Q1:

```
import torch
import torchvision
import torchvision.transforms as transforms
import torch.nn as nn
import torch.optim as optim
from torchvision.models import resnet18
from torch.utils.data import DataLoader, random_split, Subset
import numpy as np
from sklearn.metrics import f1_score, accuracy_score, precision_score, recall_score, roc_auc_score
```

We convert the images to PyTorch tensors and normalizing them with a mean and standard deviation of 0.5 for each of the three channels (RGB). This normalization step is important for stabilizing and accelerating the training process.

We load the CIFAR-10 dataset for both training and testing.

```
# Transformations for CIFAR-10 dataset
transform = transforms.Compose([
    transforms.ToTensor(),
    transforms.Normalize((0.5, 0.5, 0.5), (0.5, 0.5, 0.5))
])

# Load CIFAR-10 dataset
trainset = torchvision.datasets.CIFAR10(root='./data', train=True, download=True, transform=transform)
testset = torchvision.datasets.CIFAR10(root='./data', train=False, download=True, transform=transform)
```

```
Downloading https://www.cs.toronto.edu/~kriz/cifar-10-python.tar.gz to ./data/cifar-10-python.tar.gz
100%|██████████| 170498071/170498071 [00:03<00:00, 49053818.40it/s]
Extracting ./data/cifar-10-python.tar.gz to ./data
Files already downloaded and verified
```

This code defines a custom neural network model based on ResNet18, a popular deep learning architecture. The model is modified to fit the CIFAR-10 classification task. The fully connected (fc) layer of the pretrained ResNet18 model is replaced with a new sequential layer that includes a linear layer, ReLU activation, dropout for regularization, and another linear layer to output predictions for the 10 classes of CIFAR-10.

```
class ResNet18Modified(nn.Module):
    def __init__(self, num_classes=10):
        super(ResNet18Modified, self).__init__()
        self.model = resnet18(pretrained=True)
        self.model.fc = nn.Sequential(
            nn.Linear(self.model.fc.in_features, 512),
            nn.ReLU(),
            nn.Dropout(0.5),
            nn.Linear(512, num_classes)
        )

    def forward(self, x):
        x = self.model(x)
        # # Ensure the output is squeezed if necessary
        # if x.dim() == 4:
        #     x = x.squeeze(3).squeeze(2)
        return x
```

This function computes several evaluation metrics to assess the performance of the trained model. The metrics include the F1 score, accuracy, precision, recall, and the area under the receiver operating characteristic curve (AUROC).

```
def calculate_metrics(y_true, y_pred, y_prob):
    f1 = f1_score(y_true, y_pred, average='weighted')
    accuracy = accuracy_score(y_true, y_pred)
    precision = precision_score(y_true, y_pred, average='weighted')
    recall = recall_score(y_true, y_pred, average='weighted')
    auroc = roc_auc_score(y_true, y_prob, multi_class='ovr')
    return f1, accuracy, precision, recall, auroc
```

This function trains the neural network model. It iterates through a specified number of epochs, where each epoch consists of a full pass through the training data. The model is set to training mode, and for each batch of training data, the inputs and labels are moved to the appropriate device (CPU or GPU). The optimizer's gradients are reset, the model's predictions are computed, and the loss is calculated. The loss is then backpropagated, and the optimizer updates the model's parameters. The running loss is accumulated and averaged over the epoch to monitor training progress.

```
def train_model(model, dataloaders, criterion, optimizer, num_epochs=25):
    for epoch in range(num_epochs):
        model.train()
        running_loss = 0.0

        for inputs, labels in dataloaders['train']:
            inputs, labels = inputs.to(device), labels.to(device)

            optimizer.zero_grad()
            outputs = model(inputs)
            loss = criterion(outputs, labels)
            loss.backward()
            optimizer.step()

            running_loss += loss.item() * inputs.size(0)

        epoch_loss = running_loss / len(dataloaders['train'].dataset)
        print(f'Epoch {epoch}/{num_epochs - 1}, Loss: {epoch_loss:.4f}')

    return model
```

This function evaluates the trained model on the test data. The model is set to evaluation mode, and the gradients are not computed to save memory and computational resources. For each batch of test data, the inputs and labels are moved to the appropriate device, and the model's predictions are computed. The predicted labels and probabilities are stored, and the true labels are collected. The calculate_metrics function is then used to compute the evaluation metrics based on the collected data.

```
def evaluate_model(model, dataloaders):
    model.eval()
    y_true = []
    y_pred = []
    y_prob = []

    with torch.no_grad():
        for inputs, labels in dataloaders['test']:
            inputs, labels = inputs.to(device), labels.to(device)
            outputs = model(inputs)
            _, preds = torch.max(outputs, 1)
            probs = torch.nn.functional.softmax(outputs, dim=1)

            y_true.extend(labels.cpu().numpy())
            y_pred.extend(preds.cpu().numpy())
            y_prob.extend(probs.cpu().numpy())

    return calculate_metrics(y_true, y_pred, y_prob)
```

