Import Libraries

In [1]:
```
import os
import torch
import torch.nn as nn
from torchsummary import summary
import torch.optim as optim
import torch.nn.functional as F
from torch.utils import data
from torchvision import datasets, transforms, models
from collections import Counter
from torchvision.transforms import Resize, CenterCrop, ToTensor, N
from torchvision.datasets import ImageFolder
from torch.utils.data import DataLoader, random_split, Subset, Dat

import cv2
from PIL import Image, ImageFilter, ImageOps
from torchvision.transforms.functional import to_pil_image
from skimage.metrics import peak_signal_noise_ratio, structural_si
import matplotlib.pyplot as plt
import torch.optim as optim
from tqdm import tqdm

from sklearn.metrics import classification_report, confusion_matri
import seaborn as sns
from torch.optim import lr_scheduler

from tkinter import filedialog, Tk

from gradcam import GradCAM, GradCAMpp
from gradcam.utils import visualize_cam

import numpy as np
from sklearn.model_selection import KFold
from sklearn.metrics import accuracy_score
from sklearn.preprocessing import label_binarize
from sklearn.metrics import roc_curve, auc

import dill
import torch.multiprocessing as mp

from captum.attr import IntegratedGradients
from captum.attr import visualization as viz
```

In [2]:
```
device = torch.device("cuda" if torch.cuda.is_available() else "cp
```

Specify the path to the locally saved dataset

In [3]:
```
train_data_dir = '/Users/savin/Desktop/FYP/Implementation/kaggle_d
original_dataset = datasets.ImageFolder(train_data_dir)
```

In [4]:
```
class ContrastStretching:
    def __call__(self, img):
        # Convert PIL Image to NumPy array
        img_np = np.array(img)

        # Check if the image is grayscale or RGB
        if img_np.ndim == 2:  # Grayscale image
            img_np = self.apply_contrast_stretching(img_np)
        elif img_np.ndim == 3:  # RGB image
            # Apply contrast stretching to each channel individual
            for i in range(img_np.shape[-1]):
                img_np[:, :, i] = self.apply_contrast_stretching(i

        # Convert back to PIL Image
        return Image.fromarray(img_np.astype('uint8'))

    def apply_contrast_stretching(self, channel):
        in_min, in_max = np.percentile(channel, (0, 100))
        out_min, out_max = 0, 255
        channel = np.clip((channel - in_min) * (out_max - out_min)
        return channel

class UnsharpMask:
    def __init__(self, radius=1, percent=100, threshold=3):
        self.radius = radius
        self.percent = percent
        self.threshold = threshold

    def __call__(self, img):
        return img.filter(ImageFilter.UnsharpMask(radius=self.radi
                                                  percent=self.per
                                                  threshold=self.t
                                                 ))

class GaussianBlur:
    def __init__(self, kernel_size, sigma=(0.1, 2.0)):
        self.kernel_size = kernel_size
        self.sigma = sigma

    def __call__(self, img):
        sigma = np.random.uniform(self.sigma[0], self.sigma[1])
        img = img.filter(ImageFilter.GaussianBlur(sigma))
        return img
```

Preprocess the dataset

In [5]:
```
preprocess_transform = transforms.Compose([
    transforms.Resize((224, 224)),
    ContrastStretching(),
    UnsharpMask(radius=1, percent=100, threshold=3),
    GaussianBlur(kernel_size=(5, 5), sigma=(0.1, 0.5)),
    transforms.ToTensor()
])

preprocessed_dataset = datasets.ImageFolder(root=train_data_dir, t

data_loader = DataLoader(preprocessed_dataset, batch_size=32, shuf
```

