COMPUTER SCIENCE AND DATA ANALYTICS

Course: Guided Research I

Report 5

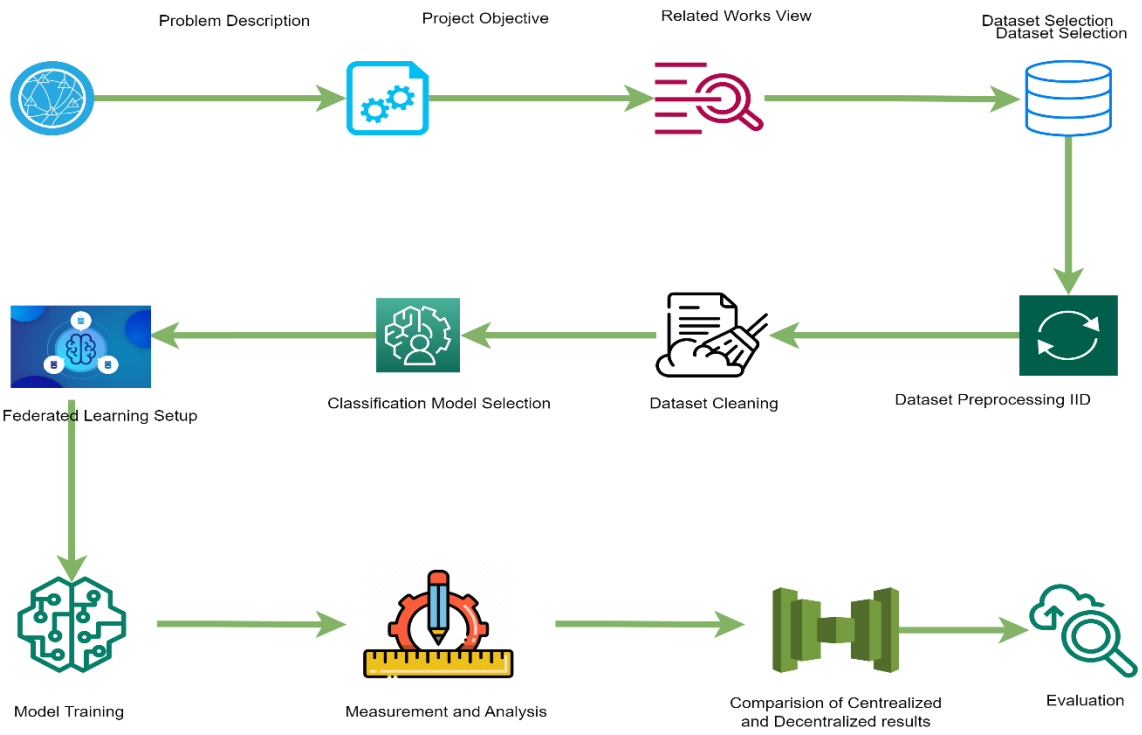
Title: Federated Machine Learning Implementation on Image Classification

Student: Ali Asgarov

Instructors: Prof. Dr. Stephen Kaisler, Assoc.Prof Jamaladdin Hasanov

Since my last report, I have accomplished:

- **Building of core functions for the Federated Learning**: Function for model and optimizer builder, function for weights sender to main node, function for averaging weights, function for setting averaged parameters to main node, function for sending main node parameters to local nodes).
- **Dataset cleaning:** In addition to Heatmap and Outlier cleaning steps, I have carried out one more experience and tried normalization on my dataset images.
- **Finalization of training and validation functions:** I have finalized the initial version of the trainer and validator functions for my models.
- **Carried experiments on measurement and analysis:** I have set up different experiments by considering the accuracy for the project as multiclassification problem as dependent variable, and number of clients, number of epochs for training, batch size as independent variables, and investigated the relationship between them.
- **Have done the training on Federated Learning with different parameters:** While doing the measurement and analysis, I have noted down the different parameters that have been used in the training, for example different batch sizes, different number of epochs and how the FL models have treated with them, and then I have used those parameters to train non-FL standard machine learning algorithms as well.
- **Have applied multiprocessing module of python for training of the local models:** For the training of the individual models in the local nodes, I have applied multiprocessing module of python to achieve training of the local nodes at the same time similar to real world practice.
- **Evaluated the results and compared the outputs with non-FL centralized learning:** Here after training of the both standard and SOTA Federated Machine Learning for training of the models, I have evaluated and analyzed the results.
- **Definition of the what were the risks and payoff throughout the project:** Here what I have done is, I have analyzed the whole project in terms of what we have achieved and what were the risks, and what will be the next steps.



