COMPUTER SCIENCE AND DATA ANALYTICS

Course: Guided Research I

Report 4

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:

- More sophisticated IID data preparation
- Model initialization
- Centralized model training
- Client model training
- Dataset cleaning techniques
- Heatmap building
- Outlier detection

More sophisticated IID data preparation:

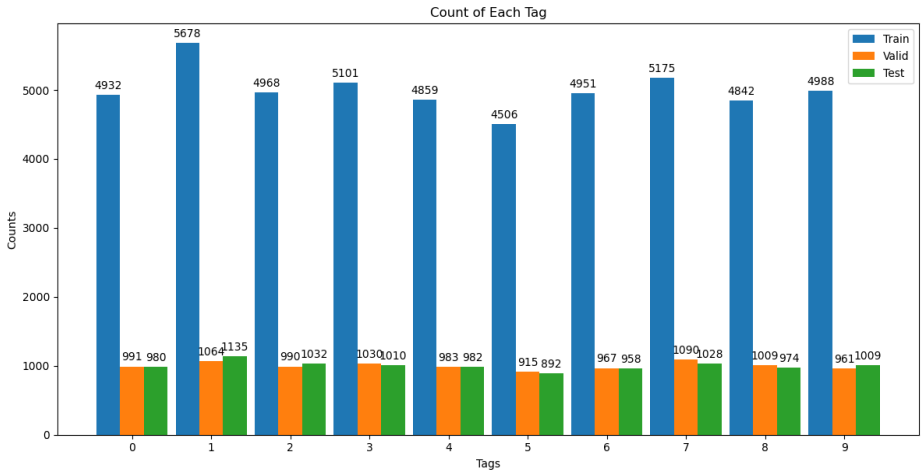
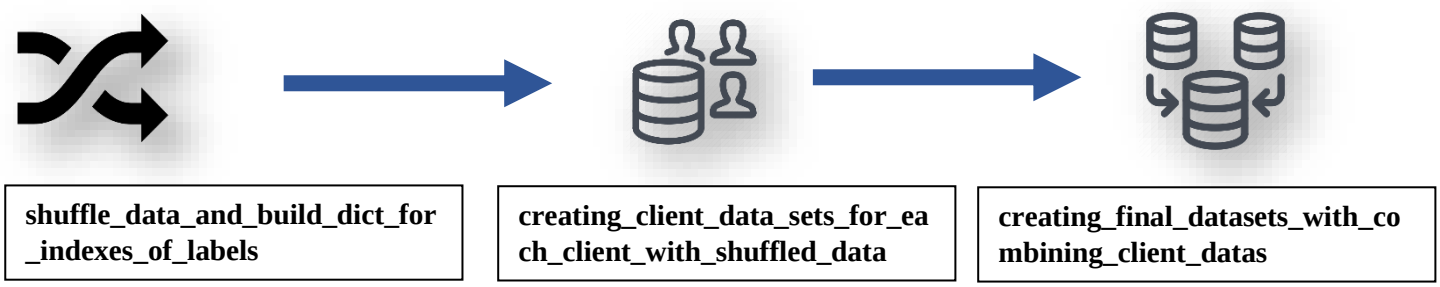


Fig 1.0 Count of each tag in the data

Implemented more sophisticated data preparation techniques for Independent and Identically Distributed (IID) data. Ensured that the data is appropriately partitioned and distributed among clients and the central server.

I have built the module for the data preprocessing and called as **preprocessing** which includes 4 main preprocessing functions:



shuffle_data_and_build_dict_for_indexes_of_labels : This function takes a dataset of labels (y_data), a random seed (seed), and an amount (amount). It returns a dictionary (label_dict) with subsets of the input data, each associated with a specific label.

```
def shuffle_data_and_build_dict_for_indexes_of_labels(y_data, seed, amount):
    y_data=pd.DataFrame(y_data,columns=["label"])
    y_data["index"]=np.arange(len(y_data))
    label_dict = {}
    for i in range(10):
        var_name = f"label_{i}"
        label_info=y_data[y_data["label"]==i]
        np.random.seed(seed)
        label_info=np.random.permutation(label_info)
        label_info=label_info[0:amount]
        label_info=pd.DataFrame(label_info, columns=["label","index"])
        label_dict.update({var_name: label_info })
    return label_dict
```

It initializes an empty dictionary (label_dict), For each label (0 to 9) a. select data rows with the current label. It randomly shuffle the data using the given seed, creates a subset of amount elements, stores the subset DataFrame in label_dict with a key named label_i. Eventually it returns the resulting label_dict.

creating_client_data_sets_for_each_client_with_shuffled_data: This function takes in label_dict, number_of_clients, and amount. It returns a dictionary clients_data_dicts containing client datasets.

