National College of Ireland

Configuration Manual

MSc Research Project
Data Analytics

Zohaib Rasool
Student ID: 23256796

School of Computing
National College of Ireland

Supervisor: Vladimir Milosavljevic

National College of Ireland

MSc Project Submission Sheet

School of Computing

Student Name: Zohaib Rasool

Student ID: 23256796

Programme: Data Analytics **Year:** 2024

Module: MSc Research Project

Supervisor: Vladimir Milosavljevic

Submission Due Date: 24/01/2025

Project Title: Optimized Convolutional-Recurrent Architecture for Detecting Diverse Crimes in Real-Time

Word Count: 1507 **Page Count** 10

I hereby certify that the information contained in this (my submission) is information pertaining to research I conducted for this project. All information other than my own contribution will be fully referenced and listed in the relevant bibliography section at the rear of the project.
ALL internet material must be referenced in the bibliography section. Students are required to use the Referencing Standard specified in the report template. To use other author's written or electronic work is illegal (plagiarism) and may result in disciplinary action.

Signature: Zohaib Rasool

Date: 24/01/2025

PLEASE READ THE FOLLOWING INSTRUCTIONS AND CHECKLIST

Attach a completed copy of this sheet to each project (including multiple copies)	□
Attach a Moodle submission receipt of the online project submission, to each project (including multiple copies).	□
You must ensure that you retain a HARD COPY of the project, both for your own reference and in case a project is lost or mislaid. It is not sufficient to keep a copy on computer.	□

Assignments that are submitted to the Programme Coordinator Office must be placed into the assignment box located outside the office.

Office Use Only	
Signature:	
Date:	
Penalty Applied (if applicable):	

Configuration Manual

Zohaib Rasool
Student ID: 23256796

1 Introduction

The configuration manual provides details about the hardware and software resources used during the project in the sections 2 and 3 respectively. Not only that but the steps that were taken to develop this project are also mentioned in section 4.

2 Hardware Configuration

The experiments were conducted on a system equipped with:

- Intel(R) Xeon(R) CPU @ 2.20GHz
- 16 GB RAM
- NVIDIA Tesla P100 GPU 16 GB

3 Software Requirements

Kaggle code IDE[1] platform is used where **Python** served as the programming language to build the infrastructure and among all the libraries, **TensorFlow** framework is used for model implementation with **OpenCV** for data pre-processing, **OS** and **Pandas** for reading file label to categorize crime and **sklearn, matplotlib** and **numpy** for plotting graphs.

4 Project Development

4.1 Setting up default variables and Loading libraries

```
import os
import cv2
import numpy as np
import pandas as pd



dataset_dir = r'/kaggle/input/thesis/DCSASS Dataset'
labels_dir = 'Labels/'
frames_per_video = 1
image_height, image_width = 256, 256
num_classes = 13
```

Figure 1 Default variables and initial libraries

[1] https://www.kaggle.com/code

Figure 1 shows the intial libraries that are used during the project that is **OS** to redirect into directories for dataset, **cv2** for image preprocessing, **pandas** to read the csv files for labels of crimes and numpy to perform operations on the images and to plot the graphs. Then there are the default variables including the dataset file directory, number of classes, image resolution, number of frames that needs to be extracted from each video and the labels directory.

4.2 Loading Video Frames

```
def load_video_frames(video_path, frames_per_video):
    cap = cv2.VideoCapture(video_path)
    frames = []

    while True:
        ret, frame = cap.read()
        if not ret:
            break

        frame = cv2.resize(frame, (image_width, image_height))
        gray_frame = cv2.cvtColor(frame, cv2.COLOR_RGB2GRAY)
        frames.append(gray_frame)
    cap.release()

    num_frames = len(frames)
    if num_frames > frames_per_video:
        indices = np.linspace(0, num_frames - 1, frames_per_video, dtype=int)
        frames = [frames[i] for i in indices]
    elif num_frames < frames_per_video:

        frames += [frames[-1]] * (frames_per_video - num_frames)

    return np.array(frames)
```

Figure 2 Function to load video frames

How a frame is extracted from a video and is converted to grayscale is shown in *Figure 2*. Each frame after extraction is firstly resized and then converted to grayscale. After that it is made sure that correct number of frames are present since the videos are of variable lengths so the last frame is repeated if the video length is less than of a second.

