Driver Drowsiness Detection Project

Image Processing Course – Dr. Alireza Akoushideh

Faculty of Engineering, University of Guilan – Fall 2024

Maryam Meshkin – Morteza Mollaei Chafi

Section 1: Introduction

Road accidents, as one of the most significant global safety challenges, claim the lives of millions of people annually. According to the World Health Organization (WHO), driver drowsiness and fatigue are among the key factors contributing to these incidents. Drivers who experience drowsiness due to sleep deprivation or fatigue from prolonged driving not only endanger their own lives but also threaten the safety of other drivers and pedestrians. Therefore, detecting drowsiness has emerged as a critical challenge in the field of road safety, emphasizing the need for the development of intelligent and efficient systems for timely identification of this condition.

In this project, we designed and implemented a suitable model for detecting driver drowsiness. This model utilizes image processing techniques and analyzes the drivers' eye conditions to identify signs of drowsiness. By selecting a reputable dataset and comparing the performance of our model with existing algorithms, we achieved higher accuracy in drowsiness detection. These results highlight the efficiency and reliability of our model in identifying drowsiness and demonstrate its potential as an effective tool for improving driving safety and reducing drowsiness-related accidents.

In the following sections, we will explore the technical details of the project and the results obtained to fully elucidate the significance and effectiveness of this system.

Dataset and Model Overview

The DDD (Driver Drowsiness Detection) dataset is one of the most reputable and comprehensive datasets in the field of driver drowsiness detection, specifically designed for research related to road safety and driver behavior analysis. This dataset includes a variety of images and videos of drivers under different conditions of drowsiness and wakefulness, enabling researchers to optimize their algorithms for detecting and identifying drowsiness. In our project, the DDD dataset was used as the primary data source due to its diversity and large volume, along with key features such as eye conditions, facial movements, and other behavioral indicators. The selection of this dataset allowed us to improve the accuracy and efficiency of our model in detecting driver drowsiness and to deliver reliable results.

In this project, the YOLOv8 (You Only Look Once version 8) model was used as the main algorithm for detecting driver drowsiness. YOLOv8 is one of the most advanced and fastest object detection models in the field of computer vision, specifically designed for real-time object detection and classification. Due to its unique architecture, this model can simultaneously detect multiple objects in an image, significantly enhancing detection speed and accuracy. YOLOv8 was chosen for our project because it not only provides high accuracy in identifying key features such as eye conditions and facial movements but also offers significantly faster processing times compared to other similar algorithms. These features enable effective and efficient real-time detection of driver drowsiness, ultimately contributing to improved road safety.

Methodology

In this project, we initially tested drowsiness detection models without applying any image processing techniques to evaluate their baseline performance. This step helped us assess the accuracy and efficiency of the models under initial conditions without data preprocessing. In the next stage, various image processing methods, including contrast enhancement, noise reduction, and image sharpening, were added to the detection process. This allowed us to observe the positive impact of preprocessing on model accuracy and performance. The results showed that using image processing techniques significantly improved drowsiness detection accuracy, enabling the models to better identify key features, ultimately enhancing the overall system performance.

Key Parameters in YOLO Model

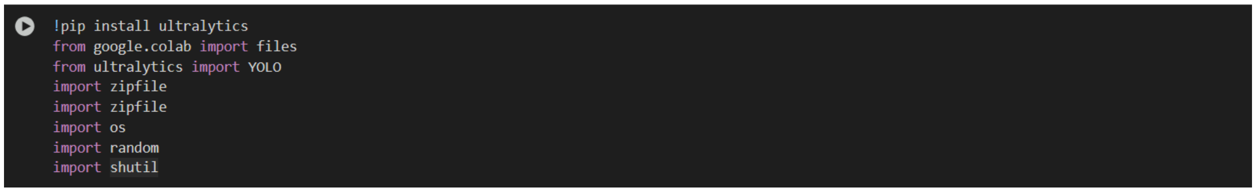
In the YOLO model, three key parameters—image size, batch size, and epoch—play a crucial role in the training process and model performance.

- **Image size** refers to the dimensions of the input images fed into the model. Choosing an appropriate size can significantly affect the model's accuracy and processing speed, as larger sizes generally contain more information but also increase processing time.
- **Batch size** refers to the number of images presented to the model in each training step. Larger batch sizes can improve stability in weight updates but require more memory.
- **Epoch** refers to the number of times the entire dataset is used to train the model. Increasing the number of epochs can help the model learn better, but excessive epochs may lead to overfitting, where the model becomes overly dependent on the training data and performs poorly on new data.

Optimally tuning these parameters is essential to achieve the best performance in detecting driver drowsiness.

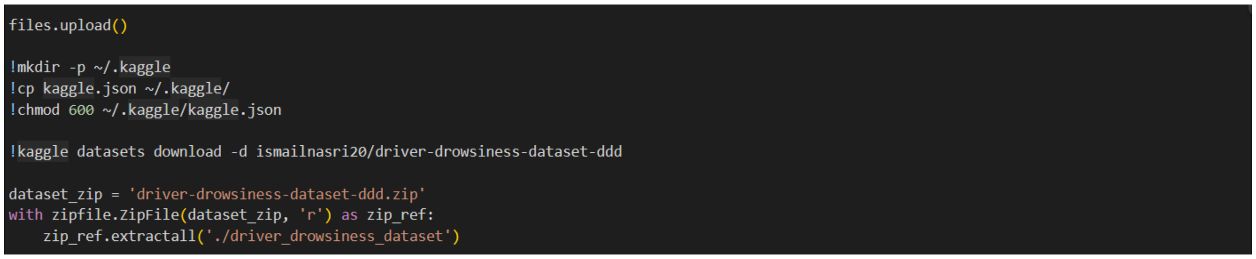
Section 2: Models Without Preprocessing

In this section, we describe the implemented model without applying any preprocessing to the dataset. All the codes mentioned in this section are available and accessible in the project's repository. (You can find these notebooks in the folder **models_without_augmentations**.)



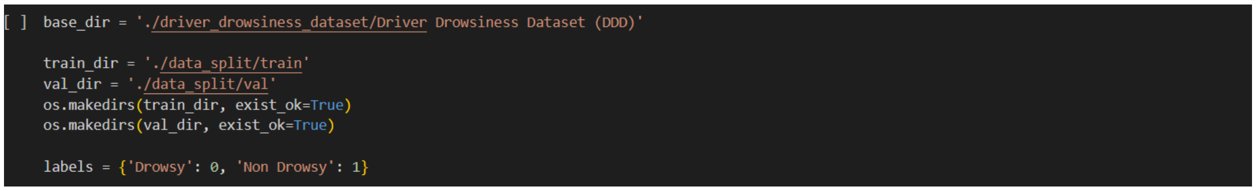
```
!pip install ultralytics
from google.colab import files
from ultralytics import YOLO
import zipfile
import zipfile
import os
import random
import shutil
```

This code is designed for setting up the YOLO environment. First, the UltraLytics library is installed, and the YOLO module is imported. Then, tools are used for uploading files in Colab, managing ZIP files, accessing the file system, generating random values, and moving or deleting files to prepare the data for training or evaluating the model.



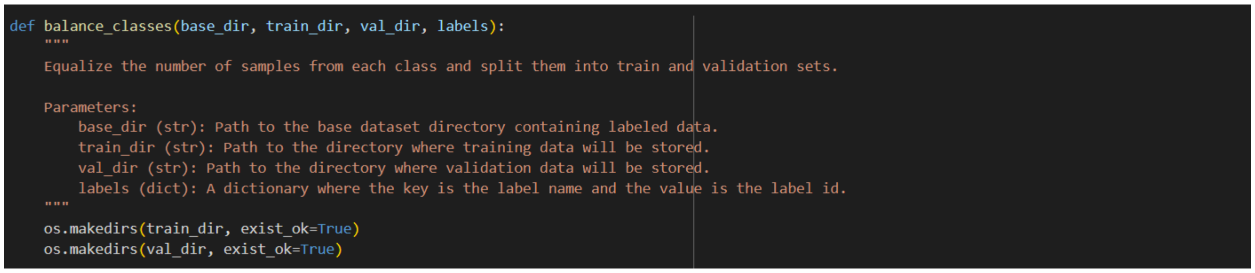
```
files.upload()

!mkdir -p ~/.kaggle
!cp kaggle.json ~/.kaggle/
!chmod 600 ~/.kaggle/kaggle.json

!kaggle datasets download -d ismailnasri20/driver-drowsiness-dataset-ddd

dataset_zip = 'driver-drowsiness-dataset-ddd.zip'
with zipfile.ZipFile(dataset_zip, 'r') as zip_ref:
    zip_ref.extractall('./driver_drowsiness_dataset')
```

This section is used to download and extract the dataset from Kaggle. First, the Kaggle.json file is uploaded and configured for authentication on Kaggle. Then, the driver drowsiness dataset is downloaded, the corresponding ZIP file is extracted, and its contents are placed in a folder named driver_drowsiness_dataset.