It calculates label_amount by dividing amount by number_of_clients It initialize an empty clients_data_dicts dictionary. For each client (indexed 0 to number_of_clients - 1):

a. Creating a unique client_data_name.

b. Initializing an empty DataFrame dumb.

c. For each label (indexed 0 to 9):

Extracting the subset of shuffled data for the current label and client.

Concatenating the subset DataFrame with dumb.

d. Resetting the index of dumb.

e. Storing dumb DataFrame in clients_data_dicts with the key client_data_i.

Return clients_data_dicts, containing the client datasets.

```
def creating_client_data_sets_for_each_client_with_shuffled_data(label_dict, number_of_clients,
    clients_data_dicts= dict()
    label_amount =int(math.floor(amount/number_of_clients))
    for i in range(number_of_clients):
        client_data_name="client_data_"+str(i)
        dumb=pd.DataFrame()
        for j in range(10):
            label_name=str("label_")+str(j)
            a=label_dict[label_name][i*label_amount:(i+1)*label_amount]
            dumb=pd.concat([dumb,a], axis=0)
        dumb.reset_index(drop=True, inplace=True)
        clients_data_dicts.update({client_data_name: dumb})
    return clients_data_dicts
```

creating_final_datasets_with_combining_client_datas : This function combines client datasets from clients_data_dicts with input features x_data and corresponding labels y_data. It creates two dictionaries, x_data_dict and y_data_dict, containing the final combined datasets for input features and labels from multiple clients, respectively. The function returns these dictionaries for further processing or model training in a Federated Learning setup.

```
def creating_final_datasets_with_combining_client_datas(clients_data_dicts, x_data, y_data, x_
    x_data_dict= dict()
    y_data_dict= dict()
    for i in range(number_of_clients):
        xname= x_name + '_' + str(i)
        yname= y_name + '_' + str(i)
        client_data_name = "client_data_" + str(i)
        indexes=np.sort(np.array(clients_data_dicts[client_data_name]["index"]))
        x_info= x_data[indexes]
        x_data_dict.update({xname : x_info})
        y_info= y_data[indexes]
        y_data_dict.update({yname : y_info})
    return x_data_dict, y_data_dict
```

The function **data_shuffler_and_distributor_for_client which** is combining all three preprocessing function inside one main function.

```
def data_shuffler_and_distributor_for_client(x_data, y_data, x_name, y_name, amount, number_of_clients):
    label_dict=shuffle_data_and_build_dict_for_indexes_of_labels(y_data, 1, amount)
    clients_data_dicts = creating_client_data_sets_for_each_client_with_shuffled_data(label_dict,
number_of_clients, amount)
    x_dict, y_dict = creating_final_datasets_with_combining_client_datas(clients_data_dicts, x_data,
y_data, x_name, y_name, number_of_clients)
    return x_dict, y_dict
```

Importing the module and functions:

```
# User Defined Function for data preprocessing
from preprocessing import data_shuffler_and_distributor_for_client
```

Model Initialization:

Developed algorithms for initializing models in both the local client and centralized server environments. Ensured that the models are set up correctly before training begins.

Created model dictionaries for each client and the central server.

These dictionaries likely store the model parameters, configurations, and other relevant information for each model.

I have built the module called as **federated** and have added the function which is doing the model initializations.

```
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=m
        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
```

This function **create_model_optimizer_criterion_**dict creates three dictionaries: model_dict, optimizer_dict, and criterion_dict for a specified number of clients. It initializes unique models, optimizers using SGD, and CrossEntropyLoss criteria for each client. The function returns these dictionaries, which are essential for setting up Federated Learning scenarios.

Centralized Model Training:

Trained the centralized model using non-Federated Learning (FL) methods.

Utilized different model parameters during training to explore how they affect the model's performance.

