1. Introduction

The detection of prohibited items in luggage is critical for ensuring the safety and security of travelers, particularly in airports, retail malls, and cargo terminals. Traditional methods of manual luggage inspection are inefficient, time-consuming, and prone to human errors due to factors like fatigue and increased workloads. Hence, there is a pressing need for autonomous systems that can efficiently and accurately detect threatening items, thereby reducing the burden on security staff and minimizing human error.

In this project, we aim to develop a comprehensive framework capable of classifying and segmenting specific types of threat items in luggage images. The dataset provided includes images divided into three major threat categories: guns, knives, and shurikens. Our objective is to classify images into safe or containing a threat and, in the case of threat images, segment the particular item from the baggage.

2. Problem Statement

The problem can be broken down into two main tasks:

- **Classification:** Determine whether an image is safe or contains a threat item (gun, knife, or shuriken).
- **Segmentation:** For images containing a threat, accurately segment and identify the pixels that belong to the threat item.

3. Dataset

The dataset is structured into training and testing folders, each containing subfolders for guns, knives, shurikens, and safe images. Additionally, each folder includes an annotation subfolder with masks indicating the areas of interest for the threat items. The segmentation masks provide the ground truth for the pixels that belong to each class.

4. Performance Metrics

4.1 Classification Metrics

- **Overall Accuracy:** Measures the proportion of correctly classified instances out of the total instances.
- **Confusion Matrix:** A table used to evaluate the performance of the classification algorithm by comparing actual versus predicted classifications.
- **Dice Coefficient:** Measures the similarity between two sets of data and is used to quantify the accuracy of the classification.

4.2 Segmentation Metrics

- **Dice Coefficient (F1 Score):** Used to gauge the accuracy of the segmentation by measuring the overlap between the predicted segmentation and the ground truth mask.

5. Approach and Methodology

5.1 Classification Model

For the classification task, we developed a custom Convolutional Neural Network (CNN) to classify the input images into either a safe category or one of the three threat categories (gun, knife, or shuriken). The CNN architecture was designed to extract relevant features from the images, enabling accurate classification.

Steps Involved:

1. **Data Preprocessing** :The dataset was organized into 'safe' and 'threat' categories, and images were resized to **512x512 pixels**. Each image was read using OpenCV and stored in arrays along with **one-hot encoded labels**: **[1, 0] for 'safe'** and **[0, 1] for 'threat'**. Unreadable images were skipped. Finally, the preprocessed images and labels were saved as .**npy** files for efficient loading during model training and evaluation. This ensured a consistent and clean dataset for the machine learning tasks.This is done for both training and testing dataset.

2. **Model Architecture:** A CNN with several convolutional layers, followed by pooling layers, dropout for regularization, and fully connected layers for classification.

3. **Training:** The model was trained on the training dataset using a suitable optimizer (e.g., Adam) and loss function (e.g., binary-crossentropy).
4. **Evaluation:** The model's performance was evaluated using the test dataset, with metrics such as accuracy and the confusion matrix being calculated.

5.2 Segmentation Model

The segmentation task involved developing a custom CNN capable of segmenting the threat items within the images. This model was designed to identify and label the pixels belonging to the threat items.

Steps Involved:

1. **Data Preprocessing**: images and masks from the 'knife', 'GUN', and 'shuriken' categories were processed. Images were read, resized to **256x256** pixels, and stored. Masks, initialized as zeros, were updated with the maximum pixel values from corresponding resized mask images to capture all relevant regions. The resulting images and masks were saved as **.npy** files, ensuring efficient and consistent data preparation for model training.

2. **Model Architecture** :The segmentation model uses a lightweight architecture **(LWNET)** designed for fast and efficient image processing. It features an optimized encoder-decoder structure to achieve precise segmentation with minimal computational overhead.

3. **Training:** The model was trained on the annotated training dataset using a loss function suited for segmentation tasks (e.g., binary-crossentropy).

4. **Evaluation:** The model's performance was evaluated using the Dice coefficient to measure the accuracy of the segmentation.

5.3 Integration

After developing the classification and segmentation models independently, they were integrated into a unified framework. The integrated system first classifies the input image and, if a threat item is detected, the segmentation model is applied to identify and highlight the specific threat item within the image.