4.3 Class Mappings

```
class_mapping = {
    'Normal': 0,
    'Abuse': 1,
    'Arrest': 2,
    'Arson': 3,
    'Assault': 4,
    'Burglary': 5,
    'Explosion': 6,
    'RoadAccidents': 7,
    'Robbery': 8,
    'Shooting': 9,
    'Stealing': 10,
    'Shoplifting': 11,
    'Vandalism': 12
}
```

Figure 3 Class Mapping for each crime category

Figure 3 tells about the mapping that how each crime category is mapped to encode the categories.

4.4 Loading the complete Dataset and Labels

```
def load_dataset(dataset_dir, labels_dir):
    X, y = [], []

    for crime_type in os.listdir(dataset_dir):

        if crime_type == 'Labels' or crime_type == 'checkFiles.ipynb':
            continue

        crime_folder_path = os.path.join(dataset_dir, crime_type)
        print(crime_folder_path)

        if not os.path.isdir(crime_folder_path):
            print(f"Warning: {crime_folder_path} is not a directory.")
            continue

        csv_file_path = os.path.join(dataset_dir, labels_dir, f"{crime_type}.csv")

        if not os.path.exists(csv_file_path):
            print(f"Warning: CSV file {csv_file_path} not found.")
            continue

        labels_df = pd.read_csv(csv_file_path)

        for _, row in labels_df.iterrows():
            video_id = row['Filename']
            label = row['Label']

            folder_name, video_number = video_id.rsplit('_', 1)
            video_file_path = os.path.join(crime_folder_path, f"{folder_name}.mp4")
            video_file_path = os.path.join(crime_folder_path, folder_name + '.mp4', f"{folder_name}{'_'}{video_number}.mp4")

            if os.path.exists(video_file_path):
                frames = load_video_frames(video_file_path, frames_per_video)
                X.append(frames)
                if label == 0:
                    y.append(label)
                else:
                    crime_label = class_mapping[crime_type]
                    y.append(crime_label)
            else:
                print(f"Warning: Video file {video_file_path} not found.")

    return np.array(X), np.array(y)
```

Figure 4 Function to load dataset and labels

The function ***load_dataset*** from *Figure 4* also uses the function ***load_video_frames*** from *Figure 2*. What the function (***load_dataset)*** does is that it loads the videos and their corresponding labels. It iterates through the directory in which all crime videos are organized in their folder and keeps reading a csv file for the name of the video file and the label for it as label 0 means that it is just a normal video and the label for class mappings from *Figure 3* are used to encode the crime label into a numerical value. Not only that but the function also makes sure if the video file is present or not. Warnings are issued if either the folder, csv or video file is not present and then the function returns video frames with **X** and its corresponding labels **y**.

4.5 Function to view frames

```
import matplotlib.pyplot as plt

def get_class_samples(X, y, class_mapping):

    reverse_class_mapping = {v: k for k, v in class_mapping.items()}
    class_samples = {}

    for i, label in enumerate(y):
        class_name = reverse_class_mapping[label]
        if class_name != "Normal" and class_name not in class_samples:
            class_samples[class_name] = X[i][0]
        if len(class_samples) == len(class_mapping) - 1:
            break

    return class_samples

class_samples = get_class_samples(X, y, class_mapping)

plt.figure(figsize=(15, 8))
num_classes = len(class_samples)

for i, (class_name, frame) in enumerate(class_samples.items()):
    plt.subplot(3, 4, i+1)
    plt.imshow(frame, cmap="gray")
    plt.title(class_name)
    plt.axis('off')

plt.tight_layout()
plt.show()
```

Figure 5 Viewing frame for each crime category

get_class_samples in *Figure 5* is just a simple function that only displays a frame from each crime category and here a **matplotlib** is used to view the frames. The class_mappings from *Figure 3* are reversed so that they can be displayed on the image since labels will not mean anything on the figure rather the crime name needs to be displayed.

