Heterogeneous Federated Transfer Learning

Overview

This tutorial builds upon the previous notebook on **Transfer Learning (TL)** and takes a step further into the field of **Federated Transfer Learning (FTL)**—specifically in **heterogeneous settings** where client models and data distributions differ.

In this tutorial, you will:

- **Implement Heterogeneous FTL**, where clients use **different model architectures** (e.g., ResNet18 vs. a simple CNN).
- **Explore the impact of varying data distributions** (IID vs. non-IID) across clients.
- **Run experiments** to compare heterogeneous vs. homogeneous setups under different data conditions.
- **Evaluate results**, focusing on convergence, performance per client, and the benefits of TL in federated contexts.
- **Leverage the Flower framework** for federated orchestration and PyTorch for model training.

By the end of this tutorial, you'll have hands-on experience implementing and analyzing a **heterogeneous FTL setup**, gaining insight into how TL, FL, and data distribution intricacies interact in real-world decentralized environments.

Note: we recommend to run this notebook in colab, there we tested out and didn't have dependency conflicts. For local environments, flower tends to be harder to install without running into conflicts.

Introduction

In many real-world applications, data isn't distributed nicely or evenly. Think of a scenario in healthcare, where different hospitals collect data from different patient populations. One hospital might see mostly older patients with diabetes, while another might serve mostly young patients with asthma. This is a typical case of non-IID data – each client (or hospital) has access to a completely different kind of data. This is exactly where standard Federated Learning methods like FedAvg start to struggle.

In our setup, we try to simulate this exact heterogeneity. While my teammate shows how basic Transfer Learning works under idealized, IID conditions, I focus on a setup that’s closer to reality. We know that in practice, data is often non-IID, and worse: the clients might not even use the same model architecture, feature space, or input distributions. This is where even use the same model architecture, feature space, or input distributions. This is where Heterogeneous Federated Transfer Learning (FTL) becomes interesting.

The goal is to still allow collaboration between clients — even if their data or models are fundamentally different — by transferring only what's useful. This notebook explores how well this works when the data is not evenly spread across clients and when they operate under their own, unique data distributions. Especially with Dirichlet partitions, we see patterns that are far more realistic than the artificially extreme partitions often used in FL literature.

In this notebook we will mainly look at how to split the data in various non-IID ways using multiple FL-Strategies. Then we will evaluate how it impacts the performance of our resulting model.

In order to be comparable to the first notebook, we will use exactely the same setup. Since non-IID data creates some edge cases, we will have to adapt a few classes and just copy the others. Feel free to skip this chapter for now, since it's not important for your understanding, rather focus on the next one where we actually partition the non-IID data.

Setup

Setup similar to first notebook (small adaptations)

```
# install all the required dependencies
%pip install -q flwr[simulation] flwr-datasets[vision] torch torchvision matplo
#%pip install -U datasets fsspec
```

```
   ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 66.7/66.7 MB 12.2 MB/s eta 0:00
   ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 363.4/363.4 MB 4.3 MB/s eta 0:0
   ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 13.8/13.8 MB 23.3 MB/s eta 0:00
   ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 24.6/24.6 MB 29.4 MB/s eta 0:00
   ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 883.7/883.7 kB 25.0 MB/s eta 0:
   ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 664.8/664.8 MB 2.3 MB/s eta 0:0
   ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 211.5/211.5 MB 6.8 MB/s eta 0:0
   ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 56.3/56.3 MB 10.3 MB/s eta 0:00
   ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 127.9/127.9 MB 11.2 MB/s eta 0:
   ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 207.5/207.5 MB 5.1 MB/s eta 0:0
   ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 21.1/21.1 MB 58.9 MB/s eta 0:00
   ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 4.2/4.2 MB 65.6 MB/s eta 0:00:0
   ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 480.6/480.6 kB 29.2 MB/s eta 0:
   ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 179.3/179.3 kB 16.3 MB/s eta 0:
   ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 294.9/294.9 kB 23.9 MB/s eta 0:
   ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 2.3/2.3 MB 54.7 MB/s eta 0:00:0
   ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 236.0/236.0 kB 17.8 MB/s eta 0:
   ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 47.3/47.3 kB 4.0 MB/s eta 0:00:
   ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 540.0/540.0 kB 34.7 MB/s eta 0:
   ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 87.0/87.0 kB 9.6 MB/s eta 0:00:
ERROR: pip's dependency resolver does not currently take into account all t
gcsfs 2025.3.2 requires fsspec==2025.3.2, but you have fsspec 2024.9.0 whic
```

```
pyopenssl 24.2.1 requires cryptography<44,>=41.0.5, but you have cryptograp
grpcio-status 1.71.0 requires protobuf<6.0dev,>=5.26.1, but you have protob
pydrive2 1.21.3 requires cryptography<44, but you have cryptography 44.0.3
ydf 0.12.0 requires protobuf<6.0.0,>=5.29.1, but you have protobuf 4.25.8 w
```

```
%pip uninstall -y cryptography
%pip install cryptography==41.0.7 --force-reinstall
```

```
Found existing installation: cryptography 44.0.3
Uninstalling cryptography-44.0.3:
  Successfully uninstalled cryptography-44.0.3
Collecting cryptography==41.0.7
  Downloading cryptography-41.0.7-cp37-abi3-manylinux_2_28_x86_64.whl.metad
Collecting cffi>=1.12 (from cryptography==41.0.7)
  Downloading cffi-1.17.1-cp311-cp311-manylinux_2_17_x86_64.manylinux2014_x
Collecting pycparser (from cffi>=1.12->cryptography==41.0.7)
  Downloading pycparser-2.22-py3-none-any.whl.metadata (943 bytes)
Downloading cryptography-41.0.7-cp37-abi3-manylinux_2_28_x86_64.whl (4.4 MB
   ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 4.4/4.4 MB 27.7 MB/s eta 0:00:0
Downloading cffi-1.17.1-cp311-cp311-manylinux_2_17_x86_64.manylinux2014_x86
   ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 467.2/467.2 kB 18.1 MB/s eta 0:
Downloading pycparser-2.22-py3-none-any.whl (117 kB)
   ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 117.6/117.6 kB 7.6 MB/s eta 0:0
Installing collected packages: pycparser, cffi, cryptography
  Attempting uninstall: pycparser
    Found existing installation: pycparser 2.22
    Uninstalling pycparser-2.22:
      Successfully uninstalled pycparser-2.22
  Attempting uninstall: cffi
    Found existing installation: cffi 1.17.1
    Uninstalling cffi-1.17.1:
      Successfully uninstalled cffi-1.17.1
  Attempting uninstall: cryptography
    Found existing installation: cryptography 3.4.8
    Uninstalling cryptography-3.4.8:
      Successfully uninstalled cryptography-3.4.8
ERROR: pip's dependency resolver does not currently take into account all t
flwr 1.18.0 requires cryptography<45.0.0,>=44.0.1, but you have cryptograph
Successfully installed cffi-1.17.1 cryptography-41.0.7 pycparser-2.22
```

```
#do all the needed imports
from collections import OrderedDict
from typing import List, Tuple

import matplotlib.pyplot as plt
import numpy as np
import torch
import torch.nn as nn
import torch.nn.functional as F
import torchvision.models as models
import torchvision.transforms as transforms
from datasets.utils.logging import disable_progress_bar
from torch.utils.data import DataLoader

import flwr
```

```
from flwr.client import Client, ClientApp, NumPyClient
from flwr.common import Metrics, Context
from flwr.server import ServerApp, ServerConfig, ServerAppComponents
from flwr.server.strategy import FedAvg
from flwr.simulation import run_simulation
from flwr_datasets import FederatedDataset
from flwr_datasets.partitioner import DirichletPartitioner
from flwr.server.strategy import FedProx
from torchvision import transforms
from torch.utils.data import DataLoader, random_split
from flwr_datasets import FederatedDataset

from IPython.display import Image, display


DEVICE = "cuda" if torch.cuda.is_available() else "cpu"
print(f"Training on {DEVICE}")
print(f"Flower {flwr.__version__} / PyTorch {torch.__version__}")
disable_progress_bar()
```

```
Training on cuda
Flower 1.18.0 / PyTorch 2.6.0+cu124
```

```
#import the pretrained model (used for TL)
class Net(nn.Module):
    def __init__(self, num_classes=10) -> None:
        super(Net, self).__init__()
        # Load a pre-trained ResNet18 model
        self.model = models.resnet18()

        # Replace the final fully connected layer to match your output size
        in_features = self.model.fc.in_features
        self.model.fc = nn.Linear(in_features, num_classes)

    def forward(self, x: torch.Tensor) -> torch.Tensor:
        return self.model(x)

def get_resnet18(num_classes=10, pretrained=False):
    """Create a ResNet18 model with specified number of output classes."""
    model = models.resnet18()

    # Modify the final fully connected layer to match our number of classes
    model.fc = nn.Linear(model.fc.in_features, num_classes)

    return model

def set_parameters(net: nn.Module, parameters: List[np.ndarray]) -> None:
    state_dict = net.state_dict()
    new_state_dict = OrderedDict()
    for (key, old_tensor), param in zip(state_dict.items(), parameters):
        new_state_dict[key] = torch.tensor(param, dtype=old_tensor.dtype)
    net.load_state_dict(new_state_dict, strict=True)
```

```
def get_parameters(net: nn.Module) -> List[np.ndarray]:
    return [val.cpu().numpy() for val in net.state_dict().values()]



#define train/test functions
def train(net, trainloader, epochs: int, verbose=False):
    """Train the network on the training set."""
    criterion = torch.nn.CrossEntropyLoss()
    optimizer = torch.optim.Adam(net.parameters())
    net.train()
    for epoch in range(epochs):
        correct, total, epoch_loss = 0, 0, 0.0
        for batch in trainloader:
            images, labels = batch["img"].to(DEVICE), batch["label"].to(DEVICE)
            optimizer.zero_grad()
            outputs = net(images)
            loss = criterion(outputs, labels)
            loss.backward()
            optimizer.step()
            # Metrics
            epoch_loss += loss
            total += labels.size(0)
            correct += (torch.max(outputs.data, 1)[1] == labels).sum().item()
        epoch_loss /= len(trainloader.dataset)
        epoch_acc = correct / total
        if verbose:
          ### write into file
            print(f"Epoch {epoch+1}: train loss {epoch_loss}, accuracy {epoch_a


def test(net, testloader):
    """Evaluate the network on the entire test set."""
    criterion = torch.nn.CrossEntropyLoss()
    correct, total, loss = 0, 0, 0.0
    net.eval()
    with torch.no_grad():
        for batch in testloader:
            images, labels = batch["img"].to(DEVICE), batch["label"].to(DEVICE)
            outputs = net(images)
            loss += criterion(outputs, labels).item()
            _, predicted = torch.max(outputs.data, 1)
            total += labels.size(0)
            correct += (predicted == labels).sum().item()
    loss /= len(testloader.dataset)
    accuracy = correct / total
    return loss, accuracy


#now we define a Heterogeneous FlowerClient (evaluate method is adapted)
class FlowerClientHeterogeneous(NumPyClient):
    def __init__(self, net, trainloader, valloader):
        self.net = net
```

```
        self.trainloader = trainloader
        self.valloader = valloader

    def get_parameters(self, config):
        return get_parameters(self.net)

    def fit(self, parameters, config):
        try:
          set_parameters(self.net, parameters)
          train(self.net, self.trainloader, epochs=1)
          return get_parameters(self.net), len(self.trainloader), {}
        except Exception as e:
            print(f"Client {self.client_id} failed during fit: {e}")
            raise
    def evaluate(self, parameters, config):
        if self.valloader is None:
          return 0.0, {}  # Or any default loss/metric

        set_parameters(self.net, parameters)
        loss, accuracy = test(self.net, self.valloader)
        return float(loss), len(self.valloader), {"accuracy": float(accuracy)}
```

Also we will define the evaluation function used for metrics(similar to first notebook):

```
def weighted_average(metrics: List[Tuple[int, Metrics]]) -> Metrics:
    # Multiply accuracy of each client by number of examples used
    accuracies = [num_examples * m["accuracy"] for num_examples, m in metrics]
    examples = [num_examples for num_examples, _ in metrics]

    # Aggregate and return custom metric (weighted average)
    return {"accuracy": sum(accuracies) / sum(examples)}
```

```
# Specify the resources each of your clients need
# By default, each client will be allocated 1x CPU and 0x GPUs
backend_config = {"client_resources": {"num_cpus": 1, "num_gpus": 0.0}}

# When running on GPU, assign an entire GPU for each client
if DEVICE == "cuda":
    backend_config = {"client_resources": {"num_cpus": 1, "num_gpus": 0.1}}
    # Refer to our Flower framework documentation for more details about Flower
    # and how to set up the `backend_config`
```

Paritioning of non-IID data Setup

First of all we define some parameters:

```
NUM_CLIENTS = 5
BATCH_SIZE = 16
```

```
BATCH_SIZE = 16
```

In this section we are defining all the necessary functions for partitioning the data in a non-IID way. Also we define functions used to print out and visualize the created distributions, in order to make it more understandable.

