

**FIT5124 Emerging topics for cybersecurity in practice**  
**Assessment 3**

**Name: Ka Yong Cheah**

**Student id: 34789596**

### **Task 3.1 Attack Description (6 Marks):**

#### **Attack Objective:**

The objective of this attack is to extract a surrogate model that closely mimic the prediction behaviour of the target model. By querying the target model using input and recording its output confidence scores, attacker can train a local model that replicates the target model's decision boundaries and accuracy. The attacker does not aim to extract the target model's parameters or architectures, rather to reproduce its functionality for selling predictions or provide model to other users.

#### **Attacker's Capability:**

The attacker has no access to the target model's training process. The architecture, training parameters and optimisation details are unknown. The attacker also has no access to the trained model or internal weights. However, the attacker has full access to the same public dataset (MNIST) used to train the model but not knowing exactly which subset.

The model is deployed as a black box API that returns the full SoftMax confidence scores for each input query. The attacker can send input images and receive the complete probability distribution over all classes. These outputs are used as training targets to build a surrogate model that mimic the original model's behaviour.

#### **Practical Scenario Example:**

An EdTech startup uses a third-party digit recognition API in its math learning app to evaluate handwritten student answers. The provider publicly states the model is trained on MNIST and charges per query. For transparency, it returns full SoftMax confidence scores for each digit. To reduce cost, the startup performs a model extraction attack. It queries the target API using MNIST digits, collects the output probabilities, and trains a local surrogate model that replicates the original model's predictive behaviour. The attacker has no access to the model's architecture, parameters or training process. However, they have full access to the MNIST dataset because it is public and unrestricted black box query access to the API, including its confidence scores.

### **Task 3.2 Attack Design (6 Marks):**

#### **Step 1: Define surrogate model**

This step is essential because the attacker needs a model to replicate the target's behaviour. As the target model's architecture is unknown, the attacker must define a custom surrogate. This aligns with the black-box threat model described in 3.1, where internal details are inaccessible. By training this surrogate on the query-response pairs, the attacker achieves the goal of creating a functional clone that can be deployed or monetised independently.

#### **Step 2: Query target model using public dataset and record it**

The attacker uses the public MNIST dataset to generate input queries. Although the API provider publicly announces that the model was trained on MNIST, the attacker has no access to the exact subset or preprocessing steps used. Querying the target model with these inputs yields SoftMax confidence scores, which are recorded to build a dataset of input-output pairs. This step is essential for training the surrogate model, as the rich output probabilities help capture the target model's decision boundaries. It aligns with the black-box

threat model, where the attacker has access to public data and can query the API but lacks internal model knowledge.

#### Step 3: Train surrogate model on collected data

After collecting the target model's outputs, the attacker trains the surrogate model using Mean Squared Error (MSE) loss between its predictions and the target's SoftMax scores. It contributes to the attack goal by enabling the surrogate to mimic not only the target's predictions but also the confidence behind them. It directly supports the attack goal of building a functionally equivalent model and fits the black-box threat model, as training is done entirely offline using only the collected input-output pairs.

#### Step 4: Evaluate surrogate mode's accuracy against target model

This step measures how well the surrogate model replicates the target model's behaviour. The attacker compares their predicted classes on the same set of input queries. Each time the surrogate matches the target's prediction, it is counted as correct. The chosen evaluation metric is extraction accuracy, which reflects how closely the surrogate mimics the target model's outputs. This metric directly supports the attack goal and is suitable for the black-box threat model, as it relies only on input queries and output predictions without requiring access to internal model details.

### Task 3.3 Implementation and Evaluation (6 Marks):

```
# Step 1: Simulate black-box API to access the target model
# Simulate the black-box API interface
def query_target_model(input_batch):
    # Load and run the target model inside a private API function
    # This simulates the server-side execution of the black-box model
    class Lenet(nn.Module):
        # Defines the target model architecture
        # Although the attacker doesn't know this architecture,
        # it is defined here internally to simulate the "hidden" server-side model
        def __init__(self):
            super(Lenet, self).__init__()
            self.conv1 = nn.Conv2d(1, 6, 3, stride=1, padding=1)
            self.conv2 = nn.Conv2d(6, 16, 5, stride=1, padding=0)
            self.dropout1 = nn.Dropout(0.25)
            self.dropout2 = nn.Dropout(0.5)
            self.fc1 = nn.Linear(400, 120)
            self.fc2 = nn.Linear(120, 84)
            self.fc3 = nn.Linear(84, 10)

        def forward(self, x):
            x = self.conv1(x)
            x = F.relu(x)
            x = F.max_pool2d(x, 2)
            x = self.conv2(x)
            x = F.relu(x)
            x = F.max_pool2d(x, 2)
            x = self.dropout1(x)
            x = torch.flatten(x, 1)
            x = self.fc1(x)
            x = F.relu(x)
            x = self.dropout2(x)
            x = self.fc2(x)
            x = F.relu(x)
            x = self.dropout2(x)
            x = self.fc3(x)
            output = F.log_softmax(x, dim=1)
            return output

    model = Lenet()          # Creates an instance of the LeNet model
    # Loads the pre-trained weights of the target model from a saved file
    model.load_state_dict(torch.load('target_model.pth'))
    # Evaluation mode, disabling dropout and other training-specific behaviors
    model.eval()
    with torch.no_grad():
        return model(input_batch)
```

- Simulated black-box API interface that loads and evaluates the target model in a server-like manner.

- Return model(input\_batch):  
This passes the input batch through the target model and returns the predicted log-SoftMax probabilities for each class. These predictions simulate what an attacker would receive from a real-world m-

- Simulated black-box API interface that loads and evaluates the target model in a server like manner.

```

"""Step 2: Define the attacker's surrogate model architecture."""
class Attack(nn.Module):
    def __init__(self):
        super(Attack, self).__init__()
        # First fully connected layer:
        # Input: 28x28 pixels (flattened to 784) → Output: 128 features
        self.fc1 = nn.Linear(28 * 28, 128)
        # Second fully connected layer:
        # Input: 128 → Output: 64 features
        self.fc2 = nn.Linear(128, 64)
        # Final fully connected layer:
        # Input: 64 → Output: 10 classes (digits 0-9)
        self.fc3 = nn.Linear(64, 10)

    def forward(self, x):
        """Forward pass with log-softmax output to match the target model."""
        x = x.view(-1, 28 * 28) # flatten input
        x = F.relu(self.fc1(x)) # Pass first layer and apply ReLU activation
        x = F.relu(self.fc2(x)) # Pass second layer and apply ReLU activation
        x = self.fc3(x) # Pass through the final classification layer
        return F.log_softmax(x, dim=1) # match output format of the target model