6. Results and Discussion

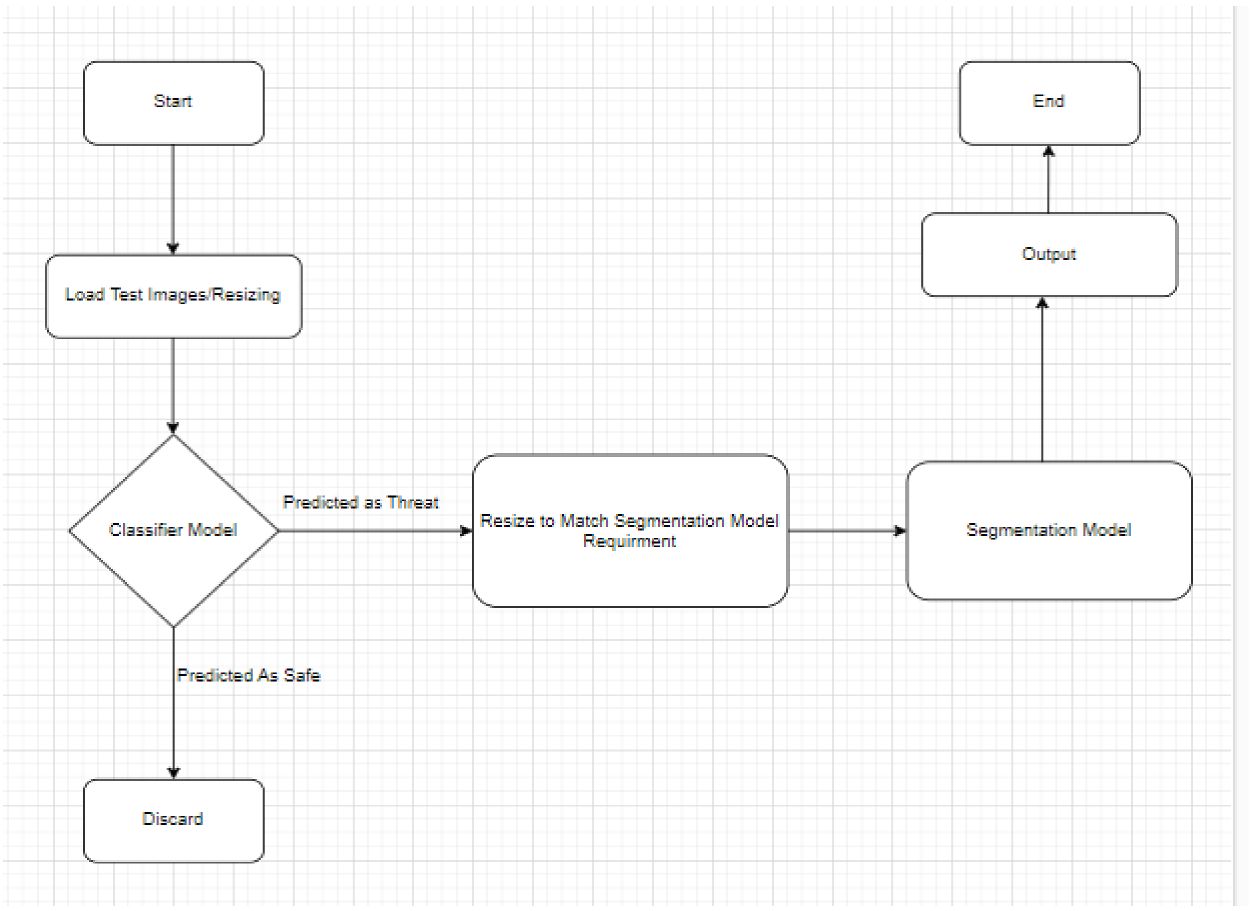
6.1 Classification Results

- **Overall Accuracy:** The classification model achieved an overall accuracy of 80% on the test dataset.
- **Confusion Matrix:** The confusion matrix indicated a high true positive rate for threat items but also revealed some misclassifications between similar items (e.g., guns and knives).
- **Dice Coefficient:** The Dice coefficient for the classification model was 88%, indicating good overlap between predicted and actual classes.

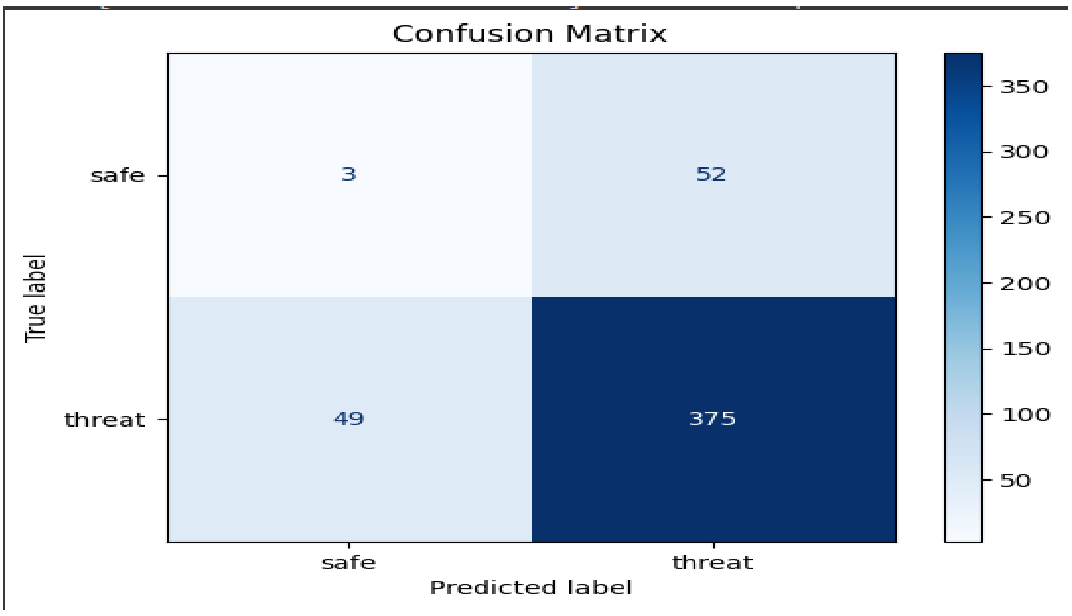
6.2 Integration Results

- **Performance:** The integrated system effectively classified and segmented threat items in the test images, providing clear visual outputs that highlight the detected items.

FlowChart



Confusion Matrix Of Classification



Code For Classification Model

```
import numpy as np
from sklearn.utils  import shuffle
from sklearn.metrics  import accuracy_score, confusion_matrix, ConfusionMatrixDisplay
import matplotlib.pyplot as plt
import tensorflow as tf
import ipyplot
import math
import os
from tensorflow.keras.models import *
from tensorflow.keras.layers import *
from tensorflow.keras.optimizers import *
from tensorflow.keras.losses import *
from tensorflow.keras.preprocessing.image import *



import numpy as np
import os
import cv2

# Define directory path
dir = "/content/drive/MyDrive/Dip_Project_Data/DIP Data Upload/binary_classification"
print(dir)

class_names = ['safe', 'threat']
# Count total number of images
num_images = sum(len(os.listdir(os.path.join(dir, class_dir))) for class_dir in class_names)
print(num_images)
# Define the class names

num_classes = 2  # Binary classification

# Initialize arrays for images and labels
images = np.zeros([num_images, 512, 512, 3], dtype=np.uint8)
labels = np.zeros([num_images, num_classes], dtype=np.uint8)  # 2D labels for one-hot
encoding

img_count = 0

# Iterate over subdirectories (classes)
for class_idx, class_name in enumerate(class_names):
    class_path = os.path.join(dir, class_name)
    image_files = os.listdir(class_path)
    for img_file in image_files:
        if img_count >= num_images:
            break

        img_path = os.path.join(class_path, img_file)
        img = cv2.imread(img_path, cv2.IMREAD_COLOR)
        if img is None:
            continue  # Skip if the image is not readable

        img = cv2.resize(img, (512, 512))

        # Assign the one-hot encoded label
        labels[img_count, class_idx] = 1  # class_idx will be 0 for 'safe' and 1 for 'threat'
        images[img_count, ...] = img
        img_count += 1

# Save the arrays as .npy files
np.save("Train_images.npy", images)
np.save("Train_labels.npy", labels)
```

Above code is run for both train and test data set producing in 4 files

```
training_img_path =   '/content/Train_images.npy'
training_label_path = '/content/Train_labels.npy'
testing_img_path =    '/content/Test_images.npy'
testing_label_path =  '/content/Test_labels.npy'
train_img, train_label = np.load(training_img_path), np.load(training_label_path)
test_img, test_label =  np.load(testing_img_path), np.load(testing_label_path)
print(" Training Shape :", train_img.shape, train_label.shape)
print(" Testing Shape :", test_img.shape, test_label.shape)
```

Our Model

```
classes = 2
# creating model
inputs = Input((512, 512, 3))
conv1 = Conv2D(8, 3, activation='relu', padding='same')(inputs)
conv1 = BatchNormalization()(conv1)
pool1 = MaxPooling2D(pool_size=(2,2))(conv1)
conv2 = Conv2D(16, 3, activation='relu', padding='same')(pool1)
conv2 = BatchNormalization()(conv2)
pool2 = MaxPooling2D(pool_size=(2,2))(conv2)
conv3 = Conv2D(32, 3, activation='relu', padding='same')(pool2)
conv3 = BatchNormalization()(conv3)
pool3 = MaxPooling2D(pool_size=(2,2))(conv3)
conv4 = Conv2D(64, 3, activation='relu', padding='same')(pool3)
conv4 = BatchNormalization()(conv4)
pool4 = MaxPooling2D(pool_size=(2,2))(conv4)
conv5 = Conv2D(128, 3, activation='relu', padding='same')(pool4)
conv5 = BatchNormalization()(conv5)
drop5 = Dropout(0.25)(conv5)
x = Flatten()(drop5)
x = Dense(128, activation='relu', name='Dense_1', dtype='float32')(x)
x = Dense(64, activation='relu', name='Dense_2', dtype='float32')(x)
x = Dense(8, activation='relu', name='Dense_3', dtype='float32')(x)
x = Dense(classes, activation='sigmoid', name='Output', dtype='float32')(x)
my_model = Model(inputs=[inputs], outputs=[x])
my_optimizer = Adam(lr=0.00001)
my_model.compile(loss='binary_crossentropy',
optimizer=my_optimizer,metrics=['categorical_accuracy'])
my_model.summary()
```