Recorded and analyzed the accuracies achieved by the centralized model with different parameter settings.

Used parameters:

learning_rate = 0.2

numEpoch = 10

batch_size = 64

momentum = 0.2

```
----------- Centralized ( Non - Distributed ) Model ------
-------------------- Training Started -------------------
Epoch:   1 | Train accuracy:  0.8908 | Test accuracy:  0.9458
Epoch:   2 | Train accuracy:  0.9626 | Test accuracy:  0.9629
Epoch:   3 | Train accuracy:  0.9734 | Test accuracy:  0.9761
Epoch:   4 | Train accuracy:  0.9808 | Test accuracy:  0.9656
Epoch:   5 | Train accuracy:  0.9849 | Test accuracy:  0.9731
Epoch:   6 | Train accuracy:  0.9885 | Test accuracy:  0.9782
Epoch:   7 | Train accuracy:  0.9919 | Test accuracy:  0.9759
Epoch:   8 | Train accuracy:  0.9930 | Test accuracy:  0.9787
Epoch:   9 | Train accuracy:  0.9951 | Test accuracy:  0.9781
Epoch:  10 | Train accuracy:  0.9970 | Test accuracy:  0.9795
-------------------- Training finished -------------------
```

learning_rate = 0.6

numEpoch = 10

batch_size = 64

momentum = 0.2

```
---------- Centralized ( Non - Distributed ) Model ------
-------------------- Training Started ------------------
Epoch:   1 | Train accuracy:  0.8874 | Test accuracy:  0.9368
Epoch:   2 | Train accuracy:  0.9624 | Test accuracy:  0.9589
Epoch:   3 | Train accuracy:  0.9711 | Test accuracy:  0.9713
Epoch:   4 | Train accuracy:  0.9789 | Test accuracy:  0.9666
Epoch:   5 | Train accuracy:  0.9812 | Test accuracy:  0.9469
Epoch:   6 | Train accuracy:  0.9860 | Test accuracy:  0.9619
Epoch:   7 | Train accuracy:  0.9872 | Test accuracy:  0.9733
Epoch:   8 | Train accuracy:  0.9902 | Test accuracy:  0.9681
Epoch:   9 | Train accuracy:  0.9914 | Test accuracy:  0.9739
Epoch:  10 | Train accuracy:  0.9909 | Test accuracy:  0.9731
-------------------- Training finished ------------------
```

learning_rate = 0.6

numEpoch = 10

batch_size = 64

momentum = 0.4

```
---------- Centralized ( Non - Distributed ) Model ------
-------------------- Training Started ------------------
Epoch:   1 | Train accuracy:  0.8981 | Test accuracy:  0.9242
Epoch:   2 | Train accuracy:  0.9638 | Test accuracy:  0.8996
Epoch:   3 | Train accuracy:  0.9750 | Test accuracy:  0.9753
Epoch:   4 | Train accuracy:  0.9823 | Test accuracy:  0.9748
Epoch:   5 | Train accuracy:  0.9853 | Test accuracy:  0.9724
Epoch:   6 | Train accuracy:  0.9893 | Test accuracy:  0.9773
Epoch:   7 | Train accuracy:  0.9914 | Test accuracy:  0.9504
Epoch:   8 | Train accuracy:  0.9927 | Test accuracy:  0.9796
Epoch:   9 | Train accuracy:  0.9956 | Test accuracy:  0.9724
Epoch:  10 | Train accuracy:  0.9965 | Test accuracy:  0.9774
-------------------- Training finished ------------------
```

Now preparing the x-y dependency graphs for them.

Model training functions:

Implemented functions for training client models in a distributed learning setup.

Ensured that clients can participate in the learning process and contribute to model improvement.

Here is the **training function:**

```
    def train_model(self, train_loader, loss_function, optimizer):
        self.train()
        train_loss = 0.0
        correct = 0
        for data, target in train_loader:
            output = self(data)
            loss = loss_function(output, target)
            optimizer.zero_grad()
            loss.backward()
            optimizer.step()
            train_loss += loss.item()
            prediction = output.argmax(dim=1, keepdim=True)
            correct += prediction.eq(target.view_as(prediction)).sum().item()

        return train_loss / len(train_loader), correct / len(train_loader.dataset)
```

This function trains a neural network model (self) using provided training data (train_loader) and a specified loss function (loss_function) with an optimizer (optimizer). It calculates the training loss and accuracy, updating the model's parameters based on the training process. The function returns the average training loss and accuracy over the entire dataset.