In [6]:
```
def calculate_psnr_ssim(original_dataset, preprocessed_dataset, nu
    psnr_values = []
    ssim_values = []

    for i in range(num_samples):
        original_img = original_dataset[i][0] # original MRI
        preprocessed_img = preprocessed_dataset[i][0] # preprocess

        if not isinstance(original_img, Image.Image):
            original_img = to_pil_image(original_img)
        if not isinstance(preprocessed_img, Image.Image):
            preprocessed_img = to_pil_image(preprocessed_img)

        # Convert MRI to grayscale
        original_img = original_img.convert("L")
        preprocessed_img = preprocessed_img.convert("L")

        # Resize images
        original_img = original_img.resize(resize)
        preprocessed_img = preprocessed_img.resize(resize)

        # Convert images to numpy arrays
        original_img_np = np.array(original_img)
        preprocessed_img_np = np.array(preprocessed_img)

        # Calculate PSNR and SSIM
        psnr = peak_signal_noise_ratio(original_img_np, preprocess
        ssim = structural_similarity(original_img_np, preprocessed

        psnr_values.append(psnr)
        ssim_values.append(ssim)

    # Compute average PSNR and SSIM
    avg_psnr = np.mean(psnr_values)
    avg_ssim = np.mean(ssim_values)

    return avg_psnr, avg_ssim

# Example usage
avg_psnr, avg_ssim = calculate_psnr_ssim(original_dataset, preproc
print(f"Average PSNR: {avg_psnr}, Average SSIM: {avg_ssim}")
```

```
Average PSNR: 27.351058453113573, Average SSIM: 0.900360553667264
2
```

preprocess_transform = transforms.Compose(

```
transforms.Resize((224, 224)),
GaussianBlur(kernel_size=(5, 5), sigma=(0.1, 2.0)),
transforms.Lambda(lambda x: x.filter(ImageFilter.UnsharpMa
sk(radius=2, percent=150, threshold=3))),
transforms.ToTensor()
```

)

Average PSNR: 9.729432125669954, Average SSIM: 0.2833625979364462

---

preprocess_transform = transforms.Compose(

```
transforms.Resize((224, 224)),
transforms.ToTensor()
```

)

Average PSNR: 10.2026194786185, Average SSIM: 0.32374658927511385

---

preprocess_transform = transforms.Compose(

```
transforms.Resize((224, 224)),
GaussianBlur(kernel_size=(5, 5), sigma=(0.1, 2.0)),
transforms.ToTensor(),
```

)

Average PSNR: 34.479706650199184, Average SSIM: 0.9638484917028203

---

preprocess_transform = transforms.Compose(

```
transforms.Resize((224, 224)),
GaussianBlur(kernel_size=(5, 5), sigma=(0.1, 1.0)),
transforms.ToTensor(),
```

)

Average PSNR: 40.03000513450271, Average SSIM: 0.9923509776637894

---

preprocess_transform = transforms.Compose(

```
transforms.Resize((224, 224)),
transforms.Lambda(lambda img: img.filter(ImageFilter.Unsha
rpMask(radius=2, percent=100, threshold=3))),
GaussianBlur(kernel_size=(5, 5), sigma=(0.1, 0.5)),
transforms.ToTensor()
```

)

Average PSNR: 28.919164968402907, Average SSIM: 0.9646276430637585 preprocess_transform = transforms.Compose(

```
transforms.Resize((224, 224)),
ContrastStretching(),
transforms.Lambda(lambda img: img.filter(ImageFilter.Unsha
rpMask(radius=1, percent=100, threshold=3))),
GaussianBlur(kernel_size=(5, 5), sigma=(0.1, 0.5)),
transforms.ToTensor()
```

)

Average PSNR: 27.194490517269728, Average SSIM: 0.8948121010151182

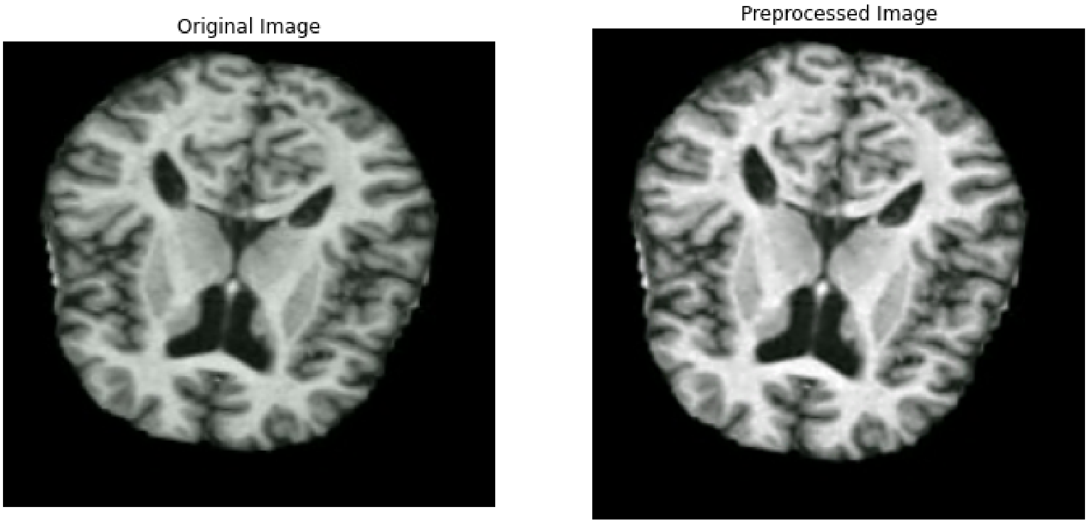
MRI scan counts in each class of the dataset