In order to test out different partitioners, we adapt load_datasets so that we can pass a partitioner as an argument:

```
def load_datasets(partition_id: int, fds: FederatedDataset, partitioner):
    """Load client-specific CIFAR-10 data with customizable non-IID partitionin

    Args:
        partition_id (int): ID of the client partition.
        dataset: A Flower federated dataset, thats already partitioned

    Returns:
        Tuple of (trainloader, valloader, testloader).
    """
    try:
      partition = fds.load_partition(partition_id=partition_id)
    except Exception as e:
      print(f"[Client {partition_id}] Failed to load partition: {e}")
      raise RuntimeError(f"Client {partition_id} failed to load partition: {e}'

    # Define standard preprocessing transforms
    transform = transforms.Compose([
        transforms.ToTensor(),
        transforms.Normalize((0.5, 0.5, 0.5), (0.5, 0.5, 0.5)),
    ])

    def apply_transforms(batch):
        batch["img"] = [transform(img) for img in batch["img"]]
        return batch

    # Apply transforms to partitioned data
    partition = partition.with_transform(apply_transforms)

    # Split into 80% train, 20% val
    total_size = len(partition)
    val_size = int(0.2 * total_size)
    train_size = total_size - val_size
    train_dataset, val_dataset = random_split(partition, [train_size, val_size]

    # DataLoaders
    trainloader = DataLoader(train_dataset, batch_size=BATCH_SIZE, shuffle=True
    valloader = DataLoader(val_dataset, batch_size=BATCH_SIZE, shuffle=False)

    # Load and transform centralized test set
    testset = fds.load_split("test").with_transform(apply_transforms)
    testloader = DataLoader(testset, batch_size=BATCH_SIZE, shuffle=False)
```

```
    return trainloader, valloader, testloader



def model_fn():
    return Net(num_classes=10).to(DEVICE)
```

Since load_datasets is called in the client_fn, we have to adapt this one as well. However, client_fn is predefined with a signature and we cannot just add a new argument to it, therefore we use the factory-pattern to define the function accordingly to a given partitioner:

```
#client_fn factory
def create_client_fn(fds, partitioner):
    def client_fn(context: Context) -> Client:
        net = get_resnet18(num_classes=10).to(DEVICE)
        partition_id = context.node_config["partition-id"]
        trainloader, valloader, testloader = load_datasets(partition_id, fds=fd
        return FlowerClientHeterogeneous(
            net=net,
            trainloader=trainloader,
            valloader=valloader,
        ).to_client()
    return client_fn
```

Also for the server_fn we create a Factory, so that we can use it with different kinds of FL-Strategies:

```
from flwr.common import parameters_to_ndarrays

def get_evaluate_fn(model, metrics_log, dataloader):
    def evaluate(server_round, parameters, config):
        # `parameters` is already a list of NumPy arrays (after conversion by F
        weights = parameters  # Keine weitere Umwandlung nötig

        # Gewichte ins Modell laden
        state_dict = dict(zip(model.state_dict().keys(), [torch.tensor(w) for w
        model.load_state_dict(state_dict, strict=True)

        correct, total = 0, 0
        with torch.no_grad():
            for batch in dataloader:
                x = batch["img"]
                y = batch["label"]

                # Falls x oder y Listen sind, in Tensoren umwandeln
                if isinstance(x, list):
```

```
            if isinstance(x, list):
                x = torch.stack(x)
            if isinstance(y, list):
                y = torch.tensor(y)

            x, y = x.to(DEVICE), y.to(DEVICE)
            preds = model(x).argmax(dim=1)
            correct += (preds == y).sum().item()
            total += y.size(0)

        accuracy = correct / total
        print(f"📊 Server round {server_round} accuracy: {accuracy:.4f}")
        metrics_log.append(accuracy)
        return 0.0, {"accuracy": accuracy}

    return evaluate
```

Make it so that multiple types of the offered Flower Strategies can be used:

```
from torch.utils.data import DataLoader
from torchvision import transforms

def strategy_fn(model, metrics_log, fds, strategy_cls, **strategy_kwargs):
    # Define transforms (same as in your training/val loader)
    transform = transforms.Compose([
        transforms.ToTensor(),
        transforms.Normalize((0.5, 0.5, 0.5), (0.5, 0.5, 0.5)),
    ])

    def apply_transforms(batch):
        batch["img"] = [transform(img) for img in batch["img"]]
        return batch

    # Load centralized test split and apply transforms
    testset = fds.load_split("test").with_transform(apply_transforms)
    testloader = DataLoader(testset, batch_size=32, shuffle=False)

    # Get evaluation function using testloader
    evaluate_fn = get_evaluate_fn(model, metrics_log, testloader)

    # Instantiate and return the strategy with the evaluate_fn and other kwargs
    return strategy_cls(evaluate_fn=evaluate_fn, **strategy_kwargs)



def server_fn_factory(strategy_fn, model_fn, metrics_log, fds, strategy_cls, nu
    def server_fn(context: Context) -> ServerAppComponents:
        model = model_fn()
        strategy = strategy_fn(model, metrics_log, fds, strategy_cls, **strateg
        config = ServerConfig(num_rounds=num_rounds, round_timeout=120)
        return ServerAppComponents(strategy=strategy, config=config)
```

```
    return server_fn
```

```
import os
import matplotlib.pyplot as plt

def plot_accuracy(metrics_log, name, save_dir):
    plt.plot(metrics_log, marker='o')
    plt.title(f"Accuracy over Rounds: {name}")
    plt.xlabel("Round")
    plt.ylabel("Accuracy")
    plt.grid(True)

    # Save the plot
    os.makedirs(save_dir, exist_ok=True)
    plt.savefig(os.path.join(save_dir, f"{name}_accuracy.png"))

    # Show the plot in interactive environments (like Colab or Jupyter)
    plt.show()

    # Close to free memory if used repeatedly
    plt.close()
```

Also we define a function to create a FederatedDataset for a specific partitioner using Cifar-10:

```
def create_federated_dataset(partitioner) -> FederatedDataset:
    """Create a FederatedDataset with the specified partitioner."""
    return FederatedDataset(
        dataset="uoft-cs/cifar10",
        partitioners={"train": partitioner}
    )
```

Now we define a function that let's us see how the Federated-Dataset is distributed (and optionally gives us a plot of it):

```
from collections import Counter
import pandas as pd
import matplotlib.pyplot as plt
import numpy as np

def print_partition_statistics(fds, num_partitions: int, plot: bool = True):
    """Prints and plots class distribution per client."""
    print("Class distribution per client partition:\n")

    class_counts = []
    all_labels = list(range(10))  # CIFAR-10 has 10 classes
```

```
    for pid in range(num_partitions):
        partition = fds.load_partition(partition_id=pid)

        # Handle both DatasetDict and Dataset
        dataset = partition["train"] if isinstance(partition, dict) and "train'
        labels = dataset["label"]
        counter = Counter(labels)
        row = [counter.get(label, 0) for label in all_labels]
        class_counts.append(row)

    df = pd.DataFrame(class_counts, columns=[f"Class {i}" for i in all_labels])
    df.index = [f"Client {i}" for i in range(num_partitions)]

    # Print horizontally compact table
    print(df.to_string())

    if plot:
        # Prepare data
        ind = np.arange(num_partitions)  # Client positions
        width = 0.08  # Width of each bar

        fig, ax = plt.subplots(figsize=(14, 6))

        # Stack bars side-by-side per client
        for i, class_label in enumerate(df.columns):
            ax.bar(ind + i * width, df[class_label], width, label=class_label)

        # Add client boundaries
        for i in range(num_partitions):
            plt.axvline(i + 0.5, color="gray", linestyle="--", alpha=0.3)

        ax.set_xticks(ind + width * 5)
        ax.set_xticklabels([f"Client {i}" for i in range(num_partitions)])
        ax.set_xlabel("Client")
        ax.set_ylabel("Number of Samples")
        ax.set_title("Class Distribution per Client")
        ax.legend(title="Class", bbox_to_anchor=(1.05, 1), loc="upper left")
        plt.tight_layout()
        plt.grid(axis="y", linestyle="--", alpha=0.5)
        plt.show()
```

Different Partitioners

Now that we have the basic setup done, we will provide a few examples on popular Partitioners, however there is a full list of them (https://flower.ai/docs/datasets/tutorial-use-partitioners.html). You can easily test them out by just redifing the partitioner in the examples below.