Validation function:

```
    def validate_model(self, test_loader, loss_function):
        self.eval()
        test_loss = 0.0
        correct = 0
        with torch.no_grad():
            for data, target in test_loader:
                output = self(data)
                test_loss += loss_function(output, target).item()
                prediction = output.argmax(dim=1, keepdim=True)
                correct += prediction.eq(target.view_as(prediction)).sum().item()

        test_loss /= len(test_loader)
        correct /= len(test_loader.dataset)
        return test_loss, correct
```

This function evaluates a neural network model (self) using provided test data (test_loader) and a specified loss function (loss_function). It calculates the test loss and accuracy without updating the model's parameters. The function returns the average test loss and accuracy over the entire test dataset.

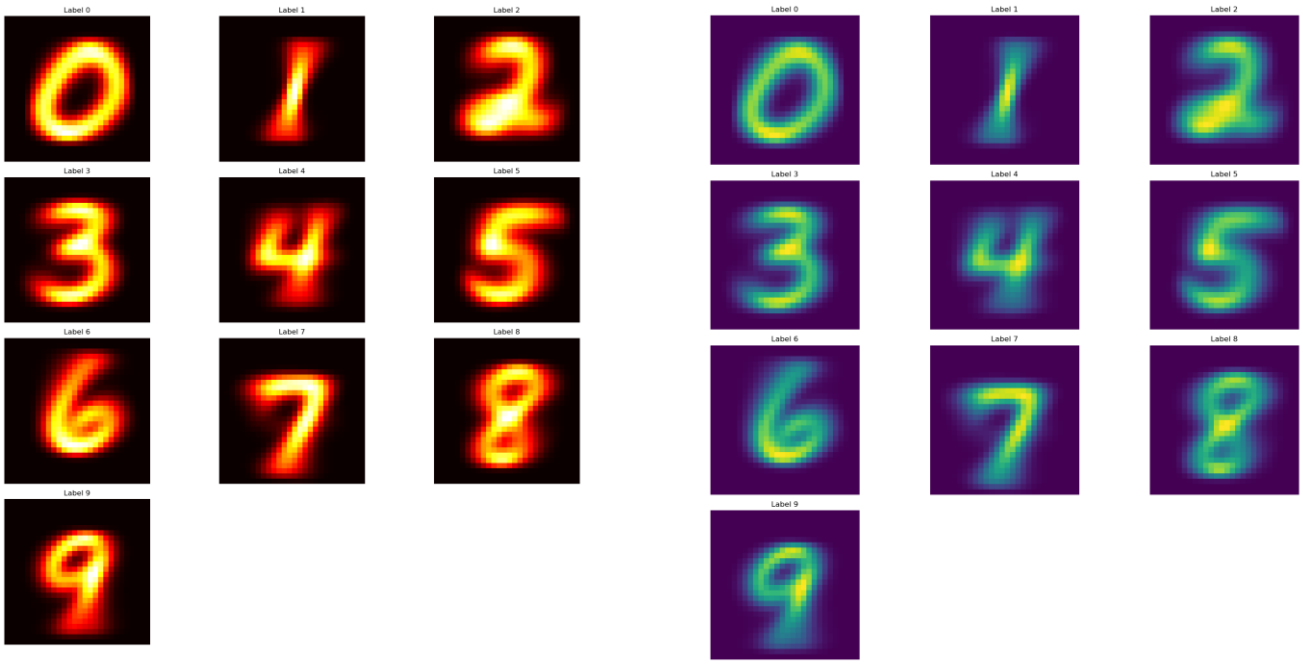
Dataset Cleaning Techniques:

Explored multiple dataset cleaning techniques to improve data quality.

Mean Image and Heatmaps:

Created mean images from the dataset to gain insights into the overall characteristics of the data.

Plotted heatmaps to visualize data distributions and patterns.



Creating a 3x3 grid of subplots, each showing the mean image of a different label from the dataset x_combined. The mean image represents the average appearance of data samples with the same label. The code randomly selects 3000 samples for each label and calculates their mean values. It then displays the mean images using a '**hot**' colormap (black to dark yellow) – first picture and '**viridis**' colormap (all colors)– second picture. The axis ticks and labels are turned off for a cleaner visualization.