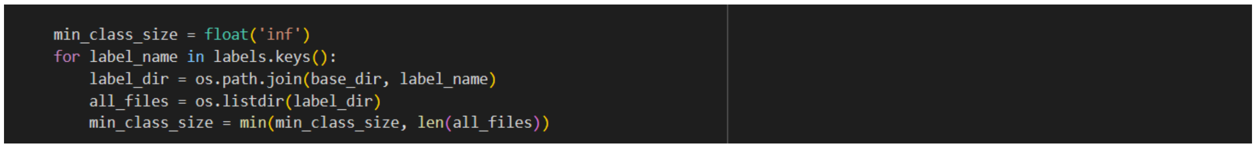


```
[ ] base_dir = './driver_drowsiness_dataset/Driver Drowsiness Dataset (DDD)'

    train_dir = './data_split/train'
    val_dir = './data_split/val'
    os.makedirs(train_dir, exist_ok=True)
    os.makedirs(val_dir, exist_ok=True)

    labels = {'Drowsy': 0, 'Non Drowsy': 1}
```

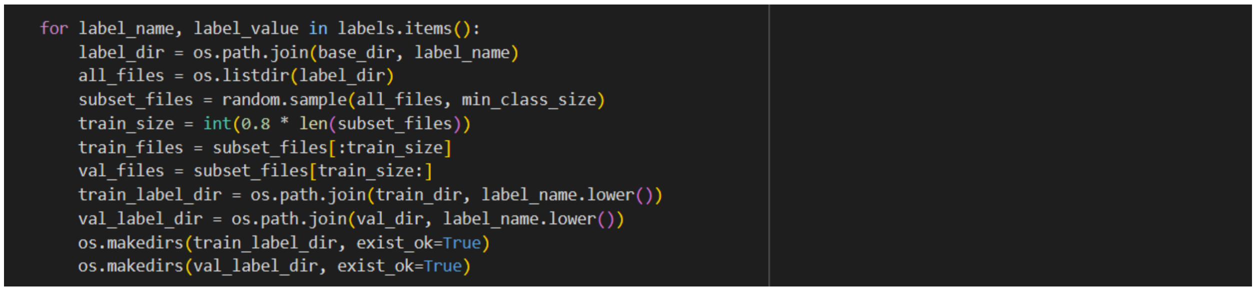
In this section, it sets the paths related to the dataset and the folders for data partitioning. The variable base_dir specifies the main path of the driver drowsiness dataset. Then, two folders, train and val, are created in the data_split path for training and validation data. Additionally, the labels dictionary defines the data labels: Drowsy with a value of 0 and Non-Drowsy with a value of 1.



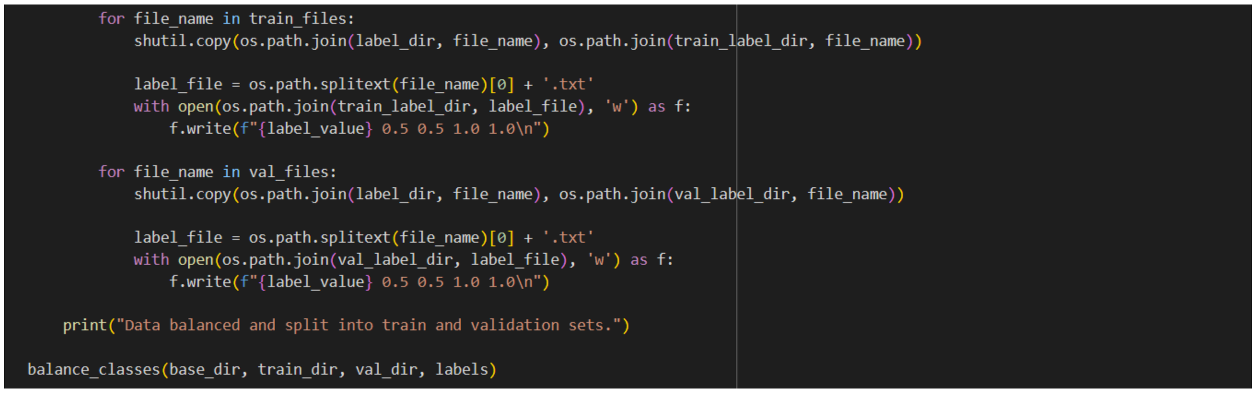
```python
def balance_classes(base_dir, train_dir, val_dir, labels):
    """
    Equalize the number of samples from each class and split them into train and validation sets.

    Parameters:
        base_dir (str): Path to the base dataset directory containing labeled data.
        train_dir (str): Path to the directory where training data will be stored.
        val_dir (str): Path to the directory where validation data will be stored.
        labels (dict): A dictionary where the key is the label name and the value is the label id.
    """
    os.makedirs(train_dir, exist_ok=True)
    os.makedirs(val_dir, exist_ok=True)
```



```python
    min_class_size = float('inf')
    for label_name in labels.keys():
        label_dir = os.path.join(base_dir, label_name)
        all_files = os.listdir(label_dir)
        min_class_size = min(min_class_size, len(all_files))
```

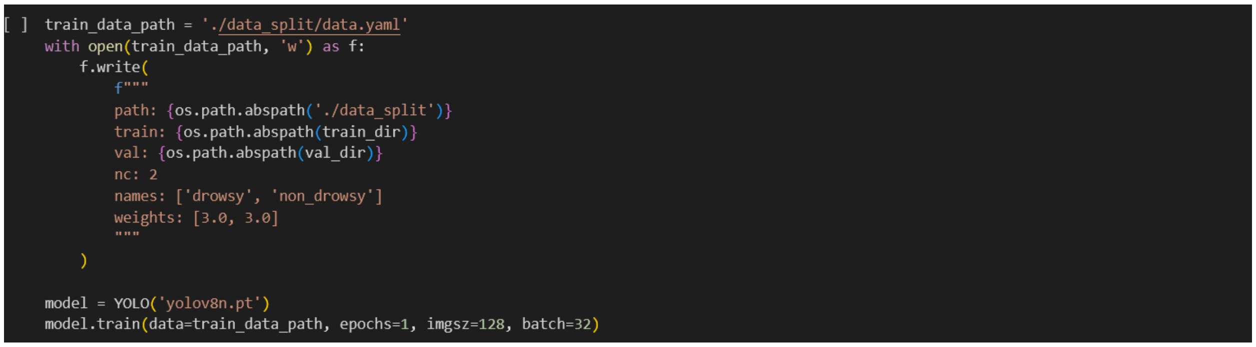


```python
    for label_name, label_value in labels.items():
        label_dir = os.path.join(base_dir, label_name)
        all_files = os.listdir(label_dir)
        subset_files = random.sample(all_files, min_class_size)
        train_size = int(0.8 * len(subset_files))
        train_files = subset_files[:train_size]
        val_files = subset_files[train_size:]
        train_label_dir = os.path.join(train_dir, label_name.lower())
        val_label_dir = os.path.join(val_dir, label_name.lower())
        os.makedirs(train_label_dir, exist_ok=True)
        os.makedirs(val_label_dir, exist_ok=True)
```