IID Partitioner for comparison

IID-Partitioner for comparison

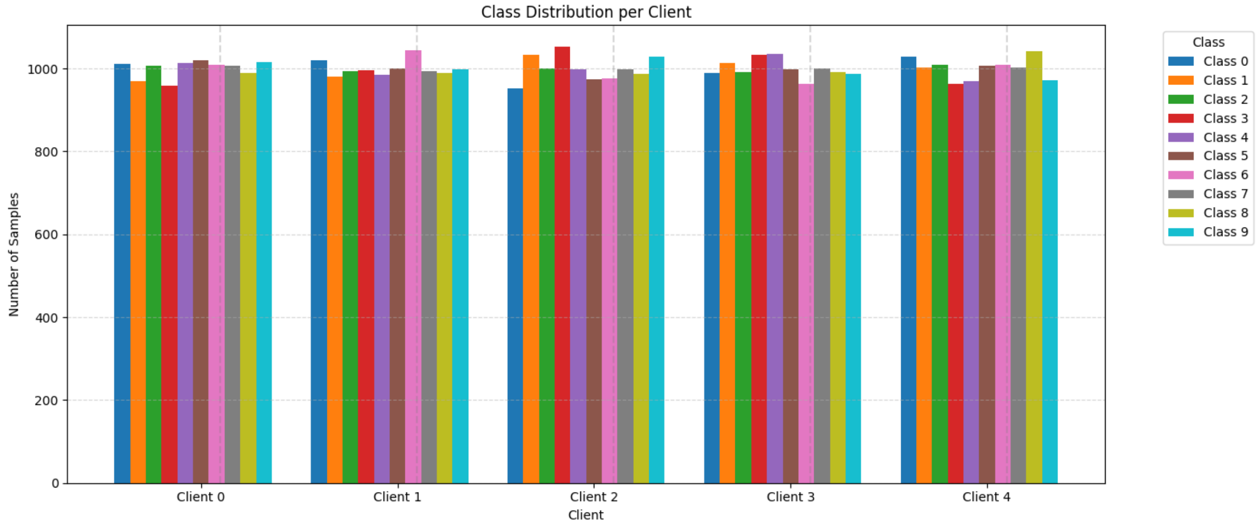
This partitioner should just divide our dataset equally among clients. It still uses random sampling, therefore it doesn't give us a completely equal split, however we can consider it as IID.

```
from flwr_datasets.partitioner import IidPartitioner

iid_partitioner = IidPartitioner(num_partitions=NUM_CLIENTS)
iid_fds = create_federated_dataset(iid_partitioner)
```

Let's take a look at the distribution:

```
print_partition_statistics(iid_fds, num_partitions=NUM_CLIENTS, plot=True)
```

```
Class distribution per client partition:

/usr/local/lib/python3.11/dist-packages/huggingface_hub/utils/_auth.py:94:
The secret `HF_TOKEN` does not exist in your Colab secrets.
To authenticate with the Hugging Face Hub, create a token in your settings
You will be able to reuse this secret in all of your notebooks.
Please note that authentication is recommended but still optional to access
  warnings.warn(
          Class 0  Class 1  Class 2  Class 3  Class 4  Class 5  Class 6  Cl
Client 0     1011      969     1007      958     1014     1021     1010
Client 1     1020      981      993      996      984     1001     1043
Client 2      952     1033     1000     1052      998      973      976
Client 3      989     1014      992     1032     1035      999      962
Client 4     1028     1003     1008      962      969     1006     1009
```



PathologicalPartitioner

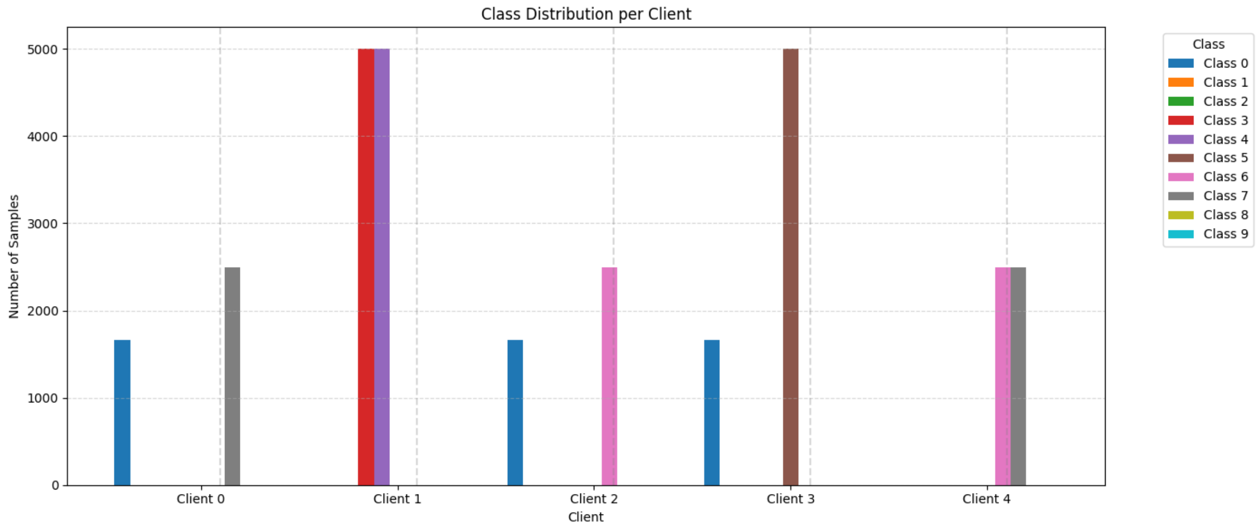
This partitioner will create a "radical" non-IID Dataset, where each partition only gets a specific amount of classes. This amount is specified by the parameter num_classes_per_partition.

So let's create a FederatedDataset with num_classes_per_partition=2 (extremely non-IID)

```
from flwr_datasets.partitioner import PathologicalPartitioner

partitioner_pathological_2 = PathologicalPartitioner(num_partitions=NUM_CLIENTS
fds_pathological_2 = create_federated_dataset(partitioner_pathological_2)
```

```
print_partition_statistics(fds_pathological_2, num_partitions=NUM_CLIENTS, plot
```

```
Class distribution per client partition:

/usr/local/lib/python3.11/dist-packages/flwr_datasets/partitioner/pathologi
  warnings.warn(
          Class 0  Class 1  Class 2  Class 3  Class 4  Class 5  Class 6  Cl
Client 0     1667        0        0        0        0        0        0
Client 1        0        0        0     5000     5000        0        0
Client 2     1667        0        0        0        0        0     2500
Client 3     1666        0        0        0        0     5000        0
Client 4        0        0        0        0        0        0     2500
```



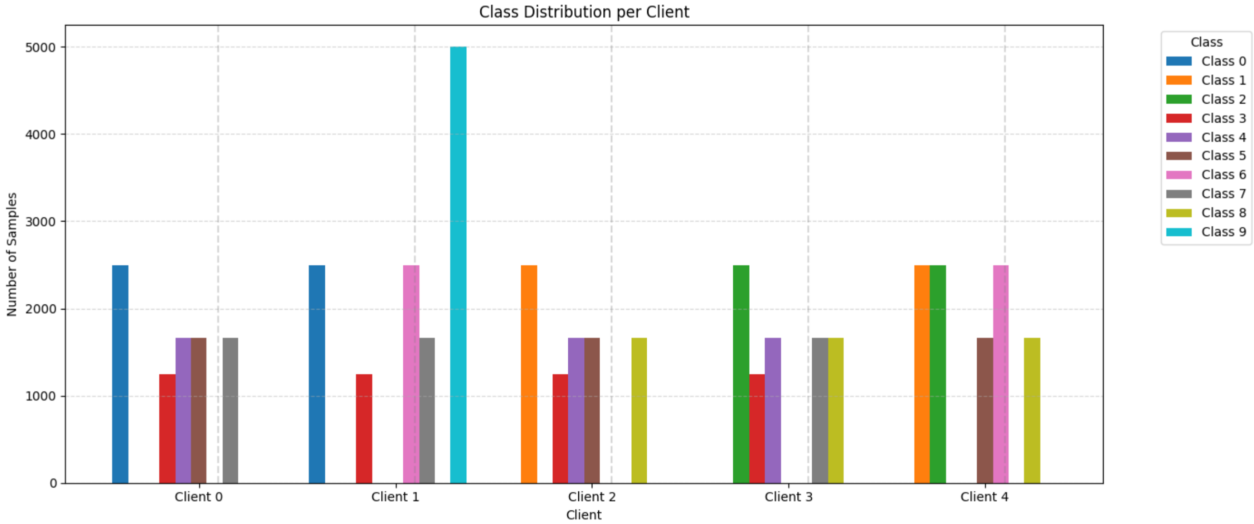
And let's also create a FederatedDataset with num_classes_per_partition=5 (still non-IID but more realistic)

```
from flwr_datasets.partitioner import PathologicalPartitioner

partitioner_pathological_5 = PathologicalPartitioner(num_partitions=NUM_CLIENTS
fds_pathological_5 = create_federated_dataset(partitioner_pathological_5)
```

```
print_partition_statistics(fds_pathological_5, num_partitions=NUM_CLIENTS, plot
```

Class distribution per client partition:

	Class 0	Class 1	Class 2	Class 3	Class 4	Class 5	Class 6	Cl
Client 0	2500	0	0	1250	1667	1667	0	
Client 1	2500	0	0	1250	0	0	2500	
Client 2	0	2500	0	1250	1667	1667	0	
Client 3	0	0	2500	1250	1666	0	0	
Client 4	0	2500	2500	0	0	1666	2500	



DirichletPartitioner

As a first non-IID Partitioner, we will use the DirichletPartitioner (https://flower.ai/docs/datasets/ref-api/flwr_datasets.partitioner.DirichletPartitioner.html#flwr_datasets.partitioner.DirichletPartitioner)

The most important parameter for this distribution is the alpha-parameter:

For low alpha (e.g. 0.1) it makes each client biased towards a few classes, highly non-IId.

For high alpha (e.g. 10) it makes a more balanced class distribution (closer to IID)

Let's first create a really non-IID partitioning (alpha=0.1)

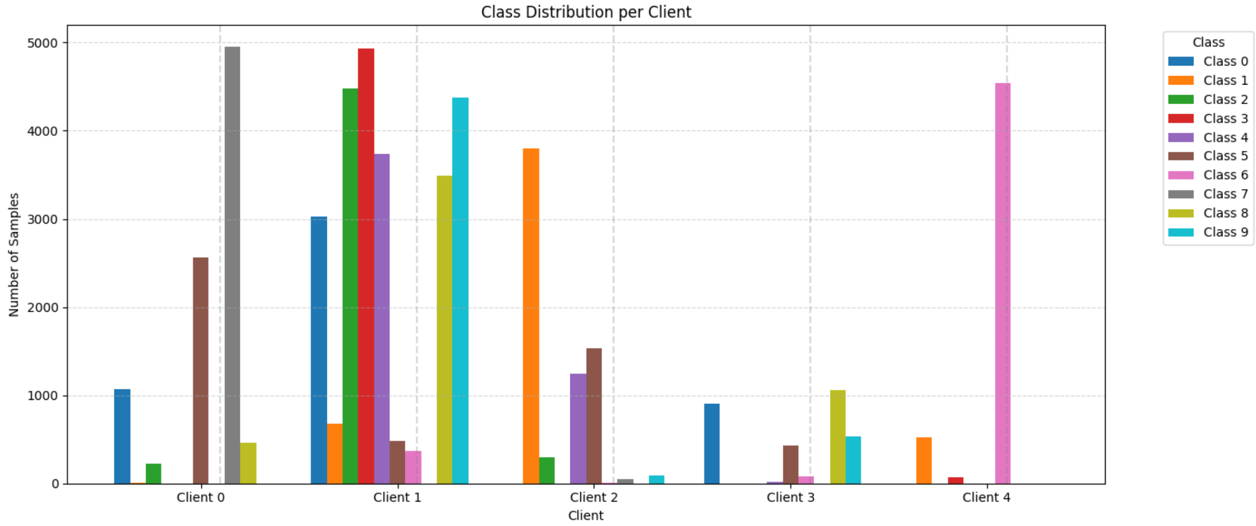
```
partitioner_low_alpha = DirichletPartitioner(num_partitions=NUM_CLIENTS, alpha=
fds_low_alpha = create_federated_dataset(partitioner_low_alpha)
```

```
print_partition_statistics(fds_low_alpha, num_partitions=NUM_CLIENTS, plot=True
```

Class distribution per client partition:

	Class 0	Class 1	Class 2	Class 3	Class 4	Class 5	Class 6	Cl

Client 0	1070	6	224	0	0	2562	1
Client 1	3023	675	4482	4930	3735	479	369
Client 2	0	3795	292	0	1248	1528	8
Client 3	906	2	1	0	16	430	83
Client 4	1	522	1	70	1	1	4539