4.6 Splitting Dataset

```
from sklearn.model_selection import train_test_split

print(f"X shape: {X.shape}, y shape: {y.shape}")
X_train, X_temp, y_train, y_temp = train_test_split(X, y, test_size=0.4, random_state=42, stratify=y)
X_val, X_test, y_val, y_test = train_test_split(X_temp, y_temp, test_size=0.5, random_state=42, stratify=y_temp)

del X
del y
del X_temp
del y_temp

print(f"Training set size: {len(X_train)}, Validation set size: {len(X_val)}, Test set size: {len(X_test)}")
```

Figure 6 Dataset splitting into 60-20-20

Dataset in *Figure 6* is split into 60% for training, 20% for testing and validation. The ***test_size*** argument in the first ***train_test_split*** fucntion actually prepares the variables ***X_train*** and ***y_train*** since these are the training data variables. If ***test_size*** is to be changed 0.3 and 0.2 then the training data will be 70% and 80% respectively. The next ***train_test_split*** function actually splits the data for testing and validation so there is a 0.5 split. The argument ***stratify*** plays a very important role since it makes sure that all the classes in ***y*** are distributed accordancing to the number of labels that they have so this makes sure that not only the data but the classes are also distributed equally. Variables ***x, y, X_temp*** and ***y_temp*** were then deleted to reserve memory (RAM).

4.7 Checking class distribution and encoding labels

```
def print_class_distribution(y, dataset_name):
    unique, counts = np.unique(y, return_counts=True)
    print(f"Class distribution in {dataset_name}: {dict(zip(unique, counts))}")

print_class_distribution(y_train, 'Training set')
print_class_distribution(y_val, 'Validation set')
print_class_distribution(y_test, 'Test set')

from tensorflow.keras.utils import to_categorical

y_train_encoded = to_categorical(y_train, num_classes=13)
y_val_encoded = to_categorical(y_val, num_classes=13)
y_test_encoded = to_categorical(y_test, num_classes=13)

print(f"X_train shape: {X_train.shape}, y_train shape: {y_train_encoded.shape}")
print(f"X_val shape: {X_val.shape}, y_val shape: {y_val_encoded.shape}")
print(f"X_test shape: {X_test.shape}, y_test shape: {y_test_encoded.shape}")
```

Figure 7 Converting labels to use in model and checking size

The size of train, test and validation data is verified by using ***print_class_distribution*** fucntion here in *Figure 7* and then the labels are converted using ***to_categorical*** from **keras** which converts the labels into a binary matrix format which is required for tensorflow to train the model. Lastly the code prints the shape of train, test and validation data so that it can be verified that data and labels are correctly formatted and aligned for the use in model training.

4.8 Models

```
import tensorflow as tf
from tensorflow.keras import layers, models

def build_model(input_shape, num_classes):

    inputs = layers.Input(shape=input_shape)

    x = layers.Conv3D(filters=64, kernel_size=(3, 3, 3), strides=(1, 2, 2), padding="same", activation="relu")(inputs)
    conv1_out = x
    lstm1 = layers.ConvLSTM2D(filters=64, kernel_size=(3, 3), padding="same", return_sequences=True)(conv1_out)
    deconv4 = layers.Conv3DTranspose(filters=64, kernel_size=(3, 3, 3), strides=(1, 2, 2), padding="same", activation="relu")(lstm1)
    x = layers.Flatten()(deconv4)
    x = layers.Dense(128, activation='relu')(x)
    x = layers.Dropout(0.5)(x)

    outputs = layers.Dense(num_classes, activation='softmax')(x)
    model = models.Model(inputs=inputs, outputs=outputs)
    return model
input_shape = (frames_per_video, image_height, image_width, 1)
model = build_model(input_shape, num_classes=13)
model.compile(optimizer='adam', loss='categorical_crossentropy', metrics=['accuracy'])
model.summary()
```