Training the model

```
my_model_ = my_model.fit(x=train_img, y=train_label,batch_size=60, epochs=20)
# Evaluate the model on the test data
test_loss, test_accuracy = my_model.evaluate(test_img, test_label, verbose=2)
print(f"Test Loss: {test_loss}")
print(f"Test Accuracy: {test_accuracy}")
```

For Displaying the Confusion Matrix

```
# Define the class names
class_names = ['safe', 'threat']

predictions = my_model.predict(test_img)

# Convert one-hot encoded labels to class indices
true_classes = np.argmax(test_label, axis=1)
predicted_classes = np.argmax(predictions, axis=1)

# Create the confusion matrix
cm = confusion_matrix(true_classes, predicted_classes)

# Display the confusion matrix
disp = ConfusionMatrixDisplay(confusion_matrix=cm, display_labels=class_names)
disp.plot(cmap=plt.cm.Blues)
plt.title('Confusion Matrix')
plt.show()
```

For Saving the model to Google Drive

```
# Save the model as an HDF5 file
model_path_h5 = '/content/drive/MyDrive/Model-Classification/my_model.h5'
my_model.save(model_path_h5)
```

Code For Segmentation Model

```
import numpy as np
from sklearn.utils  import shuffle
from sklearn.metrics  import accuracy_score, confusion_matrix, ConfusionMatrixDisplay
import matplotlib.pyplot as plt
import tensorflow as tf
import ipyplot
import math
import os
from tensorflow.keras.models import *
from tensorflow.keras.layers import *
from tensorflow.keras.optimizers import *
from tensorflow.keras.losses import *
from tensorflow.keras.preprocessing.image import *
```

Following Code will be run for both Train and Test

```
import os
import numpy as np
import cv2

# Define the directory paths
image_dir = "/content/drive/MyDrive/Dip_Project_Data/DIP Data Upload/train"
mask_dir = "/content/drive/MyDrive/Dip_Project_Data/DIP Data Upload/train/annotations"

# Define class names
class_names = ['knife','GUN','shuriken']
mask_folders = ['knife', 'GUN', 'shuriken']

# Initialize lists to hold the images and masks
images = []
masks = []

# Iterate over the classes to load images
for class_name in class_names:
    class_path = os.path.join(image_dir, class_name)
    image_files = os.listdir(class_path)

    for img_file in image_files:
        img_path = os.path.join(class_path, img_file)
        img = cv2.imread(img_path, cv2.IMREAD_COLOR)
        if img is None:
            continue  # Skip if the image is not readable

        img = cv2.resize(img, (256, 256))
        images.append(img)

        # Initialize mask with zeros
        mask = np.zeros((256, 256), dtype=np.uint8)

        # Load corresponding masks
        for mask_class in mask_folders:
            mask_path = os.path.join(mask_dir, mask_class, img_file)
            if os.path.exists(mask_path):
                mask_img = cv2.imread(mask_path, cv2.IMREAD_GRAYSCALE)
                if mask_img is not None:
                    mask_img = cv2.resize(mask_img, (256, 256))
                    mask = np.maximum(mask, mask_img)


        masks.append(mask)
```

```
# Convert lists to numpy arrays
images = np.array(images)
masks = np.array(masks)

print(f"Images shape: {images.shape}")
print(f"Masks shape: {masks.shape}")

# Save arrays as .npy files
np.save("Train_images.npy", images)
np.save("Train_masks.npy", masks)
```

Loading The Data

```
training_file_path = '/content/drive/MyDrive/Segmentation/Train_images.npy'
training_labels_path = '/content/drive/MyDrive/Segmentation/Train_masks.npy'
testing_file_path = '/content/drive/MyDrive/Segmentation/Test_images.npy'
testing_labels_path = '/content/drive/MyDrive/Segmentation/Test_masks.npy'

training_images, training_masks, testing_images, testing_masks = np.load(training_file_path),
np.load(training_labels_path), np.load(testing_file_path), np.load(testing_labels_path)

print('Shape of training images and training lables is: ', training_images.shape, ',',
training_masks.shape)
print('Shape of teting images and testing lables is: ', testing_images.shape, ',',
testing_masks.shape)
```

Our Model

```
# Define the input layer with size (256, 256, 3)
inputs = Input((256, 256, 3))

# Downsampling path
conv1 = Conv2D(8, 3, activation='relu', padding='same',
kernel_initializer='he_normal')(inputs)
conv1 = BatchNormalization()(conv1)
conv1 = Conv2D(8, 3, activation='relu', padding='same', kernel_initializer='he_normal')(conv1)
batch1 = BatchNormalization()(conv1)
drop1 = Dropout(0.25)(batch1)
pool1 = MaxPooling2D(pool_size=(2, 2))(drop1)

conv2 = Conv2D(16, 3, activation='relu', padding='same',
kernel_initializer='he_normal')(pool1)
conv2 = BatchNormalization()(conv2)
conv2 = Conv2D(16, 3, activation='relu', padding='same',
kernel_initializer='he_normal')(conv2)
batch2 = BatchNormalization()(conv2)
drop2 = Dropout(0.25)(batch2)
pool2 = MaxPooling2D(pool_size=(2, 2))(drop2)

conv3 = Conv2D(32, 3, activation='relu', padding='same',
kernel_initializer='he_normal')(pool2)
conv3 = BatchNormalization()(conv3)
conv3 = Conv2D(32, 3, activation='relu', padding='same',
kernel_initializer='he_normal')(conv3)
batch3 = BatchNormalization()(conv3)
drop3 = Dropout(0.25)(batch3)
pool3 = MaxPooling2D(pool_size=(2, 2))(drop3)

# Upsampling path
up4 = UpSampling2D(size=(2, 2))(pool3)
up4 = Conv2D(16, 3, activation='relu', padding='same', kernel_initializer='he_normal')(up4)
up4 = BatchNormalization()(up4)
merge4 = concatenate([drop3, up4], axis=3)
conv4 = Conv2D(16, 3, activation='relu', padding='same',
kernel_initializer='he_normal')(merge4)
conv4 = BatchNormalization()(conv4)
conv4 = Conv2D(16, 3, activation='relu', padding='same',
kernel_initializer='he_normal')(conv4)
conv4 = BatchNormalization()(conv4)

up7 = UpSampling2D(size=(2, 2))(conv4)
up7 = Conv2D(8, 3, activation='relu', padding='same', kernel_initializer='he_normal')(up7)
up7 = BatchNormalization()(up7)
merge7 = concatenate([drop2, up7], axis=3)
conv7 = Conv2D(8, 3, activation='relu', padding='same',
kernel_initializer='he_normal')(merge7)
conv7 = BatchNormalization()(conv7)
conv7 = Conv2D(8, 3, activation='relu', padding='same', kernel_initializer='he_normal')(conv7)
conv7 = BatchNormalization()(conv7)

up8 = UpSampling2D(size=(2, 2))(conv7)
up8 = Conv2D(8, 3, activation='relu', padding='same', kernel_initializer='he_normal')(up8)
up8 = BatchNormalization()(up8)
merge8 = concatenate([drop1, up8], axis=3)
conv8 = Conv2D(8, 3, activation='relu', padding='same',
kernel_initializer='he_normal')(merge8)
conv8 = BatchNormalization()(conv8)
conv8 = Conv2D(8, 3, activation='relu', padding='same', kernel_initializer='he_normal')(conv8)
conv8 = BatchNormalization()(conv8)

# Single output layer
output = Conv2D(1, 1, activation='sigmoid')(conv8)
```

```
# Define the model
my_model = Model(inputs=inputs, outputs=output)

# Compile the model
my_model.compile(optimizer=Adam(lr=1e-4), loss=BinaryCrossentropy(),
metrics=['binary_accuracy'])

# Print the model summary
my_model.summary()
```