Green – **Done**

Blue – **In progress**

Figure 1. Project Roadmap

Building of core functions for the Federated Learning

Here I have built the module called as **federated** which holds all the functions for the implementation of the Federated Averaging methods.

```
def create_model_optimizer_criterion_dict(number_of_clients, learning_rate, momentum):
    model_dict = dict()
    optimizer_dict= dict()
    criterion_dict = dict()

    for i in range(number_of_clients):
        model_name=f"client_{i}_model"
        model_info=net2nn()
        model_dict.update({model_name : model_info })

        optimizer_name="optimizer"+str(i)
        optimizer_info = torch.optim.SGD(model_info.parameters(), lr=learning_rate, momentum=momentum)
        optimizer_dict.update({optimizer_name : optimizer_info })

        criterion_name = "criterion"+str(i)
        criterion_info = nn.CrossEntropyLoss()
        criterion_dict.update({criterion_name : criterion_info})

    return model_dict, optimizer_dict, criterion_dict
```

create_model_optimizer_criterion_dict : This function takes three parameters: number_of_clients, learning_rate, and momentum. It returns three dictionaries: model_dict, optimizer_dict, and criterion_dict.

model_dict: It contains a set of models for each client, where each model is initialized using the net2nn() function.

optimizer_dict: It contains a set of optimizers for each client's model. Each optimizer is initialized as a Stochastic Gradient Descent (SGD) optimizer with the specified learning_rate and momentum.

criterion_dict: It contains a set of loss criteria for each client's model. Each criterion is initialized as a CrossEntropyLoss, which was used for our multi-class classification problem.

The function creates separate models, optimizers, and loss criteria for each client specified by number_of_clients, making it suitable for federated learning.

get_averaged_weights : This function calculates the averaged weights and biases of three fully connected layers (fc1, fc2, and fc3) from a set of models specified in model_dict. The function assumes that each model in model_dict contains these three layers (fc1, fc2, and fc3).

```
def get_averaged_weights(model_dict, number_of_clients, name_of_models):

    fc1_mean_weight = torch.zeros(size=model_dict[name_of_models[0]].fc1.weight.shape)
    fc1_mean_bias = torch.zeros(size=model_dict[name_of_models[0]].fc1.bias.shape)

    fc2_mean_weight = torch.zeros(size=model_dict[name_of_models[0]].fc2.weight.shape)
    fc2_mean_bias = torch.zeros(size=model_dict[name_of_models[0]].fc2.bias.shape)

    fc3_mean_weight = torch.zeros(size=model_dict[name_of_models[0]].fc3.weight.shape)
    fc3_mean_bias = torch.zeros(size=model_dict[name_of_models[0]].fc3.bias.shape)

    with torch.no_grad():
        for i in range(number_of_clients):
            fc1_mean_weight += model_dict[name_of_models[i]].fc1.weight.data.clone()
            fc1_mean_bias += model_dict[name_of_models[i]].fc1.bias.data.clone()
            (variable) fc2_mean_bias: Any
                                        ame_of_models[i]].fc2.weight.data.clone()
            fc2_mean_bias += model_dict[name_of_models[i]].fc2.bias.data.clone()

            fc3_mean_weight += model_dict[name_of_models[i]].fc3.weight.data.clone()
            fc3_mean_bias += model_dict[name_of_models[i]].fc3.bias.data.clone()


        fc1_mean_weight =fc1_mean_weight/number_of_clients
        fc1_mean_bias = fc1_mean_bias/ number_of_clients

        fc2_mean_weight =fc2_mean_weight/number_of_clients
        fc2_mean_bias = fc2_mean_bias/ number_of_clients

        fc3_mean_weight =fc3_mean_weight/number_of_clients
        fc3_mean_bias = fc3_mean_bias/ number_of_clients

    return fc1_mean_weight, fc1_mean_bias, fc2_mean_weight, fc2_mean_bias, fc3_mean_weight, fc3_mean_bias
```

The function uses the following steps to calculate the averaged weights and biases:

Initialize tensors (fc1_mean_weight, fc1_mean_bias, fc2_mean_weight, fc2_mean_bias, fc3_mean_weight, and fc3_mean_bias) with zeros. These will be used to store the cumulative sum of weights and biases from all the models.

Iterate through each client model (indexed by i) specified in name_of_models.

Add the weights and biases of each layer from the client's model to the corresponding initialized tensors.

After iterating through all the client models, divide the cumulative sum of weights and biases by the number_of_clients to get the averaged values.

Returning the averaged weights and biases for all three layers.

set_averaged_weights_as_main_model_weights_and_update_main_model: This function takes a main_model and updates its weights and biases to be the averaged values obtained from the get_averaged_weights function, using the model_dict and name_of_models.

```
def set_averaged_weights_as_main_model_weights_and_update_main_model(main_model, model_dict, number_of_clients, name_of_models):
    fc1_mean_weight, fc1_mean_bias, fc2_mean_weight, fc2_mean_bias, fc3_mean_weight, fc3_mean_bias = get_averaged_weights(model_dict, number_of_clients, name_o
    with torch.no_grad():
        main_model.fc1.weight.data = fc1_mean_weight.data.clone()
        main_model.fc2.weight.data = fc2_mean_weight.data.clone()
        main_model.fc3.weight.data = fc3_mean_weight.data.clone()

        main_model.fc1.bias.data = fc1_mean_bias.data.clone()
        main_model.fc2.bias.data = fc2_mean_bias.data.clone()
        main_model.fc3.bias.data = fc3_mean_bias.data.clone()
    return main_model
```

The steps involved in this function are as follows:

Calls the get_averaged_weights function to calculate the averaged weights and biases from the model_dict using name_of_models.