This function implements the SISA (Sharded, Isolated, Sliced, and Aggregated) training algorithm. The training dataset is divided into S shards, each of which is further divided into R slices. For each shard, a new model instance is created and trained incrementally using the slices. The initial training is done with the first slice, and subsequent training iterations use the union of the current and all previous slices. This incremental training approach allows the model to be trained on increasingly larger subsets of the shard data, improving its performance while maintaining isolation. The trained models from all shards are collected and returned.

```
def sisa_training(trainset, S, R, num_epochs, device):
    shard_size = len(trainset) // S
    shard_indices = [list(range(i * shard_size, (i + 1) * shard_size)) for i in range(S)]

    all_models = []
    for s in range(S):
        shard_data = Subset(trainset, shard_indices[s])
        slice_size = len(shard_data) // R
        slice_indices = [list(range(i * slice_size, (i + 1) * slice_size)) for i in range(R)]

        model = ResNet18Modified().to(device)
        criterion = nn.CrossEntropyLoss()
        optimizer = optim.SGD(model.parameters(), lr=0.001, momentum=0.9)

        for r in range(R):
            if r == 0:
                # Initial training with the first slice
                dataloaders = {
                    'train': DataLoader(Subset(shard_data, slice_indices[r]), batch_size=64, shuffle=True),
                    'test': DataLoader(testset, batch_size=64, shuffle=False)
                }
                model = train_model(model, dataloaders, criterion, optimizer, num_epochs)
            else:
                # Incremental training with union of current and previous slices
                union_indices = [idx for i in range(r+1) for idx in slice_indices[i]]
                dataloaders['train'] = DataLoader(Subset(shard_data, union_indices), batch_size=64, shuffle=True)
                model = train_model(model, dataloaders, criterion, optimizer, num_epochs)

        all_models.append(model)

    return all_models



device = torch.device("cuda:0" if torch.cuda.is_available() else "cpu")

S_values = [5, 10, 20]
R_values = [5, 10, 20]
num_epochs = 5
S = 10
R = 10

print(f"Training with S = {S}, R = {R}")
models = sisa_training(trainset, S, R, num_epochs, device)
# Save models for later evaluation
torch.save(models, f'sisa_models_S{S}_R{R}.pt')
```

```
Epoch 2/4, Loss: 0.1330
Epoch 3/4, Loss: 0.1030
Epoch 4/4, Loss: 0.0891
Epoch 0/4, Loss: 2.4195
Epoch 1/4, Loss: 2.2552
Epoch 2/4, Loss: 2.1389
Epoch 3/4, Loss: 1.9749
Epoch 4/4, Loss: 1.8126
Epoch 0/4, Loss: 1.8678
Epoch 1/4, Loss: 1.6207
Epoch 2/4, Loss: 1.4003
Epoch 3/4, Loss: 1.1743
Epoch 4/4, Loss: 0.9341
Epoch 0/4, Loss: 1.0559
Epoch 1/4, Loss: 0.7913
Epoch 2/4, Loss: 0.6208
Epoch 3/4, Loss: 0.4565
Epoch 4/4, Loss: 0.3464
Epoch 0/4, Loss: 0.5975
Epoch 1/4, Loss: 0.3813
Epoch 2/4, Loss: 0.2655
Epoch 3/4, Loss: 0.2159
Epoch 4/4, Loss: 0.1632
Epoch 0/4, Loss: 0.4356
Epoch 1/4, Loss: 0.4454
```

Aggregating Model Predictions using Average Prediction Vectors: This function aggregates predictions from multiple models using the average of their prediction vectors (softmax probabilities).

Model Evaluation Mode: Each model is set to evaluation mode to ensure that layers like dropout are not active during inference.

Predicting Probabilities: For each model, the function iterates over the dataloader, computes the softmax probabilities for the inputs, and stores these probabilities.

Averaging Probabilities: After collecting the probabilities from all models, the average probability for each class is calculated across all models.

Final Prediction: The class with the highest average probability is chosen as the final prediction. The function returns these predictions along with the averaged probabilities.

```
# Using average prediction vectors
def aggregate_models(models, dataloader):
    all_probs = []

    for model in models:
        model.eval()
        probs = []

        with torch.no_grad():
            for inputs, _ in dataloader:
                inputs = inputs.to(device)
                outputs = model(inputs)
                prob = torch.nn.functional.softmax(outputs, dim=1)
                probs.extend(prob.cpu().numpy())

        all_probs.append(probs)

    avg_probs = np.mean(all_probs, axis=0)
    return np.argmax(avg_probs, axis=1), avg_probs
```

Aggregating Model Predictions using Majority Voting: This function aggregates predictions from multiple models using majority voting.

Model Evaluation Mode: Each model is set to evaluation mode to ensure consistent inference behavior.

Predicting Classes: For each model, the function iterates over the dataloader, computes the predicted class for each input (using the class with the highest output score), and stores these predictions.

Majority Voting: After collecting the predictions from all models, the final prediction for each input is determined by finding the most frequently predicted class (mode) among the models.