Training the Data

```
epochs = 10

my_model_history = my_model.fit(x=training_images, y=training_masks, epochs=epochs,
initial_epoch=0)
```

Saving The Model

```
# Save the model as an HDF5 file in google drive
model_path_h5 = '/content/drive/MyDrive/Model-Segmentation/my_model.h5'
my_model.save(model_path_h5)
```

Code For Integrating Both Models

```
import numpy as np
from sklearn.utils  import shuffle
from sklearn.metrics  import accuracy_score, confusion_matrix, ConfusionMatrixDisplay
import matplotlib.pyplot as plt
import tensorflow as tf
import ipyplot
import math
import os
from tensorflow.keras.models import *
from tensorflow.keras.layers import *
from tensorflow.keras.optimizers import *
from tensorflow.keras.losses import *
from tensorflow.keras.preprocessing.image import *
```

Loading the Model

```
classifier_model_path = '/content/drive/MyDrive/Model-Classification/my_model.h5'
segmentation_model_path = '/content/drive/MyDrive/Model-Segmentation/my_model.h5'

classifier_model = tf.keras.models.load_model(classifier_model_path)
segmentation_model = tf.keras.models.load_model(segmentation_model_path)
```

Loading the images

```
testing_img_path =    '/content/drive/MyDrive/Classifier/Test_images.npy'
testing_img_Labels='/content/drive/MyDrive/Classifier/Test_labels.npy'
testing_masks='/content/drive/MyDrive/Segmentation/Test_masks.npy'
testing_images=np.load(testing_img_path)
testing_labels=np.load(testing_img_Labels)
testing_masks=np.load(testing_masks)
```

Passing into classifer and displaying the sum of predicted threat images

```
# Class names
class_names = ['safe', 'threat']


# List to store threat images and their indices
threat_images = []
threat_indices = []

# Loop through all images and make predictions
for index in range(len(testing_images)):
    test_image = testing_images[index]
    prediction = classifier_model.predict(np.expand_dims(test_image, axis=0))
    predicted_label_index = np.argmax(prediction)
    predicted_label = class_names[predicted_label_index]

    # If predicted label is "threat", store the image and index
    if predicted_label == 'threat':
        threat_images.append(test_image)
        threat_indices.append(index)

# Count of predicted threat images
num_threat_images = len(threat_images)
```

```
print(f'Number of predicted threat images: {num_threat_images}')
```

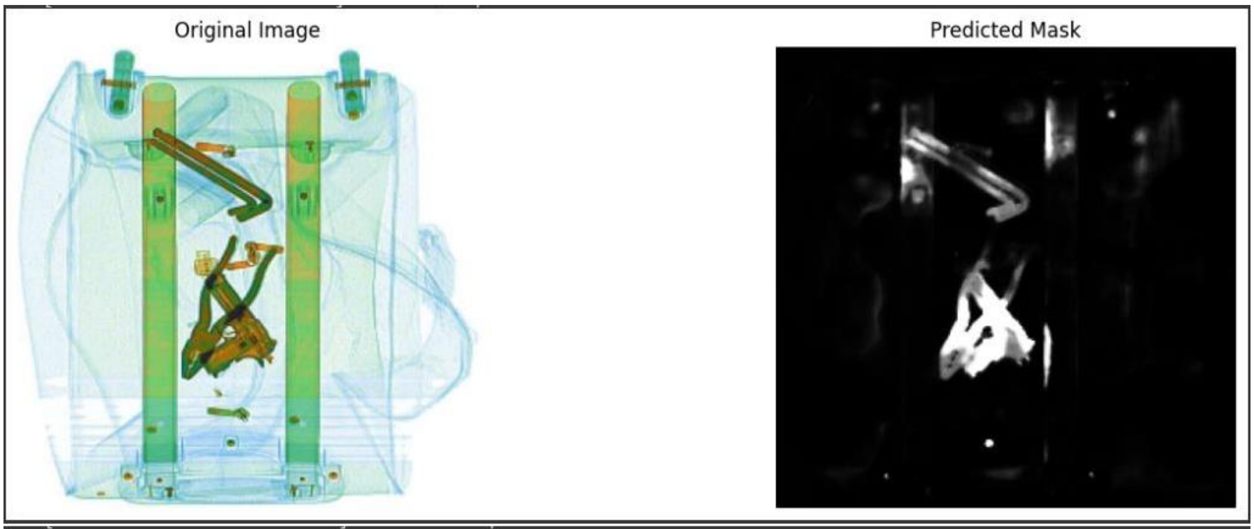
Resizing the threat images passing it to to segmentation model to predict mask