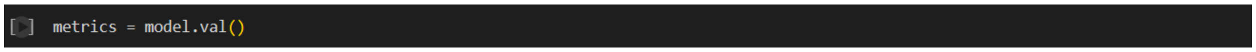


```python
        for file_name in train_files:
            shutil.copy(os.path.join(label_dir, file_name), os.path.join(train_label_dir, file_name))

            label_file = os.path.splitext(file_name)[0] + '.txt'
            with open(os.path.join(train_label_dir, label_file), 'w') as f:
                f.write(f"{label_value} 0.5 0.5 1.0 1.0\n")

        for file_name in val_files:
            shutil.copy(os.path.join(label_dir, file_name), os.path.join(val_label_dir, file_name))

            label_file = os.path.splitext(file_name)[0] + '.txt'
            with open(os.path.join(val_label_dir, label_file), 'w') as f:
                f.write(f"{label_value} 0.5 0.5 1.0 1.0\n")

    print("Data balanced and split into train and validation sets.")

balance_classes(base_dir, train_dir, val_dir, labels)
```

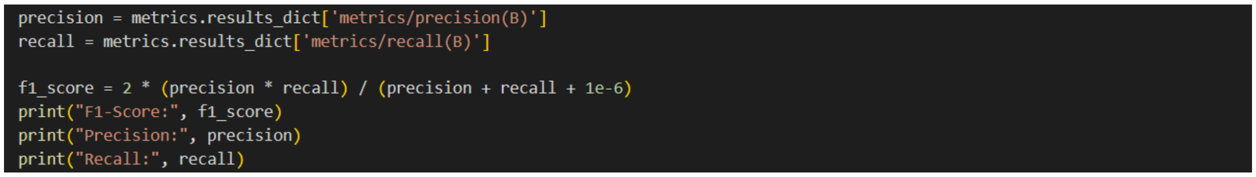
This code is designed to prepare the file structure and label the driver drowsiness dataset. First, the path to the main dataset is defined with the base_dir variable. Then, two folders, train and val, are created to split the data into training and validation sets. If these folders do not already exist, they are created using os.makedirs. The labels dictionary specifies the binary labels for classifying the data: Drowsy with a value of 0 represents the drowsy state, and Non-Drowsy with a value of 1 represents the alert state. This structure is used to prepare the data for model training.


```
train_data_path = './data_split/data.yaml'
with open(train_data_path, 'w') as f:
    f.write(
        f"""
        path: {os.path.abspath('./data_split')}
        train: {os.path.abspath(train_dir)}
        val: {os.path.abspath(val_dir)}
        nc: 2
        names: ['drowsy', 'non_drowsy']
        weights: [3.0, 3.0]
        """
    )

model = YOLO('yolov8n.pt')
model.train(data=train_data_path, epochs=1, imgsz=128, batch=32)
```

In this section, we first create a YAML file named data.yaml that includes the paths related to the training and validation data, the number of classes, and their names. Using os.path.abspath(), we obtain the absolute paths of the directories and write them into the file. Then, using the YOLO model, we load the pre-trained weights (yolov8n.pt) and train the model for one epoch (epochs=1) with an image size of 128 and a batch size of 32. These steps help us train the model for driver drowsiness detection.


```
metrics = model.val()
```

In this line of code, we use the YOLO model that we previously trained to evaluate its performance. By calling the model.val() function, we obtain the metrics related to the model's validation. These metrics typically include measures such as Precision, Recall, mean Average Precision (mAP), and other performance-related criteria for detecting and classifying data. This evaluation helps us assess the quality of the model's training and determine whether the model performs well on the validation data.


```
precision = metrics.results_dict['metrics/precision(B)']
recall = metrics.results_dict['metrics/recall(B)']

f1_score = 2 * (precision * recall) / (precision + recall + 1e-6)
print("F1-Score:", f1_score)
print("Precision:", precision)
print("Recall:", recall)
```

In this part of the code, we extract the Precision and Recall values from the model's evaluation results. First, we obtain the Precision and Recall values from the results dictionary using metrics.results_dict['metrics/precision(B)'] and metrics.results_dict['metrics/recall(B)']. Then, to calculate the F1 score, we use the following formula:

F1-Score = 2 × (Precision × Recall) / (Precision + Recall + 1e-6)

In this formula, the small value 1e-6 is added to prevent division by zero in case both Precision and Recall are zero. Finally, we print the F1-Score, Precision, and Recall values to assess the model's performance in detecting driver drowsiness. These values help us make an accurate evaluation of the model's quality.

Section 3: Results from Running the Model on Datasets

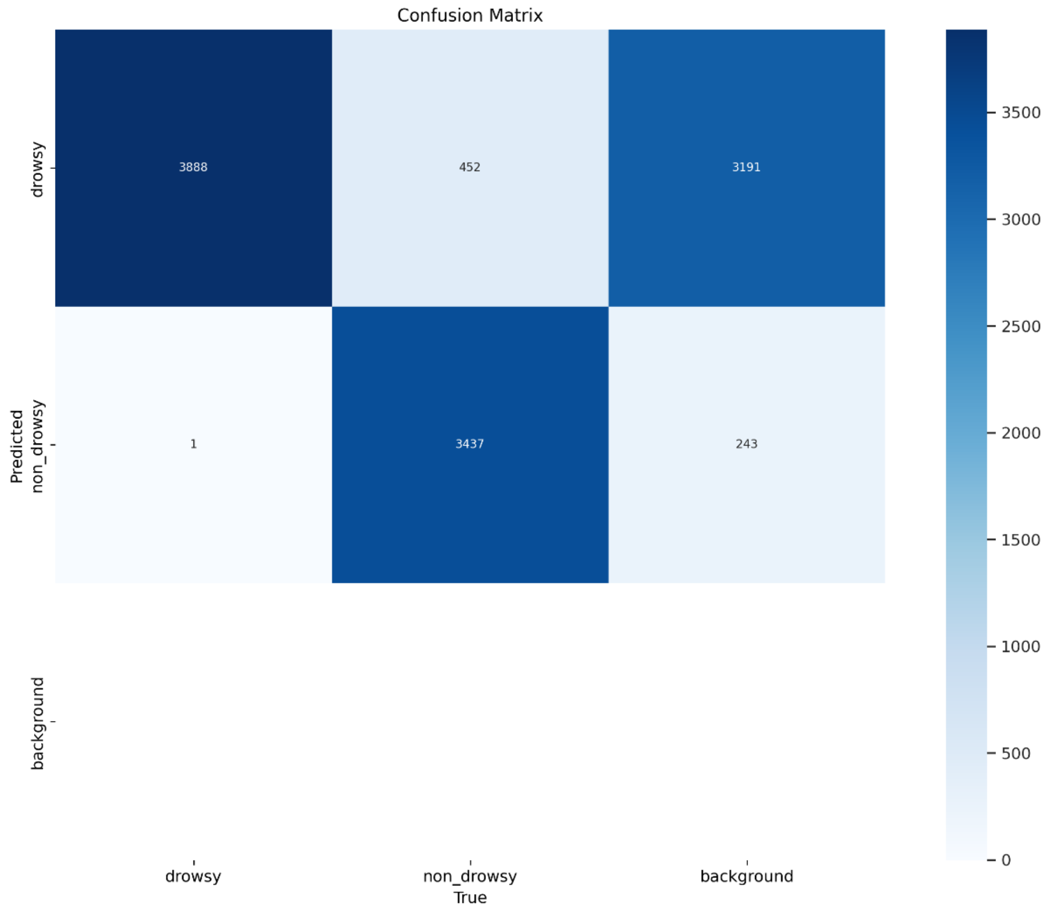
In the table below, the values of F1-score, Precision, and Recall obtained from running the models on the available dataset are displayed. All these values are taken from the test and evaluation section at the end of the models, which can be observed.