Final Prediction: The function returns the final predictions based on majority voting.

```
def majority_voting(models, dataloader):
    all_preds = []

    for model in models:
        model.eval()
        preds = []

        with torch.no_grad():
            for inputs, _ in dataloader:
                inputs = inputs.to(device)
                outputs = model(inputs)
                _, pred = torch.max(outputs, 1)
                preds.extend(pred.cpu().numpy())

        all_preds.append(preds)

    all_preds = np.array(all_preds)
    majority_preds = np.apply_along_axis(lambda x: np.bincount(x).argmax(), axis=0, arr=all_preds)
    return majority_preds



# Example of using the aggregate_models function
test_loader = DataLoader(testset, batch_size=64, shuffle=False)
models = torch.load('sisa_models_S10_R10.pt')  # Load your saved models

# Extract y_true from the test DataLoader
y_true = []
for _, labels in test_loader:
    y_true.extend(labels.numpy())

# Get aggregated predictions
y_pred, y_prob = aggregate_models(models, test_loader)

# Calculate metrics
final_metrics = calculate_metrics(y_true, y_pred, y_prob)
print(f"Final aggregated model metrics: {final_metrics}")
```

```
Final aggregated model metrics: (0.7613011829529175, 0.7622, 0.761723100021814, 0.7622, 0.9688322333333333)
```

```
device = torch.device("cuda:0" if torch.cuda.is_available() else "cpu")
S_values = [5]
R_values = [5, 10, 20]
num_epochs = 5

models_dict = {}

# Training and saving models
for S in S_values:
    models_dict[S] = {}
    for R in R_values:
        print(f"Training with S = {S}, R = {R}")
        models = sisa_training(trainset, S, R, num_epochs, device)
        models_dict[S][R] = models
        torch.save(models, f'sisa_models_S{S}_R{R}.pt')
```

```
Epoch 4/4, Loss: 0.3394
Epoch 0/4, Loss: 0.5984
Epoch 1/4, Loss: 0.3685
Epoch 2/4, Loss: 0.2685
Epoch 3/4, Loss: 0.2094
Epoch 4/4, Loss: 0.1591
Epoch 0/4, Loss: 0.4209
Epoch 1/4, Loss: 0.8571
Epoch 2/4, Loss: 0.5302
Epoch 3/4, Loss: 0.3306
Epoch 4/4, Loss: 0.3011
Epoch 0/4, Loss: 0.5207
Epoch 1/4, Loss: 0.2453
Epoch 2/4, Loss: 0.1541
Epoch 3/4, Loss: 0.1053
Epoch 4/4, Loss: 0.0805
Epoch 0/4, Loss: 0.3027
Epoch 1/4, Loss: 0.1619
Epoch 2/4, Loss: 0.0900
Epoch 3/4, Loss: 0.0742
Epoch 4/4, Loss: 0.0562
Epoch 0/4, Loss: 0.2326
Epoch 1/4, Loss: 0.1218
Epoch 2/4, Loss: 0.0831
Epoch 3/4, Loss: 0.0611
Epoch 4/4, Loss: 0.0389
Epoch 0/4, Loss: 0.2287
Epoch 1/4, Loss: 0.1277
Epoch 2/4, Loss: 0.0823
Epoch 3/4, Loss: 0.0577
Epoch 4/4, Loss: 0.0455
Epoch 0/4, Loss: 0.2144
Epoch 1/4, Loss: 0.1401
Epoch 2/4, Loss: 0.1347
```

```
S_values = [5]
R_values = [5, 10, 20]
num_epochs = 5

test_loader = DataLoader(testset, batch_size=64, shuffle=False)

# Extract y_true from the test DataLoader
y_true = []
for _, labels in test_loader:
    y_true.extend(labels.numpy())

for S in S_values:
    for R in R_values:
        models = torch.load(f'sisa_models_S{S}_R{R}.pt')  # Load your saved models
        models_dict[S][R] = models
        # Get aggregated predictions
        y_pred, y_prob = aggregate_models(models, test_loader)
        # Calculate metrics
        final_metrics = calculate_metrics(y_true, y_pred, y_prob)
        print(f"Final aggregated model metrics for S = {S}, R = {R}: {final_metrics}")
```

```
Final aggregated model metrics for S = 5, R = 5: (0.7730889342326599, 0.7743, 0.7727521380380119, 0.7743, 0.9699974888888889)
Final aggregated model metrics for S = 5, R = 10: (0.787026570413323, 0.7884, 0.7871762116488508, 0.7884, 0.9727793333333332)
Final aggregated model metrics for S = 5, R = 20: (0.7956192134175321, 0.7973, 0.7952194992752175, 0.7973, 0.9746610666666667)
```

```
device = torch.device("cuda:0" if torch.cuda.is_available() else "cpu")
S_values = [10]
R_values = [5, 10, 20]
num_epochs = 5

models_dict = {}

# Training and saving models
for S in S_values:
    models_dict[S] = {}
    for R in R_values:
        print(f"Training with S = {S}, R = {R}")
        models = sisa_training(trainset, S, R, num_epochs, device)
        models_dict[S][R] = models
        torch.save(models, f'sisa_models_S{S}_R{R}.pt')
```

```
Epoch 4/4, Loss: 0.0564
Epoch 0/4, Loss: 0.2442
Epoch 1/4, Loss: 0.1312
Epoch 2/4, Loss: 0.0936
Epoch 3/4, Loss: 0.0696
Epoch 4/4, Loss: 0.0716
Epoch 0/4, Loss: 0.2529
Epoch 1/4, Loss: 0.3319
Epoch 2/4, Loss: 0.3899
Epoch 3/4, Loss: 0.5199
Epoch 4/4, Loss: 0.3550
Epoch 0/4, Loss: 0.4035
Epoch 1/4, Loss: 0.1914
Epoch 2/4, Loss: 0.1072
Epoch 3/4, Loss: 0.0807
Epoch 4/4, Loss: 0.0465
Epoch 0/4, Loss: 0.1879
Epoch 1/4, Loss: 0.0999
Epoch 2/4, Loss: 0.0590
Epoch 3/4, Loss: 0.0412
Epoch 4/4, Loss: 0.0391
Epoch 0/4, Loss: 0.1436
Epoch 1/4, Loss: 0.0605
Epoch 2/4, Loss: 0.0573
Epoch 3/4, Loss: 0.0360
Epoch 4/4, Loss: 0.0265
Epoch 0/4, Loss: 0.1625
Epoch 1/4, Loss: 0.0802
Epoch 2/4, Loss: 0.0475