In [7]:
```
MildDemented = '/Users/savin/Desktop/FYP/Implementation/kaggle_dat
ModerateDemented = '/Users/savin/Desktop/FYP/Implementation/kaggle
NonDemented = '/Users/savin/Desktop/FYP/Implementation/kaggle_data
VeryMildDemented = '/Users/savin/Desktop/FYP/Implementation/kaggle

count_MildDemented = len(os.listdir(MildDemented))
count_ModerateDemented = len(os.listdir(ModerateDemented))
count_NonDemented = len(os.listdir(NonDemented))
count_VeryMildDemented = len(os.listdir(VeryMildDemented))

print(f"Number of images in MildDemented: {count_MildDemented}")
print(f"Number of images in ModerateDemented: {count_ModerateDemen
print(f"Number of images in NonDemented: {count_NonDemented}")
print(f"Number of images in VeryMildDemented: {count_VeryMildDemen

print(f"\nTotal MRIs in the dataset = {count_MildDemented+count_Mo
```

```
Number of images in MildDemented: 8960
Number of images in ModerateDemented: 6464
Number of images in NonDemented: 9600
Number of images in VeryMildDemented: 8960

Total MRIs in the dataset = 33984
```

Sample MRI before and after preprocessing

In [8]:
```
sample_image_path = os.path.join(MildDemented, os.listdir(MildDeme

original_image = Image.open(sample_image_path)

# Apply the preprocessing transforms
preprocessed_image = preprocess_transform(original_image)

preprocessed_image = transforms.ToPILImage()(preprocessed_image)

# Display the images
plt.figure(figsize=(12, 6))

plt.subplot(1, 2, 1)
plt.imshow(original_image)
plt.title("Original Image")
plt.axis('off')

plt.subplot(1, 2, 2)
plt.imshow(preprocessed_image)
plt.title("Preprocessed Image")
plt.axis('off')

plt.show()
```



Dataset splitting & creating DataLoaders

In [9]:
```
train_size = int(0.70 * len(preprocessed_dataset))
val_size = int(0.15 * len(preprocessed_dataset))
test_size = len(preprocessed_dataset) - train_size - val_size

train_dataset, val_dataset, test_dataset = random_split(preprocess

# Create DataLoaders
train_loader = DataLoader(train_dataset, batch_size=32, shuffle=Tr
val_loader = DataLoader(val_dataset, batch_size=32, shuffle=True)
test_loader = DataLoader(test_dataset, batch_size=32, shuffle=Fals

dataloaders = {'train': train_loader, 'val': val_loader, 'test': t
```

In [40]:
```
all_labels = [label for _, label in train_dataset]
class_distribution = Counter(all_labels)
print(class_distribution)
```

```
Counter({2: 6737, 0: 6238, 3: 6236, 1: 4577})
```