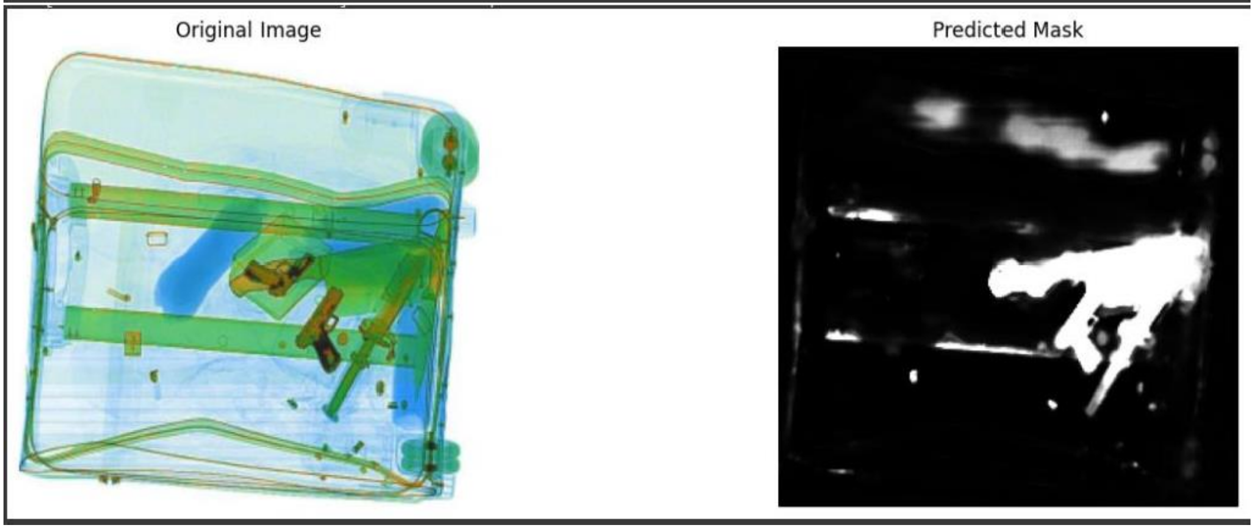
```
 #  Total images classified as threat by our model were 427 .

def display_image_with_masks(index):
    # Ensure the index is within range
    if index >= len(threat_images):
        print("Index out of range. Please choose a smaller index.")
        return

    image = threat_images[index]
    image = cv2.resize(image, (256, 256))

    # Predict the mask for the image
    image_input = np.expand_dims(image, axis=0)  # Expand dims to match model input
    predicted_mask = segmentation_model.predict(image_input)
    predicted_mask = np.squeeze(predicted_mask)  # Remove the extra dimension

    # Create a figure with three subplots
    fig, ax = plt.subplots(1, 2, figsize=(15, 5))

    # Display the original image
    ax[0].imshow(image)
    ax[0].set_title('Original Image')
    ax[0].axis('off')



    # Display the predicted mask
    ax[1].imshow(predicted_mask, cmap='gray')
    ax[1].set_title('Predicted Mask')
    ax[1].axis('off')

    plt.show()

# Example usage: Display the image and masks at index 0
display_image_with_masks(102)
```

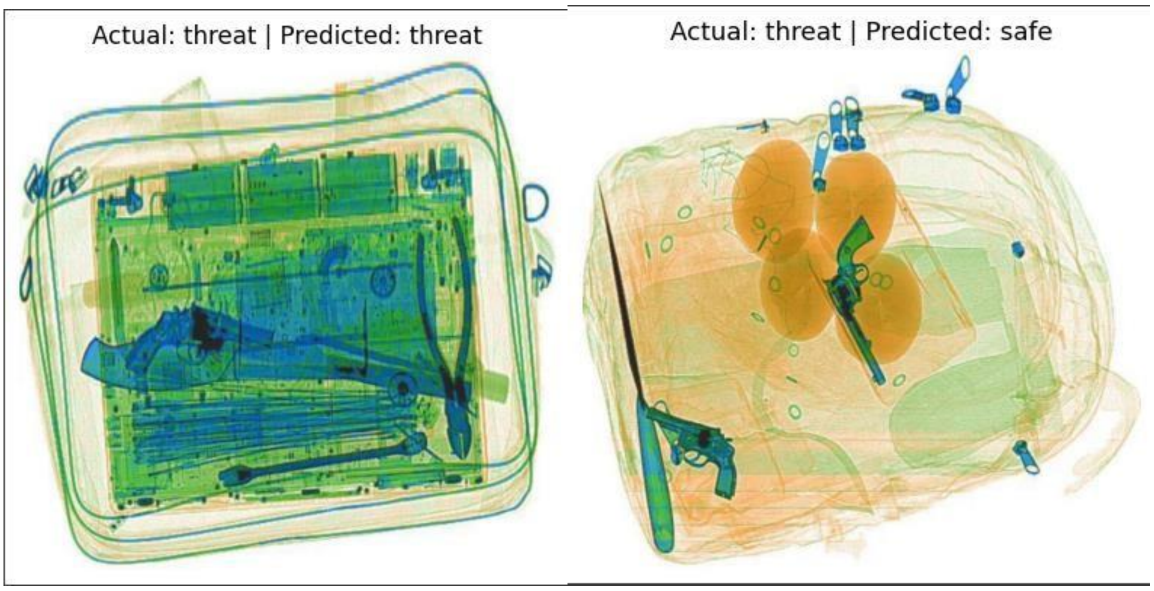
Dice Coefficent

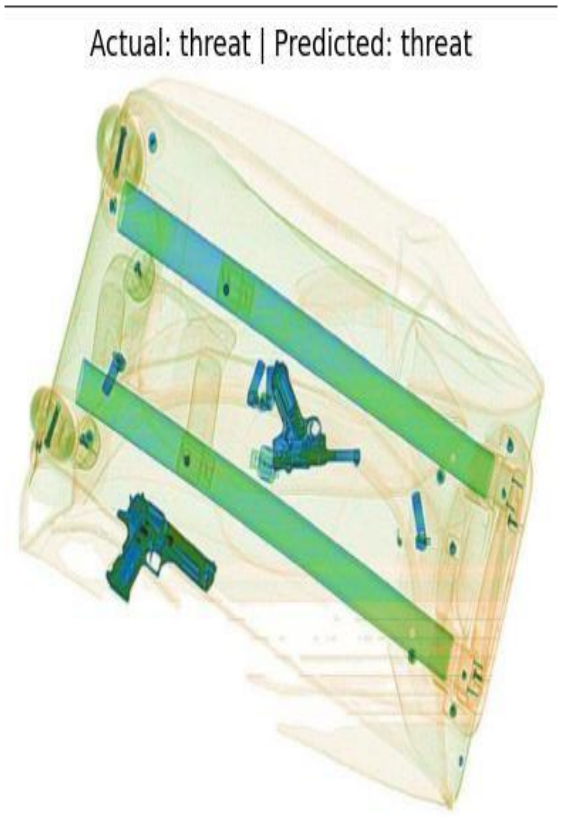
```
import numpy as np


def calculate_dice_coefficient(predicted_labels, actual_labels):
    # Convert labels to binary format (0 for 'safe', 1 for 'threat')
    actual_binary = np.argmax(actual_labels, axis=1)
    predicted_binary = np.argmax(predicted_labels, axis=1)

    # Calculate TP, FP, FN
    TP = np.sum((predicted_binary == 1) & (actual_binary == 1))
    FP = np.sum((predicted_binary == 1) & (actual_binary == 0))
    FN = np.sum((predicted_binary == 0) & (actual_binary == 1))

    # Calculate Dice coefficient
    dice_coefficient = (2 * TP) / (2 * TP + FP + FN)

    return dice_coefficient

predictions = classifier_model.predict(testing_images)
dice_coefficient = calculate_dice_coefficient(predictions, testing_labels)
print(f'Dice Coefficient: {dice_coefficient:.4f}')
```