Assigns the returned averaged weights and biases to fc1_mean_weight, fc1_mean_bias, fc2_mean_weight, fc2_mean_bias, fc3_mean_weight, and fc3_mean_bias.

Updates the main_model's fc1, fc2, and fc3 layers' weights and biases with the calculated averaged values. This is done using the main_model.fcX.weight.data and main_model.fcX.bias.data attributes where X is 1, 2, or 3, corresponding to the layers.

Returns the updated main_model.

send_main_model_to_nodes_and_update_model_dict: This function updates the models in model_dict with the weights and biases of the main_model. It uses the name_of_models list to identify the corresponding models in the model_dict.

```
def send_main_model_to_nodes_and_update_model_dict(main_model, model_dict, number_of_clients,name_of_models):
    with torch.no_grad():
        for i in range(number_of_clients):
            model_dict[name_of_models[i]].fc1.weight.data =main_model.fc1.weight.data.clone()
            model_dict[name_of_models[i]].fc2.weight.data =main_model.fc2.weight.data.clone()
            model_dict[name_of_models[i]].fc3.weight.data =main_model.fc3.weight.data.clone()
            model_dict[name_of_models[i]].fc1.bias.data =main_model.fc1.bias.data.clone()
            model_dict[name_of_models[i]].fc2.bias.data =main_model.fc2.bias.data.clone()
            model_dict[name_of_models[i]].fc3.bias.data =main_model.fc3.bias.data.clone()
    return model_dict
```

The steps involved in this function are as follows:

Iterating through each client (indexed by i) specified in name_of_models.

For each client model, updating its fc1, fc2, and fc3 layers' weights and biases with the values from the main_model.

Cloning the weights and biases of main_model to avoid directly referencing the tensors, ensuring that changes to one model do not affect the others.

After updating the weights and biases for all clients, return the updated model_dict.

start_train_end_node_process : This function trains and validates multiple clients/nodes concurrently using ThreadPoolExecutor. It uses the train_and_validate function to handle training and validation for each client. The clients' data and models are managed using dictionaries (x_train_dict, y_train_dict, x_test_dict, y_test_dict, model_dict, criterion_dict, and optimizer_dict).

```
def start_train_end_node_process(number_of_clients, model_dict, name_of_criterions, name_of_optimizers, criterion_dict, name_of_models, optimizer_dict, x_train
    def train_and_validate(client_idx):
        x_train, y_train = x_train_dict[list(x_train_dict.keys())[client_idx]], y_train_dict[list(y_train_dict.keys())[client_idx]]
        x_test, y_test = x_test_dict[list(x_test_dict.keys())[client_idx]], y_test_dict[list(y_test_dict.keys())[client_idx]]

        train_ds = TensorDataset(x_train, y_train)
        train_dl = DataLoader(train_ds, batch_size=batch_size, shuffle=True)

        test_ds = TensorDataset(x_test, y_test)
        test_dl = DataLoader(test_ds, batch_size=batch_size * 2)

        model = model_dict[name_of_models[client_idx]]
        criterion = criterion_dict[name_of_criterions[client_idx]]
        optimizer = optimizer_dict[name_of_optimizers[client_idx]]

        for epoch in range(numEpoch):
            train_loss, train_accuracy = model.train_model(train_dl, criterion, optimizer)
            test_loss, test_accuracy = model.validate_model(test_dl, criterion)
```