```

- A simple 3-layer MLP used to mimic the target model.
- It accepts flattened 28×28 MNIST images.
- It outputs log SoftMax probabilities over 10 classes just like the target model.

```

"""Step 3: Define normalisation transform for MNIST input images."""
transform = transforms.Compose([
    transforms.ToTensor(),
    transforms.Normalize((0.1307,), (0.3081,))])

""" Step 4: Load dataset (attacker uses only test set)."""
# test set → 10,000 images -> for attacker to test query
dataset2 = datasets.MNIST('./data', train=False, transform=transform)
test_loader = DataLoader(dataset2, batch_size=64, shuffle=True)

""" Step 5: Initialize the attacker's surrogate model."""
extracted_model = Attack()

"""

Step 6: Configure number of queries
Simulate attacker querying the black-box model and collecting responses
Use the test set as the attacker's query source
"""

attack_percentage = 0.3
attack_number = int(attack_percentage * len(dataset2))

```

- Preparing attacker input data and configuring surrogate model.
- This code normalises MNIST images using standard mean and std values, loads the public MNIST test set used as attacker query inputs, initialises the surrogate model and defines the number of queries 30% of the test set (can be control). The attacker does not use the training set, aligning with the black-box threat model.

```

"""

Step 7: Query the black-box model and collect (input, output) response pairs.
These pairs form the training set for the attacker's surrogate model.
"""

# Initialize lists to store input images and
# corresponding softmax outputs from the target model
x_data = []
y_data = []

# Collect outputs by querying the target model via the simulated API
with torch.no_grad():
    for data, _ in test_loader:
        # Query the black-box target model with input data
        preds = query_target_model(data)
        # Record the input batch and the model's predicted softmax outputs
        x_data.append(data)
        y_data.append(preds)
        # Stop collecting once the desired number of queries is reached
        if len(torch.cat(x_data)) >= attack_number:
            break

# Concatenate and truncate the collected data to exactly match the configured query count
x_all = torch.cat(x_data)[:attack_number]
y_all = torch.cat(y_data)[:attack_number]

```

- This code collects input, SoftMax output pairs by querying the black-box target model with MNIST test images. The input data and predicted SoftMax scores are stored and truncated to match the configured query count. These pairs form the dataset used to train the surrogate model.

```

"""
Step 8: Train the surrogate model using Mean Squared Error loss.
The attacker attempts to mimic the full confidence distribution of the target.
"""

# Define the loss function (MSE) to match predicted and target softmax scores
criterion = nn.MSELoss()
# Use the Adam optimizer to train the surrogate model
optimizer = torch.optim.Adam(extracted_model.parameters(), lr=0.001)

print("Training stolen model on queries...")

for epoch in range(10): # adjust this for the number of epochs
    extracted_model.train() # set the model to training mode
    running_loss = 0.0
    # Loop through the pseudo-labeled dataset in batches of 64
    for i in range(0, len(x_all), 64):
        x_batch = x_all[i:i+64] # Input batch
        y_batch = Y_all[i:i+64] # Corresponding soft labels

        optimizer.zero_grad() # Reset gradients
        out = extracted_model(x_batch) # Forward pass
        loss = criterion(out, y_batch) # Compute MSE loss
        loss.backward() # Backpropagate
        optimizer.step() # Update weights

        print(f"Epoch {epoch+1} Loss: {loss.item():.6f}")

"""

Step 9: Evaluate the extraction effectiveness by comparing predictions of the
surrogate model against the original target model on the test set.
"""

correct = 0 # Counter for correct predictions
total = 0 # Total number of samples evaluated

extracted_model.eval() # Set surrogate model to evaluation mode
with torch.no_grad():
    for data, _ in test_loader:
        # Query the target and surrogate models on the same data
        target_pred = query_target_model(data).argmax(dim=1) # Target model's predicted class
        extracted_pred = extracted_model(data).argmax(dim=1) # Surrogate model's predicted class

        # Compare predictions and count correct matches
        correct += extracted_pred.eq(target_pred).sum().item()
        total += len(data)
    print(
        f"\nExtraction Accuracy: {correct}/{total} "
        f"({100 * correct / total:.2f}%) using {attack_number} queries"
    )

```

- This code trains the surrogate model using Mean Squared Error (MSE) loss to match the SoftMax outputs of the target model. The attacker uses the Adam optimizer with a learning rate of 0.001 and iterates over the pseudo-labelled dataset for 10 epochs in batches of 64. The surrogate learns to mimic the full confidence distribution of the target model, aligning with the attack goal of reproducing its prediction behaviour.

- The code evaluates the surrogate model by comparing its predictions with those of the target model on all 10000 MNIST test images. For each input, it checks whether the surrogate and target model predict the same class. It then calculates the extraction accuracy as the percentage of matching predictions. This metric reflects how closely the surrogate replicates the target's behaviour using only black box access.

```

kche0213@FIT5124-VM:~/shared_folder/a3_real$ python3 black_model_extraction_attack.py
Training stolen model on queries...
Epoch 1 Loss: 111.549919
Epoch 2 Loss: 85.076927
Epoch 3 Loss: 68.624695
Epoch 4 Loss: 59.160896
Epoch 5 Loss: 53.521065
Epoch 6 Loss: 46.567089
Epoch 7 Loss: 35.539482
Epoch 8 Loss: 30.063629
Epoch 9 Loss: 21.464376
Epoch 10 Loss: 16.205873
Total extraction time: 0.97 seconds

Extraction Accuracy: 8966/10000 (89.66%) using 3000 queries

```

## Results and Evaluation

The attack's performance was evaluated by measuring extraction accuracy score, which is the percentage of inputs for which the surrogate model predicted the same class as the target model. The evaluation was conducted on all 10000 MNIST test images, providing a reliable measure of behavioural similarity. By using only 30% of the 10000 MNIST test images which is 3000 queries, the surrogate model achieves an accuracy of 89.66% after 10 training epochs. This high accuracy demonstrates the effectiveness and successful of the attack in replicating the target model's output behaviour. Extraction accuracy was chosen as the metric because it directly reflects the attack goal of how well the model mimicking the target's predictions in a black box setting.

### Task 3.4 Attack Analysis (7 Marks):

**Effectiveness:** This attack successfully achieves its goal of replicating the target model's behaviour define in task 3.1. As shown in result and evaluation, the surrogate model reached an accuracy of 89.66%, indicating strong alignment between the surrogate and target predictions. This evident that the surrogate can effectively mimic the target model in a black box setting.

**Efficiency:** The model extraction is very efficient because extraction was completed in just 10 training epochs, with both querying and training taking approximately 0.97 seconds on a standard CPU. The attacker was able to replicate the model's behaviour using only 3000 samples, making the process highly cost-effective compared to training a model from scratch with the full 60,000-image MNIST training set, which would require significantly more time and resources. This statistic demonstrates the efficiency and practicality of the attack, especially in scenarios where access to the original training pipeline is limited or costly.