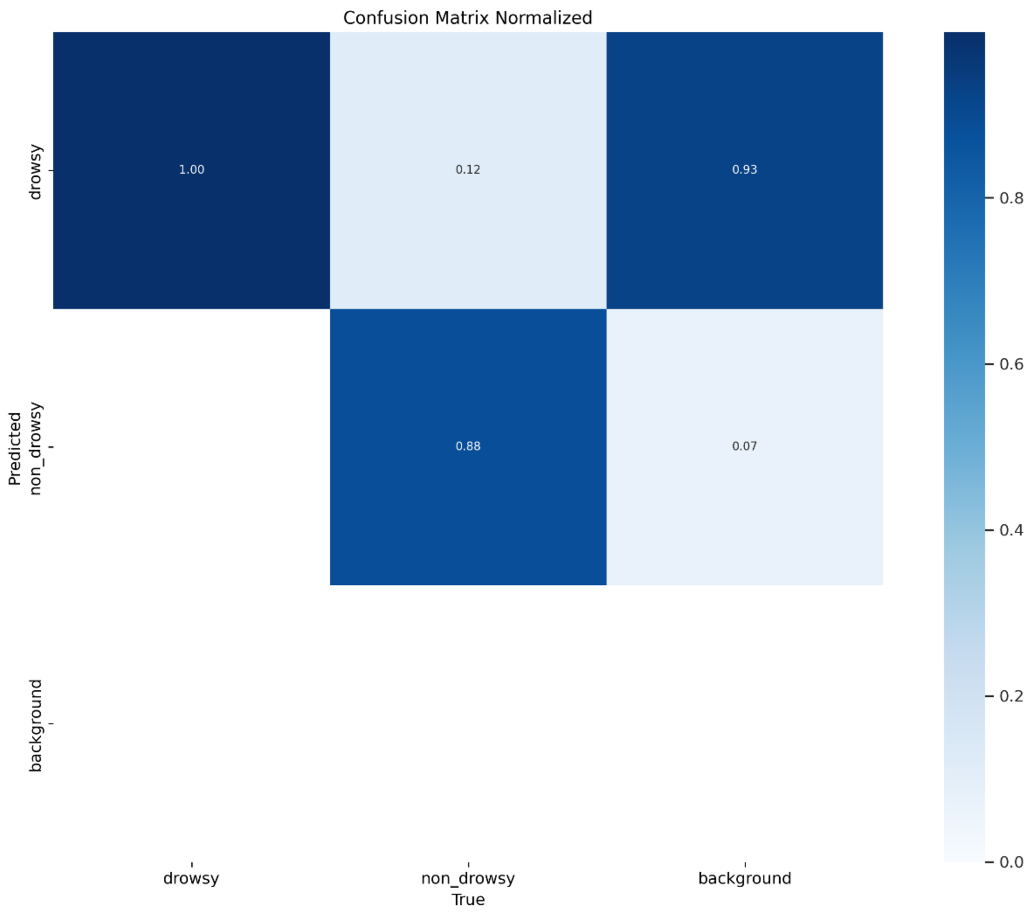
	f1-score	precision	recall
imgsz_128_batch_24	0.956	0.946	0.967
imgsz_128_batch_32	0.932	0.926	0.938
imgsz_224_batch_24	0.980	0.984	0.976
imgsz_224_batch_32	0.877	0.839	0.917

Below are some charts and tables displayed as a result.

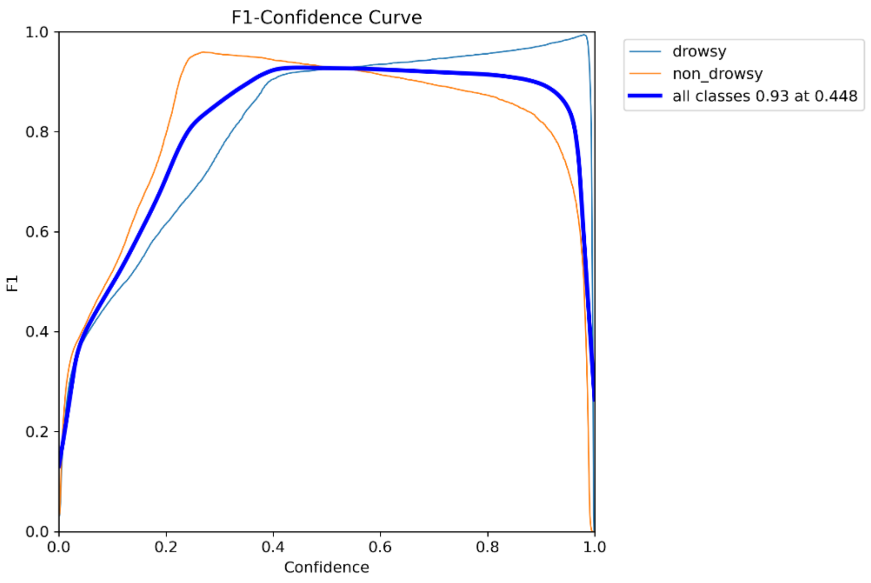
For the model with image size = 128 and batch size = 24:

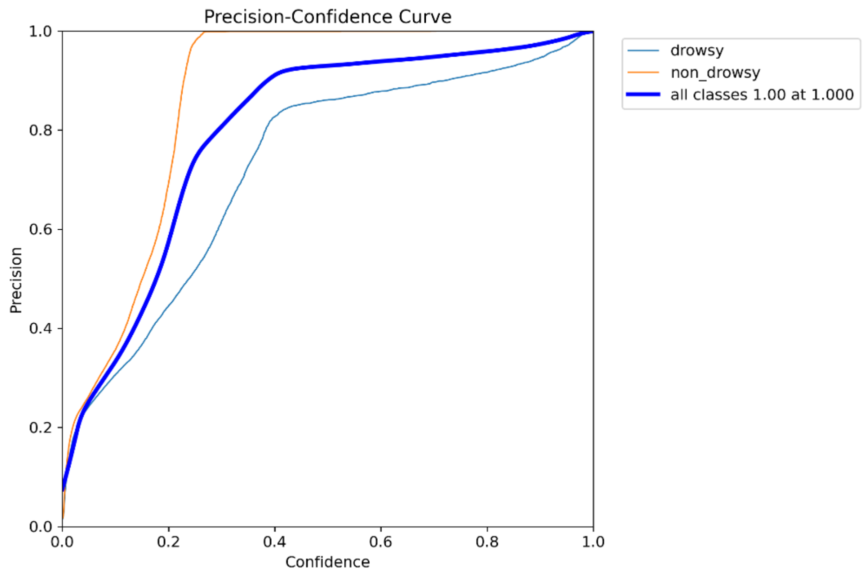
• The confusion matrix along with its normalized version:

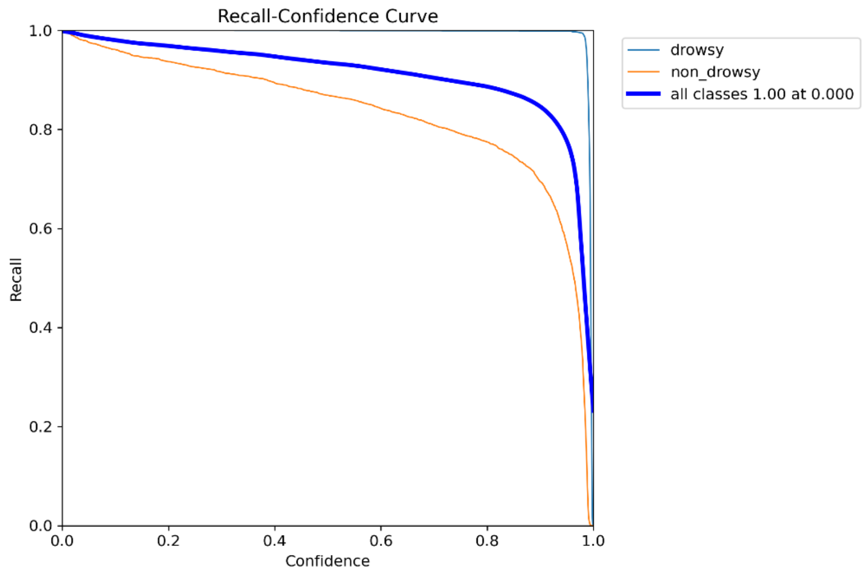




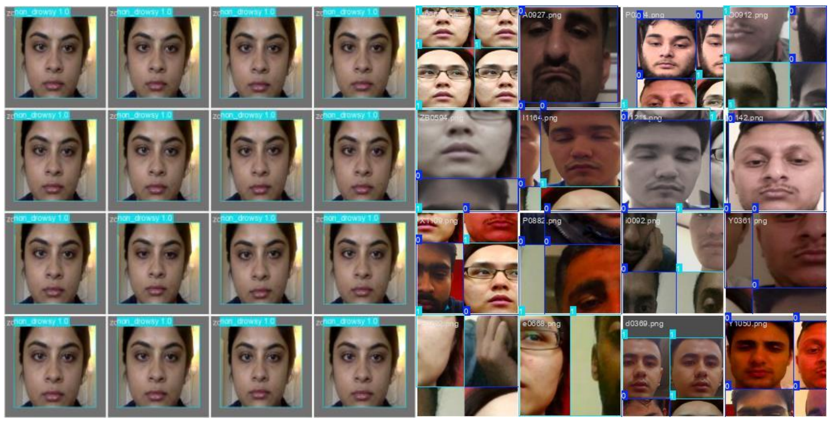
• Curves related to F1-score, Precision, and Recall:







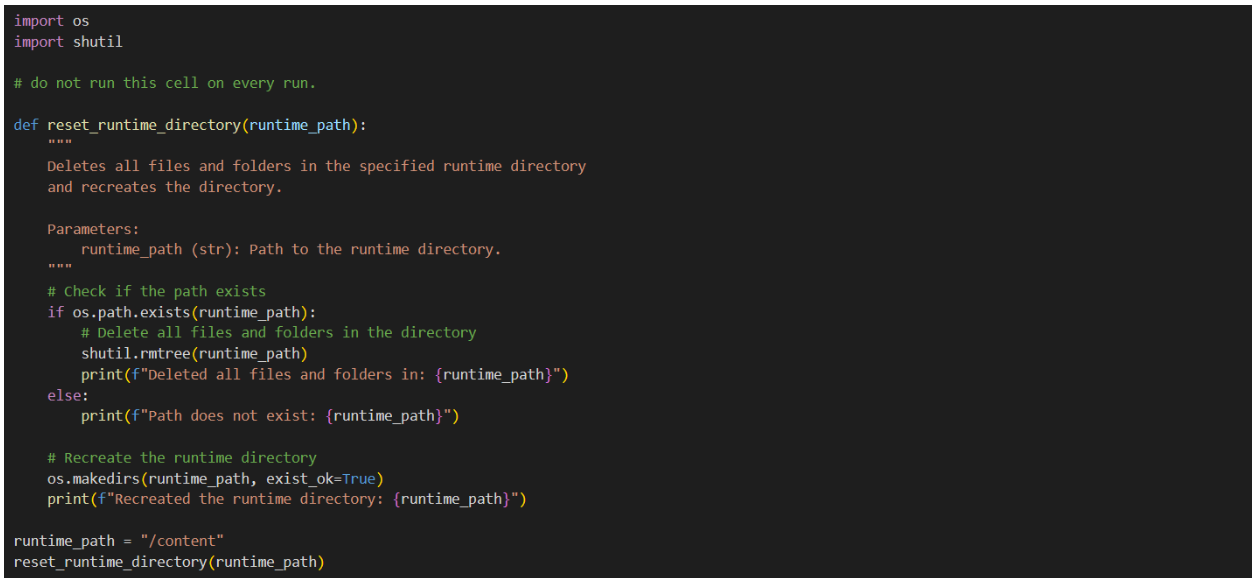
• A sample of the train and validation batches:



For the other models as well, the confusion matrices, corresponding curves, a sample of the train and validation batches, along with other model data, are available in the project repository.

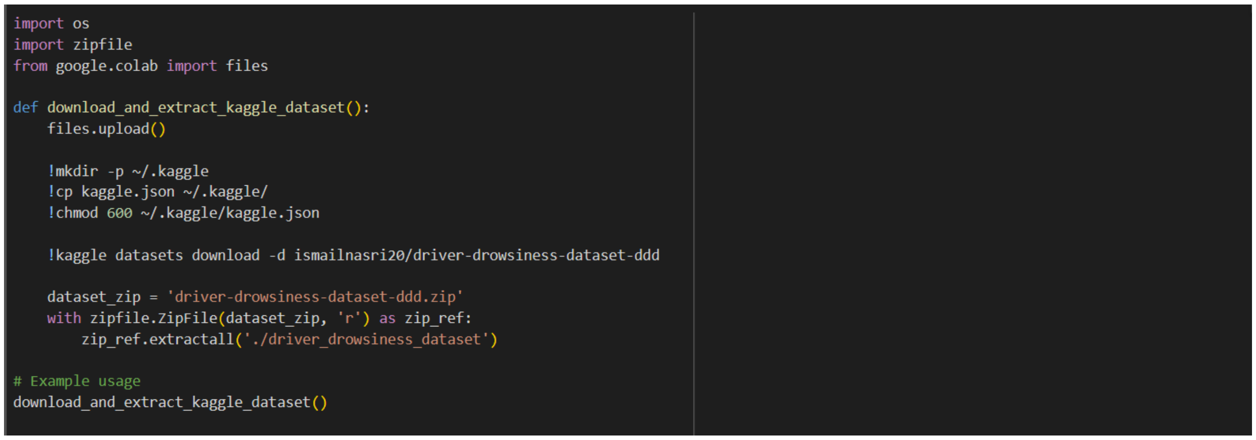
Section 4: Models with Preprocessing Applied to the Data

In this section, we have refactored the code for the models without preprocessing and added the functions for performing preprocessing on the dataset images. As in Section 2, a detailed description of all the functions used is provided.



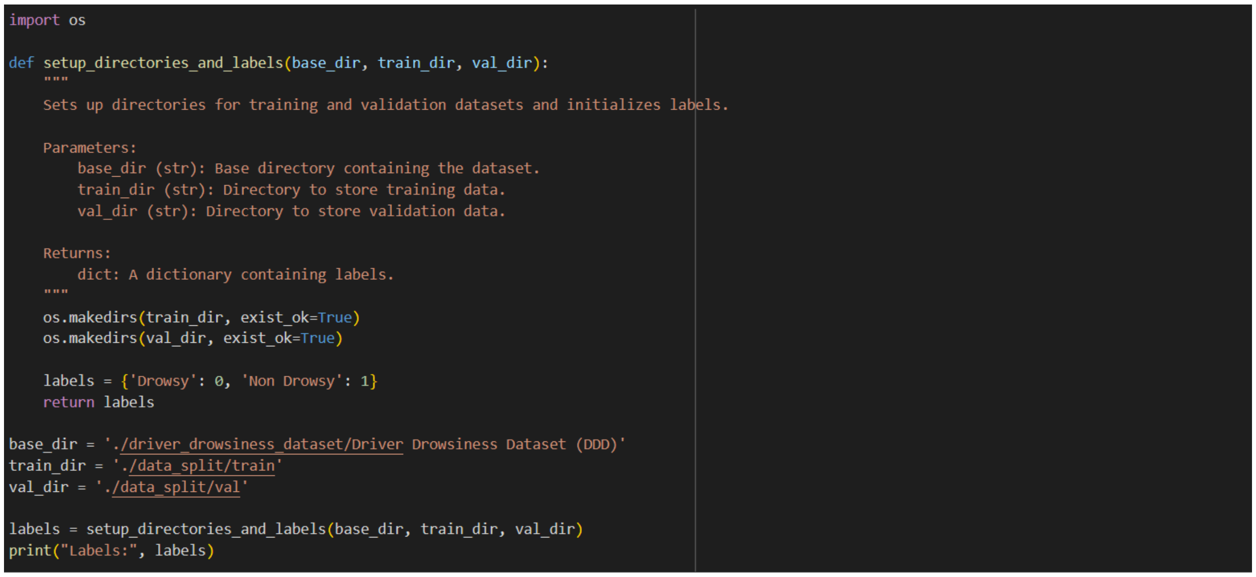
```python
import os
import shutil

# do not run this cell on every run.

def reset_runtime_directory(runtime_path):
    """
    Deletes all files and folders in the specified runtime directory
    and recreates the directory.

    Parameters:
        runtime_path (str): Path to the runtime directory.
    """
    # Check if the path exists
    if os.path.exists(runtime_path):
        # Delete all files and folders in the directory
        shutil.rmtree(runtime_path)
        print(f"Deleted all files and folders in: {runtime_path}")
    else:
        print(f"Path does not exist: {runtime_path}")

    # Recreate the runtime directory
    os.makedirs(runtime_path, exist_ok=True)
    print(f"Recreated the runtime directory: {runtime_path}")

runtime_path = "/content"
reset_runtime_directory(runtime_path)
```

In the code above, we have defined a function called reset_runtime_directory whose task is to clean and rebuild a specified directory. This function first checks whether the target directory (specified in the runtime_path parameter) exists or not; if it exists, it deletes all the files and folders within it using shutil.rmtree. Then, it recreates the directory using os.makedirs and, with the option exist_ok=True, prevents an error in case the directory already exists. Finally, messages are printed to the console to inform the user about the status of the operation. This function is eventually called with the path /content to reset the corresponding directory.