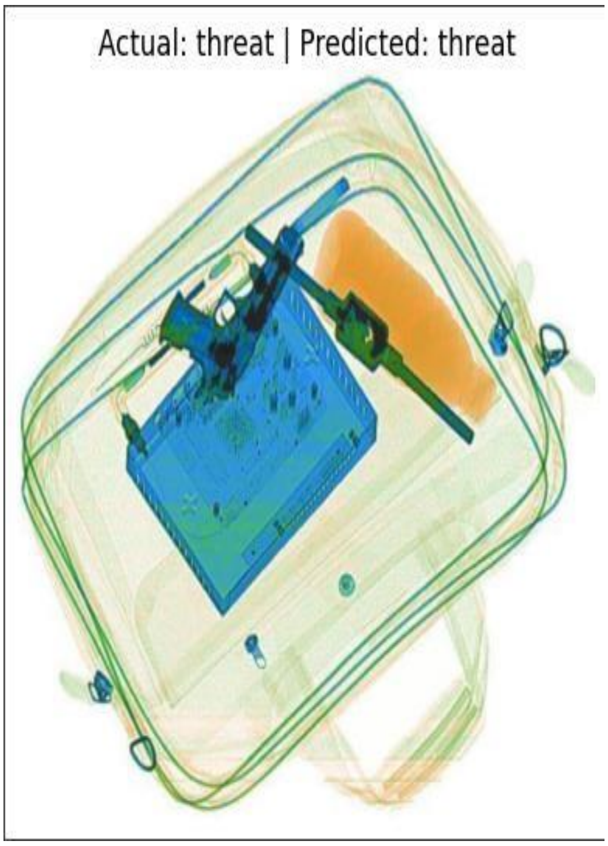
Outputs

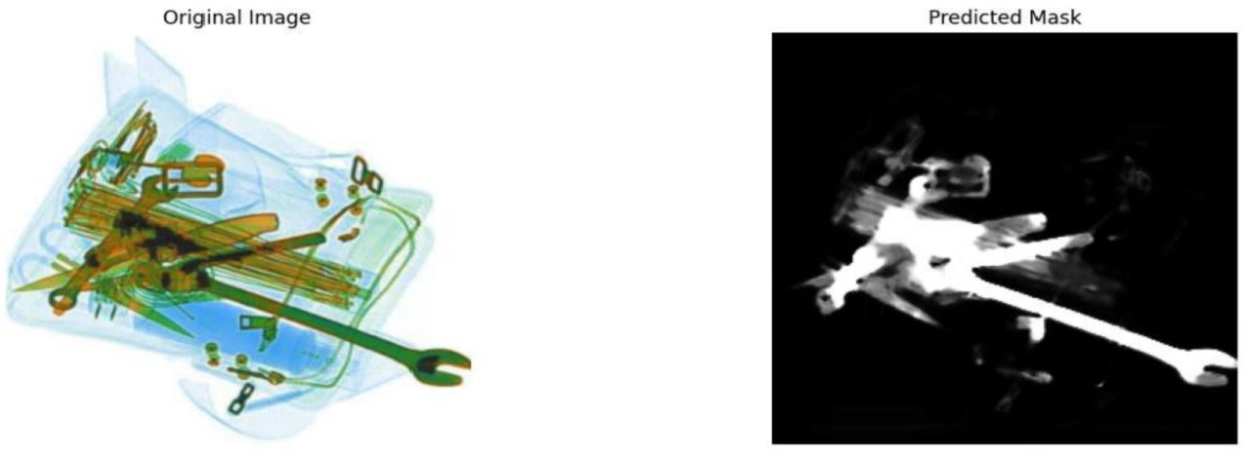

Original Image
Predicted Mask

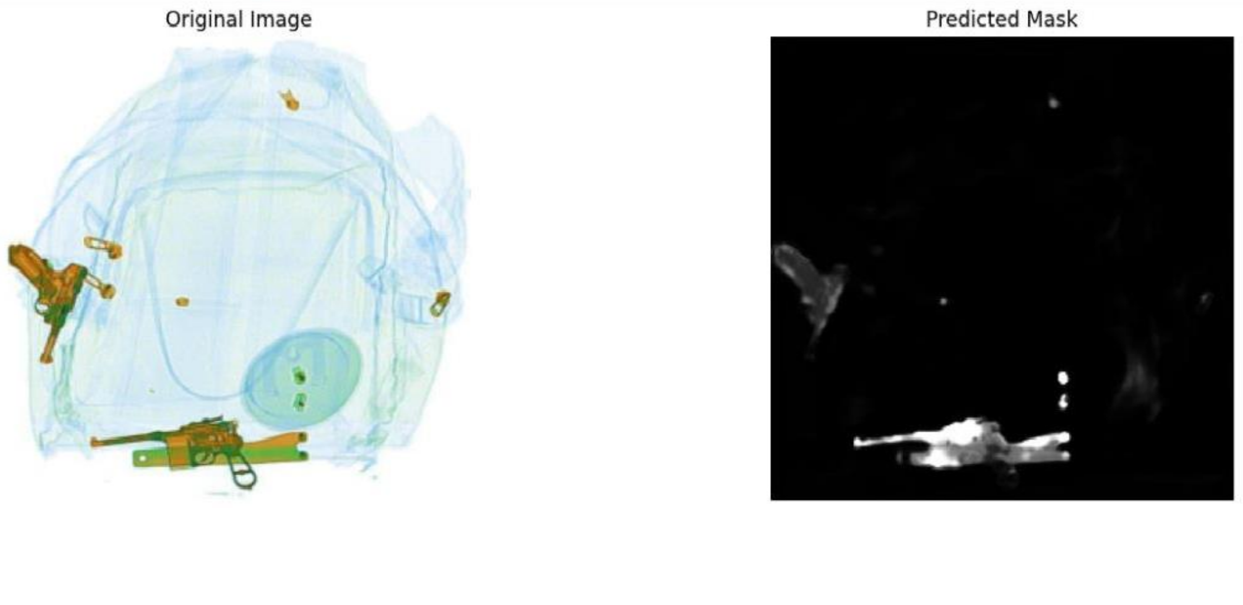

Original Image
Predicted Mask

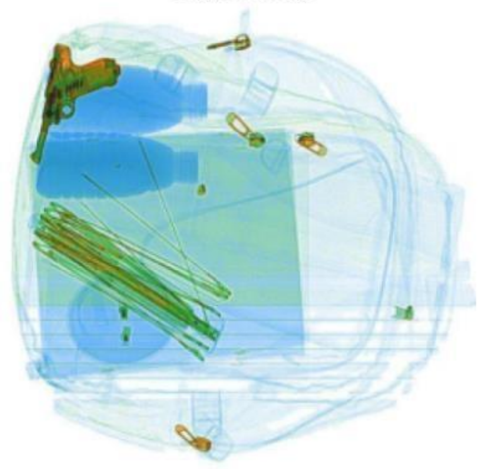

Actual: threat | Predicted: threat
Actual: threat | Predicted: safe

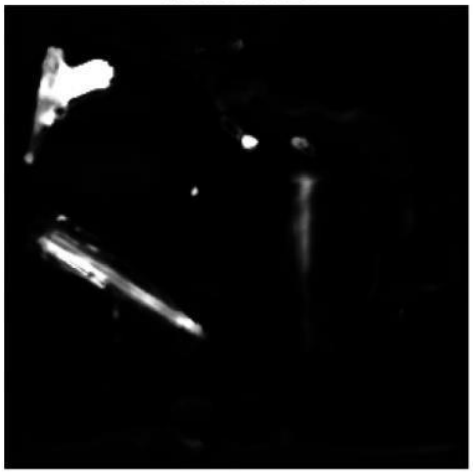

Actual: threat | Predicted: threat

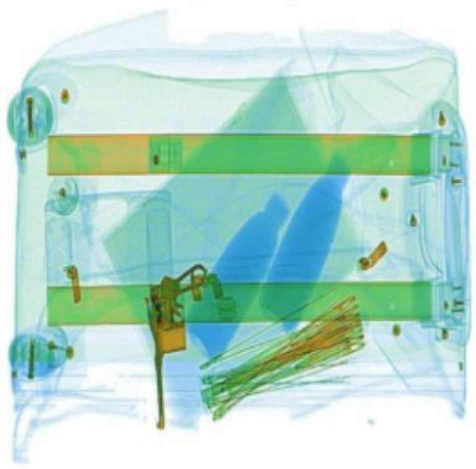

Actual: threat | Predicted: threat

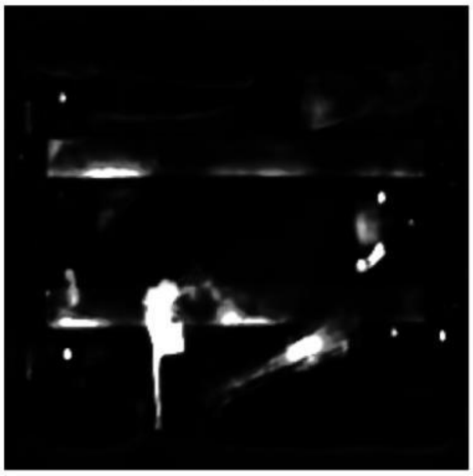

Original Image
Predicted Mask