Figure 8 Version 1 (V1) Architecture

This code from *Figure 8* constructs and trains a video classification model into a by using **tensorflow and keras**. It begins by creating a ***build_model*** function that accepts the number of dimensions in the input video data (***input_shape***) and the total number of classes (***num_classes***) in the output. First we concatenate the video frames and apply a convolution operation in the spatial & temporal domain which is called ***Conv3D***. After, they pass trough ***ConvLSTM2D*** that enables to model temporal dependencies based on convolutional operations. The features extracted is then passed through a deconvolutional layer (***Conv3DTranspose***) to increase its spatial dimensions then flattened to a 1D array. A ***Dense*** layer having 128 neurons forms the fully connected layers and the network includes a ***Dropout*** layer, with a drop rate of 50% to minimize overfitting. Lastly, the model produces the ***softmax*** layer which gives the 13 class probability distribution for multi-class classification. The above model is built with ***Adam*** optimizer, ***categorical_crossentropy*** loss function and accuracy as the metric where the optimizer's learning rate is set to 0.001 by default. The structure and the details such as parameters of built model are displayed using the ***model.summary()***. The model in ready for training, while the input shape variables which include the ***frames_per_video, image_height and image_width*** which are properly defined in *Figure 1*. *Figure 9, Figure 10, Figure 11* are architectures of Version 2 (V2), Version 3 (V3), Version 4 (V4) and Version 5 (V5) respectively. They all follow the same code structure except the fact that they have more of the core layers due to incremental development of the model.

```
#Conv3D layers
x = layers.Conv3D(filters=64, kernel_size=(3, 3, 3), strides=(1, 2, 2), padding="same", activation="relu")(inputs)
conv1_out = x  # For LSTM1
x = layers.Conv3D(filters=32, kernel_size=(3, 3, 3), strides=(1, 2, 2), padding="same", activation="relu")(x)
conv2_out = x  # For LSTM2
# LSTM layers
lstm1 = layers.ConvLSTM2D(filters=64, kernel_size=(3, 3), padding="same", return_sequences=True)(conv1_out)
lstm2 = layers.ConvLSTM2D(filters=32, kernel_size=(3, 3), padding="same", return_sequences=True)(conv2_out)
# De-conv Layers
deconv3 = layers.Conv3DTranspose(filters=64, kernel_size=(3, 3, 3), strides=(1, 2, 2), padding="same", activation="relu")(lstm2 + conv2_out)
deconv4 = layers.Conv3DTranspose(filters=64, kernel_size=(3, 3, 3), strides=(1, 2, 2), padding="same", activation="relu")(lstm1 + deconv3)
```

Figure 9 Version 2 (V2) Architecture

```
# Conv3D layers
x = layers.Conv3D(filters=64, kernel_size=(3, 3, 3), strides=(1, 2, 2), padding="same", activation="relu")(inputs)
conv1_out = x  # For LSTM1
x = layers.Conv3D(filters=32, kernel_size=(3, 3, 3), strides=(1, 2, 2), padding="same", activation="relu")(x)
conv2_out = x  # For LSTM2
x = layers.Conv3D(filters=16, kernel_size=(3, 3, 3), strides=(1, 2, 2), padding="same", activation="relu")(x)
conv3_out = x  # For LSTM3
# LSTM layers
lstm1 = layers.ConvLSTM2D(filters=64, kernel_size=(3, 3), padding="same", return_sequences=True)(conv1_out)
lstm2 = layers.ConvLSTM2D(filters=32, kernel_size=(3, 3), padding="same", return_sequences=True)(conv2_out)
lstm3 = layers.ConvLSTM2D(filters=16, kernel_size=(3, 3), padding="same", return_sequences=True)(conv3_out)
# De-conv layers
deconv2 = layers.Conv3DTranspose(filters=32, kernel_size=(3, 3, 3), strides=(1, 2, 2), padding="same", activation="relu")(lstm3 + conv3_out)
deconv3 = layers.Conv3DTranspose(filters=64, kernel_size=(3, 3, 3), strides=(1, 2, 2), padding="same", activation="relu")(lstm2 + deconv2)
deconv4 = layers.Conv3DTranspose(filters=64, kernel_size=(3, 3, 3), strides=(1, 2, 2), padding="same", activation="relu")(lstm1 + deconv3)
```