### Impact Factors:

Epoch	Number of data	Learning rate	Extraction accuracy score
5	3000	0.001	37.45%
10	3000	0.001	89.66%
15	3000	0.001	91.19%

This table experiment conclude that extraction accuracy increased significantly with more training epochs, rising from 37.45% (5 epochs) to 89.35% (10 epochs), and then to 91.19% (15 epochs). This indicates that longer training improves surrogate performance.

Epoch	Number of data	Learning rate	Extraction accuracy score
10	1500	0.001	67.00%
10	3000	0.001	89.66%
10	4500	0.001	91.58%

This table experiment conclude that extraction accuracy improved with larger query budgets, increasing from 67.00% (1,500 queries) to 89.35% (3,000 queries) and 91.58% (4,500 queries). This indicates that more query budgets improve surrogate performance.

Epoch	Number of data	Learning rate	Extraction accuracy score
10	3000	0.0001	9.97%
10	3000	0.001	89.66%
10	3000	0.01	92.28%

This table experiment conclude that extraction accuracy is highly sensitive to the learning rate. A very low learning rate (0.0001) results in poor surrogate performance (9.97%), while

higher learning rates (0.001 and 0.01) significantly improve extraction accuracy to 89.35% and 92.28%, respectively. This indicates that model extraction success depends strongly on the attacker's training configuration.

## Danger and Limitations

The success of this attack demonstrates that even without access to a model's architecture or exact training data, an attacker can reconstruct a highly accurate surrogate using only query access and public data. This poses a serious risk to machine learning models deployed via APIs, particularly when full confidence scores are exposed, as they can be exploited to reverse engineer the model's behaviour with little resources. However, there are limitations, first the attack assumes the training data distribution (MNIST) is publicly known and like what the target model was trained on, which may not be true in all cases some companies do not announce what data had been used. Secondly, the effectiveness drops significantly if the model returns only predicted class labels instead of full SoftMax outputs, as class labels provide less information and prevent the surrogate from learning the target model's confidence and decision boundaries.

## Task 4.1 Defence Overview (6 Marks)

### Defence Objective:

The objective of this defence is to make model extraction attacks more costly by reducing how much useful information the attacker gets from each query. By reducing the usability of each query response, attackers must send significantly more queries to achieve comparable performance.

The success of this defence is evaluated using the following criteria:

- Increased number of queries or training epochs required to achieve performance comparable to the original extraction, indicating higher cost for the attacker.
- Reduced extraction accuracy under a 3000 query budget, demonstrating the defence's ability to weaken the surrogate model's ability to replicate the target.

### Defender Capability:

The defender is the owner of the model and has full access to the target model's training and inference processes, enabling modifications to the output format before serving predictions. They also have complete access to the trained model, including its architecture, parameters and original training data.

### Practical Scenario Example:

A technology company trains and deploys a digit recognition model through a public API used by third-party educational apps. Initially, the API returned full confidence scores for each prediction. However, the company later discovered that these scores could be exploited to reverse engineer the model. To address this, the company decides to implement a label-only defence, returning only the predicted class label to reduce information leakage. As the model owner, the company has full control over the training and inference processes, including access to the model architecture, training data and the ability to modify output

formats. These capabilities enable the implementation of output-level defences without affecting the model's internal performance.

## 4.2 Defence Design (6 Marks)

Step 1: Modify API to return only the top predicted label

This step replaces the original output, which included full SoftMax confidence scores, with a single predicted class label. It is included to limit the amount of information leaked per query, as full confidence scores reveal decision boundaries that attackers can exploit. By exposing only the final decision, the attacker loses access to the model's confidence distribution, making it significantly inefficient to train an accurate surrogate. This supports the defence goal of increasing extraction difficulty while providing functional predictions for users and addresses the model extraction attack described in Task 3, which relied on using full SoftMax outputs for training.

Step 2: Convert the predicted label into a log one-hot vector

To maintain the same output format (log-SofMax) expected by the system, the predicted class label is converted into a one-hot vector, where only the correct class is marked with 1 and the rest are 0. A small constant is added to avoid taking the log of zero, and then a logarithmic transformation is applied. This step ensures that the output shape remains unchanged while leaking no confidence information beyond the predicted class. It fits the defence objective by keeping the API stable for legitimate use while reducing the learning signal available to an attacker.

## 4.3 Implementation and Evaluation (6 Marks)

```
def run_attack(query_model, label="Original"):
    start_time = time.time() # Start total runtime timer
    # Initialize the attacker's surrogate model
    extracted_model = Attack()
    # Define the loss function (MSE) to match predicted and target softmax scores
    criterion = nn.MSELoss()
    # Use the Adam optimizer to train the surrogate model
    optimizer = torch.optim.Adam(extracted_model.parameters(), lr=0.001)

    # Initialize lists to store input images and
    # corresponding softmax outputs from the target model
    x_data = []
    y_data = []
    # Collect outputs by querying the target model via the simulated API
    with torch.no_grad():
        for data, _ in test_loader:
            # Query the black-box target model with input data
            preds = query_model(data)
            # Record the input batch and the model's predicted softmax outputs
            x_data.append(data)
            y_data.append(preds)
            # Stop collecting once the desired number of queries is reached
            if len(torch.cat(x_data)) >= attack_number:
                break
    # Concatenate and truncate the collected data to exactly match the configured query count
    x_all = torch.cat(x_data)[:attack_number]
    y_all = torch.cat(y_data)[:attack_number]

    total_loss = 0.0 # Track total loss over all epochs
    for epoch in range(10): # adjust this for the number of epochs
        extracted_model.train() # set the model to training mode
        epoch_loss = 0.0
        # Loop through the pseudo-labeled dataset in batches of 64
        for i in range(0, len(x_all), 64):
            x_batch = x_all[i:i+64] # Input batch
            y_batch = y_all[i:i+64] # Corresponding soft labels
            optimizer.zero_grad() # Reset gradients
            out = extracted_model(x_batch) # Forward pass
            loss = criterion(out, y_batch) # Compute MSE loss
            loss.backward() # Backpropagate
            optimizer.step() # Update weights
            epoch_loss += loss.item()
        print(f"[{label}] Epoch {epoch+1} Loss: {loss.item():.6f}")
        total_loss += epoch_loss
    avg_loss = total_loss / 10
```

- This function Executes a model extraction attack by training a surrogate model based on outputs from a target model.

- Parameters:

`query_model` (callable): A function that takes a batch of input images and returns the model's output either full SoftMax/log-SofMax or label-only).

`label` (str): A label to identify the evaluation output "Original" or "Label-Only Defence".