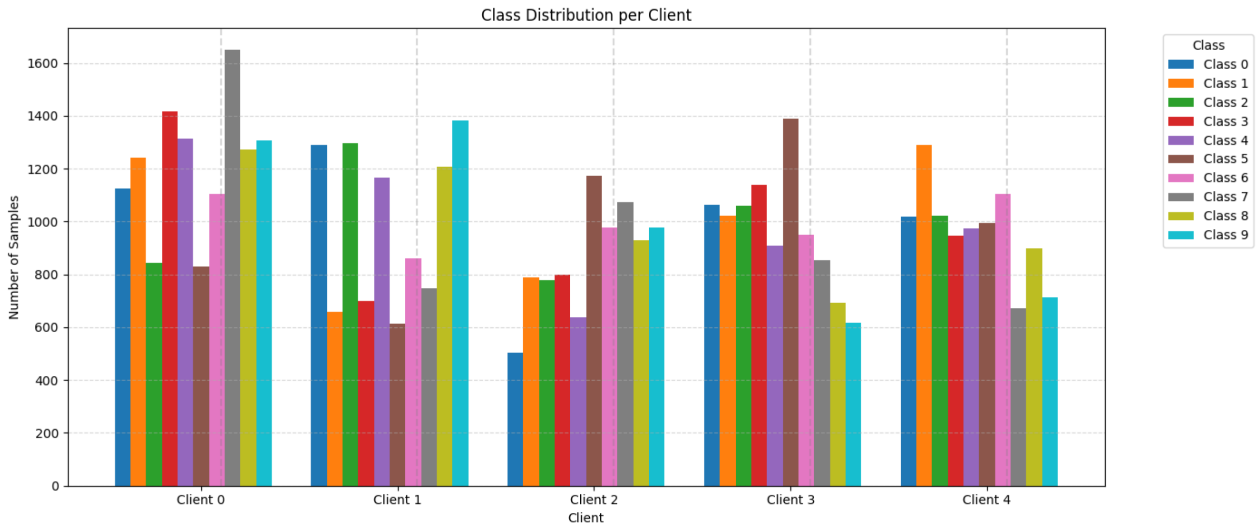
For comparison, we will also create one FederatedDataset with high_alpha

```
partitioner_high_alpha = DirichletPartitioner(num_partitions=NUM_CLIENTS, alpha
fds_high_alpha = create_federated_dataset(partitioner_high_alpha)
```

```
print_partition_statistics(fds_high_alpha, num_partitions=NUM_CLIENTS, plot=Tru
```

Class distribution per client partition:

	Class 0	Class 1	Class 2	Class 3	Class 4	Class 5	Class 6	Cl
Client 0	1125	1240	844	1417	1313	831	1106	
Client 1	1289	659	1297	700	1165	613	862	
Client 2	505	789	779	798	639	1173	977	
Client 3	1064	1023	1059	1139	910	1389	949	
Client 4	1017	1289	1021	946	973	994	1106	



Testing/Evaluating different Partitioners/Strategies

Now that we have a nice and modular setup and showed how different Partitioners split up data, we evaluate how well they perform.

For this, we will test out always how well FedAvg works and then compare it to FedProx. FedProx has been shown to work really well on highly skewed data, which could be beneficial in our Non-IID cases.

First of all, we define a run_experiment Function, in order to call a simulation with specific setup in one line:

```
def run_experiment(fds, model_fn, strategy_fn, experiment_name, backend_config,
    print(f"\n🚀 Running experiment: {experiment_name}")
    metrics_log = []

    client_fn = create_client_fn(fds, partitioner)
    client = ClientApp(client_fn=client_fn)

    server_fn = server_fn_factory(strategy_fn, model_fn, metrics_log, fds, stra
    server = ServerApp(server_fn=server_fn)

    run_simulation(
        server_app=server,
        client_app=client,
        num_supernodes=num_clients,
        backend_config=backend_config,
    )

    if metrics_log:
        plot_accuracy(metrics_log, experiment_name, save_dir)

    print(f"✅ Finished: {experiment_name}")
    return metrics_log
```

Now that we defined this function, we can test out all kinds of Model, Aggregation Strategy and Data combinations. For this we loop over all our fds and then use both strategies on it.

```
from flwr.server.strategy import FedAvg, FedProx

# List of federated datasets
fds_list = [
    ("base_IID", iid_fds, iid_partitioner),
```

```
    ("low_alpha_dirichlet", fds_low_alpha, partitioner_low_alpha),
    ("high_alpha_dirichlet", fds_high_alpha, partitioner_high_alpha),
    ("pathological_2", fds_pathological_2, partitioner_pathological_2),
    ("pathological_5", fds_pathological_5, partitioner_pathological_5),
]
strategies = [FedAvg, FedProx]

NUM_ROUNDS=8

# Loop through each dataset and each strategy
for fds_name, fds, partitioner in fds_list:
    for strategy_cls in strategies:
        experiment_name = f"{fds_name}_{strategy_cls.__name__.lower()}_experime

        if strategy_cls == FedProx:
          run_experiment(
              fds=fds,
              partitioner=partitioner,
              model_fn=model_fn,
              strategy_fn=strategy_fn,
              experiment_name=experiment_name,
              backend_config=backend_config,
              num_clients=NUM_CLIENTS,
              strategy_cls=strategy_cls,
              save_dir="results",
              num_rounds=8,
              proximal_mu= 0.01)

        else:
          run_experiment(
            fds=fds,
            partitioner=partitioner,
            model_fn=model_fn,
            strategy_fn=strategy_fn,
            experiment_name=experiment_name,
            backend_config=backend_config,
            num_clients=NUM_CLIENTS,
            strategy_cls=strategy_cls,
            save_dir="results",
            num_rounds=8
          )
```

```
DEBUG:flwr:Asyncio event loop already running.

🚀 Running experiment: base_IID_fedavg_experiment
INFO :      Starting Flower ServerApp, config: num_rounds=8, round_timeout=
INFO :
INFO :      [INIT]
INFO :      Requesting initial parameters from one random client
(pid=3867) 2025-06-05 12:39:49.803074: E external/local_xla/xla/stream_exec
(pid=3867) WARNING: All log messages before absl::InitializeLog() is called
(pid=3867) E0000 00:00:1749127189.836519    3867 cuda_dnn.cc:8310] Unable t
(pid=3867) E0000 00:00:1749127189.848270    3867 cuda_blas.cc:1418] Unable
(ClientAppActor pid=3868) /usr/local/lib/python3.11/dist-packages/jupyter_c
(ClientAppActor pid=3868) given by the platformdirs library.  To remove thi
```

```
(ClientAppActor pid=3868) see the appropriate new directories, set the envi
(ClientAppActor pid=3868) `JUPYTER_PLATFORM_DIRS=1` and then run `jupyter -
(ClientAppActor pid=3868) The use of platformdirs will be the default in `j
(ClientAppActor pid=3868)   from jupyter_core.paths import jupyter_data_dir
(pid=3868) 2025-06-05 12:39:49.826355: E external/local_xla/xla/stream_exec
(pid=3868) WARNING: All log messages before absl::InitializeLog() is called
(pid=3868) E0000 00:00:1749127189.860762    3868 cuda_dnn.cc:8310] Unable t
(pid=3868) E0000 00:00:1749127189.870813    3868 cuda_blas.cc:1418] Unable 
INFO :      Received initial parameters from one random client
INFO :      Starting evaluation of initial global parameters
INFO :      initial parameters (loss, other metrics): 0.0, {'accuracy': 0.1
INFO :
INFO :      [ROUND 1]
INFO :      configure_fit: strategy sampled 5 clients (out of 5)
📊 Server round 0 accuracy: 0.1029
(ClientAppActor pid=3867) /usr/local/lib/python3.11/dist-packages/jupyter_c
(ClientAppActor pid=3867) given by the platformdirs library.  To remove thi
(ClientAppActor pid=3867) see the appropriate new directories, set the envi
(ClientAppActor pid=3867) `JUPYTER_PLATFORM_DIRS=1` and then run `jupyter -
(ClientAppActor pid=3867) The use of platformdirs will be the default in `j
(ClientAppActor pid=3867)   from jupyter_core.paths import jupyter_data_dir
INFO :      aggregate_fit: received 5 results and 0 failures
WARNING :   No fit_metrics_aggregation_fn provided
INFO :      fit progress: (1, 0.0, {'accuracy': 0.3735}, 59.15123558999994)
INFO :      configure_evaluate: strategy sampled 5 clients (out of 5)
📊 Server round 1 accuracy: 0.3735
INFO :      aggregate_evaluate: received 5 results and 0 failures
WARNING :   No evaluate_metrics_aggregation_fn provided
INFO :
INFO :      [ROUND 2]
INFO :      configure_fit: strategy sampled 5 clients (out of 5)
INFO :      aggregate_fit: received 5 results and 0 failures
INFO :      fit progress: (2, 0.0, {'accuracy': 0.4593}, 121.30504739399998
INFO :      configure_evaluate: strategy sampled 5 clients (out of 5)
📊 Server round 2 accuracy: 0.4593
INFO :      aggregate_evaluate: received 5 results and 0 failures
INFO :
INFO :      [ROUND 3]
INFO :      configure_fit: strategy sampled 5 clients (out of 5)
INFO :      aggregate_fit: received 5 results and 0 failures
INFO :      fit progress: (3, 0.0, {'accuracy': 0.5185}, 185.94308447499998
INFO :      configure_evaluate: strategy sampled 5 clients (out of 5)
📊 Server round 3 accuracy: 0.5185
INFO :      aggregate_evaluate: received 5 results and 0 failures
INFO :
INFO :      [ROUND 4]
INFO :      configure_fit: strategy sampled 5 clients (out of 5)
INFO :      aggregate_fit: received 5 results and 0 failures
INFO :      fit progress: (4, 0.0, {'accuracy': 0.569}, 247.24495441600004)
INFO :      configure_evaluate: strategy sampled 5 clients (out of 5)
📊 Server round 4 accuracy: 0.5690
INFO :      aggregate_evaluate: received 5 results and 0 failures
INFO :
INFO :      [ROUND 5]
INFO :      configure_fit: strategy sampled 5 clients (out of 5)
INFO :      aggregate_fit: received 5 results and 0 failures
INFO :      fit progress: (5, 0.0, {'accuracy': 0.6018}, 310.62665868)
INFO :      configure_evaluate: strategy sampled 5 clients (out of 5)
📊 Server round 5 accuracy: 0.6018
```

Interpretation of Results

After doing a lot of test-runs which are quite heavy computation, we decided to group the results here to get a good overview.

We used 5 Clients for our experiment, with 8 rounds in total. On the client fit function we just compute the loss once each time (epochs=1). In reality this parameters should be tested out and especially epochs should be set higher.

Let's look at how well our different Strategies FedAvg and FedProx performed on the different

Data-Distributions.

NOTE: In this notebook the files get saved to a directory called results. However the computation took us multiple hours on Google Colab using the GPU backend, therefore you can also get the required plots on github and upload them into Colab or your local directory to be able to interpret the results.