Building CNN Model 2

In [10]:
```
class SEBlock(nn.Module):
    def __init__(self, in_channels, reduction=32):
        super(SEBlock, self).__init__()
        self.avg_pool = nn.AdaptiveAvgPool2d(1)
        self.fc = nn.Sequential(
            nn.Linear(in_channels, in_channels // reduction, bias=
            nn.ReLU(inplace=True),
            nn.Linear(in_channels // reduction, in_channels, bias=
            nn.Sigmoid()
        )

    def forward(self, x):
        b, c, _, _ = x.size()
        y = self.avg_pool(x).view(b, c)
        y = self.fc(y).view(b, c, 1, 1)
        return x * y.expand_as(x)

class ResidualBlock(nn.Module):
    def __init__(self, in_channels):
        super(ResidualBlock, self).__init__()
        self.conv = nn.Conv2d(in_channels, in_channels, kernel_siz
        self.bn = nn.BatchNorm2d(in_channels)

    def forward(self, x):
        residual = x
        out = F.relu(self.bn(self.conv(x)))
        out += residual
        return F.relu(out)

class CustomEfficientNet(nn.Module):
    def __init__(self, num_classes=4):
        super(CustomEfficientNet, self).__init__()
        self.base_model = models.efficientnet_b0(pretrained=True)

        for param in self.base_model.parameters():
            param.requires_grad = False

        # Replace the classifier with a new one
        num_ftrs = self.base_model.classifier[1].in_features
        self.classifier = nn.Sequential(
            nn.Dropout(0.2),
            nn.Linear(num_ftrs, 512),
            nn.ReLU(),
            nn.Dropout(0.5),
            nn.Linear(512, num_classes),
        )

        # Add SEBlock and ResidualBlock to the end of the features
        self.base_model.features.add_module("SEBlock", SEBlock(128

    def forward(self, x):
        # Process through EfficientNet up to before avgpool
        x = self.base_model.features(x)

        # Now x is a 4D tensor, and we can apply SEBlock and Resid
        # No need for separate calls, as they are part of the feat

        # Apply avgpool and classifier
        x = self.base_model.avgpool(x)
        x = torch.flatten(x, 1)
        x = self.classifier(x)
```

```
        return x

# Creating the model and moving to device
model2 = CustomEfficientNet(num_classes=4).to(device)

# For a summary, ensure the input size matches your dataset
summary(model2, (3, 224, 224))
```

```
==============================================================
===
            Conv2d-1         [-1, 32, 112, 112]             8
64
       BatchNorm2d-2         [-1, 32, 112, 112]
64
              SiLU-3         [-1, 32, 112, 112]
0
            Conv2d-4         [-1, 32, 112, 112]             2
88
       BatchNorm2d-5         [-1, 32, 112, 112]
64
              SiLU-6         [-1, 32, 112, 112]
0
 AdaptiveAvgPool2d-7             [-1, 32, 1, 1]
0
            Conv2d-8              [-1, 8, 1, 1]             2
64
              SiLU-9              [-1, 8, 1, 1]
0
```

Train Customized EfficientNet-B0

In [12]:
```
criterion = nn.CrossEntropyLoss()
trainable_params = filter(lambda p: p.requires_grad, model2.parame
optimizer = torch.optim.Adam(trainable_params, lr=0.001)
scheduler = lr_scheduler.StepLR(optimizer, step_size=7, gamma=0.1)
torch.autograd.set_detect_anomaly(True)

def train_epoch(epoch_index, train_loader, model, optimizer):
    model.train()
    running_loss = 0.0
    correct_pred = 0
    total_pred = 0

    for inputs, labels in tqdm(train_loader, desc=f"Epoch {epoch_i
        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()
        _, predictions = torch.max(outputs, 1)
        correct_pred += (predictions == labels).sum().item()
        total_pred += labels.size(0)

    avg_loss = running_loss / len(train_loader)
    avg_acc = correct_pred / total_pred
    print(f'train Loss: {avg_loss:.4f} Acc: {avg_acc:.4f}')
    return avg_loss, avg_acc

def validate_epoch(epoch_index, val_loader, model):
    model.eval()
    running_loss = 0.0
    correct_pred = 0
    total_pred = 0

    with torch.no_grad():
        for inputs, labels in tqdm(val_loader, desc=f"Epoch {epoch
            inputs, labels = inputs.to(device), labels.to(device)
            outputs = model(inputs)
            loss = criterion(outputs, labels)

            running_loss += loss.item()
            _, predictions = torch.max(outputs, 1)
            correct_pred += (predictions == labels).sum().item()
            total_pred += labels.size(0)

    avg_loss = running_loss / len(val_loader)
    avg_acc = correct_pred / total_pred
    print(f'val Loss: {avg_loss:.4f} Acc: {avg_acc:.4f}')
    return avg_loss, avg_acc

# Training loop
num_epochs = 25
train_losses, train_accuracies = [], []
val_losses, val_accuracies = [], []
best_val_loss = float('inf')
patience = 8

for epoch in range(num_epochs):
    train_loss, train_acc = train_epoch(epoch, train_loader, model
```

```
    val_loss, val_acc = validate_epoch(epoch, val_loader, model2)

    if val_loss < best_val_loss:
        best_val_loss = val_loss
        trigger_times = 0
        torch.save(model2.state_dict(), 'model2_test1.pth')
#     else:
#         trigger_times += 1
#         if trigger_times >= patience:
#             print(f"Early stopping at epoch {epoch+1}")
#             break

    train_losses.append(train_loss)
    train_accuracies.append(train_acc)
    val_losses.append(val_loss)
    val_accuracies.append(val_acc)

# Plotting
plt.figure(figsize=(12, 5))
plt.subplot(1, 2, 1)
plt.plot(train_losses, label='Training Loss')
plt.plot(val_losses, label='Validation Loss')
plt.legend()
plt.title('Loss over epochs')

plt.subplot(1, 2, 2)
plt.plot(train_accuracies, label='Training Accuracy')
plt.plot(val_accuracies, label='Validation Accuracy')
plt.legend()
plt.title('Accuracy over epochs')

plt.show()
```

```
Epoch 20 [val] Progress: 100%|██████████| 160/160 [03:48<0
0:00,  1.43s/batch]

val Loss: 0.6427 Acc: 0.7108

Epoch 21 [train] Progress: 100%|██████████| 744/744 [19:31<0
0:00,  1.58s/batch]

train Loss: 0.6003 Acc: 0.7444

Epoch 21 [val] Progress: 100%|██████████| 160/160 [03:49<0
0:00,  1.43s/batch]

val Loss: 0.6550 Acc: 0.7141

Epoch 22 [train] Progress: 100%|██████████| 744/744 [18:59<0
0:00,  1.53s/batch]

train Loss: 0.5978 Acc: 0.7463

Epoch 22 [val] Progress: 100%|██████████| 160/160 [03:50<0
```