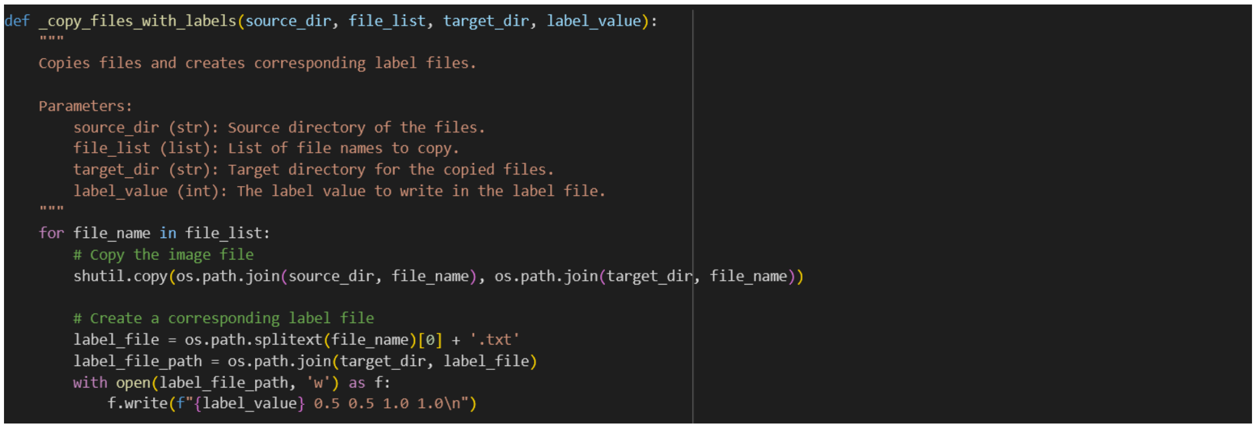
```python
import os
import zipfile
from google.colab import files

def download_and_extract_kaggle_dataset():
    files.upload()

    !mkdir -p ~/.kaggle
    !cp kaggle.json ~/.kaggle/
    !chmod 600 ~/.kaggle/kaggle.json

    !kaggle datasets download -d ismailnasri20/driver-drowsiness-dataset-ddd

    dataset_zip = 'driver-drowsiness-dataset-ddd.zip'
    with zipfile.ZipFile(dataset_zip, 'r') as zip_ref:
        zip_ref.extractall('./driver_drowsiness_dataset')

# Example usage
download_and_extract_kaggle_dataset()
```

In the code above, we have defined a function called download_and_extract_kaggle_dataset designed to download and extract a dataset from Kaggle. First, we prompt the user to upload the kaggle.json file to provide authentication information for accessing the Kaggle API. Then, a directory named .kaggle is created in the home directory, and the authentication file is moved there, with the necessary permissions set. After that, the desired dataset is downloaded using the Kaggle command, and finally, its ZIP file is extracted, placing its contents in the ./driver_drowsiness_dataset directory. By calling this function, all the steps of downloading and extracting are performed automatically.



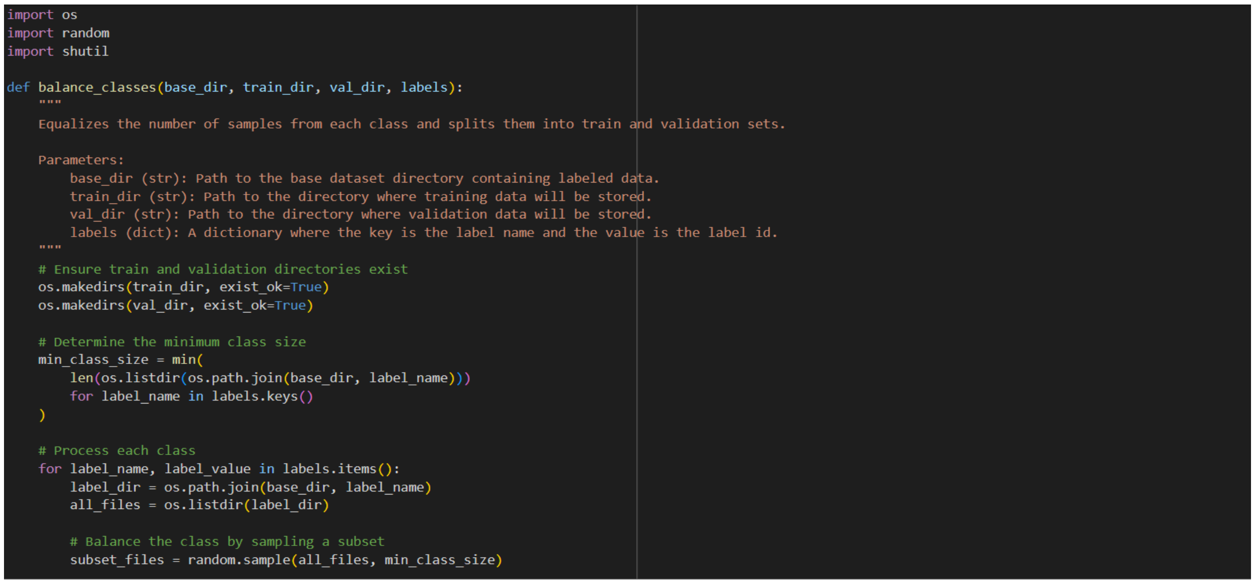
```python
import os

def setup_directories_and_labels(base_dir, train_dir, val_dir):
    """
    Sets up directories for training and validation datasets and initializes labels.

    Parameters:
        base_dir (str): Base directory containing the dataset.
        train_dir (str): Directory to store training data.
        val_dir (str): Directory to store validation data.

    Returns:
        dict: A dictionary containing labels.
    """
    os.makedirs(train_dir, exist_ok=True)
    os.makedirs(val_dir, exist_ok=True)

    labels = {'Drowsy': 0, 'Non Drowsy': 1}
    return labels

base_dir = './driver_drowsiness_dataset/Driver Drowsiness Dataset (DDD)'
train_dir = './data_split/train'
val_dir = './data_split/val'

labels = setup_directories_and_labels(base_dir, train_dir, val_dir)
print("Labels:", labels)
```

In the code above, we have defined a function called setup_directories_and_labels designed to set up the directories for the training and validation datasets, as well as create the labels. First, we create the directories for the training and validation data using os.makedirs, and prevent them from being created again if they already exist. Then, we define a dictionary for the labels, which includes the labels "Drowsy" and "Non-Drowsy," assigning the numeric values 0 and 1 to them. Finally, by calling this function and specifying the corresponding paths, we retrieve the labels and print them.