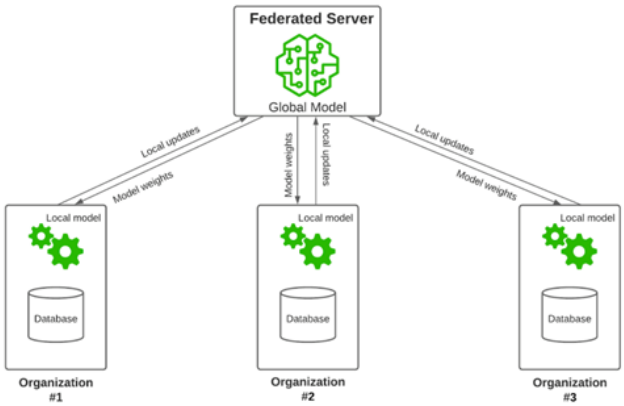


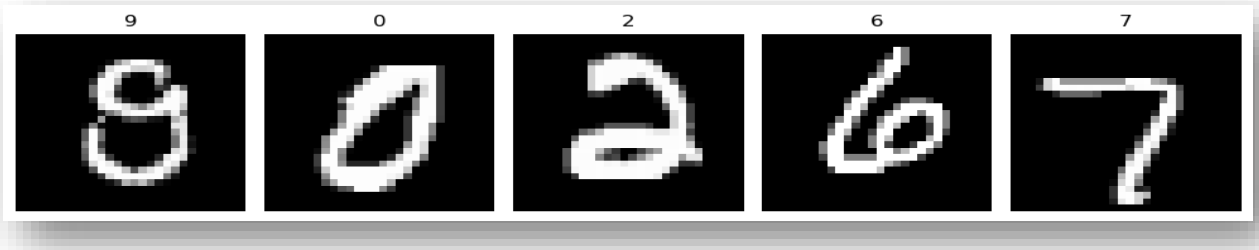
Figure 2. Federated Learning Setup

Dataset cleaning - Normalization Technique Here purpose of the code was to visualize a small subset of digits from our dataset and show the effect of normalization on the pixel values. Normalization is a common preprocessing step to scale features within a similar range, which can improve the performance of learning algorithms. In this case, the pixel values are normalized to be between 0 and 1, which makes the data suitable for training neural networks. The function plot_digits helps visualize the images in a grid-like format with their corresponding labels.

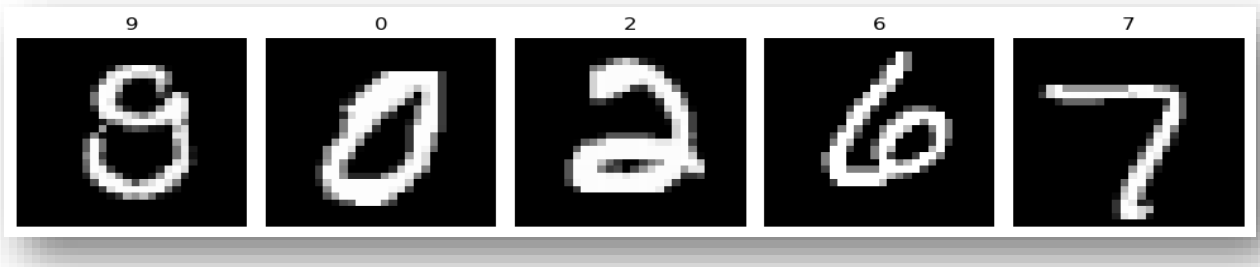
```
x_combined = np.concatenate([x_train, x_valid])
y_combined = np.concatenate([y_train, y_valid])

x_combined_normalized = x_combined / 255
x_train_normalized = x_combined_normalized[:len(x_train)]
x_valid_normalized = x_combined_normalized[len(x_train):]
x_test_normalized = x_test / 255


def plot_digits(images, titles, num_digits=5, rows=1, figsize=(10, 4)):
    fig, axs = plt.subplots(rows, num_digits, figsize=figsize)
    axs = axs.ravel()
    for i in range(rows * num_digits):
        axs[i].imshow(images[i].reshape(28, 28), cmap='gray')
        axs[i].set_title(titles[i])
        axs[i].axis('off')
    plt.tight_layout()
    plt.show()

i = 80
j = 85

sample_digits = x_train[i:j]
sample_titles = y_train[i:j]
plot_digits(sample_digits, sample_titles, num_digits=5, rows=1, figsize=(10, 4))

sample_digits_normalized = x_train_normalized[i:j]
sample_titles_normalized = y_train[i:j]
plot_digits(sample_digits_normalized, sample_titles_normalized, num_digits=5, rows=1, figsize=(10, 4))
```

Here are the outputs before and after normalization.

Before normalization



After normalization



Finalization of training and validation functions: Here what I have done is, I have finalized the training and validation function steps and used for the training of both local and central nodes.