Epoch 3/4, Loss: 0.0337
Epoch 4/4, Loss: 0.0311
Epoch 0/4, Loss: 0.1318
Epoch 1/4, Loss: 0.0615
Epoch 2/4, Loss: 0.0368
Epoch 3/4, Loss: 0.0252
Epoch 4/4, Loss: 0.0250
Epoch 0/4, Loss: 0.1237
Epoch 1/4, Loss: 0.0604
Epoch 2/4, Loss: 0.0391
Epoch 3/4, Loss: 0.0286
Epoch 4/4, Loss: 0.0206
Epoch 0/4, Loss: 0.1165
Epoch 1/4, Loss: 0.0675
Epoch 2/4, Loss: 0.0498
Epoch 3/4, Loss: 0.0331
Epoch 4/4, Loss: 0.0209
Epoch 0/4, Loss: 0.1112
Epoch 1/4, Loss: 0.0509
Epoch 2/4, Loss: 0.0322
Epoch 3/4, Loss: 0.0318
Epoch 4/4, Loss: 0.0253
Epoch 0/4, Loss: 0.1186
Epoch 1/4, Loss: 0.0669
Epoch 2/4, Loss: 0.0512
```

```
S_values = [10]
R_values = [5, 10, 20]
num_epochs = 5

test_loader = DataLoader(testset, batch_size=64, shuffle=False)

# Extract y_true from the test DataLoader
y_true = []
for _, labels in test_loader:
    y_true.extend(labels.numpy())

for S in S_values:
    for R in R_values:
        models = torch.load(f'sisa_models_S{S}_R{R}.pt')  # Load your saved models
        models_dict[S][R] = models
        # Get aggregated predictions
        y_pred, y_prob = aggregate_models(models, test_loader)
        # Calculate metrics
        final_metrics = calculate_metrics(y_true, y_pred, y_prob)
        print(f"Final aggregated model metrics for S = {S}, R = {R}: {final_metrics}")
```

```
Final aggregated model metrics for S = 10, R = 5: (0.737528290273186, 0.7388, 0.7377902083160518, 0.7388, 0.9630484333333336)
Final aggregated model metrics for S = 10, R = 10: (0.7652568512964656, 0.7667, 0.7658048621663245, 0.7667, 0.9683451666666667)
Final aggregated model metrics for S = 10, R = 20: (0.7711972773795097, 0.773, 0.771217815860338, 0.773, 0.9707286777777776)
```

```
device = torch.device("cuda:0" if torch.cuda.is_available() else "cpu")
S_values = [20]
R_values = [5, 10, 20]
num_epochs = 5

models_dict = {}

# Training and saving models
for S in S_values:
    models_dict[S] = {}
    for R in R_values:
        print(f"Training with S = {S}, R = {R}")
        models = sisa_training(trainset, S, R, num_epochs, device)
        models_dict[S][R] = models
        torch.save(models, f'sisa_models_S{S}_R{R}.pt')
```

```
Training with S = 20, R = 5
/usr/local/lib/python3.10/dist-packages/torchvision/models/_utils.py:208: UserWarning: The parameter 'pretrained' is deprecated since
  warnings.warn(
/usr/local/lib/python3.10/dist-packages/torchvision/models/_utils.py:223: UserWarning: Arguments other than a weight enum or `None` f
  warnings.warn(msg)
Epoch 0/4, Loss: 2.3993
Epoch 1/4, Loss: 2.2371
Epoch 2/4, Loss: 2.1215
Epoch 3/4, Loss: 1.9771
Epoch 4/4, Loss: 1.7687
Epoch 0/4, Loss: 1.8424
Epoch 1/4, Loss: 1.6186
Epoch 2/4, Loss: 1.3609
Epoch 3/4, Loss: 1.1256
Epoch 4/4, Loss: 0.9290
Epoch 0/4, Loss: 1.0149
Epoch 1/4, Loss: 0.7512
Epoch 2/4, Loss: 0.5953
Epoch 3/4, Loss: 0.4441
Epoch 4/4, Loss: 0.3208
Epoch 0/4, Loss: 0.5634
Epoch 1/4, Loss: 0.3614
Epoch 2/4, Loss: 0.2445
Epoch 3/4, Loss: 0.2026
Epoch 4/4, Loss: 0.1453
Epoch 0/4, Loss: 0.4146
Epoch 1/4, Loss: 0.5082
Epoch 2/4, Loss: 0.9296
Epoch 3/4, Loss: 0.5625
Epoch 4/4, Loss: 0.5125
Epoch 0/4, Loss: 2.4157
Epoch 1/4, Loss: 2.2525
Epoch 2/4, Loss: 2.1034
Epoch 3/4, Loss: 1.9325
Epoch 4/4, Loss: 1.7778
Epoch 0/4, Loss: 1.8735
Epoch 1/4, Loss: 1.6204
Epoch 2/4, Loss: 1.3645
Epoch 3/4, Loss: 1.1224
Epoch 4/4, Loss: 0.9255
Epoch 0/4, Loss: 1.0291
Epoch 1/4, Loss: 0.7994
Epoch 2/4, Loss: 0.6165
Epoch 3/4, Loss: 0.4490
Epoch 4/4, Loss: 0.3423
Epoch 0/4, Loss: 0.5702
Epoch 1/4, Loss: 0.3869
Epoch 2/4, Loss: 0.2840
Epoch 3/4, Loss: 0.2076
Epoch 4/4, Loss: 0.1646
Epoch 0/4, Loss: 0.4212
Epoch 1/4, Loss: 0.7031
Epoch 2/4, Loss: 0.4704
Epoch 3/4, Loss: 0.3811
Epoch 4/4, Loss: 0.4759
Epoch 0/4, Loss: 2.4266
Epoch 1/4, Loss: 2.2738
```

```
S_values = [20]
R_values = [5, 10, 20]
num_epochs = 5

test_loader = DataLoader(testset, batch_size=64, shuffle=False)

# Extract y_true from the test DataLoader
y_true = []
for _, labels in test_loader:
    y_true.extend(labels.numpy())

for S in S_values:
    for R in R_values:
        models = torch.load(f'sisa_models_S{S}_R{R}.pt')  # Load your saved models
        models_dict[S][R] = models
        # Get aggregated predictions
        y_pred, y_prob = aggregate_models(models, test_loader)
        # Calculate metrics
        final_metrics = calculate_metrics(y_true, y_pred, y_prob)
        print(f"Final aggregated model metrics for S = {S}, R = {R}: {final_metrics}")
```

```
Final aggregated model metrics for S = 20, R = 5: (0.7089042339610256, 0.7109, 0.7086992438045219, 0.7109, 0.9568803444444445)
Final aggregated model metrics for S = 20, R = 10: (0.7244662773425617, 0.726, 0.724612719037516, 0.726, 0.9603272777777778)
Final aggregated model metrics for S = 20, R = 20: (0.7321854585590531, 0.7343, 0.7319627474351771, 0.7343, 0.9631915111111111)
```