Classification Report of the trained Modified EfficientNet B0

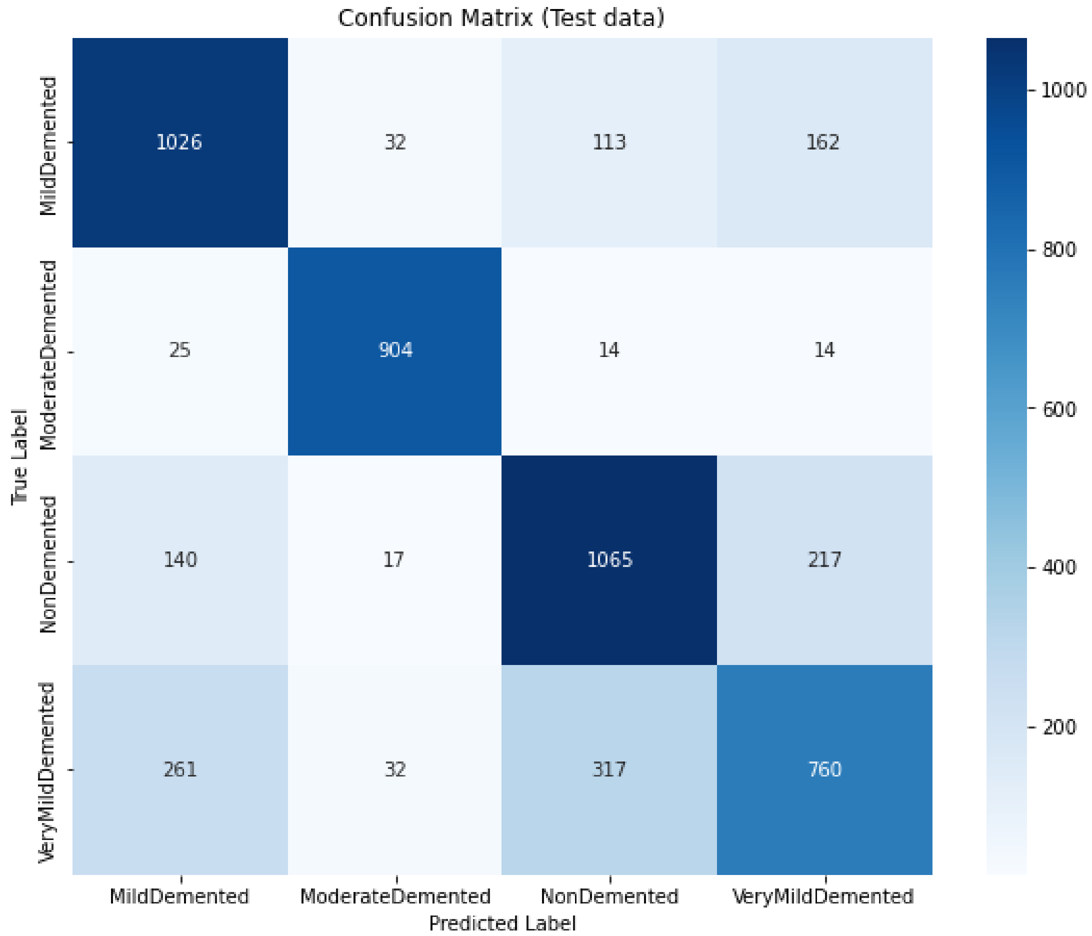
In [13]:
```
# classification report (val loader)
model2 = CustomEfficientNet(num_classes=4).to(device)

model2.load_state_dict(torch.load('model2_test1.pth'))

def evaluate_model(model, dataloader):
    model.eval()
    true_labels = []
    predictions = []

    with torch.no_grad():
        for inputs, labels in dataloader:
            inputs, labels = inputs.to(device), labels.to(device)

            outputs = model(inputs)
            _, preds = torch.max(outputs, 1)

            true_labels.extend(labels.cpu().numpy())
            predictions.extend(preds.cpu().numpy())

    return true_labels, predictions

# Evaluate the model
true_labels, predictions = evaluate_model(model2, val_loader)

# Print classification report
print(classification_report(true_labels, predictions, target_names
```

```
                  precision    recall  f1-score   support

    MildDemented       0.70      0.77      0.73      1331
ModerateDemented       0.92      0.96      0.94       994
     NonDemented       0.70      0.71      0.71      1470
VeryMildDemented       0.62      0.54      0.57      1302

        accuracy                           0.73      5097
       macro avg       0.74      0.74      0.74      5097
    weighted avg       0.72      0.73      0.73      5097
```