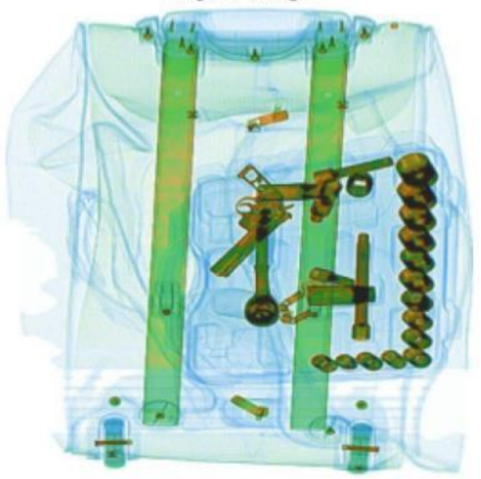

Original Image
Predicted Mask

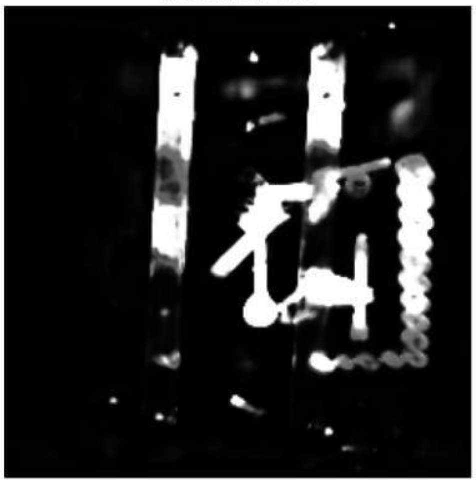

Original Image

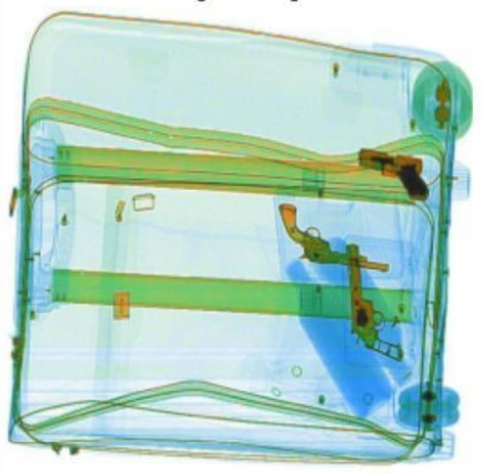

Predicted Mask

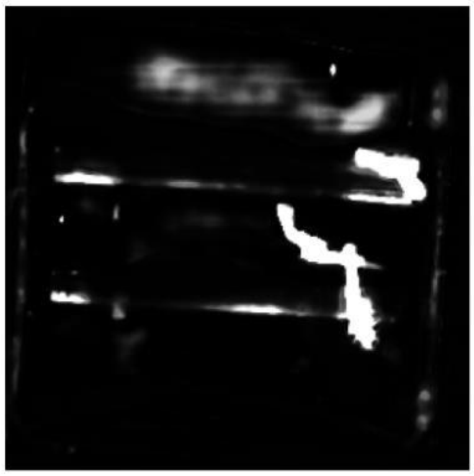

Original Image

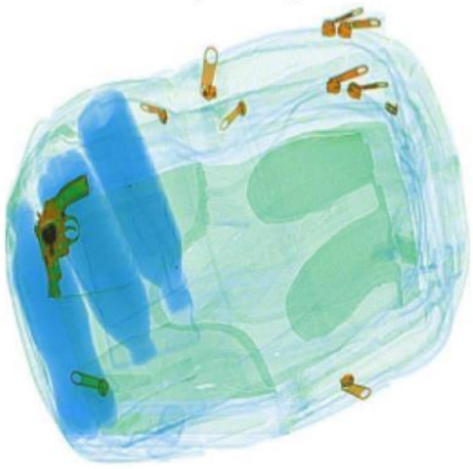

Predicted Mask

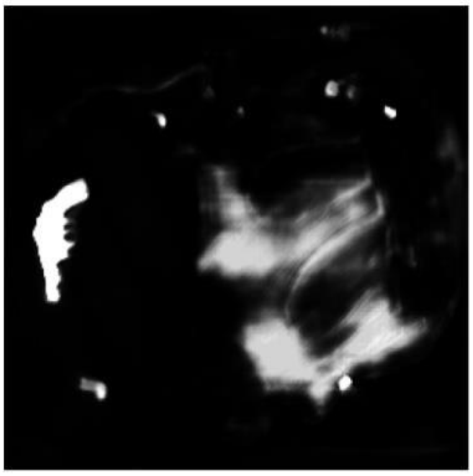

Original Image

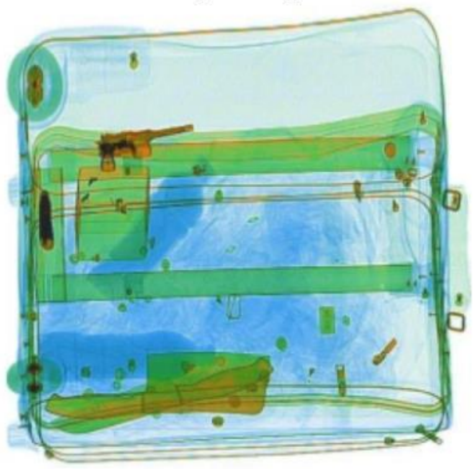

Predicted Mask


Original Image

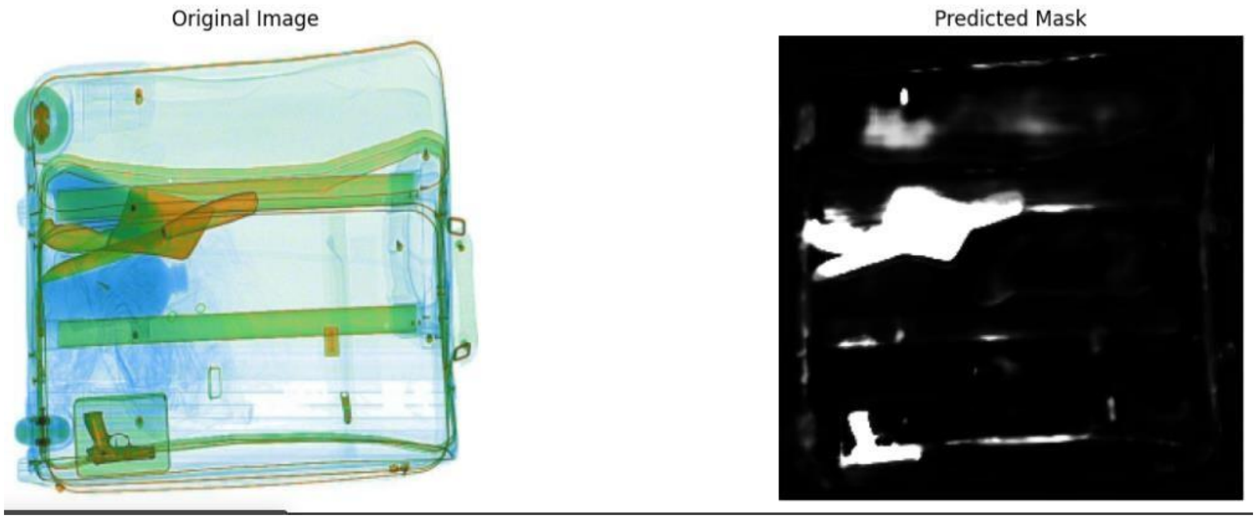

Predicted Mask

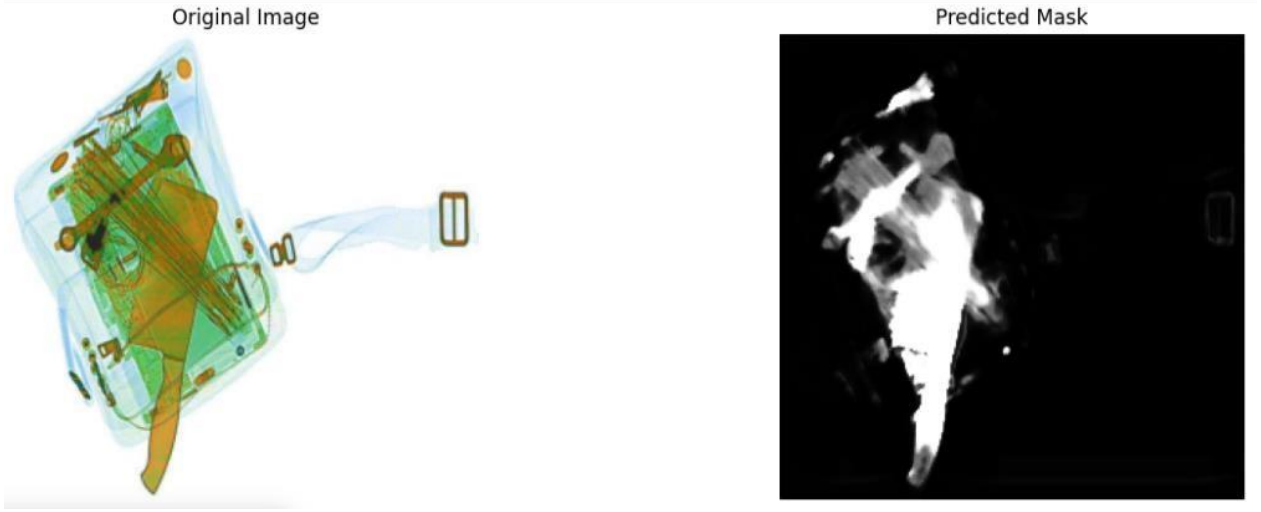