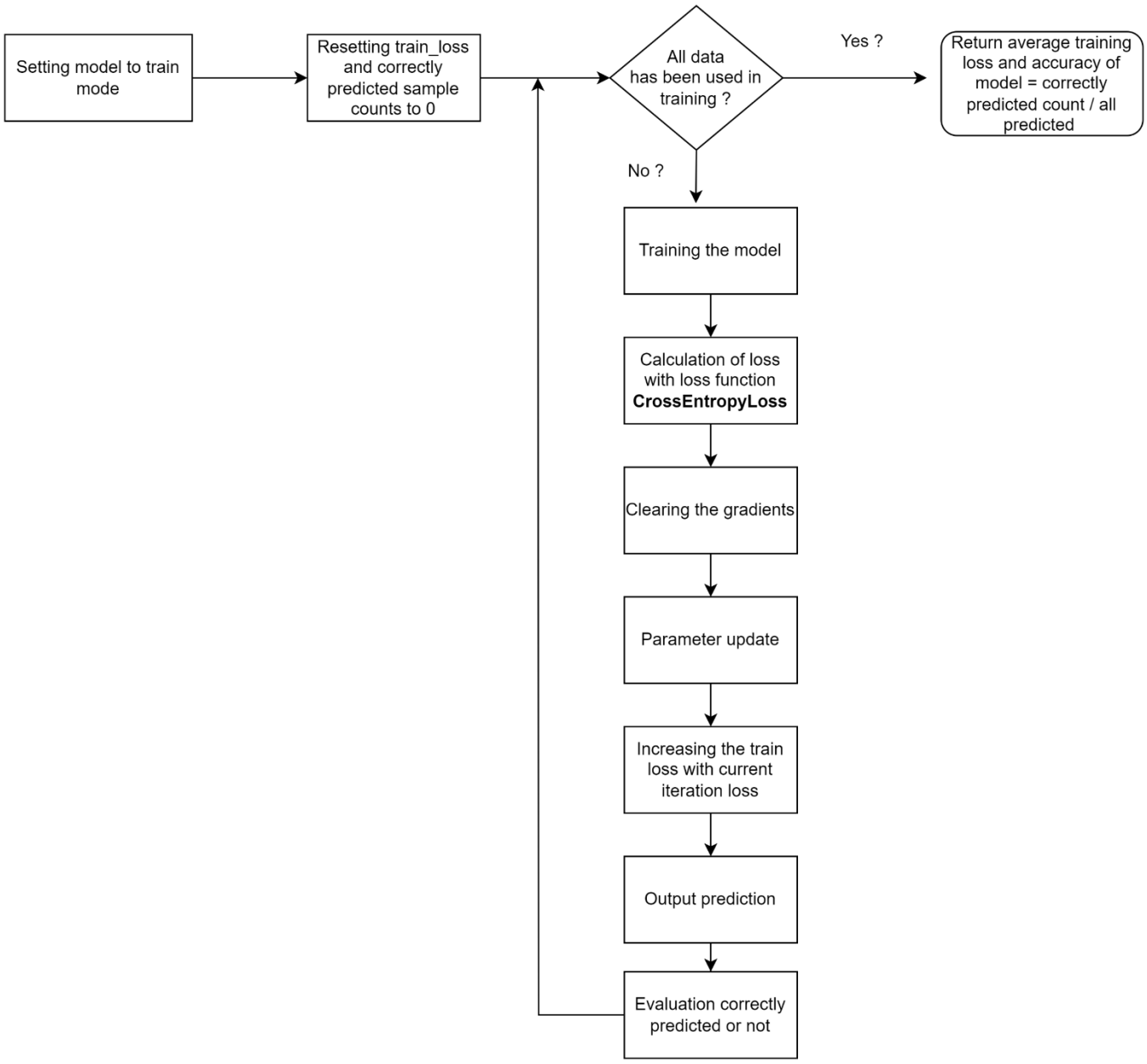


Figure 3. Training Function Setup

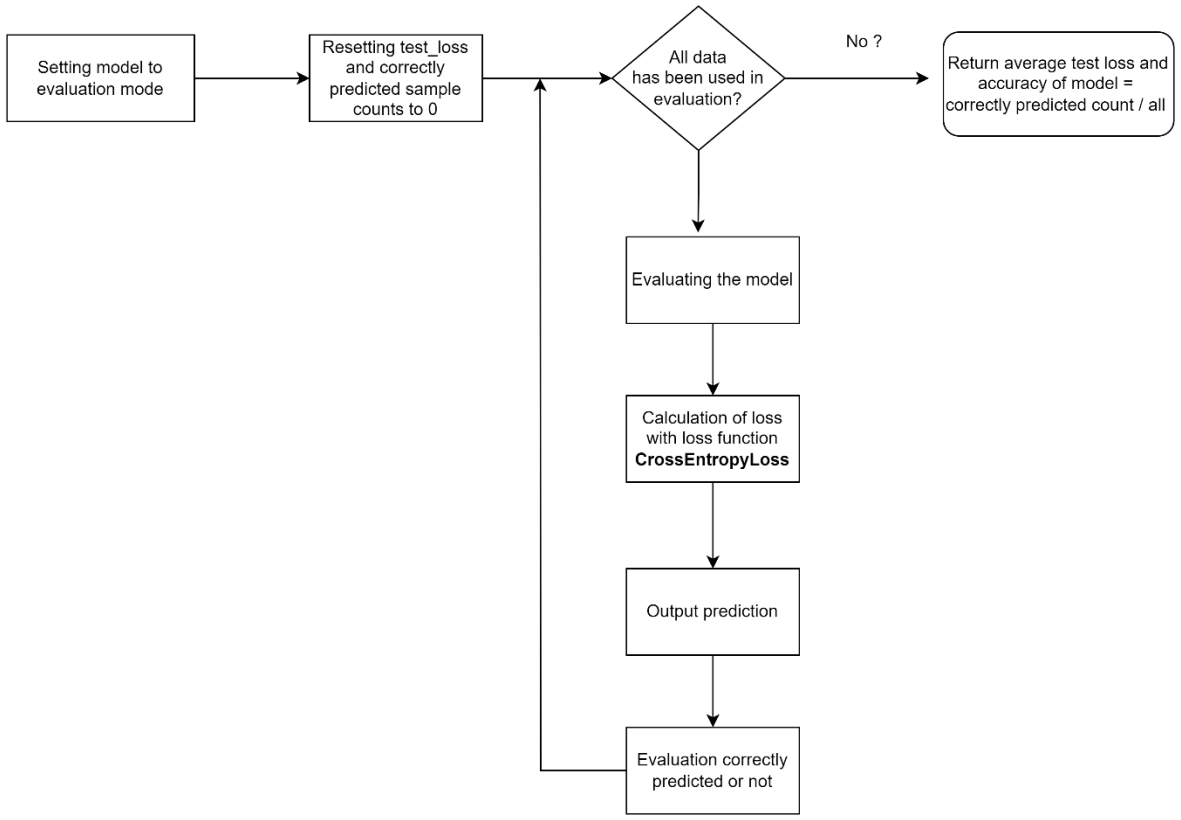


Figure 4. Validation Function Setup

Carried experiments on measurement and analysis:

Defined the dependent variable as accuracy of the main multiclassification model and defined independent variables as:

- Number of Clients
- Learning Rate
- Number of Epochs for Training
- Batch Size
- Experimental Design:

Established a baseline configuration for the model with initial values of independent variables.

Trained the model with the baseline configuration and recorded accuracy.

Systematically varied one independent variable at a time while keeping others constant.

Trained the model with each new configuration and record accuracy.

Analyzed the results to identify the impact of each independent variable on model accuracy.