Github: (https://github.com/aherzinger/federated_transfer_learning/blob/main/Heterogeneous_FTL.ipynb)

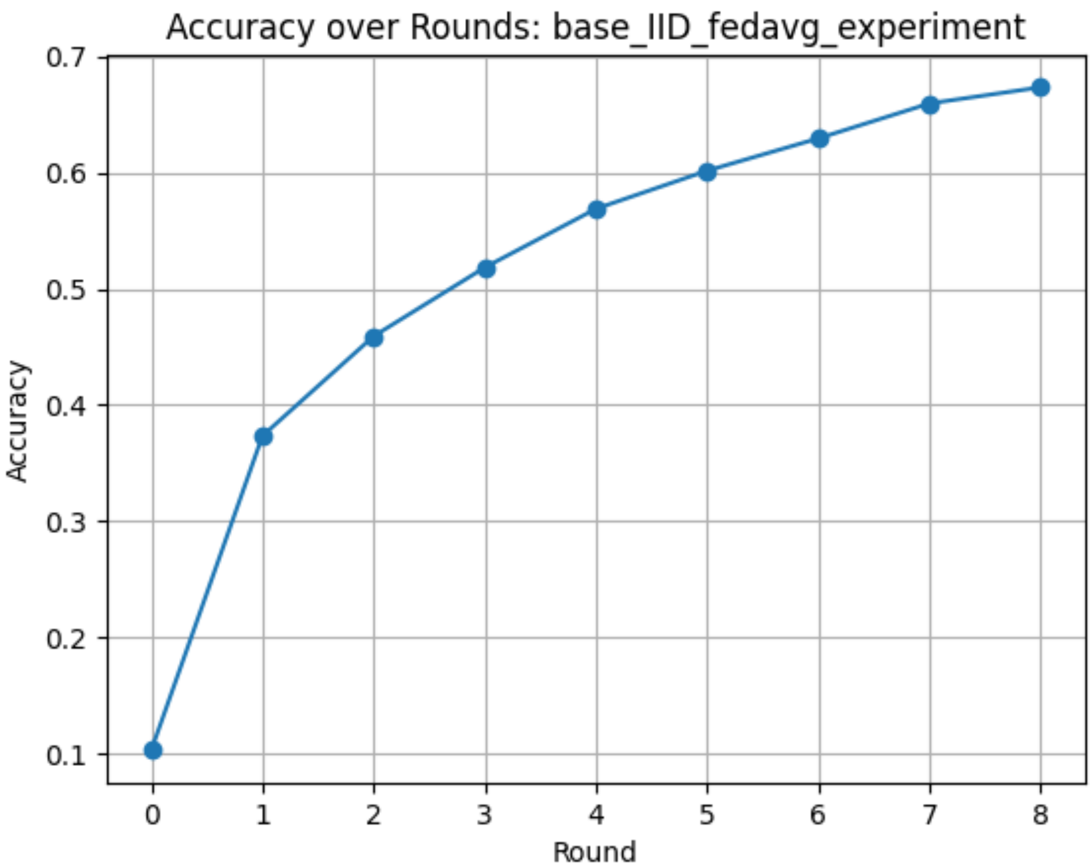
Results on IID-Data

For reference, we first ran our experiments on our IID FederatedDataset. With both strategies we reached about 68% accuracy after 8 epochs, using more rounds we might get up to 70%.

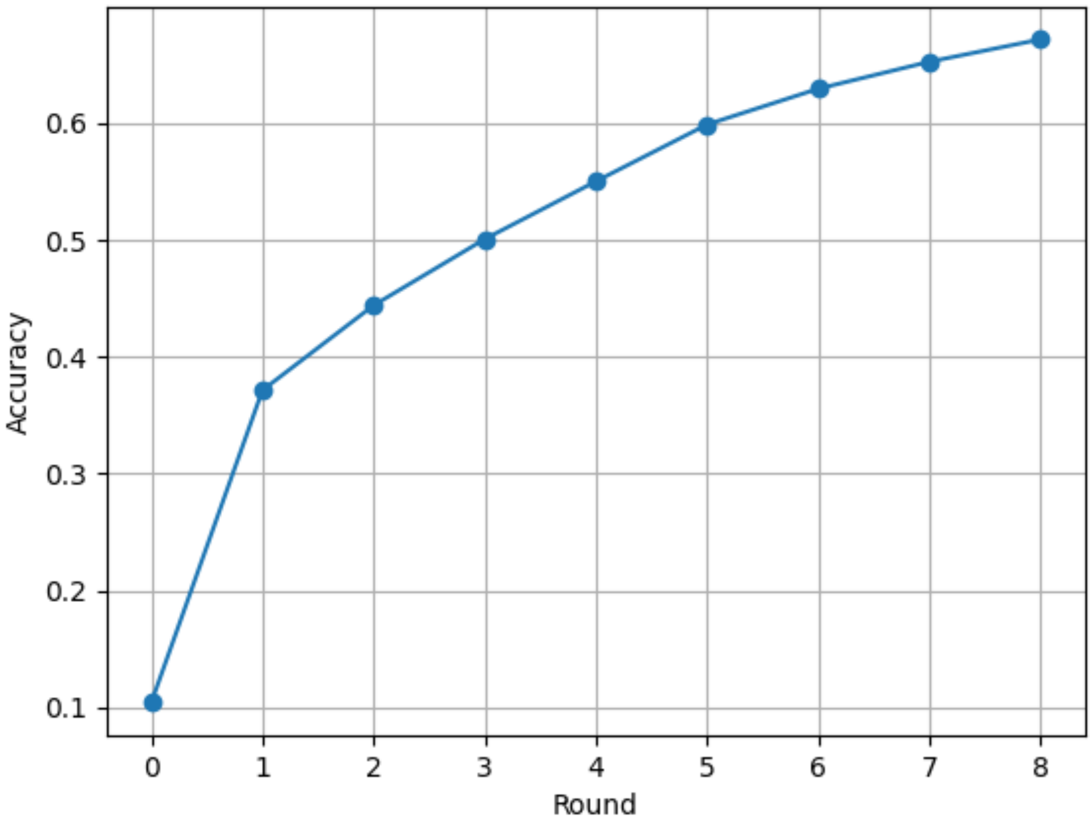
It can be seen on the graphs that the strategy does not really matter in this case, which was also expected.

```
INFO :      aggregate_evaluate: received 5 results and 0 failures
INFO :
INFO :      [ROUND 6]
INFO :      configure_fit: strategy sampled 5 clients (out of 5)
INFO :      aggregate_fit: received 5 results and 0 failures
INFO :      fit progress: (6, 0.0, {'accuracy': 0.6296}, 373.3084767590001)
INFO :      configure_evaluate: strategy sampled 5 clients (out of 5)
📊 Server round 6 accuracy: 0.6296
INFO :      aggregate_evaluate: received 5 results and 0 failures
INFO :
INFO :      [ROUND 7]
INFO :      configure_fit: strategy sampled 5 clients (out of 5)
INFO :      aggregate_fit: received 5 results and 0 failures
INFO :      fit progress: (7, 0.0, {'accuracy': 0.6597}, 436.5367848000001)
INFO :      configure_evaluate: strategy sampled 5 clients (out of 5)
📊 Server round 7 accuracy: 0.6597
INFO :      aggregate_evaluate: received 5 results and 0 failures
INFO :
INFO :      [ROUND 8]
INFO :      configure_fit: strategy sampled 5 clients (out of 5)
INFO :      aggregate_fit: received 5 results and 0 failures
INFO :      fit progress: (8, 0.0, {'accuracy': 0.6736}, 504.014833468)
INFO :      configure_evaluate: strategy sampled 5 clients (out of 5)
📊 Server round 8 accuracy: 0.6736
```

```
display(Image(filename="/content/results/base_IID_fedavg_experiment_accuracy.pr
display(Image(filename="/content/results/base_IID_fedprox_experiment_accuracy.p
```



Accuracy over Rounds: base_IID_fedprox_experiment




```
✅ Finished: base_IID_fedavg_experiment

🚀 Running experiment: base_IID_fedprox_experiment
INFO :      Starting Flower ServerApp, config: num_rounds=8, round_timeout=
INFO :
INFO :      [INIT]
INFO :      Requesting initial parameters from one random client
(pid=6468) 2025-06-05 12:48:48.776296: E external/local_xla/xla/stream_exec
(pid=6468) WARNING: All log messages before absl::InitializeLog() is called
(pid=6468) E0000 00:00:1749127729.817472    6468 cuda_dnn.cc:8310] Unable t
(pid=6468) E0000 00:00:1749127729.830757    6468 cuda_blas.cc:1418] Unable
(ClientAppActor pid=6469) /usr/local/lib/python3.11/dist-packages/jupyter_c
(ClientAppActor pid=6469) given by the platformdirs library.  To remove thi
(ClientAppActor pid=6469) see the appropriate new directories, set the envi
(ClientAppActor pid=6469) `JUPYTER_PLATFORM_DIRS=1` and then run `jupyter -
(ClientAppActor pid=6469) The use of platformdirs will be the default in `j
(ClientAppActor pid=6468)   from jupyter_core.paths import jupyter_data_dir
(pid=6469) 2025-06-05 12:48:49.777683: E external/local_xla/xla/stream_exec
(pid=6469) WARNING: All log messages before absl::InitializeLog() is called
```

As we can see it doesn't seem to matter which strategy we use. Now we can use this as our baseline - how much worse will Non-IID data be?

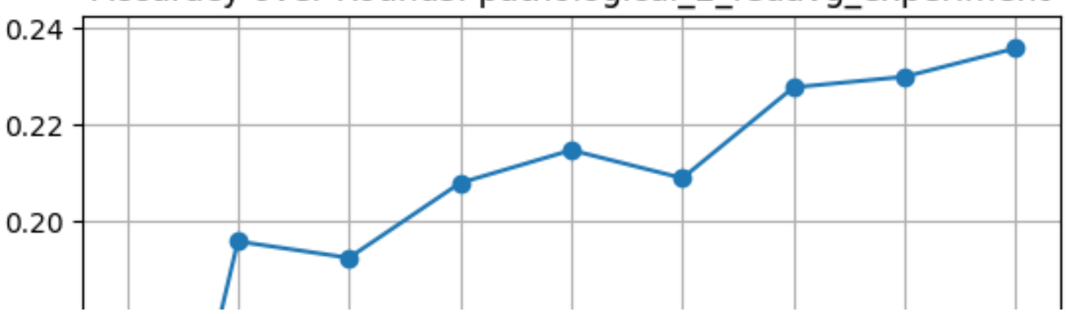
Results on PathologicalPartitioner

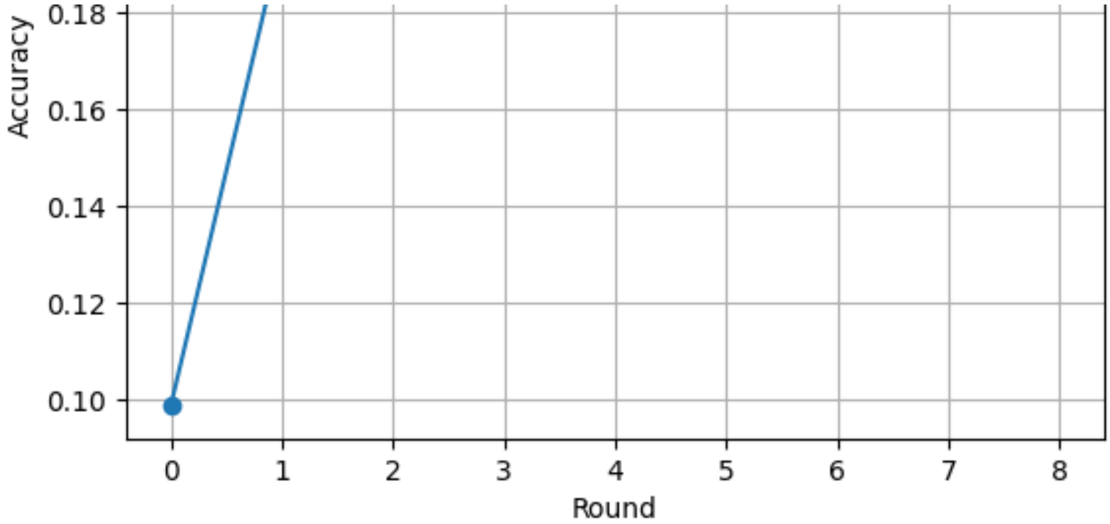
For the PathologicalPartitioner we ran our experiments twice: once with a maximum of 2 labels per client (extremly Non-IID) and once ith a maximum of 5 labels per client (a bit better but still quite extreme).