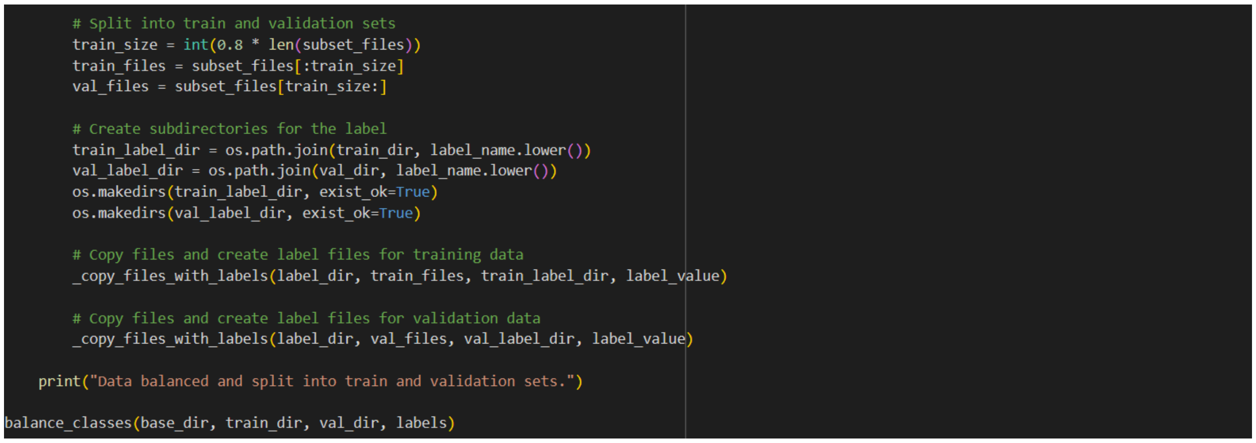


```python
def _copy_files_with_labels(source_dir, file_list, target_dir, label_value):
    """
    Copies files and creates corresponding label files.

    Parameters:
        source_dir (str): Source directory of the files.
        file_list (list): List of file names to copy.
        target_dir (str): Target directory for the copied files.
        label_value (int): The label value to write in the label file.
    """
    for file_name in file_list:
        # Copy the image file
        shutil.copy(os.path.join(source_dir, file_name), os.path.join(target_dir, file_name))

        # Create a corresponding label file
        label_file = os.path.splitext(file_name)[0] + '.txt'
        label_file_path = os.path.join(target_dir, label_file)
        with open(label_file_path, 'w') as f:
            f.write(f"{label_value} 0.5 0.5 1.0 1.0\n")
```

In the code above, we have defined a function called _copy_files_with_labels whose task is to copy the files and create the corresponding label files. This function first receives a source directory (source_dir), a list of file names (file_list), a target directory (target_dir), and a label value (label_value) as inputs. Then, for each file name in the list, it copies the image file from the source directory to the target directory. Additionally, it creates a label text file with the same name as the image file (with a .txt extension) and stores the label value along with constant values for the coordinates in it. This helps us store the image files and their corresponding labels in an organized and synchronized manner in the target directory.

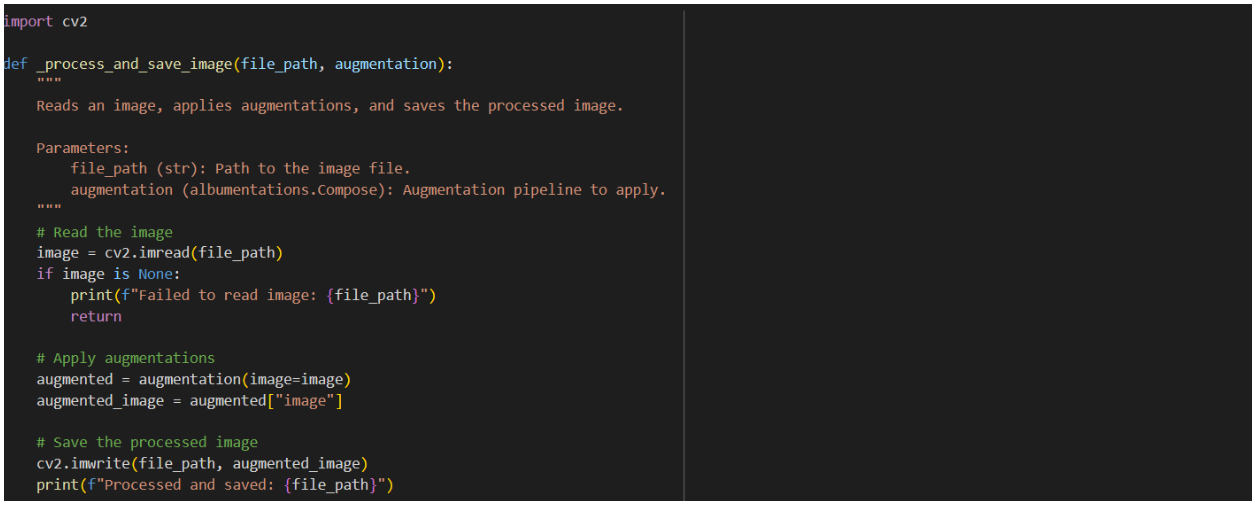


```python
import os
import random
import shutil

def balance_classes(base_dir, train_dir, val_dir, labels):
    """
    Equalizes the number of samples from each class and splits them into train and validation sets.

    Parameters:
        base_dir (str): Path to the base dataset directory containing labeled data.
        train_dir (str): Path to the directory where training data will be stored.
        val_dir (str): Path to the directory where validation data will be stored.
        labels (dict): A dictionary where the key is the label name and the value is the label id.
    """
    # Ensure train and validation directories exist
    os.makedirs(train_dir, exist_ok=True)
    os.makedirs(val_dir, exist_ok=True)

    # Determine the minimum class size
    min_class_size = min(
        len(os.listdir(os.path.join(base_dir, label_name)))
        for label_name in labels.keys()
    )

    # Process each class
    for label_name, label_value in labels.items():
        label_dir = os.path.join(base_dir, label_name)
        all_files = os.listdir(label_dir)

        # Balance the class by sampling a subset
        subset_files = random.sample(all_files, min_class_size)
```



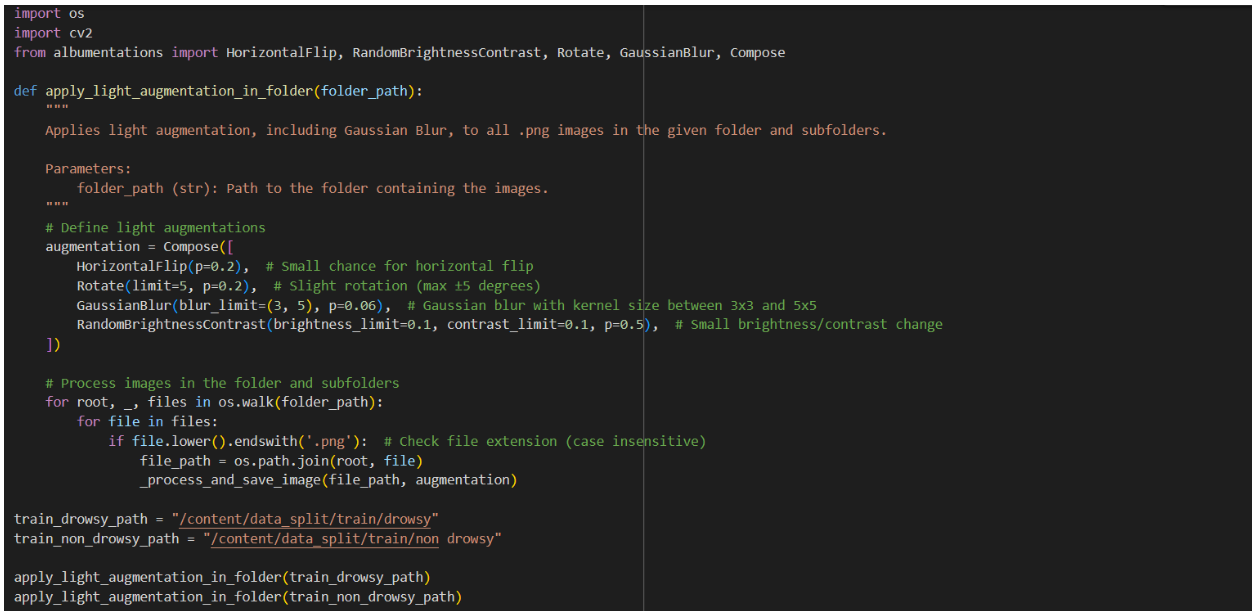
```python
        # Split into train and validation sets
        train_size = int(0.8 * len(subset_files))
        train_files = subset_files[:train_size]
        val_files = subset_files[train_size:]

        # Create subdirectories for the label
        train_label_dir = os.path.join(train_dir, label_name.lower())
        val_label_dir = os.path.join(val_dir, label_name.lower())
        os.makedirs(train_label_dir, exist_ok=True)
        os.makedirs(val_label_dir, exist_ok=True)

        # Copy files and create label files for training data
        _copy_files_with_labels(label_dir, train_files, train_label_dir, label_value)

        # Copy files and create label files for validation data
        _copy_files_with_labels(label_dir, val_files, val_label_dir, label_value)

    print("Data balanced and split into train and validation sets.")

balance_classes(base_dir, train_dir, val_dir, labels)
```

In the code above, we have defined a function called balance_classes whose goal is to equalize the number of samples from each class and split them into training and validation sets. First, we ensure that the directories for the training and validation data exist. Then, by checking the size of the smallest class, we determine the number of samples for each class. For each class, we list the available files and select a random subset equal in size to the smallest class. This subset is then divided into training and validation sets, with 80% allocated to the training data and 20% to the validation data. After that, the corresponding directories for each class in the training and validation sets are created, and using the _copy_files_with_labels function, the files and their corresponding label files are copied to these directories. Finally, a message is printed indicating that the data has been balanced and split.