- Process:

1. Query the target model using MNIST test data to build the attack dataset.
2. Train a surrogate model using the collected (input, output) pairs.

```

# Evaluation
correct = 0 # Counter for correct predictions
total = 0 # Total number of samples evaluated
with torch.no_grad():
    for data, _ in test_loader:
        # Query the target and surrogate models on the same data
        target_pred = query_model(data).argmax(dim=1) # Target model's predicted class
        extracted_pred = extracted_model(data).argmax(dim=1) # Surrogate model's predicted class
        # Compare predictions and count correct matches
        correct += extracted_pred.eq(target_pred).sum().item()
        total += len(data)

accuracy = 100 * correct / total
end_time = time.time() # End total runtime timer
total_runtime = end_time - start_time

print(f"\n[{label}] Extraction Accuracy: {correct}/{total} ({accuracy:.2f}%)")
print(f"Queries used: {attack_number}")
print(f"Average Training Loss: {avg_loss:.6f}")
print(f"Total Runtime: {total_runtime:.2f} seconds")

```

```

def query_defended_model(input_batch):
    # Load the trained target model (same architecture as used during training)
    model = LeNet()
    model.load_state_dict(torch.load("target_model.pth"))
    model.eval() # evaluation mode
    with torch.no_grad():
        # Run the model on the input batch to get log-softmax output
        output = model(input_batch) # log_softmax

        # Label-only defence
        # Get the index of the most confident class (argmax)
        predicted_labels = torch.argmax(output, dim=1)
        # Convert the predicted label(s) to a one-hot encoded vector
        one_hot = F.one_hot(predicted_labels, num_classes=10).float()
        # Apply log transformation to match original log-softmax format
        # A small constant (1e-8) is added to avoid log(0)
        defended_output = torch.log(one_hot + 1e-8)
    return defended_output

```

```

# Run attacks
print("\n ATTACK ON ORIGINAL MODEL")
run_attack(query_target_model, label="Original")

print("\n ATTACK ON DEFENDED MODEL (LABEL-ONLY)")
run_attack(query_defended_label_only, label="Label-Only Defence")

```

```

ATTACK ON ORIGINAL MODEL
[Original] Epoch 1 Loss: 107.470093
[Original] Epoch 2 Loss: 82.830322
[Original] Epoch 3 Loss: 68.160660
[Original] Epoch 4 Loss: 60.601925
[Original] Epoch 5 Loss: 47.803085
[Original] Epoch 6 Loss: 22.963680
[Original] Epoch 7 Loss: 19.074705
[Original] Epoch 8 Loss: 16.513044
[Original] Epoch 9 Loss: 14.648273
[Original] Epoch 10 Loss: 13.746671

[Original] Extraction Accuracy: 8970/10000 (89.70%)
Queries used: 3000
Average Training Loss: 2834.658691
Total Runtime: 3.10 seconds

```

```

ATTACK ON DEFENDED MODEL (LABEL-ONLY)
[Label-Only Defence] Epoch 1 Loss: 64.932045
[Label-Only Defence] Epoch 2 Loss: 56.888203
[Label-Only Defence] Epoch 3 Loss: 51.231052
[Label-Only Defence] Epoch 4 Loss: 45.572773
[Label-Only Defence] Epoch 5 Loss: 39.074848
[Label-Only Defence] Epoch 6 Loss: 33.926109
[Label-Only Defence] Epoch 7 Loss: 29.552155
[Label-Only Defence] Epoch 8 Loss: 25.455402
[Label-Only Defence] Epoch 9 Loss: 22.799885
[Label-Only Defence] Epoch 10 Loss: 21.621376

[Label-Only Defence] Extraction Accuracy: 6744/10000 (67.44%)
Queries used: 3000
Average Training Loss: 2157.032652
Total Runtime: 3.09 seconds

```

3. Evaluate the surrogate by comparing its predictions with the target model.

4. Report extraction accuracy, average training loss, and total runtime.

- The model first performs a standard inference to produce log-softmax outputs. Then it replaces the full softmax output with a one-hot vector containing only the predicted class. This vector is log-transformed to match the original output format. The approach hides confidence scores, limiting the information available to attackers.

- Parameters:

input\_batch: A batch of input images from MNIST

- This code runs extraction attacks on both the original and defended models using different query functions. It allows direct comparison of extraction accuracy to evaluate the impact of the label-only defence.

## **Results and Evaluation:**

The objective of the defence is to increase the cost of model extraction by limiting the amount of useful information provided in each query response. To assess its effectiveness, surrogate models were trained using outputs from both the original target model, which returned full confidence scores, and the defended model, which returned only predicted labels. Extraction accuracy was used as the primary evaluation metric, as it directly reflects the surrogate model's ability to replicate the target model's predictions. This metric aligns with the defence goal of reducing extractability. The label only defence reduced extraction accuracy from 89.70% to 67.44% under the same query budget of 3000, indicating a significant decline in the surrogate's effectiveness. Although the average training loss decreased from 2834.66 to 2157.03, this reflects the reduced complexity of the defended output, which consists of log one-hot vectors. These simplified outputs are easier to fit, but carry significantly less information, confirming that the defence effectively limits learning signal without increasing training difficulty. Total runtime was also measured and showed no significant change, remaining approximately 3.10 seconds in both settings. These metrics were selected to evaluate the defence's effectiveness in degrading extraction performance while preserving system efficiency and functionality. These findings confirm that both defence objectives outlined in Task 4.1 have been achieved.

### **4.4 Defence Analysis (4 Marks)**

**Effectiveness:** The defence objective in Task 4.1 was to weaken and increase the cost of attacker to extract a surrogate model that mimics the target model. Results from Task 4.3 show that the extraction accuracy dropped from 89.70% to 67.44% under the same query budget of 3,000. This significant reduction confirms that the label only defence effectively limit the surrogate model's ability to replicate the target, thus meeting the objectives

**Efficiency:** The defence modifies only the output by converting log-SoftMax outputs to log-transformed one-hot vectors. This output level adjustment introduces no additional inference time, with total runtime remaining around 3.10 second in both defended and undefended settings. Since the defence does not alter the model architecture or training process, its computational complexity remains  $O(N)$ , where  $N$  is the batch size and the number of classes is constant. This makes the defence lightweight, efficient and practical for real-world deployment.

#### **Impact factors:**

Epoch	Number of data	Learning rate	Extraction accuracy scores defend model	Extraction accuracy scores original model
5	3000	0.001	20.54%	41.83%
10	3000	0.001	67.44%	89.70%
15	3000	0.001	84.97%	91.13%

Increasing the number of training epochs improves extraction accuracy for both the defended and original models. However, the defended model yields lower accuracy compared to the original, even as training time increases. This suggests the defence remains effective across training durations, but its strength diminishes with prolonged training, allowing the surrogate to partially adapt over time.

Epoch	Number of data	Learning rate	Extraction accuracy scores defend model	Extraction accuracy scores original model
-------	----------------	---------------	---	---

10	1500	0.001	35.70%	55.08%
10	3000	0.001	67.44%	89.70%
10	4500	0.001	86.09%	91.94%

Extraction accuracy increases with more training data for both models. However, the defended model consistently remains less extractable, demonstrating defence effectiveness. This also indicates that achieving high accuracy requires the attacker to incur higher costs.