Q2:

```
import random
from copy import deepcopy
```

his function randomly selects a specified number (num_to_forget) of data points from the dataset to be forgotten.

Random Sampling: The function uses random.sample to randomly select indices from the dataset.

Output: The selected indices represent the data points to be forgotten during the unlearning process.

```
# Assume dataset is a list of (data, label) tuples
def get_forget_data_indices(dataset, num_to_forget=500):
    return random.sample(range(len(dataset)), num_to_forget)

forget_indices = get_forget_data_indices(trainset)
```

Updating Models to Forget Specific Data: This function updates the models to unlearn specific data points.

Identifying Data to Forget: For each shard, it identifies which data points in the shard need to be forgotten based on the forget_indices.

Removing Data to Forget: It removes these data points from the shard, retaining only the remaining data.

Training on Remaining Data: The remaining data is then divided into slices. For each slice, the model is retrained, starting with the initial model and incrementally updating it with each new slice of data.

Storing Updated Models: The final updated model for each shard is stored

```
def update_relevant_slices(models, shards, forget_indices, num_slices, num_epochs):
    updated_models = []

    for shard_idx, shard in enumerate(shards):
        shard_indices = [i for i in shard]
        forget_in_shard = [i for i in shard_indices if i in forget_indices]

        if forget_in_shard:
            remaining_data = [trainset[i] for i in shard_indices if i not in forget_in_shard]
            slice_size = len(remaining_data) // num_slices
            updated_slices = []

            for slice_idx in range(num_slices):
                slice_data = remaining_data[: (slice_idx + 1) * slice_size]
                slice_loader = DataLoader(slice_data, batch_size=64, shuffle=True)

                if slice_idx == 0:
                    model = deepcopy(models[shard_idx])
                    model.train()
                else:
                    model = deepcopy(updated_slices[-1])
```

```
                optimizer = optim.SGD(model.parameters(), lr=0.001, momentum=0.9)

                for epoch in range(num_epochs):
                    for inputs, labels in slice_loader:
                        inputs = inputs.to(device)
                        labels = labels.to(device)

                        optimizer.zero_grad()
                        outputs = model(inputs)
                        loss = criterion(outputs, labels)
                        loss.backward()
                        optimizer.step()

                updated_slices.append(model)

            updated_models.append(updated_slices[-1])
        else:
            updated_models.append(models[shard_idx])

    return updated_models



def aggregate_models(models, dataloader):
    all_probs = []

    for model in models:
        model.eval()
        probs = []

        with torch.no_grad():
            for inputs, _ in dataloader:
                inputs = inputs.to(device)
                outputs = model(inputs)
                prob = torch.nn.functional.softmax(outputs, dim=1)
                probs.extend(prob.cpu().numpy())

        all_probs.append(probs)

    all_probs = np.array(all_probs)
    avg_probs = np.mean(all_probs, axis=0)
    return np.argmax(avg_probs, axis=1), avg_probs



S = 10
R = 10
forget_indices = get_forget_data_indices(trainset)
# Get the shards from the sisa training process
shard_size = len(trainset) // S
shards = [list(range(i * shard_size, (i + 1) * shard_size)) for i in range(S)]
criterion = nn.CrossEntropyLoss()
updated_models = update_relevant_slices(models, shards, forget_indices, num_slices=10, num_epochs=5)
# Aggregation methods (choose one to use)
test_loader = DataLoader(testset, batch_size=64, shuffle=False)
# Extract y_true from the test DataLoader
y_true = []
for _, labels in test_loader:
    y_true.extend(labels.numpy())
y_true = np.array(y_true)

# Averaging prediction vectors
y_pred, y_prob = aggregate_models(updated_models, test_loader)
final_metrics = calculate_metrics(y_true, y_pred, y_prob)
print(f"Metrics after unlearning: {final_metrics}")
```

```
Metrics after unlearning: (0.7641618542289331, 0.7657, 0.7638189919018865, 0.7657, 0.9683451944444444)
```

```
S_values = [5]
R_values = [5, 10]
num_epochs = 5

updated_models_dict = {}

criterion = nn.CrossEntropyLoss()
forget_indices = get_forget_data_indices(trainset)

for S in S_values:
    updated_models_dict[S] = {}
    for R in R_values:
        # Get the shards from the sisa training process
        shard_size = len(trainset) // S
        shards = [list(range(i * shard_size, (i + 1) * shard_size)) for i in range(S)]
        models = torch.load(f'sisa_models_S{S}_R{R}.pt')  # Load your saved models

        updated_models = update_relevant_slices(models, shards, forget_indices, num_slices=R, num_epochs=5)
        # Aggregation methods (choose one to use)
        test_loader = DataLoader(testset, batch_size=64, shuffle=False)
        # Extract y_true from the test DataLoader
        y_true = []
        for _, labels in test_loader:
            y_true.extend(labels.numpy())
        y_true = np.array(y_true)

        # Averaging prediction vectors
        y_pred, y_prob = aggregate_models(updated_models, test_loader)
        final_metrics = calculate_metrics(y_true, y_pred, y_prob)
        print(f"Metrics after unlearning with S = {S}, R = {R}: {final_metrics}")
```

```
Metrics after unlearning with S = 5, R = 5: (0.7792516679349943, 0.7805, 0.7801513304419404, 0.7805, 0.9709321)
Metrics after unlearning with S = 5, R = 10: (0.7954096899799903, 0.797, 0.7955113469567775, 0.797, 0.9746380666666667)
```

Q3:

```
from sklearn.linear_model import LogisticRegression
from sklearn.model_selection import cross_val_score
```

Aggregating Losses: This function calculates the losses for the aggregated model predictions over a given dataloader using a specified loss criterion.

Aggregate Model Predictions: First, it calls aggregate_models to get the average prediction probabilities from the ensemble of models.

Loop to Get Labels: It iterates through the dataloader again to get the true labels of the samples.

Calculate Loss for Each Sample: For each sample, it computes the loss using the criterion and the aggregated probabilities, storing these losses in a list.

Output: The function returns a list of loss values for each sample in the dataloader.

```
def get_aggregated_losses(models, dataloader, criterion):
    _, avg_probs = aggregate_models(models, dataloader)
    losses = []

    # Ensure avg_probs is a numpy array
    avg_probs = np.array(avg_probs)

    # Dataloader needs to be looped again to get labels
    all_labels = []
    for _, labels in dataloader:
        all_labels.extend(labels.numpy())

    for prob, label in zip(avg_probs, all_labels):
        label_tensor = torch.tensor([label], device=device)
        prob_tensor = torch.tensor([prob], device=device)
        loss = criterion(prob_tensor, label_tensor)
        losses.append(loss.item())

    return losses
```

Loading Pre-trained Models: Loads the pre-trained models from disk.