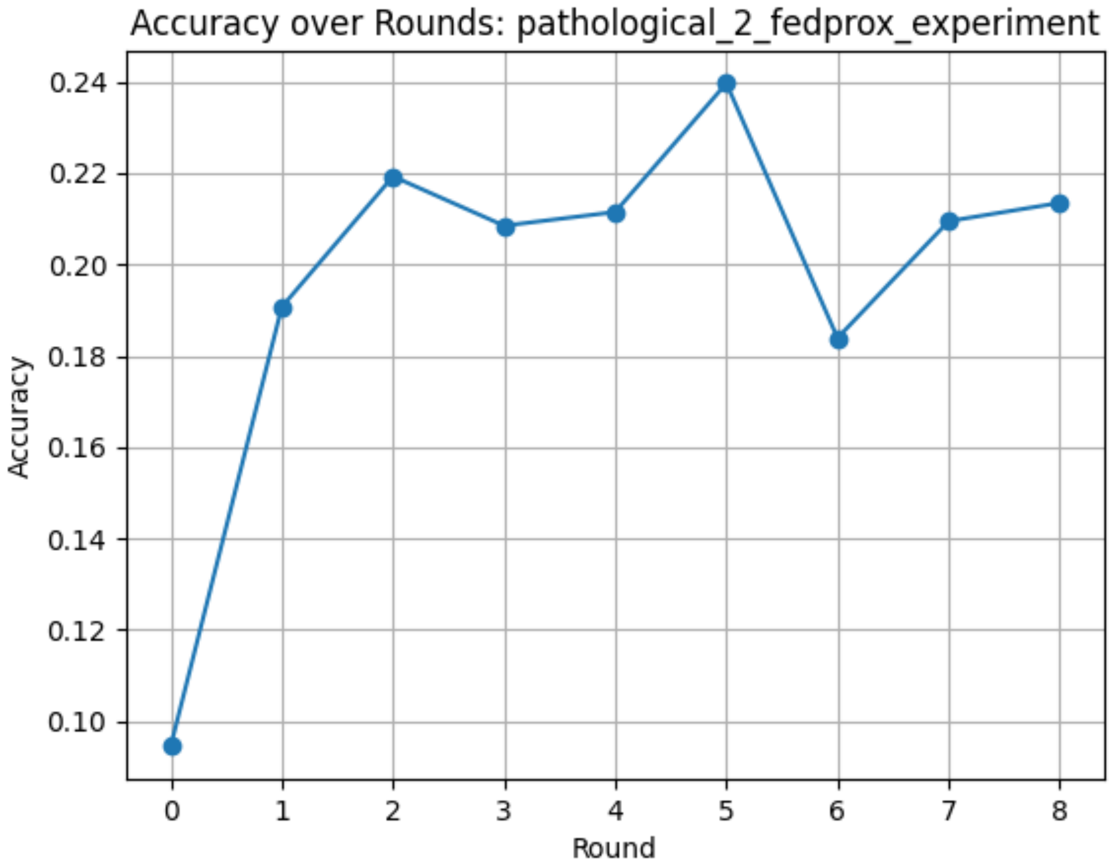
Both of our approaches really struggled with performance in our experiments. First of all we will look at the 2 labels per partition case:

```
display(Image(filename="/content/results/pathological_2_fedavg_experiment_accur
display(Image(filename="/content/results/pathological_2_fedprox_experiment_accu
```

Accuracy over Rounds: pathological_2_fedavg_experiment








```
INFO :      configure_evaluate: strategy sampled 5 clients (out of 5)
📊 Server round 6 accuracy: 0.6294
INFO :      aggregate_evaluate: received 5 results and 0 failures
INFO :
INFO :      [ROUND 7]
INFO :      configure_fit: strategy sampled 5 clients (out of 5)
INFO :      aggregate_fit: received 5 results and 0 failures
INFO :      fit progress: (7, 0.0, {'accuracy': 0.6525}, 434.8482026280001)
INFO :      configure_evaluate: strategy sampled 5 clients (out of 5)
📊 Server round 7 accuracy: 0.6525
INFO :      aggregate_evaluate: received 5 results and 0 failures
INFO :
INFO :      [ROUND 8]
INFO :      configure_fit: strategy sampled 5 clients (out of 5)
INFO :      aggregate_fit: received 5 results and 0 failures
```

We can see that both approaches don't really learn well. However what we can conclude is that it might be safer to use the FedAvg strategy in this case, with a lot more rounds we might get better performance.

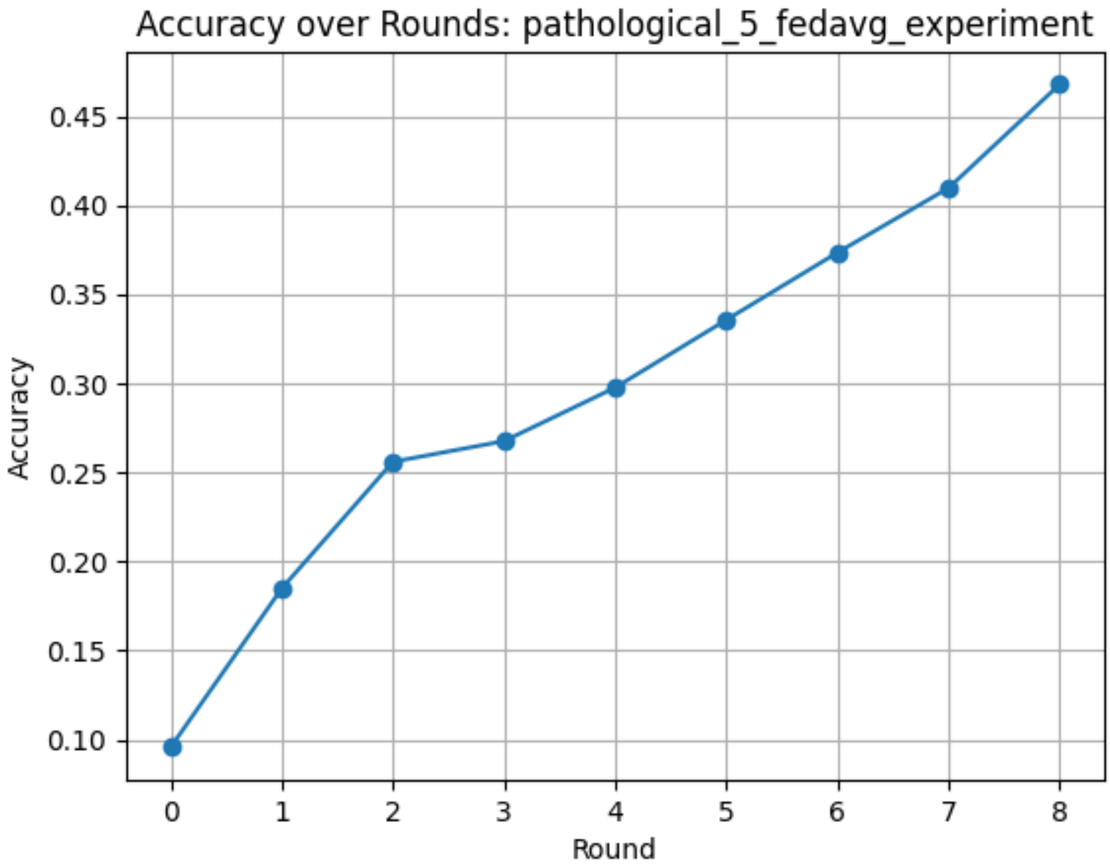
But this also works the other way, if one could find a really advanced aggregation strategy, it might perform a lot better.

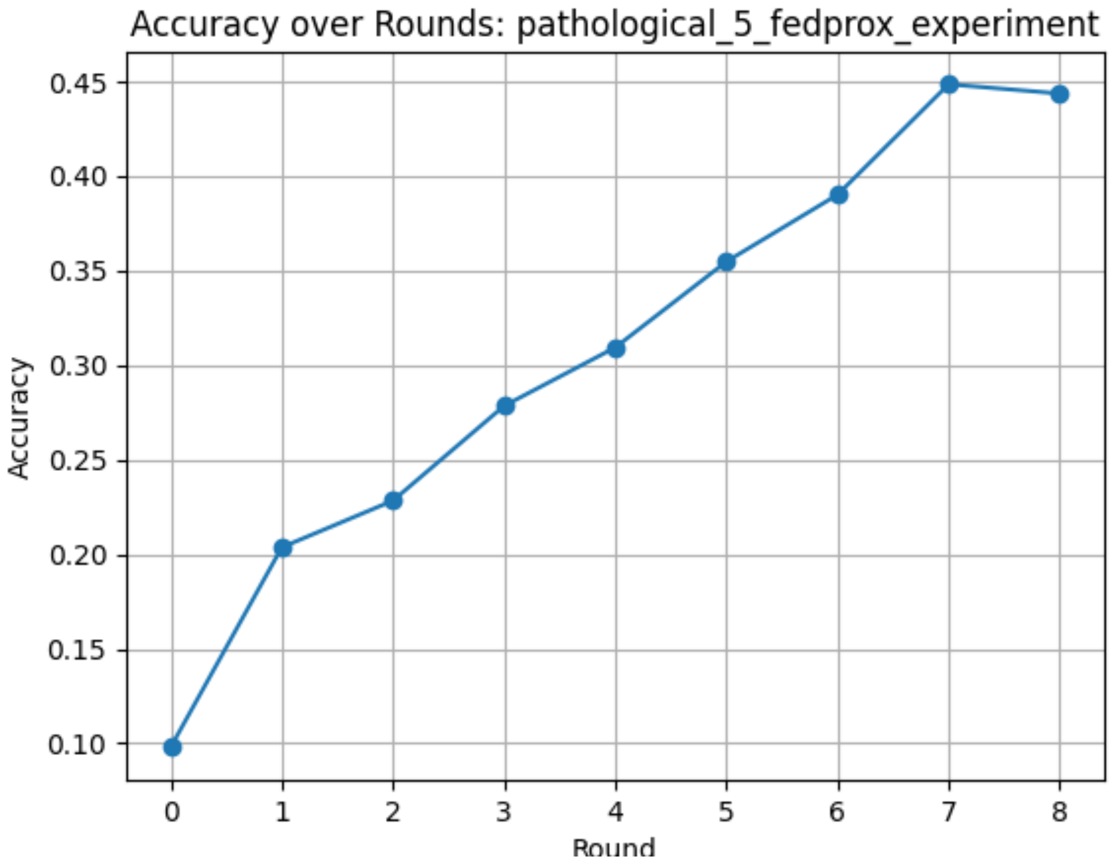
In our experiments however we can not be satisfied with the result and this shows that it is a really hard task to work with this extremely skewed data.