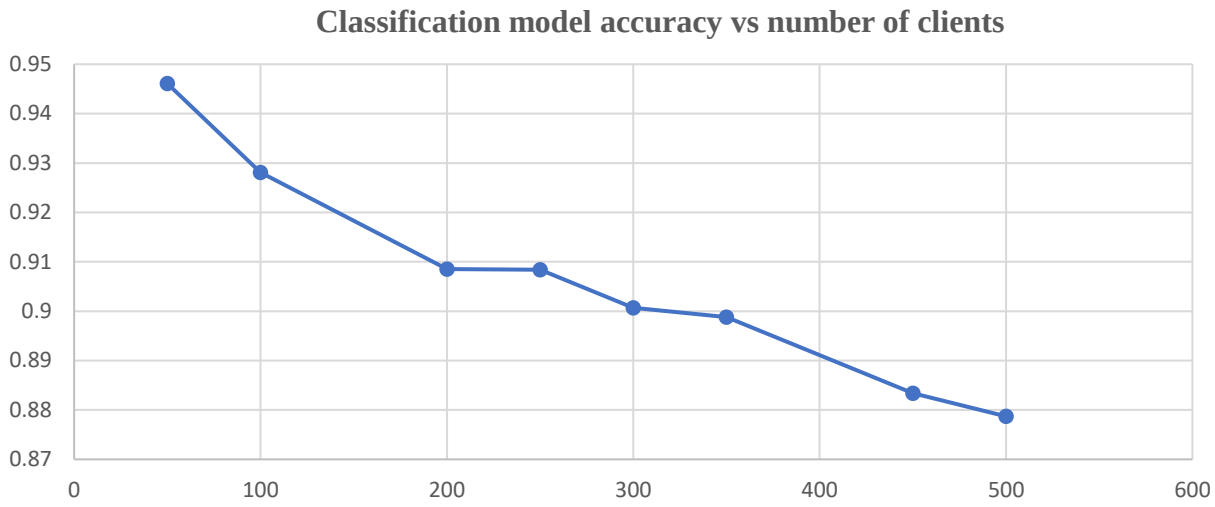


Figure 5 Classification model accuracy vs number of clients

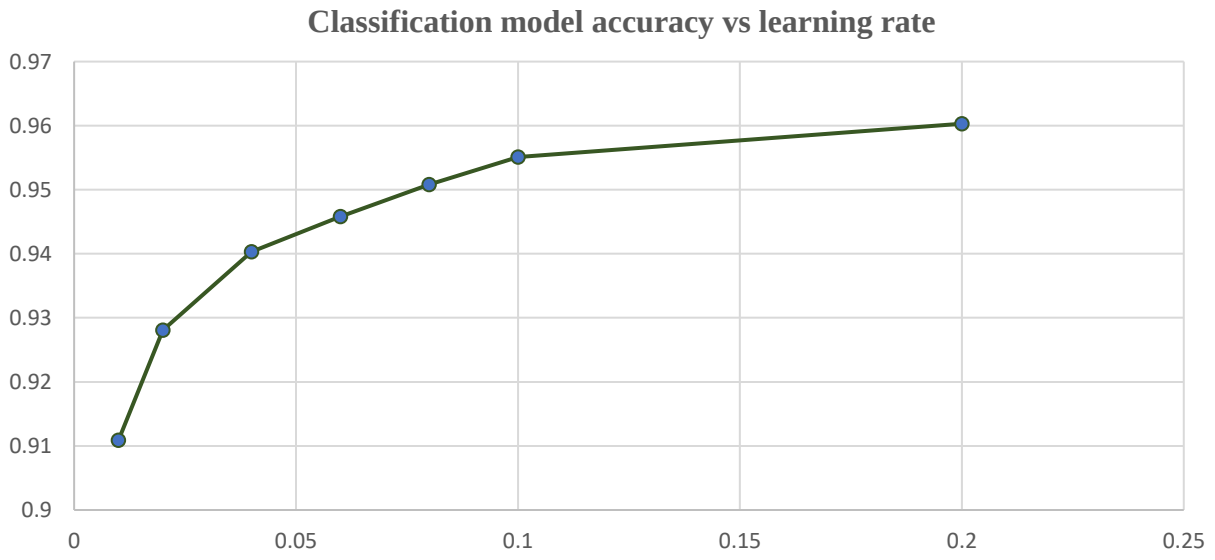


Figure 6 Classification model accuracy vs learning rate

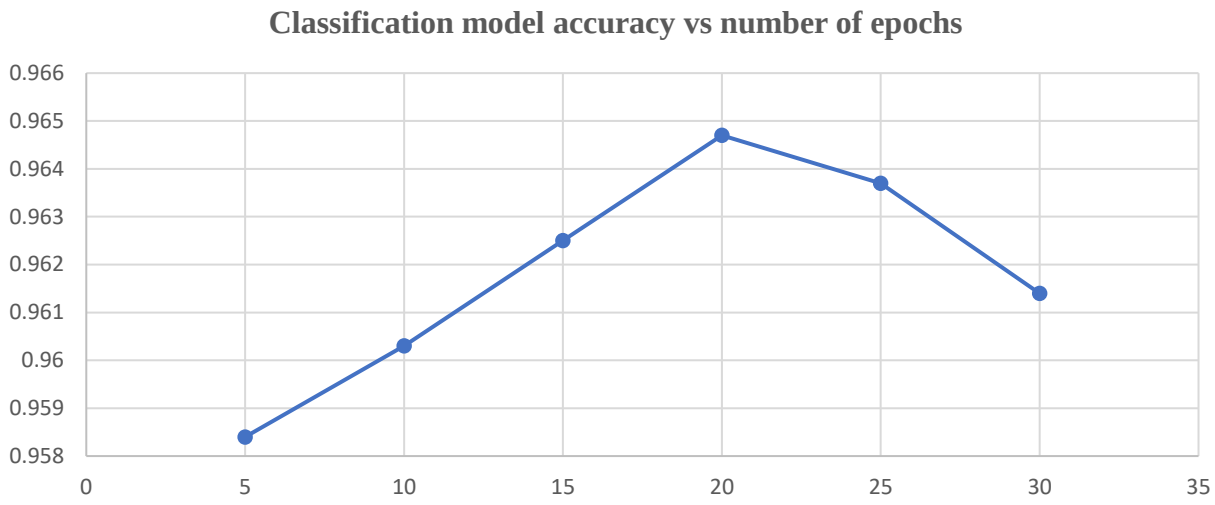


Figure 7 Classification model accuracy vs number of epochs

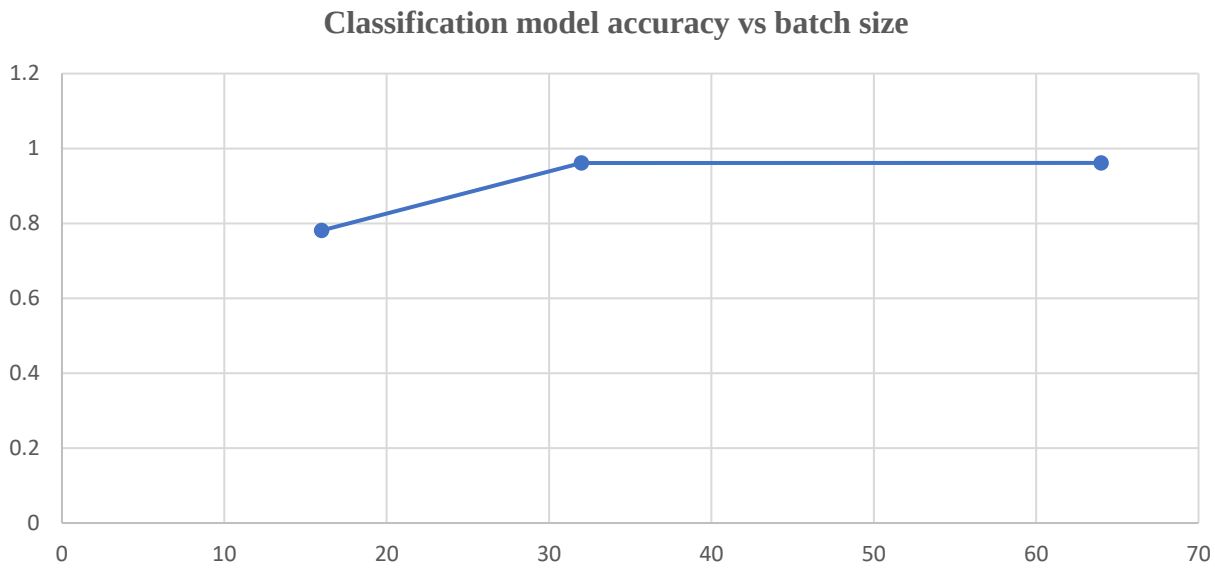


Figure 8 Classification model accuracy vs batch size

Evaluated the results and compared the outputs with non-FL centralized learning: Here after training of the both standard and SOTA Federated Machine Learning for training of the models, I have evaluated and analyzed the results. In order to be able to have apple to apple comparison I have done the comparison with the same parameters. Here are the results for the standard machine learning model and Federated Machine Learning implementation.

Standard Machine Learning Model

```
---------- Centralized ( Non - Distributed ) Model ------
-------------------- Training Started -------------------
Epoch:   1 | Train accuracy:  0.8893 | Test accuracy:  0.8629
Epoch:   2 | Train accuracy:  0.9617 | Test accuracy:  0.9652
Epoch:   3 | Train accuracy:  0.9741 | Test accuracy:  0.9713
Epoch:   4 | Train accuracy:  0.9797 | Test accuracy:  0.9743
Epoch:   5 | Train accuracy:  0.9855 | Test accuracy:  0.9736
Epoch:   6 | Train accuracy:  0.9885 | Test accuracy:  0.9738
Epoch:   7 | Train accuracy:  0.9909 | Test accuracy:  0.9782
Epoch:   8 | Train accuracy:  0.9935 | Test accuracy:  0.9761
Epoch:   9 | Train accuracy:  0.9945 | Test accuracy:  0.9624
Epoch:  10 | Train accuracy:  0.9965 | Test accuracy:  0.9791
-------------------- Training finished -------------------
```

Train (50000) and test (10000) amounts are full train and test data

learning_rate is 0.2