Figure 10 Version 3 (V3) Architecture

```
# Conv3D layers
x = layers.Conv3D(filters=64, kernel_size=(3, 3, 3), strides=(1, 2, 2), padding="same", activation="relu")(inputs)
conv1_out = x  # For LSTM1
x = layers.Conv3D(filters=32, kernel_size=(3, 3, 3), strides=(1, 2, 2), padding="same", activation="relu")(x)
conv2_out = x  # For LSTM2
x = layers.Conv3D(filters=32, kernel_size=(3, 3, 3), strides=(1, 2, 2), padding="same", activation="relu")(x)
conv3_out = x  # For LSTM3
x = layers.Conv3D(filters=16, kernel_size=(3, 3, 3), strides=(1, 1, 1), padding="same", activation="relu")(x)
conv4_out = x  # For LSTM4

# LSTM layers
lstm1 = layers.ConvLSTM2D(filters=64, kernel_size=(3, 3), padding="same", return_sequences=True)(conv1_out)
lstm2 = layers.ConvLSTM2D(filters=32, kernel_size=(3, 3), padding="same", return_sequences=True)(conv2_out)
lstm3 = layers.ConvLSTM2D(filters=32, kernel_size=(3, 3), padding="same", return_sequences=True)(conv3_out)
lstm4 = layers.ConvLSTM2D(filters=16, kernel_size=(3, 3), padding="same", return_sequences=True)(conv4_out)

# De-conv1 layers
deconv1 = layers.Conv3DTranspose(filters=32, kernel_size=(3, 3, 3), strides=(1, 1, 1), padding="same", activation="relu")(lstm4 + conv4_out)
deconv2 = layers.Conv3DTranspose(filters=32, kernel_size=(3, 3, 3), strides=(1, 2, 2), padding="same", activation="relu")(lstm3 + deconv1)
deconv3 = layers.Conv3DTranspose(filters=64, kernel_size=(3, 3, 3), strides=(1, 2, 2), padding="same", activation="relu")(lstm2 + deconv2)
deconv4 = layers.Conv3DTranspose(filters=64, kernel_size=(3, 3, 3), strides=(1, 2, 2), padding="same", activation="relu")(lstm1 + deconv3)
```

Figure 11 Version 4 (V4) and Version 5 (V5) Architecture

V4 and V5 are the same because V5 is only trained for 100 epochs and V4 is trained for 10 as to see that how epochs affect the model's performance.

4.9 Model Evaluation and Plotting Results

```
test_loss, test_accuracy = model.evaluate(X_test, y_test_encoded, batch_size=8)

print(f"Test Accuracy: {test_accuracy * 100:.2f}%")
print(f"Test Loss: {test_loss * 100:.2f}%")
```

Figure 12 Test accuracy and loss

The code in *Figure 12* is used to check that how the model is performing over test data and to print the results. *Figure 13* displays the code where model's training and validation accuracy and loss is being plotted to visualize the model's performance when it was in its training phase over each epoch.

```
plt.plot(history.history['accuracy'])
plt.plot(history.history['val_accuracy'])
plt.title('Model accuracy')
plt.ylabel('Accuracy')
plt.xlabel('Epoch')
plt.legend(['Train', 'Validation'], loc='upper left')
plt.show()

plt.plot(history.history['loss'])
plt.plot(history.history['val_loss'])
plt.title('Model loss')
plt.ylabel('Loss')
plt.xlabel('Epoch')
plt.legend(['Train', 'Validation'], loc='upper left')
plt.show()
```

Figure 13 Plots for training accuracy and loss

The ***plot_prediction*** function in *Figure 14* is used to visualize model's prediction over test dataset by displaying the true and predicted labels over the images. After creating a reverse mapping of the original ***class_mappings,*** the fuction selects 10 images for normal class and rest are for the other different classes meaning 20 since 30 is the default parameter set for sample of images. After then selecting the random indexes, it then appends the variables and then shuffles them and after that a plot is being made by using the index (***idx***) and selecting images from ***X_test***, true labels from ***y_test*** and then predicting the labels using ***model.predict***.

```
def plot_predictions(X_test, y_test, model, class_mapping, num_samples=30):

    reverse_class_mapping = {v: k for k, v in class_mapping.items()}
    normal_indices = [i for i, label in enumerate(y_test) if reverse_class_mapping[label] == "Normal"]
    other_indices = [i for i, label in enumerate(y_test) if reverse_class_mapping[label] != "Normal"]
    selected_normal_indices = np.random.choice(normal_indices, 10, replace=False)

    selected_other_indices = np.random.choice(other_indices, min(len(other_indices), 20), replace=False)
    selected_indices = np.concatenate((selected_normal_indices, selected_other_indices))
    np.random.shuffle(selected_indices)
    plt.figure(figsize=(15, 15))

    for i, idx in enumerate(selected_indices[:num_samples]):
        image = X_test[idx]
        true_label = y_test[idx]
        image_batch = np.expand_dims(image, axis=0)
        pred_label = np.argmax(model.predict(image_batch), axis=1)[0]
        plt.subplot(6, 5, i+1)
        plt.imshow(image[0], cmap='gray')
        plt.title(f"True: {reverse_class_mapping[true_label]}, Pred: {reverse_class_mapping[pred_label]}")
        plt.axis('off')

    plt.tight_layout()
    plt.show()
plot_predictions(X_test, y_test, model, class_mapping, num_samples=30)
```