In [14]:
```
def evaluate_model(model, dataloader):
    model.eval()
    true_labels = []
    predictions = []

    with torch.no_grad():
        for inputs, labels in dataloader:
            inputs, labels = inputs.to(device), labels.to(device)

            outputs = model(inputs)
            _, preds = torch.max(outputs, 1)

            true_labels.extend(labels.cpu().numpy())
            predictions.extend(preds.cpu().numpy())

    return true_labels, predictions

# Evaluate the model
true_labels, predictions = evaluate_model(model2, test_loader)

# Print classification report
print(classification_report(true_labels, predictions, target_names
```

```
                  precision    recall  f1-score   support

    MildDemented       0.71      0.76      0.74      1333
ModerateDemented       0.91      0.94      0.93       957
     NonDemented       0.70      0.75      0.72      1439
VeryMildDemented       0.66      0.55      0.60      1370

        accuracy                           0.74      5099
       macro avg       0.75      0.75      0.75      5099
    weighted avg       0.73      0.74      0.73      5099
```

In [15]:
```
# Confusion Matrix on Test Loader - modified EfficientNet B0

def get_predictions(model, dataloader):
    model.eval()
    true_labels = []
    predictions = []

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

            true_labels.extend(labels.cpu().numpy())
            predictions.extend(preds.cpu().numpy())

    return true_labels, predictions

# Evaluate the model
true_labels, predictions = get_predictions(model2, test_loader)

# Compute the confusion matrix
cm = confusion_matrix(true_labels, predictions)
class_names = ['MildDemented', 'ModerateDemented', 'NonDemented',

# Plot the confusion matrix
plt.figure(figsize=(10, 8))
sns.heatmap(cm, annot=True, fmt='d', cmap='Blues', xticklabels=cla
plt.title('Confusion Matrix (Test data)')
plt.ylabel('True Label')
plt.xlabel('Predicted Label')
plt.show()
```



In [16]:
```
# Save the best model locally (model2 - modified EfficientNetB0)

model_save_path = '/Users/savin/Desktop/FYP/final_chapters/Model_T
os.makedirs(model_save_path, exist_ok=True)
model_save_file = os.path.join(model_save_path, 'model2_test3.pth'

torch.save(model2.state_dict(), model_save_file)

print(f'Model saved to {model_save_file}')
```

```
Model saved to /Users/savin/Desktop/FYP/final_chapters/Model_Test
ing/model2_test3.pth
```