Creating Dataloaders: Creates dataloaders for the forget set (subset of the training data to be forgotten) and a random subset of the test set.

Initialize Loss Criterion: Initializes the loss criterion, in this case, cross-entropy loss.

Calculate Losses: Computes the losses for the forget set and the random test set using the previously defined get_aggregated_losses function.

Prepare Data: Prepares the input (X) and labels (y) for logistic regression. The input X is a combination of the losses from the forget set and the random test set, and the labels y are 1 for forget set losses and 0 for test set losses.

Train Logistic Regression: Trains a logistic regression model on the prepared data and computes the cross-validation accuracy score. This score measures the ability of the model to distinguish between the forget set and the test set, indicating the membership inference attack's success rate.

Random Train Set Losses: Creates a dataloader for a random subset of the training set and computes the losses for these samples.

Prepare Data: Prepares the input (X) and labels (y) for logistic regression. The input X is a combination of the losses from the random train set and the random test set, and the labels y are 1 for train set losses and 0 for test set losses.

Train Logistic Regression: Trains a logistic regression model on the prepared data and computes the cross-validation accuracy score. This score measures the ability of the model to distinguish between the train set and the test set, indicating the membership inference attack's success rate.

Calculate Losses for Unlearned Models: Repeats the process of calculating the losses for the forget set, random test set, and random train set using the updated models (after unlearning).

Prepare Data for Logistic Regression: Prepares the input (X) and labels (y) for logistic regression for both the forget set vs. test set and the train set vs. test set scenarios.

Train Logistic Regression: Trains a logistic regression model on the prepared data for the unlearned models and computes the cross-validation accuracy scores. These scores measure the ability of the model to distinguish between the forget/train set and the test set after unlearning, indicating the effectiveness of the unlearning process in mitigating the membership inference attack.

```
models = torch.load(f'sisa_models_S{5}_R{10}.pt')
# Create dataloaders for the forget set and a random subset of the test set
forget_set_loader = DataLoader(Subset(trainset, forget_indices), batch_size=64, shuffle=False)
random_test_indices = random.sample(range(len(testset)), 500)
random_test_loader = DataLoader(Subset(testset, random_test_indices), batch_size=64, shuffle=False)

# Initialize the criterion
criterion = nn.CrossEntropyLoss()

# Forget set and test set for the trained model (before unlearning)
forget_set_losses = get_aggregated_losses(models, forget_set_loader, criterion)
random_test_set_losses = get_aggregated_losses(models, random_test_loader, criterion)

# Prepare data for logistic regression
X = forget_set_losses + random_test_set_losses
y = [1] * len(forget_set_losses) + [0] * len(random_test_set_losses)
X = np.array(X).reshape(-1, 1)
y = np.array(y)

# Train logistic regression and compute cross-validation score
logistic_regression = LogisticRegression()
cv_score_trained1 = cross_val_score(logistic_regression, X, y, cv=5, scoring='accuracy').mean()
print(f'Cross-validation score for Membership Inference Attack (trained model and forget data): {cv_score_trained1}')

# Train set and test set for the trained model (before unlearning)
random_train_indices = random.sample(range(len(trainset)), 500)
random_train_loader = DataLoader(Subset(trainset, random_train_indices), batch_size=64, shuffle=False)
train_set_losses = get_aggregated_losses(models, random_train_loader, criterion)

# Prepare data for logistic regression
X = train_set_losses + random_test_set_losses
y = [1] * len(train_set_losses) + [0] * len(random_test_set_losses)
X = np.array(X).reshape(-1, 1)
y = np.array(y)

# Train logistic regression and compute cross-validation score
logistic_regression = LogisticRegression()
cv_score_trained2 = cross_val_score(logistic_regression, X, y, cv=5, scoring='accuracy').mean()
print(f'Cross-validation score for Membership Inference Attack (trained model and train data): {cv_score_trained2}')

# Repeat the same for the unlearned models
forget_set_losses_unlearned = get_aggregated_losses(updated_models, forget_set_loader, criterion)
random_test_set_losses_unlearned = get_aggregated_losses(updated_models, random_test_loader, criterion)
train_set_losses_unlearned = get_aggregated_losses(updated_models, random_train_loader, criterion)

# Prepare data for logistic regression for unlearned model
X_unlearned_forget = forget_set_losses_unlearned + random_test_set_losses_unlearned
y_unlearned_forget = [1] * len(forget_set_losses_unlearned) + [0] * len(random_test_set_losses_unlearned)
X_unlearned_forget = np.array(X_unlearned_forget).reshape(-1, 1)
y_unlearned_forget = np.array(y_unlearned_forget)

X_unlearned_train = train_set_losses_unlearned + random_test_set_losses_unlearned
y_unlearned_train = [1] * len(train_set_losses_unlearned) + [0] * len(random_test_set_losses_unlearned)
X_unlearned_train = np.array(X_unlearned_train).reshape(-1, 1)
y_unlearned_train = np.array(y_unlearned_train)

# Train logistic regression and compute cross-validation score for unlearned model
cv_score_unlearned1 = cross_val_score(logistic_regression, X_unlearned_forget, y_unlearned_forget, cv=5, scoring='accuracy').mean()
print(f'Cross-validation score for Membership Inference Attack (unlearned model and forget data): {cv_score_unlearned1}')

cv_score_unlearned2 = cross_val_score(logistic_regression, X_unlearned_train, y_unlearned_train, cv=5, scoring='accuracy').mean()
print(f'Cross-validation score for Membership Inference Attack (unlearned model and train data): {cv_score_unlearned2}')
```

```
Cross-validation score for Membership Inference Attack (trained model and forget data): 0.513
Cross-validation score for Membership Inference Attack (trained model and train data): 0.524
Cross-validation score for Membership Inference Attack (unlearned model and forget data): 0.5129999999999999
Cross-validation score for Membership Inference Attack (unlearned model and train data): 0.51
```

For a perfectly unlearned model, we would expect:

1. **Forget Data vs. Test Data Cross-Validation Score:**
    - The score should be close to 50%. This indicates that the Logistic Regression model cannot distinguish between the forget set and the test set better than random guessing. This would mean the model has successfully forgotten the data it was supposed to unlearn.

