sent to the server, this can easily be affected by network connection issues when sending those models to server takes so much time.

6 Conclusion

Medical issues are the most critical problems in human's daily life. By applying the power of machine learning, people now can know more about their health status as well as predict the risk of diseases in the future.

Heart disease causes thousands of deaths each year, which makes it one of the most dangerous medical issue. Using the power of machine learning, early precaution can be given to the patients so that they can prepare and find for themselves suitable treatments. However, to train those machine learning models we need a lot of data from patients which are usually credential and private among medical facilities. To solve this problem, federated learning is used.

After using our federated learning system for the problem of heart disease detection, we know that this method is suitable and general enough for wide usage. Moreover, when using this method, patients' data is kept secretly and maintains privacy. We also consider three different model fusion functions on three separating federated learning and have that the averaging model fusion function best suits with our problem. Although the final results are still not good, the global model has reflected and generalized successfully data in all the clients.

However, there are still many disadvantages in our federated learning system. Some of them can be listed out as not good enough datasets, too simple implementation, do not consider outer factors such as network disconnection, returned data is null, ... In the future, we will try to resolve those problems and develop a more complete federated learning system for this problem.

References

[Ludwig et al., 2020] Ludwig, H., Baracaldo, N., Thomas, G., Zhou, Y., Anwar, A., Rajamoni, S., Ong, Y., Radhakrishnan, J., Verma, A., Sinn, M., Purcell, M., Rawat, A., Minh, T., Holohan, N., Chakraborty, S., Whitherspoon, S., Steuer, D., Wynter, L., Hassan, H., and Abay, A. (2020). IBM Federated Learning: an Enterprise Framework White Paper V0.1.

[ML-CMU, 2019] ML-CMU (2019). Federated Learning: Challenges, Methods, and Future Directions.