Epoch	Number of data	Learning rate	Extraction accuracy scores defend model	Extraction accuracy scores original model
10	3000	0.0001	9.84%	9.90%
10	3000	0.001	67.44%	89.70%
10	3000	0.01	90.40%	92.19%

Extraction accuracy increases with higher learning rates, showing that surrogate model training quality impacts attack success. However, the defended model remains harder to extract, demonstrating robustness of the defence even under aggressive learning configurations.

#### 4.5 Using ChatGPT for Further Analysis (3 Marks)

<https://chatgpt.com/share/684c4e0c-6d38-8007-bd38-b3d817ea8cf8>

##### GPT prompt:

**Defence method:** The defence method I use is change the model's API output by replacing full confidence scores with only the predicted class label. This label is converted into a one-hot encoded vector and transformed using the logarithm function, with a small constant added to prevent log(0). The output format mimics the original log-softmax structure but reveals no confidence distribution. This weaken the attacker ability to train accurate surrogate models while maintaining prediction functionality for legitimate users.

**Define attacker capabilities:** The advanced attacker is aware of the defence strategy, including the removal of softmax scores and the use of log-transformed one-hot vectors. The attacker also understands the transformation steps and associated hyperparameters. While they do not have access to the model's architecture or training data, they can optimise query inputs and tune surrogate models to improve extraction attempts within a limited query budget.

ChatGPT's answer is correct, it clearly capturing the limitations of a label-only defence. It correctly notes that removing confidence scores disrupt common attack strategies and increases the cost for attackers. It also states that attackers can still perform label-only attacks using optimised queries. This aligns with experimental results showing that while the defence does not fully prevent extraction, it effectively limits information leakage and raises the cost of attack, achieving goal define in task 4.1.

Below is partial answer from Chat GPT, original answer is too long to include.

 **Why It Still Works (Partially)**

1. You remove softmax/confidence scores, which:
  - Breaks gradient-based surrogate learning.
  - Blocks common membership inference attacks.
  - Prevents confidence-calibrated model stealing.
2. The attacker only gets log(one-hot) labels:
  - Gives them just a binary answer (correct class or not).
  - No ranking or distance information between classes.
  - Makes decision boundary probing more difficult and query-expensive.

So, yes — your defense limits the attacker's effectiveness, especially under a query budget. Their surrogate will be less accurate, especially in complex or ambiguous regions of the input space.

 **Why It Doesn't Fully Work**

Your attacker is:

- Aware of your defense strategy and its hyperparameters.
- Optimizing their queries to efficiently explore decision boundaries.
- Training surrogates with only labels (label-only attacks, e.g., Copycat CNN, Knockoff Nets, Label-only MIA).

So they can still:

- Perform adaptive label-only attacks.
- Use synthetic or augmented datasets.
- Approximate the decision boundaries with clever input selection.

Even with just labels, research shows it's possible to extract functional surrogates, especially when:

- The attacker has a limited but smartly used query budget.
- The original model has sharp decision boundaries.

## **5. Membership Inference (25 marks)**

### **5.1 Attack Overview (5 Marks)**

#### **Attacker's Capability:**

The attacker operates in a black box setting with no access to the model's architecture, parameters or training process. Attacker did not know which datasets were used and training data distribution. However, the attacker suspects that the training data may include public digit datasets like MNIST. Despite this uncertainty, the attacker can query the deployed model through a public API that returns full SoftMax confidence vectors for any given input. By leveraging only these output predictions, the attacker aims to infer whether MNIST dataset were likely used during training.

#### **Practical Scenario Example:**

An EdTech startup uses a third-party digit recognition API in its math learning app to evaluate handwritten student answers. While the API delivers high accuracy, its training data remains unknown. Curious about the origin of the model's performance, the startup suspects that the model may have been trained on the publicly available MNIST dataset. However, they lack confirmation and do not know the exact samples or distribution used in training. To validate their suspicion, operating under a black box setting, the attacker has no knowledge of the model's architecture or training process but can query the API and receive full SoftMax confidence scores in return. They simulate member and non-member queries using samples from the MNIST and EMNIST datasets, respectively, and train an attack model to distinguish between them based on the confidence outputs. They aim to determine whether the model memorises MNIST examples, thereby revealing their presence in the training set. The potential impact of such an attack includes exposing proprietary training data practices, revealing dataset leakage, and raising privacy concerns, especially if the model had been trained on sensitive or licensed data.

### **5.2 Attack Design (6 Marks)**

#### **Step 1: Simulate member and non-member samples**

To simulate a membership inference setting, the attacker selects two subsets from the MNIST and EMNIST datasets. The member samples consist of 10000 images randomly drawn from the MNIST training set, assumed to represent "in-training" data. In contrast, the non-member samples are 10000 images taken from the EMNIST training set, chosen to represent "out-of-training" data from a similar but distinct distribution. Although the attacker does not know the exact subset of MNIST data used to train the target model, they suspect it was trained on data from the same distribution. This approach aligns with the noisy real data strategy often used in practical membership inference attacks, where the goal is to approximate training behaviour using publicly available or similar datasets. This step is critical because it allows the attacker to generate labelled data for training an attack model, which learns to distinguish between the behaviour of member and non-member samples based on the output responses from the target model.

#### **Step 2: Query the target model**

Both member and non-member samples are queried against the target model, which returns SoftMax confidence vectors. Member outputs are labelled as 1, non-members as 0, forming

a labelled dataset for training an attack model. This step is critical because it enables the attacker to distinguish between training and non-training data solely based on output behaviour, which directly supports the attack goal of inferring whether the MNIST dataset was used during training. It also aligns with the black-box threat model described in Subtask 5.1, where the attacker relies only on model outputs without access to internal parameters or training data.

#### Step 3: Train the attack model

The attacker uses the labelled SoftMax outputs get from target model to train a Logistic Regression model that predicts whether an input is a member of the target model's training set. The combined dataset is randomly split into 80% for training and 20% for testing to ensure reliable evaluation. By learning statistical patterns such as higher confidence or lower entropy in member outputs, the attack model captures distinctions in prediction behaviour. This step directly supports the attack goal of inferring training data presence and is well-suited to the black-box threat model described in Subtask 5.1, as it relies solely on observable outputs without requiring access to the target model's internal structure or training data.

#### Step 4: Evaluate the attack

The trained attack model is evaluated using a holdout set of member and non-member outputs to assess its ability to infer membership status. Effectiveness is measured using metrics such as accuracy and AUC, which reflect how well the model distinguishes between samples seen during training and those that were not. This step validates the success of the attack under the black-box threat model by quantifying how much membership information is leaked through observable output behaviour.