Figure 14 Model performance over test images

```
from sklearn.metrics import roc_auc_score
from sklearn.preprocessing import label_binarize

y_test_one_hot = label_binarize(y_test, classes=list(class_mapping.values()))
y_pred_proba = model.predict(X_test)

try:
    auc = roc_auc_score(y_test_one_hot, y_pred_proba, average='macro', multi_class='ovr')
    print(f"AUC: {auc:.2f}")
except ValueError as e:
    print(f"Error calculating AUC: {e}")
```

Figure 15 AUC-ROC score

For AUC-ROC score, **sklearn** has been used as shown *Figure 15*. Firstly y_test labels are one hot encoded using ***label_binarize*** and predictions are made using ***model.predict*** and then the AUC is calculated by using ***roc _auc_score*** function where ***average='macro'*** is used to compute the average AUC across all classes and ***multi_class='ovr'*** for a one-vs-rest strategy. In *Figure 16*, the plot for AUC-ROC is made where the false and true postive rate is first calculated (***fpr*** and ***tpr***) and then the plot is being made where a diagonal dashed line is made from [0, 1] to [0, 1] in order to represent a baseline of 0.5 AUC and ROC curves for rest of the classes are plotted.

```
from sklearn.metrics import roc_curve, auc

y_pred_prob = model.predict(X_test)
n_classes = len(class_mapping)
fpr = dict()
tpr = dict()
roc_auc = dict()
y_test_binarized = tf.keras.utils.to_categorical(y_test, num_classes=n_classes)

for i in range(n_classes):
    fpr[i], tpr[i], _ = roc_curve(y_test_binarized[:, i], y_pred_prob[:, i])
    roc_auc[i] = auc(fpr[i], tpr[i])

plt.figure(figsize=(10, 8))
for i in range(n_classes):
    plt.plot(fpr[i], tpr[i], label=f'Class {i}: AUC = {roc_auc[i]:.2f}')

plt.plot([0, 1], [0, 1], 'k--', label='Random Guessing (AUC = 0.5)')
plt.xlim([0.0, 1.0])
plt.ylim([0.0, 1.05])
plt.xlabel('False Positive Rate')
plt.ylabel('True Positive Rate')
plt.title('Receiver Operating Characteristic (ROC) Curve')
plt.legend(loc='lower right')
plt.grid()
plt.show()
```

Figure 16 AUC-ROC plots

```python
from sklearn.metrics import precision_recall_fscore_support

def evaluate_and_visualize_metrics(y_true, y_pred, class_mapping):

    reverse_class_mapping = {v: k for k, v in class_mapping.items()}
    class_names = [reverse_class_mapping[i] for i in sorted(reverse_class_mapping.keys())]
    per_class_precision, per_class_recall, per_class_f1, _ = precision_recall_fscore_support(y_true, y_pred, average=None)
    bar_width = 0.25
    indices = np.arange(len(class_names))

    plt.figure(figsize=(15, 8))
    plt.bar(indices - bar_width, per_class_precision, bar_width, label='Precision', color='skyblue')
    plt.bar(indices, per_class_recall, bar_width, label='Recall', color='lightgreen')
    plt.bar(indices + bar_width, per_class_f1, bar_width, label='F1-Score', color='salmon')
    plt.xticks(indices, class_names, rotation=45, ha='right', fontsize=10)
    plt.ylabel("Score")
    plt.title("Precision, Recall, and F1-Score for Each Class")
    plt.ylim([0, 1])
    plt.legend()
    plt.tight_layout()
    plt.show()

    precision, recall, f1, _ = precision_recall_fscore_support(y_true, y_pred, average='weighted')
    print(f"Weighted Precision: {precision:.4f}")
    print(f"Weighted Recall: {recall:.4f}")
    print(f"Weighted F1-Score: {f1:.4f}")

y_pred = np.argmax(y_pred_prob, axis=1)
evaluate_and_visualize_metrics(y_test, y_pred, class_mapping)
```

Figure 17 Bar plots for precision, recall and F1-Score

The code in *Figure 17* evaluates and plots the precision, recall and F1-Score where the plotting is done on the basis of each class. The metrics of each class are called with easy to identify labels on the bars as well as color differences for easy distinction. Also the weighted average is printed to provide a summary of all the mertrics.