momentum is 0.2

numEpoch is 30

Implementation is done on 16 core CPU, 3.2 GHz base speed

Federated Machine Learning Implementation

Chosen parameters from the measurement and analysis

number_of_clients is 100

learning_rate is 0.2

numEpoch is 30

batch_size is 64

momentum is 0.2

train_amount is 4000 for each label (build IID data)

test_amount is 1000 for each label (build IID data)

```
Iteration 2 : main_model accuracy on all test data:  0.8915
Iteration 3 : main_model accuracy on all test data:  0.9134
Iteration 4 : main_model accuracy on all test data:  0.9243
Iteration 5 : main_model accuracy on all test data:  0.9319
Iteration 6 : main_model accuracy on all test data:  0.9394
Iteration 7 : main_model accuracy on all test data:  0.9438
Iteration 8 : main_model accuracy on all test data:  0.9427
Iteration 9 : main_model accuracy on all test data:  0.9490
Iteration 10 : main_model accuracy on all test data:  0.9504
Iteration 11 : main_model accuracy on all test data:  0.9526
Iteration 12 : main_model accuracy on all test data:  0.9542
Iteration 13 : main_model accuracy on all test data:  0.9565
Iteration 14 : main_model accuracy on all test data:  0.9577
Iteration 15 : main_model accuracy on all test data:  0.9601
```

As I mentioned in the previous reports, the convergence is guaranteed for the IID data and in this training. I have done the 15 iterations and model has reached 96.01 % of accuracy.

Definition of the what were the risks and payoff throughout the project

Throughout the project there were some risks, limits and payoff in terms of data preprocessing, data privacy, client selection, model training, and others. At the end of the day we have achieved 96 % of impressive accuracy, that was impressive because Federated Learning achieved this without seeing the data.