2. **Train Data vs. Test Data Cross-Validation Score:**
    - Similarly, this score should also be close to 50%. This would indicate that the model treats the forget set as if it were never seen during training, making it indistinguishable from the test set.

The closest value to 0.5 should be the last one, as it is.

Add On.Q1:

```
import torch
import torchvision
from torchvision import transforms, datasets
import torch.nn as nn
import torch.optim as optim
from torch.utils.data import DataLoader, Subset, Dataset
from sklearn.metrics import f1_score, accuracy_score, precision_score, recall_score, roc_auc_score
import numpy as np
import random
from copy import deepcopy
```

Functionality: This function adds a backdoor trigger (a black 3x3 block) to a random position in the image. Random Position: The x and y coordinates for the top-left corner of the block are chosen randomly, ensuring the block fits within the image dimensions. Trigger Addition: The specified region in the image is set to 0, creating the black block.

```
# Function to add backdoor trigger to an image
def add_backdoor_trigger(image):
    # Add a black 3x3 block to a random position in the image
    x = random.randint(0, image.shape[1] - 3)
    y = random.randint(0, image.shape[2] - 3)
    image[:, x:x+3, y:y+3] = 0
    return image
```

Functionality: This function poisons the training data by adding a backdoor trigger to a specified number of samples from a target class. Target Class: It iterates over the dataset, and if the label matches the target class, it adds a backdoor trigger to the image. Poisoned Indices: The indices of the poisoned samples are stored and returned.

```
# Function to poison the training data
def poison_training_data(dataset, target_class, num_samples=500):
    poisoned_indices = []
    for i, (image, label) in enumerate(dataset):
        if label == target_class and len(poisoned_indices) < num_samples:
            image = add_backdoor_trigger(image)
            poisoned_indices.append(i)
    return poisoned_indices


device = torch.device("cuda:0" if torch.cuda.is_available() else "cpu")
S_values = [5]
R_values = [10]
num_epochs = 5

models_dict = {}

# Training and saving models
for S in S_values:
    models_dict[S] = {}
    for R in R_values:
        print(f"Training with S = {S}, R = {R}")
        models = sisa_training(trainset, S, R, num_epochs, device)
        models_dict[S][R] = models
        torch.save(models, f'sisa_models_S{S}_R{R}.pt')
```

```
Training with S = 5, R = 10
/usr/local/lib/python3.10/dist-packages/torchvision/models/_utils.py:208: UserWarning: The parameter 'pretrained' is deprecated since
  warnings.warn(
/usr/local/lib/python3.10/dist-packages/torchvision/models/_utils.py:223: UserWarning: Arguments other than a weight enum or `None` f
  warnings.warn(msg)
Downloading: "https://download.pytorch.org/models/resnet18-f37072fd.pth" to /root/.cache/torch/hub/checkpoints/resnet18-f37072fd.pth
100%|██████████| 44.7M/44.7M [00:00<00:00, 182MB/s]
Epoch 0/4, Loss: 2.3780
```

```
Epoch 1/4, Loss: 2.1526
Epoch 2/4, Loss: 1.9477
Epoch 3/4, Loss: 1.7080
Epoch 4/4, Loss: 1.4681
Epoch 0/4, Loss: 1.4641
Epoch 1/4, Loss: 1.1232
Epoch 2/4, Loss: 0.8769
Epoch 3/4, Loss: 0.6666
Epoch 4/4, Loss: 0.4989
Epoch 0/4, Loss: 0.7139
Epoch 1/4, Loss: 0.4743
Epoch 2/4, Loss: 0.3170
Epoch 3/4, Loss: 0.2169
Epoch 4/4, Loss: 0.1549
Epoch 0/4, Loss: 0.4922
Epoch 1/4, Loss: 0.2782
Epoch 2/4, Loss: 0.1700
Epoch 3/4, Loss: 0.1209
Epoch 4/4, Loss: 0.1011
Epoch 0/4, Loss: 0.3877
Epoch 1/4, Loss: 0.2352
Epoch 2/4, Loss: 0.2072
Epoch 3/4, Loss: 0.1550
Epoch 4/4, Loss: 0.1331
Epoch 0/4, Loss: 0.3624
Epoch 1/4, Loss: 0.1975
Epoch 2/4, Loss: 0.1256
Epoch 3/4, Loss: 0.0837
Epoch 4/4, Loss: 0.0600
Epoch 0/4, Loss: 0.2621
Epoch 1/4, Loss: 0.1463
Epoch 2/4, Loss: 0.0950
Epoch 3/4, Loss: 0.0734
Epoch 4/4, Loss: 0.0481
Epoch 0/4, Loss: 0.2333
Epoch 1/4, Loss: 0.1298
Epoch 2/4, Loss: 0.0730
Epoch 3/4, Loss: 0.0511
Epoch 4/4, Loss: 0.0385
Epoch 0/4, Loss: 0.2005
Epoch 1/4, Loss: 0.1043
Epoch 2/4, Loss: 0.0603
Epoch 3/4, Loss: 0.0458
Epoch 4/4, Loss: 0.0337
Epoch 0/4, Loss: 0.1750
Epoch 1/4, Loss: 0.0870
Epoch 2/4, Loss: 0.0611
Epoch 3/4, Loss: 0.0403
```

```
device = torch.device("cuda:0" if torch.cuda.is_available() else "cpu")
S = 5
R = 10
num_epochs = 5

print(f"Training with S = {S}, R = {R} after applying the attack")
poisoned_models = train_poisoned_model(trainset, S, R, num_epochs, device)

# Evaluate on clean test data
test_loader = DataLoader(testset, batch_size=64, shuffle=False)
y_true = []
for _, labels in test_loader:
    y_true.extend(labels.numpy())
y_true = np.array(y_true)

y_pred, y_prob = aggregate_models(poisoned_models, test_loader)
final_metrics = calculate_metrics(y_true, y_pred, y_prob)
print(f"Metrics after poisoning and training: {final_metrics}")

# Calculate Attack Success Rate (ASR)
target_class = 3  # Replace with the specific class targeted in the attack
asr = calculate_asr(y_true, y_pred, target_class)
print(f"Attack Success Rate (ASR) on clean test data: {asr:.2f}%")
```

```
Training with S = 5, R = 10 after applying the attack
/usr/local/lib/python3.10/dist-packages/torchvision/models/_utils.py:208: UserWarning: The parameter 'pretrained' is deprecated since
  warnings.warn(
/usr/local/lib/python3.10/dist-packages/torchvision/models/_utils.py:223: UserWarning: Arguments other than a weight enum or `None` f
  warnings.warn(msg)
Epoch 0/4, Loss: 2.3861
```

```
Epoch 1/4, Loss: 2.1586
Epoch 2/4, Loss: 1.9532
Epoch 3/4, Loss: 1.7255
Epoch 4/4, Loss: 1.4886
Epoch 0/4, Loss: 1.4798
Epoch 1/4, Loss: 1.1233
Epoch 2/4, Loss: 0.8823
Epoch 3/4, Loss: 0.6842
Epoch 4/4, Loss: 0.4948
Epoch 0/4, Loss: 0.6969
Epoch 1/4, Loss: 0.4759
Epoch 2/4, Loss: 0.3125
Epoch 3/4, Loss: 0.2124
Epoch 4/4, Loss: 0.1586
Epoch 0/4, Loss: 0.5155
Epoch 1/4, Loss: 0.2719
Epoch 2/4, Loss: 0.1813
Epoch 3/4, Loss: 0.1134
Epoch 4/4, Loss: 0.0845
Epoch 0/4, Loss: 0.3841
Epoch 1/4, Loss: 0.3029
Epoch 2/4, Loss: 0.2025
Epoch 3/4, Loss: 0.1187
Epoch 4/4, Loss: 0.1096
Epoch 0/4, Loss: 0.3856
Epoch 1/4, Loss: 0.1793
Epoch 2/4, Loss: 0.1070
Epoch 3/4, Loss: 0.0825
Epoch 4/4, Loss: 0.0509
Epoch 0/4, Loss: 0.2644
Epoch 1/4, Loss: 0.1394
Epoch 2/4, Loss: 0.0840
Epoch 3/4, Loss: 0.0582
Epoch 4/4, Loss: 0.0510
Epoch 0/4, Loss: 0.2399
Epoch 1/4, Loss: 0.1144
Epoch 2/4, Loss: 0.0764
Epoch 3/4, Loss: 0.0549
Epoch 4/4, Loss: 0.0406
Epoch 0/4, Loss: 0.1879
Epoch 1/4, Loss: 0.1075
Epoch 2/4, Loss: 0.0677
Epoch 3/4, Loss: 0.0474
Epoch 4/4, Loss: 0.0465
Epoch 0/4, Loss: 0.1665
Epoch 1/4, Loss: 0.1058
Epoch 2/4, Loss: 0.0688
Epoch 3/4, Loss: 0.0526
Epoch 4/4, Loss: 0.0466
Epoch 0/4, Loss: 2.3868
```

```
import random
import numpy as np
import torch
from torch.utils.data import Dataset, DataLoader
from torch import nn, optim
from copy import deepcopy

device = torch.device("cuda" if torch.cuda.is_available() else "cpu")

# Function to add backdoor trigger to an image
def add_backdoor_trigger(image):
    # Add a black 3x3 block to a random position in the image
    x = random.randint(0, image.shape[1] - 3)
    y = random.randint(0, image.shape[2] - 3)
    image[:, x:x+3, y:y+3] = 0
    return image

# Function to poison the training data
def poison_training_data(trainset, target_class, num_samples=500):
    poisoned_indices = random.sample(
```