Let's see how well our setup works with the Pathological setup of 5 classes per client:

```
INFO :      fit progress: (8, 0.0, {'accuracy': 0.6716}, 495.89619979200006
INFO :      configure evaluate: strategy sampled 5 clients (out of 5)
```

```
display(Image(filename="/content/results/pathological_5_fedavg_experiment_accur
display(Image(filename="/content/results/pathological_5_fedprox_experiment_accu
```





Round

With 5 classes per client our model seems to learn quite well, however it is a bit slower. With more runs we might get a bit more performance out of it. Also in this case FedAvg is the prefered strategy in our usecase, since it learns steadier and gives us a better result.

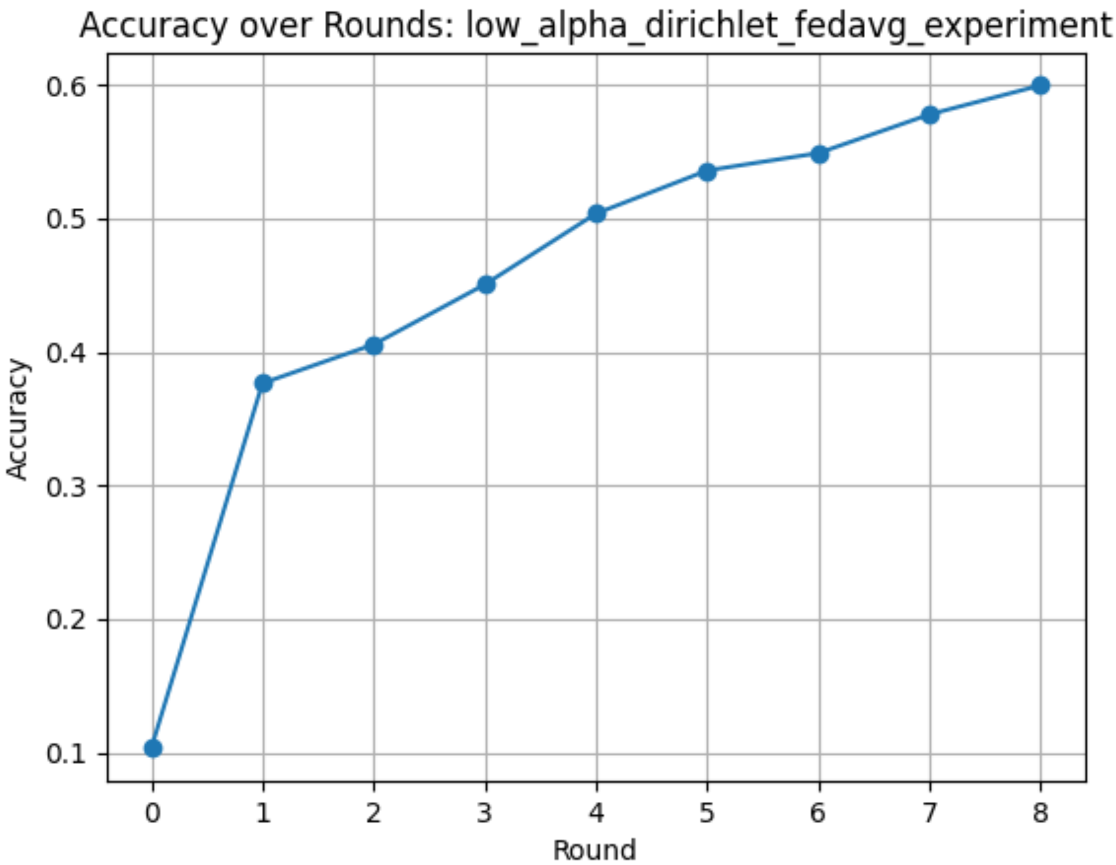
Zum Bearbeiten doppelklicken (oder Eingabe)

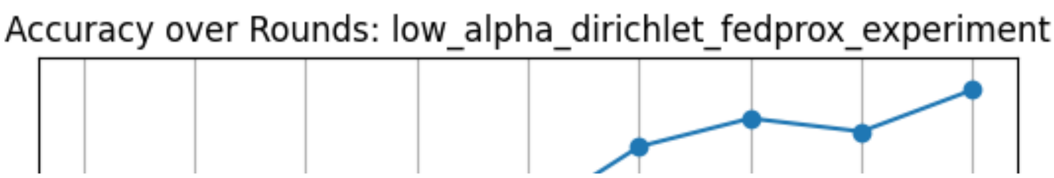
Results on DirichletPartitioner

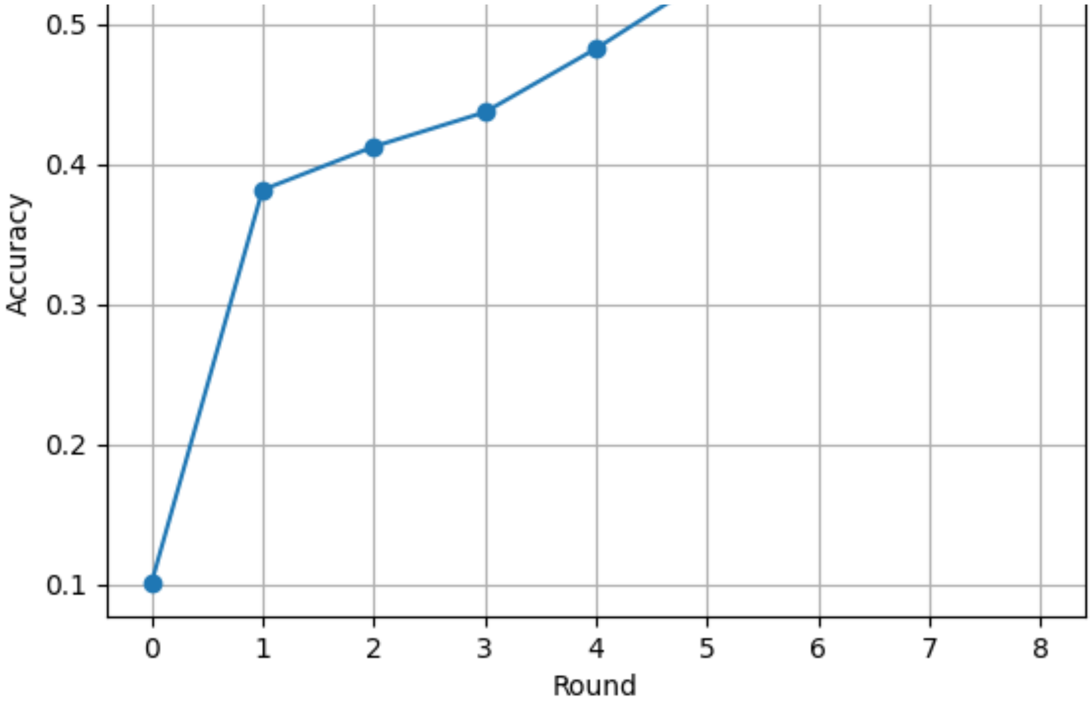
Also for the DirichletPartitioner we ran our experiments twice, once with low alpha and once with high alpha. As illurstrated in the Distribution Section, low alpha makes the data really non-IID, while high alpha just slightly does so.

```
✅ Finished: base_IID_fedprox_experiment

🚀 Running experiment: low_alpha_dirichlet_fedavg_experiment
INFO :      Starting Flower ServerApp, config: num_rounds=8, round_timeout=
INFO :
INFO :      [INIT]
INFO :      Requesting initial parameters from one random client
(pid=9083) 2025-06-05 12:57:41.298547: E external/local_xla/xla/stream_exec
(pid=9083) WARNING: All log messages before absl::InitializeLog() is called
(pid=9083) E0000 00:00:1749128261.333908    9083 cuda_dnn.cc:8310] Unable t
(pid=9083) E0000 00:00:1749128261.343490    9083 cuda_blas.cc:1418] Unable
(ClientAppActor pid=9085) /usr/local/lib/python3.11/dist-packages/jupyter_c
(ClientAppActor pid=9085) given by the platformdirs library.  To remove thi
(ClientAppActor pid=9085) see the appropriate new directories, set the envi
(ClientAppActor pid=9085) `JUPYTER_PLATFORM_DIRS=1` and then run `jupyter -
(ClientAppActor pid=9085) The use of platformdirs will be the default in `j
(ClientAppActor pid=9085)   from jupyter_core.paths import jupyter_data_dir
```

```
display(Image(filename="/content/results/low_alpha_dirichlet_fedavg_experiment_
display(Image(filename="/content/results/low_alpha_dirichlet_fedprox_experiment
```







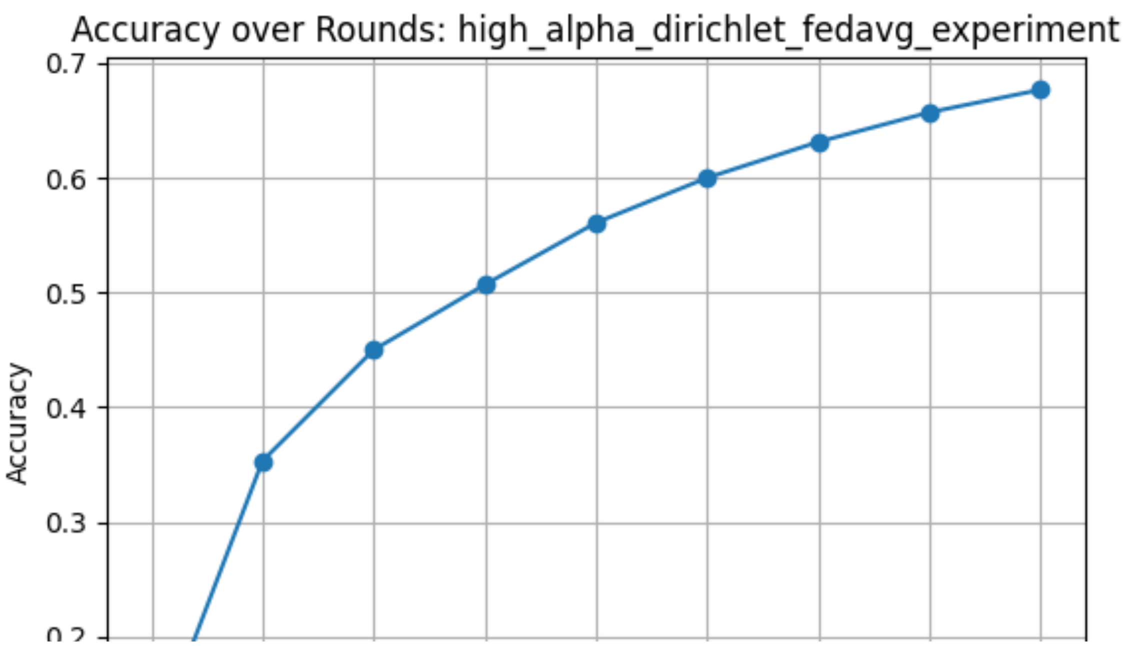

```
INFO :      fit progress: (6, 0.0, {'accuracy': 0.5489}, 371.56044096999995
INFO :      configure_evaluate: strategy sampled 5 clients (out of 5)
📊 Server round 6 accuracy: 0.5489
INFO :      aggregate_evaluate: received 5 results and 0 failures
INFO :
INFO :      [ROUND 7]
INFO :      configure_fit: strategy sampled 5 clients (out of 5)
INFO :      aggregate_fit: received 5 results and 0 failures
INFO :      fit progress: (7, 0.0, {'accuracy': 0.5779}, 432.6090579470001)
INFO :      configure_evaluate: strategy sampled 5 clients (out of 5)
📊 Server round 7 accuracy: 0.5779
INFO :      aggregate_evaluate: received 5 results and 0 failures
INFO :
INFO :      [ROUND 8]
```