We have achieved:

Data Privacy

As we have trained all the models in the local nodes, data privacy has been saved.

Decentralized Training

We have achieved the decentralized learning as we have done the training on local nodes, each node has used its own computation power for the training which has resulted in faster training – to simulate it I have used multiprocessing module of the python.

Lower Bandwidth and Power Usage

As the training goes inside the local nodes, we do not need to have high transmission speed, bandwidth or power since only thing being transferred are the parameters.

Control Over Data

During the training all the nodes are able to participate or choose not to, since the control of the data is on the user and no part of the data is being shared in any case.

Scalability

The number of clients could be increased and decreased at the same time without any additional engineering, solution is scalable.

Cost-Efficient

The solution is cost efficient since we have not needed any transmission power which is costly.

Here are some of the next steps defined to work in the future:

Handling Heterogeneity: As the Federated Learning is dependent on the IID data, one of the next steps is working on non-IID data, where data will be non-independently and identically distributed.

Adaptive Learning Rates: While working on non-IID data we would need to have adaptive learning rates for each client, since data size will be different in all clients

Fault Tolerance and Dynamic Client Selection: There could be cases that some of the devices/nodes are faulty or malicious devices or inaccurate updates. Would want to build model for detection of those clients needed in this scenario with selecting devices to participate in training.