Outlier Detection:

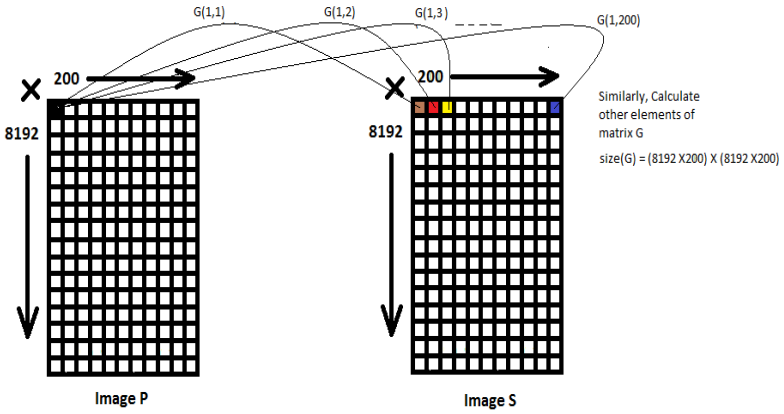
Established an outlier threshold based on the mean image and other statistical measures.

Calculated Euclidean distances between data points and the mean image to identify potential outliers.

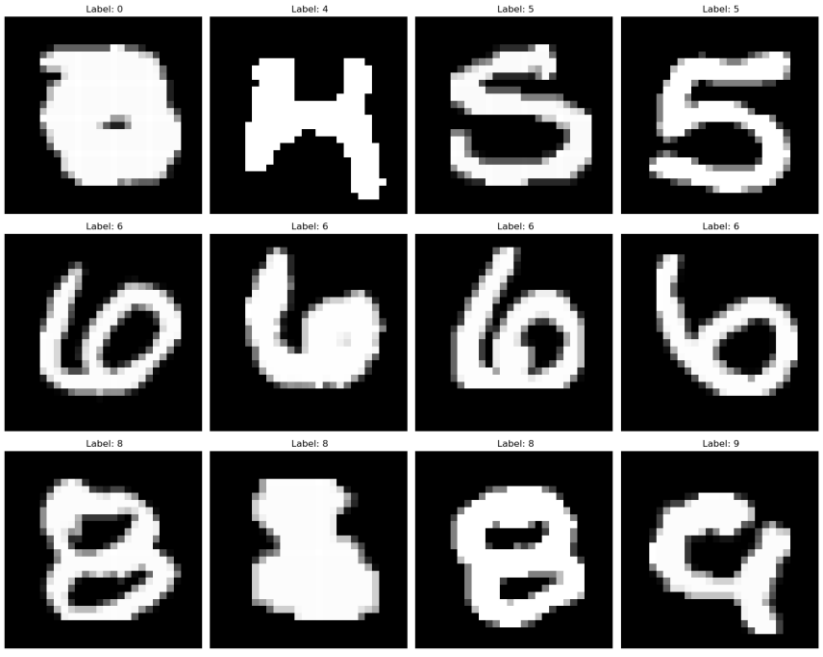
Detected images that are candidates for outliers based on the calculated Euclidean distances.

This step helps in identifying and handling potentially noisy or mislabeled data.

Euclidean distance:



Below are the samples for the outliers.



```
outlier_threshold = 10
outlier_indices = []
for label in range(10):
    indices = np.where(y_combined == label)[0]
    np.random.shuffle(indices)
    indices = indices[:3000]
    data_subset = x_combined[indices]
    mean_values = np.mean(data_subset, axis=0)
    distances = np.linalg.norm(data_subset - mean_values, axis=1)
    outlier_indices_label = np.where(distances > outlier_threshold)[0]
    outlier_indices.extend(indices[outlier_indices_label])
```

Calculating outlier data samples for each label in the dataset x_combined using the specified outlier_threshold. It finds the data samples whose distances from the mean values of their corresponding labels exceed the threshold and adds their indices to the outlier_indices list.

Indices of sample outlier images:

[41453, 24798, 25315, 36193, 29489, 25317, 8488, 59423, 8586, 18598]