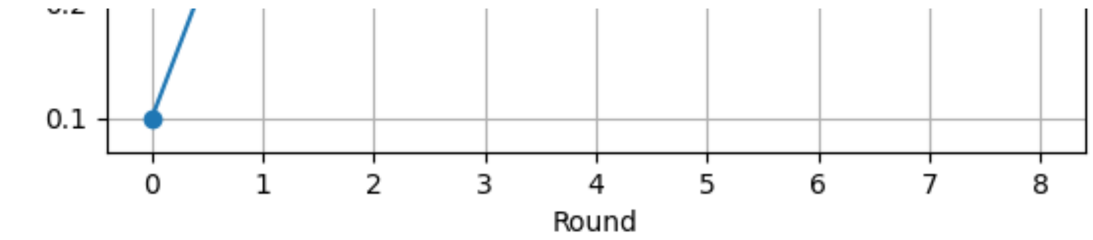
We can observe that also in these case the FedAvg strategy is the better choice, achieving a more stable and overall better result.

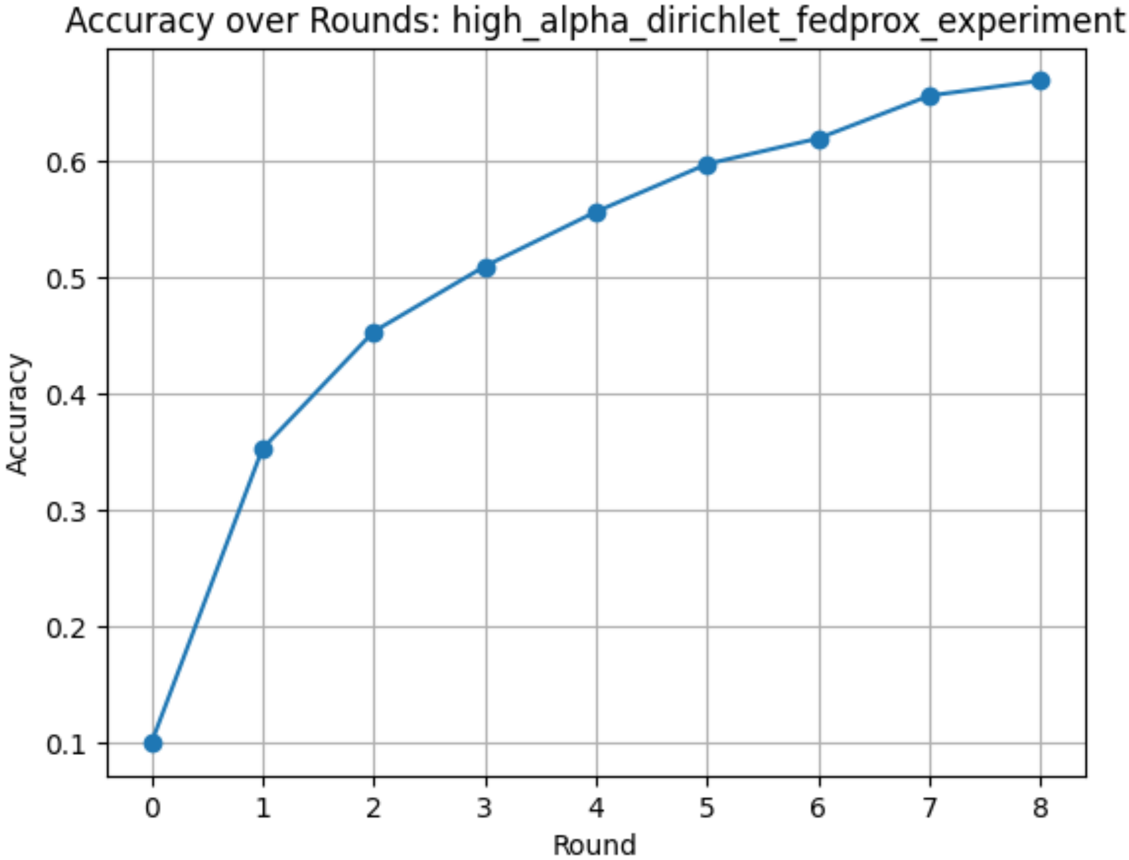
It is remarkable that a model performs so well, considering that this data is highly skewed. However here every client has all labels, just in different quantities and that seems to make the difference.

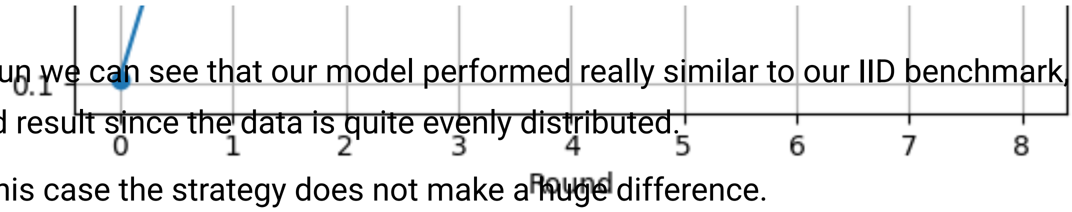
Let's also look at the Dirichlet Partition with high alpha (more regular but still non-IID):

```
display(Image(filename="/content/results/high_alpha_dirichlet_fedavg_experiment
display(Image(filename="/content/results/high_alpha_dirichlet_fedprox_experimer
```









On this run we can see that our model performed really similar to our IID benchmark, an expected result since the data is quite evenly distributed.

Also in this case the strategy does not make a huge difference.

The Dirichlet Distribution also has the advantage that it replicates real world scenarios really well - we might have really different data amount, however in a usual setup most clients will have at least a little bit of data of each class.

Especially the run with low_alpha has shown that FTL works really well on these setups.

```
✅ Finished: low_alpha_dirichlet_fedavg_experiment
🚀 Running experiment: low_alpha_dirichlet_fedprox_experiment
INFO :      Starting Flower ServerApp, config: num_rounds=8, round_timeout=
INFO :
INFO :      [INIT]
INFO :      Requesting initial parameters from one random client
(pid=11694) 2025-06-05 13:06:29.094096: E external/local_xla/xla/stream_exe
(pid=11694) WARNING: All log messages before absl::InitializeLog() is calle
(pid=11694) E0000 00:00:1749128789.130594   11694 cuda_dnn.cc:8310] Unable
(pid=11694) E0000 00:00:1749128789.140498   11694 cuda_blas.cc:1418] Unable
(ClientAppActor pid=11694) /usr/local/lib/python3.11/dist-packages/jupyter_
(ClientAppActor pid=11694) [illegible] platformdirs library. To remove th
(ClientAppActor pid=11694) see the appropriate new directories, set the env
(ClientAppActor pid=11694)   JUPYTER_PLATFORM_DIRS=1` and then run `jupyter
(ClientAppActor pid=11694) The use of platformdirs will be the default in `
(ClientAppActor pid=11694)   from jupyter_core.paths import jupyter_data_di
```

Conclusion

The results have shown to be really dependant on the distribution which we gave to our clients. The distribution has shown to be the main factor in most cases FedAvg worked similarly well as FedProx. Our intuition is that it is really hard to find the right hyperparameters for a Strategy different than FedAvg to be significantly better.

Also regarding the Setup in general, there are a whole set of hyperparameters that would need to be optimized, how ever this needs a significant amount of computational power, which we don't have at this moment.

In reality we really often encounter the Dirichlet Distribution, it makes a lot more sense to use this one instead of the Pathological Partitioners. While they might be good for experimentation, they likely don't replicate real world scenarios. Our models performed a lot better on Dirichlet, which is remarkable. It seems that while having really skewed data, the model learns really well if the classes per client are more evenly distributed. Since its a kind of reversed Normal Distribution, this is usually the case.

All in all the results clearly show how a basic FTL setup works and what one has to be careful of - especially regarding the Distribution of Non-IID Data.

```
(pid=11693) 2025-06-05 13:06:29.156090: E external/local_xla/xla/stream_exe
(pid=11693) WARNING: All log messages before absl::InitializeLog() is calle
(pid=11693) E0000 00:00:1749128789.189756   11693 cuda_dnn.cc:8310] Unable
(pid=11693) E0000 00:00:1749128789.199767   11693 cuda_blas.cc:1418] Unable
INFO :      Received initial parameters from one random client
INFO :      Starting evaluation of initial global parameters
INFO :      initial parameters (loss, other metrics): 0.0, {'accuracy': 0.1
INFO :
INFO :      [ROUND 1]
INFO :      configure_fit: strategy sampled 5 clients (out of 5)
📊 Server round 0 accuracy: 0.1007
(ClientAppActor pid=11693) /usr/local/lib/python3.11/dist-packages/jupyter_
(ClientAppActor pid=11693) given by the platformdirs library.  To remove th
(ClientAppActor pid=11693) see the appropriate new directories, set the env
(ClientAppActor pid=11693)  JUPYTER_PLATFORM_DIRS=1` and then run `jupyter
(ClientAppActor pid=11693) The use of platformdirs will be the default in `
(ClientAppActor pid=11693)   from jupyter_core.paths import jupyter_data_di
INFO :      aggregate_fit: received 5 results and 0 failures
WARNING :   No fit_metrics_aggregation_fn provided
INFO :      fit progress: (1, 0.0, {'accuracy': 0.3819}, 61.556240574000185
INFO :      configure_evaluate: strategy sampled 5 clients (out of 5)
📊 Server round 1 accuracy: 0.3819
INFO :      aggregate_evaluate: received 5 results and 0 failures
WARNING :   No evaluate_metrics_aggregation_fn provided
INFO :
INFO :      [ROUND 2]
INFO :      configure_fit: strategy sampled 5 clients (out of 5)
INFO :      aggregate_fit: received 5 results and 0 failures
INFO :      fit progress: (2, 0.0, {'accuracy': 0.4128}, 122.29452376600011
INFO :      configure_evaluate: strategy sampled 5 clients (out of 5)
📊 Server round 2 accuracy: 0.4128
INFO :      aggregate_evaluate: received 5 results and 0 failures
INFO :
INFO :      [ROUND 3]
INFO :      configure_fit: strategy sampled 5 clients (out of 5)
INFO :      aggregate_fit: received 5 results and 0 failures
INFO :      fit progress: (3, 0.0, {'accuracy': 0.4374}, 184.78126636099978
INFO :      configure_evaluate: strategy sampled 5 clients (out of 5)
📊 Server round 3 accuracy: 0.4374
INFO :      aggregate_evaluate: received 5 results and 0 failures
INFO :
INFO :      [ROUND 4]
INFO :      configure_fit: strategy sampled 5 clients (out of 5)
```