### 5.3 Using ChatGPT for Implementation (8 Marks)

<https://chatgpt.com/share/684c6431-e500-8007-921e-40bba102882c>

The link give access to the full conversation, include original prompt and answer.

Majority membership inference attack implemented by Chat GPT aligns with the goals and capabilities defined in Subtask 5.2. only small fine tune is needed.

```
class Lenet(nn.Module):
    # Defines the target model architecture
    # Although the attacker doesn't know this architecture,
    # it is defined here internally to simulate the "hidden" server-side model
    def __init__(self):
        super(Lenet, self).__init__()
        self.conv1 = nn.Conv2d(1, 6, 3, stride=1, padding=1)
        self.conv2 = nn.Conv2d(6, 16, 5, stride=1, padding=0)
        self.dropout1 = nn.Dropout(0.25)
        self.dropout2 = nn.Dropout(0.5)
        self.fc1 = nn.Linear(400, 120)
        self.fc2 = nn.Linear(120, 84)
        self.fc3 = nn.Linear(84, 10)

    def forward(self, x):
        x = self.conv1(x)
        x = F.relu(x)
        x = F.max_pool2d(x, 2)
        x = self.conv2(x)
        x = F.relu(x)
        x = F.max_pool2d(x, 2)
        x = self.dropout1(x)
        x = torch.flatten(x, 1)
        x = self.fc1(x)
        x = F.relu(x)
        x = self.dropout2(x)
        x = self.fc2(x)
        x = F.relu(x)
        x = self.dropout2(x)
        x = self.fc3(x)
        output = F.log_softmax(x, dim=1)
        return output

target_model = Lenet()
target_model.load_state_dict(torch.load("target_model.pth"))
target_model.eval()
```

- The original code provided by Chat GPT used a simple fully connected target model, which did not match the original architecture. To align the implementation, the original LeNet CNN model is replaced.

```

# ----- Load MNIST (members) & EMNIST (non-members) ----- #
transform = transforms.Compose([transforms.ToTensor()])
mnist_train = datasets.MNIST(root='./data', train=True, download=True, transform=transform)
emnist_test = datasets.EMNIST(root='./data', split='letters', train=False, download=True, transform=transform)

mnist_indices = np.random.choice(len(mnist_train), 10000, replace=False)
emnist_indices = np.random.choice(len(emnist_test), 10000, replace=False)

mnist_subset = Subset(mnist_train, mnist_indices)
emnist_subset = Subset(emnist_test, emnist_indices)

mnist_loader = DataLoader(mnist_subset, batch_size=128, shuffle=False)
emnist_loader = DataLoader(emnist_subset, batch_size=128, shuffle=False)

# Define a transformation: convert images to PyTorch tensors
transform = transforms.Compose([transforms.ToTensor()])

# Load 60,000 training images from the MNIST dataset (digit images 0-9)
mnist_train = datasets.MNIST(root='./data', train=True, download=True, transform=transform)
# Load 40,000 test images from the EMNIST dataset (digit split: 0-9)
# EMNIST 'digits' split resembles MNIST but is disjoint in samples
emnist_test = datasets.EMNIST(root='./data', split='digits', train=False, download=True, transform=transform)

# Randomly select 10,000 samples from MNIST and EMNIST to simulate "member" and "non-member" data
mnist_indices = np.random.choice(len(mnist_train), 10000, replace=False)
emnist_indices = np.random.choice(len(emnist_test), 10000, replace=False)

# Create subset datasets using the selected indices
mnist_subset = Subset(mnist_train, mnist_indices)
emnist_subset = Subset(emnist_test, emnist_indices)

# Create DataLoaders for batching and iterating over the subsets
mnist_loader = DataLoader(mnist_subset, batch_size=128, shuffle=False)
emnist_loader = DataLoader(emnist_subset, batch_size=128, shuffle=False)

```

On the left, the diagram shows the code provided by ChatGPT, which uses the EMNIST letters split as non-member data. On the right is the revised version of the code that I have corrected. I replaced `split='letters'` with `split='digits'` because using letters introduces a significant distributional gap from MNIST, making the attack unrealistic. By switching to `split='digits'`, the EMNIST samples better align with MNIST's digit format, creating a more valid and realistic membership inference setting where the attacker tests on data from a similar but unseen distribution, consistent with the threat model described in Subtask 5.1.

```

def get_softmax_outputs(dataloader, label):
    outputs = [] # List to store softmax output vectors
    labels = [] # List to store corresponding membership labels (1 or 0)
    for images, _ in dataloader:
        with torch.no_grad(): # Disable gradient calculation for inference
            logits = target_model(images) # Forward pass through target model
            softmax = F.softmax(logits, dim=1).numpy() # Convert logits to softmax probabilities
            outputs.append(softmax) # Store softmax outputs
            labels.append(np.full(len(images), label)) # Assign membership label to all images in the batch
    return np.vstack(outputs), np.hstack(labels) # Return stacked arrays of outputs and labels

# Get softmax outputs and labels for member (MNIST) and non-member (EMNIST) data
member_outputs, member_labels = get_softmax_outputs(mnist_loader, 1)
nonmember_outputs, nonmember_labels = get_softmax_outputs(emnist_loader, 0)

# Combine member and non-member data into a single feature matrix (X) and label vector (y)
X = np.vstack((member_outputs, nonmember_outputs))
y = np.hstack((member_labels, nonmember_labels))

```

There is no error in this part, this function queries the target model with input samples and collects their SoftMax confidence vectors. It labels each output as 1 for members or 0 for non-members, based on the dataset used. The `get_softmax_outputs(dataloader, label)` function takes two parameters, a dataloader containing input samples, and a label indicating membership status (1 for members, 0 for non-members). It returns the SoftMax confidence vectors produced by the target model along with the corresponding membership labels for each input.

```

# ----- Train the Attack Model ----- #
# Split the softmax outputs and labels into training and test sets (80% train, 20% test)
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=42)

# Initialize a logistic regression model as the attack classifier
# Increase max_iter to ensure convergence during training
attack_model = LogisticRegression(max_iter=1000)

# Train the attack model using the training softmax vectors and membership labels
attack_model.fit(X_train, y_train)

```

There is no error in this part, it aligns with the attack design. The labelled SoftMax outputs are split into training and testing sets using an 80/20 ratio. A logistic regression model is then trained on the training set to learn patterns in the output vectors that distinguish member samples from non-members. This attack model aims to infer membership status based solely on prediction behaviour.

```

# ----- Evaluate the Attack ----- #
# Use the trained attack model to predict membership on the test set
predictions = attack_model.predict(x_test)
# Get the predicted probability for the positive class (member = 1)
probabilities = attack_model.predict_proba(x_test)[:, 1]

# Calculate evaluation metrics
# Check how accurate the predictions are
acc = accuracy_score(y_test, predictions)
# Check how well the model separates members from non-members
auc = roc_auc_score(y_test, probabilities)

print(f"Attack Accuracy: {acc:.4f}")
print(f"Attack AUC: {auc:.4f}")