Functionality: This custom dataset class is designed to handle poisoned data. Initialization: The constructor takes the original dataset, a list of poisoned indices, and an optional target class.

Get Item: The **getitem** method returns an image and its label. If the index is in the list of poisoned indices, it adds a backdoor trigger to the image before returning it.

Length: The **len** method returns the length of the dataset.

```
def evaluate_model(model, dataloaders):

# Custom dataset class to handle poisoned data
class PoisonedDataset(Dataset):
    def __init__(self, original_dataset, poisoned_indices):
        self.dataset = original_dataset
        self.poisoned_indices = poisoned_indices

    def __getitem__(self, index):
        image, label = self.dataset[index]
        if index in self.poisoned_indices:
            image = add_backdoor_trigger(image)
        return image, label

    def __len__(self):
        return len(self.dataset)
```

Explanation:

Functionality: This function evaluates the effectiveness of a backdoor attack on aggregated models. Dataloaders: Creates dataloaders for the poisoned training set and clean test set. Training: Trains each model on the poisoned dataset for a specified number of epochs. Evaluation on Clean Data: Evaluates the aggregated models on the clean test set, calculating and printing the performance metrics. Evaluation on Backdoor Data: Evaluates the aggregated models on a backdoor test set, calculating and printing the attack success rate (ASR), which measures the proportion of backdoor samples classified as the target class.

```
# Function to evaluate backdoor attack for aggregated models
def evaluate_backdoor_attack_aggregated(models, target_class):
    # Poison the test dataset
    backdoor_testset = deepcopy(testset)
    poisoned_testset = PoisonedDataset(backdoor_testset, range(len(backdoor_testset)))

    clean_test_loader = DataLoader(testset, batch_size=64, shuffle=False)
    backdoor_test_loader = DataLoader(poisoned_testset, batch_size=64, shuffle=False)
```