Original Image

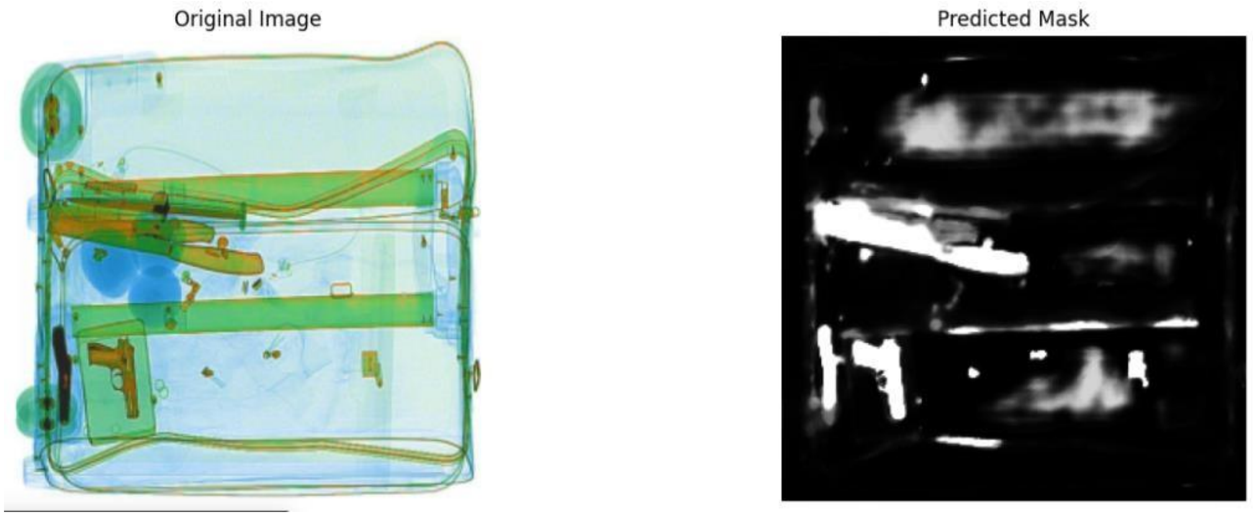

Predicted Mask


Original Image

Predicted Mask


Original Image
Predicted Mask


Original Image
Predicted Mask


Original Image
Predicted Mask

Google Collab Link

Segmentation
https://colab.research.google.com/drive/17QgfakIjptoaNfdEiLqgm3ZUfNh0GRSv?usp=sharing

Classification
https://colab.research.google.com/drive/1Y8zX7AhqlnEg525zO0Igs0CDJtcnetFg?usp=sharing

Intergrating Both
https://colab.research.google.com/drive/1vHGJTWMfWwjQT4bLZF4UfVLaNG15YWyO?usp=sharing