```

There is no error in this part, to evaluate the effectiveness of the membership inference attack, two metrics were used: accuracy and AUC (Area Under the ROC Curve). Accuracy measures how often the attack model correctly predicts whether a sample is from the target model's training set. AUC reflects how well the model separates member from non-member samples based on prediction confidence. Together, these metrics provide a clear view of how successfully the attack can infer membership status using only the model's output.

```

kche0213@FIT5124-VM:~/shared_folder/a3_real$ python3 mia_attack.py
Attack Accuracy: 0.6925
Attack AUC: 0.6628
Runtime: 0.05 seconds

```

### **Results and evaluation:**

The attack model's performance was evaluated using two main metrics, accuracy and Area Under the ROC Curve (AUC), time is use for efficiency in later task. Accuracy measures how often the model correctly predicts whether a sample was part of the target model's training set. AUC evaluates how well the model distinguishes members from non-members based on prediction confidence, regardless of the threshold. These metrics were chosen because the attack goal in Subtask 5.1 and 5.2 is to infer membership status in a black box setting using only SoftMax outputs. Specifically, the attack aims to determine whether the model memorises MNIST examples, thereby revealing their presence in the training set. The results show an attack accuracy of 0.6927 and an AUC of 0.6511, indicating that the target model leaks moderate membership information, suggesting that MNIST samples have high probability were used in training.

### **5.4 Attack Analysis (6 Marks)**

The attack demonstrates moderate effectiveness, as it only has accuracy of 0.6927 and an AUC of 0.651, indicating the ability to infer membership status better than random guessing. In term of efficiency, the attack is computationally lightweight as it only requires querying the target model and training a simple logistic regression classifier on SoftMax outputs. No access to the model's internals is required, and it only use 0.05 seconds. The attack is practical in real-world settings where the attacker can query the target model and use similar public data. It is low-cost and effective at inferring membership from outputs, but its success may be limited by real-world defences like output masking, rate limits, or absence of confidence scores.

### **Impact factors of the attacker's capability:**

Number of data - Member	Number of data – Non-Member	Accuracy	AUC
5000	5000	68.45%	65.57%
10000	10000	69.25%	66.28%
15000	15000	71.15%	69.29%

The experiment shows that increasing the number of member and non-member samples improves the performance of the membership inference attack. As the dataset size grows from 5000 to 15000 samples per class, both accuracy and AUC steadily increase. This indicates that the attack model benefits from more training data, learning clearer distinctions between member and non-member outputs. Therefore, attacker capability in terms of data availability has a positive impact on attack effectiveness.

```
[Design: EMNIST Train = Member, EMNIST Test = Non-Member]
Attack Accuracy: 0.4923
Attack AUC: 0.5026
kche0213@FIT5124-VM:~/shared_folder/a3_real$ |
```

To further examine the impact of the attacker's capabilities, other structure remains the same an experiment was conducted using EMNIST training data as member samples and EMNIST test data as non-member samples. Since both sets are drawn from the same distribution and dataset split, this design removes any distributional differences that could give the attacker an unfair advantage. The resulting attack accuracy of 49.23% and AUC of 50.26% equivalent to guessing, suggesting that the target model generalised well and did not leak meaningful membership information. This result reinforces that the effectiveness of membership inference attacks strongly depends on the attacker's ability to access data that is distinguishable from the target model's training distribution.

### **Impact factors of the target model:**

Epoch on target model	Accuracy	AUC
5	69.25%	66.28%
25	68.70%	64.47%
50	72.00%	67.42%

This experiment shows that the number of training epochs affects how vulnerable a model is to membership inference attacks. At 5 and 25 epochs, the attack performs similarly, meaning the model does not overfit much and is harder to attack. However, at 50 epochs, the attack becomes more effective, suggesting the model starts to memorise training data. This means that training for too long can increase privacy risks, so the number of training epochs should be chosen carefully.

### **Danger and limitations:**

This attack demonstrates how even in a black box setting an attacker can infer whether specific data was used to train a model. This raises concerns about potential data leakage, especially if the model contains sensitive data. In this scenario, confirming that the target model uses MNIST data could raise serious concerns. If the model is advertised as being trained on unique educational data, discovering that it uses a public dataset like MNIST suggests a lack of originality or transparency. This could damage the company creditability leading to loss of profit and maybe lawsuit. However, the attack has limitations. Its success

depends on access to full confidence scores and on the difference between member and non-member data distributions. If the API masks output confidences, the attack's accuracy may drop significantly. Despite these challenges, the attack is a powerful tool to audit model privacy and training transparency.

## **6 Defending against membership inference attack (25 marks)**

### **6.1 Defence Overview (6 Marks)**

#### **Defender's Capability:**

The defender is the model owner and therefore has full access to the target model's training and inference processes, including its architecture, parameters and original training data. This access enables the defender to implement post-processing defences such as revealing only the top k class prediction scores and injecting noise into the output vector. These measures aim to confuse membership inference attacks while still preserving model usability for regular users.

#### **Practical Scenario Example:**

An EdTech company deploys a digit recognition API trained on public datasets such as MNIST. While the model achieves high accuracy, the company treats its specific training data selection and fine-tuning process as a competitive advantage. To prevent attackers from using membership inference attacks to determine which exact MNIST samples were used during training, the company applies a post-processing defence. This defence adds controlled noise to flatten the confidence distribution, reducing the attacker's ability to distinguish member from non-member inputs. As the model owner, the company has full access to the architecture, data, and inference process, enabling seamless deployment of this defence without impacting legitimate users. The goal is to prevent the success of membership inference attacks such that an attacker performs no better than random guessing, while keeping the defence lightweight and efficient with minimal computational overhead.

### **6.2 Defence Design (6 Marks):**

#### **Step 1: Adding noise with distribution flattening**

This defence introduces controlled noise to generate a near uniform SoftMax output for each prediction by assigning random noise to all classes. To ensure the top 1 predicted class remain the same, the highest random noise was track down and add in another small noise. The output is subsequently normalised to maintain a valid probability distribution. This step is critical as it weakens the confidence differences between member and non-member samples, which are exploited in membership inference attacks in task 5. By flattening the confidence distribution, it becomes harder for the attacker to distinguish these patterns. As a lightweight strategy, this defence effectively lowers attack accuracy and confuses the attack model without compromising model functionality for regular users.

### 6.3 Implementation and Evaluation (6 Marks)

```
def defence(dataloader, label, name=""):
    outputs = []      # List to store softmax output vectors
    labels = []       # List to store corresponding membership labels (1 or 0)
    start_time = time.perf_counter()

    predicted_classes = []
    original_confidences = []

    for images, _ in dataloader:
        with torch.no_grad():    # Disable gradient calculation for inference
            logits = target_model(images)    # Forward pass through target model
            softmax = F.softmax(logits, dim=1).numpy()    # get softmax probabilities

            batch_output = []    # store output
            for sample in softmax:
                top_idx = np.argmax(sample)    # get top 1 predicted class
                predicted_classes.append(top_idx)    # save it
                original_confidences.append(np.max(sample))    # save the original confidence

                num_classes = len(sample)
                defended = np.ones(num_classes) * random.uniform(0.09, 0.1)    # random uniform noise
                # tiny bias for predicted class
                defended[top_idx] = defended.max() + 0.001    # can be tune
                defended = defended / defended.sum()    # normalise

            batch_output.append(defended)    # append the defended sample output

        outputs.append(np.array(batch_output))    # add all batch outputs
        labels.append(np.full(len(images), label))    # append corresponding labels

    end_time = time.perf_counter()
    print(f"[{name}] Defence time: {end_time - start_time:.2f} seconds")

    # combine all batches into a single 2D array (samples x classes) and label array
    return np.vstack(outputs), np.hstack(labels)

# Get softmax outputs and labels for member (MNIST) and non-member (EMNIST) data
member_outputs, member_labels = defence(mnist_loader, 1, name="Member")
nonmember_outputs, nonmember_labels = defence(emnist_loader, 0, name="Non-member")
```

The defence function takes three parameters. The dataloader parameter is a PyTorch DataLoader object that provides batches of input data, such as member or non-member samples. The label parameter is an integer 1 for member and 0 for non-member used to tag the outputs for attack model training. The name parameter is a string such as member or Non-Member for logging purpose to identify the source of the data when printing the total runtime of the defence function.

The defence function returns two NumPy arrays one with modified SoftMax vectors and one with corresponding binary labels, with 1 for members and 0 for non-members. These are used to train or evaluate the attack model under the defence setting.

This function implements the post-processing defence design described in Task 6.2. Its purpose is to modify and mask the model's SoftMax probabilities before they are exposed to

an external user, aligning with the defence goal defined in Task 6.1. Instead of returning the full SoftMax output, the function replaces them with controlled noise and flattened values while ensuring the original top 1 class remains ranked highest to preserve the predicted label. This allows the model's classification outcome for users while reducing information leakage that could be exploited by membership inference attacks.

```
kche0213@FIT5124-VM:~/shared_folder/a3_real$ python3 mia_defence.py
[Member] Defence time: 0.64 seconds
[Non-member] Defence time: 0.71 seconds

Attack Accuracy: 0.4953
Attack AUC: 0.7879
```

To evaluate the effectiveness of the defence, the performance of the membership inference attack model is measured using two metrics including attack accuracy and AUC (Area Under the ROC Curve). These metrics directly quantify the attacker's ability to distinguish between member and non-member samples and align with the defence objective defined in Subtask 6.1, which aims to reduce the attacker's ability to distinguish between member and non-member samples, bringing the performance close to random guessing (50%). The results show an attack accuracy of 49.53% and an AUC of 78.79%, indicating that the attacker performs worse than guessing proving that the defence effectively masks membership signals while maintaining prediction utility, with minimal computational overhead.

#### 6.4 Defence Analysis (7 Marks)

**Defence cost:** The defence incurs minimal runtime overhead, with processing times of 0.64 and 0.71 seconds for both member and non-member datasets. Its computational complexity is approximately  $O(n \times C)$  per batch, where  $n$  is the number of samples and  $C$  is the number of classes, mainly from operations like uniform vector creation, top 1 class identification, and normalisation. The method requires no extra memory or hardware and is highly practical for real-world deployment without degrading performance or user experience.

#### Impact factors:

Bias value for top 1 predicted class	Accuracy	AUC
0.01	73.70%	78.30%
0.001	49.53%	78.79%,
0.0001	49.53%	78.34%

This table experiment was conducted to evaluate the impact of the top 1 bias value on defence effectiveness. When the bias was set to 0.01, the attack accuracy was high at 73.70%, indicating the defence failed to prevent membership inference. Reducing the bias to 0.001 and 0.0001 dropped the attack accuracy to 49.53%, effectively bringing it down to random guessing. This suggests that larger top 1 bias values leak membership information, enabling successful attacks, while smaller biases flatten the output, reducing the confidence gap and lowering attack accuracy to 50%. Thus, minimising the bias is key to effective defence. However, the attack AUC remained consistently high across all settings around 78%, suggesting the attacker could still rank member samples above non-members. This indicates that while lowering the bias disrupts confidence-based membership signals, the defence does not eliminate class distribution leakage, which remains a key vulnerability.

### **Against advanced attacker:**

While this defence reduces membership inference attack accuracy to nearly 50%, a high AUC of 78.79% indicates that the attack model can still consistently rank member samples above non-members. This leakage occurs because the preserved top 1 class reveals statistical biases. For instance, certain classes may appear more frequently as top predictions for member data than for non-member data. An advanced attacker operating in a black box setting meaning they do not know target model training data, inference process and model architecture but only knowing the defence mechanism, even with only SoftMax outputs can exploit this. By sending both member and non-member samples to the model and recording the top 1 predicted class, the attacker can identify class distribution patterns that differ between the two groups. For example, they may find that class 4 appears as the top 1 prediction more frequently in member samples, while class 8 appears more in non-member samples. These biases, although very small, might leak enough statistical signals for the attacker to use frequency-based patterns to infer membership status. This demonstrates that while the defence masks confidence scores, it does not fully hide structural patterns, leaving it vulnerable to label-only attacks.

### **References List**

Tramèr, F., Zhang, F., Juels, A., Reiter, M. K., & Ristenpart, T. (2016). Stealing Machine Learning Models via Prediction APIs. *In 25th USENIX Security Symposium (USENIX Security 16)*, pp. 601-618. <https://arxiv.org/abs/1609.02943>

### **Monash AI acknowledgement**

I acknowledge the use of ChatGPT (<https://chat.openai.com/>) to refine the academic language and accuracy of my own work. I submitted my entire report with the instruction to “polish my writing, fix my grammar and make it smooth”. The output was then modified further to better represent my own tone and style of writing.

I acknowledge the use of ChatGPT (<https://chat.openai.com/>) and Claude AI (<https://claude.ai/new>) to assist me in debugging the code and including comment. The output was then modified further to better represent my own work.

Link to the code:

[https://drive.google.com/drive/folders/1tG3po\\_wjPDyQVSnwuaWWXHuSQLbYPQU?usp=sharing](https://drive.google.com/drive/folders/1tG3po_wjPDyQVSnwuaWWXHuSQLbYPQU?usp=sharing)

<https://github.com/Tyson369-369/KaYongCheah-34789596-FIT5124-A3>