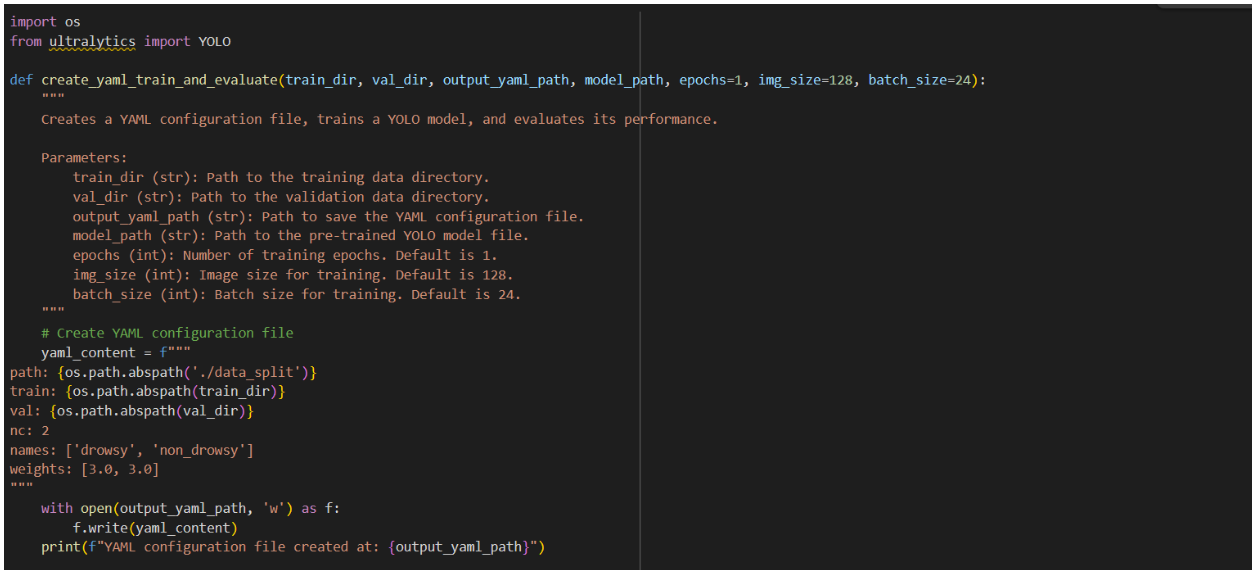
```python
import cv2

def _process_and_save_image(file_path, augmentation):
    """
    Reads an image, applies augmentations, and saves the processed image.

    Parameters:
        file_path (str): Path to the image file.
        augmentation (albumentations.Compose): Augmentation pipeline to apply.
    """
    # Read the image
    image = cv2.imread(file_path)
    if image is None:
        print(f"Failed to read image: {file_path}")
        return

    # Apply augmentations
    augmented = augmentation(image=image)
    augmented_image = augmented["image"]

    # Save the processed image
    cv2.imwrite(file_path, augmented_image)
    print(f"Processed and saved: {file_path}")
```

In the code above, we have defined a function called _process_and_save_image whose task is to read an image, apply specified augmentations to it, and save the processed image. First, we read the image from the specified path using OpenCV, and if reading the image fails, we print an error message. Then, we apply the augmentations specified in the augmentation parameter to the image and save the modified image. Finally, after successfully saving the processed image, we print a message indicating that the image has been processed and saved.

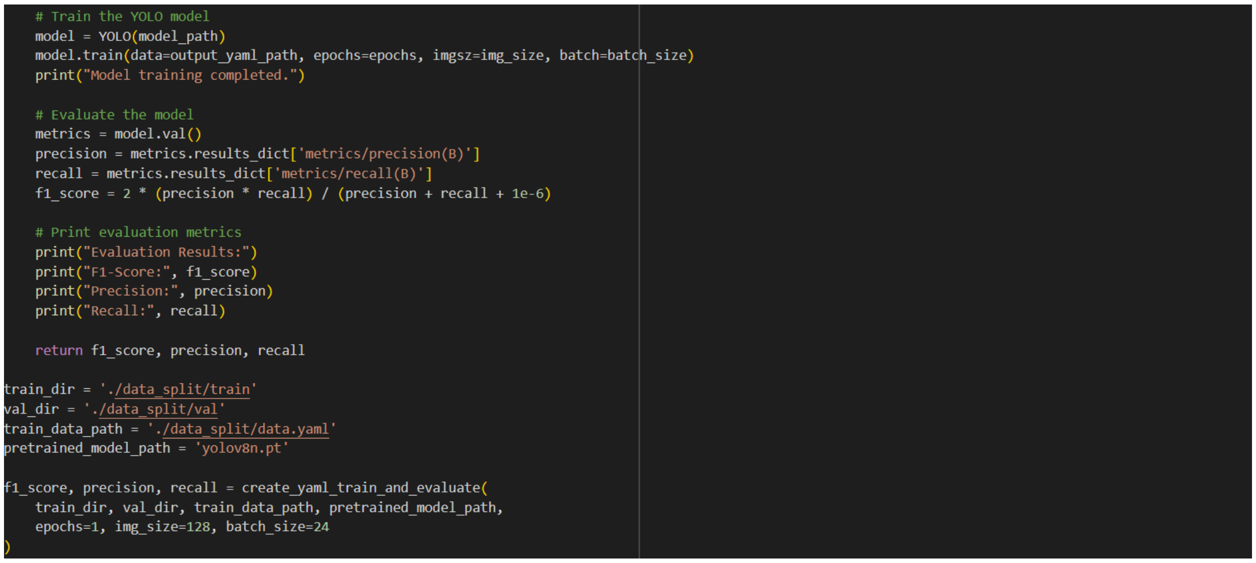


```python
import os
import cv2
from albumentations import HorizontalFlip, RandomBrightnessContrast, Rotate, GaussianBlur, Compose

def apply_light_augmentation_in_folder(folder_path):
    """
    Applies light augmentation, including Gaussian Blur, to all .png images in the given folder and subfolders.

    Parameters:
        folder_path (str): Path to the folder containing the images.
    """
    # Define light augmentations
    augmentation = Compose([
        HorizontalFlip(p=0.2),  # Small chance for horizontal flip
        Rotate(limit=5, p=0.2),  # Slight rotation (max ±5 degrees)
        GaussianBlur(blur_limit=(3, 5), p=0.06),  # Gaussian blur with kernel size between 3x3 and 5x5
        RandomBrightnessContrast(brightness_limit=0.1, contrast_limit=0.1, p=0.5),  # Small brightness/contrast change
    ])

    # Process images in the folder and subfolders
    for root, _, files in os.walk(folder_path):
        for file in files:
            if file.lower().endswith('.png'):  # Check file extension (case insensitive)
                file_path = os.path.join(root, file)
                _process_and_save_image(file_path, augmentation)

train_drowsy_path = "/content/data_split/train/drowsy"
train_non_drowsy_path = "/content/data_split/train/non drowsy"

apply_light_augmentation_in_folder(train_drowsy_path)
apply_light_augmentation_in_folder(train_non_drowsy_path)
```

In the code above, we have defined a function called apply_light_augmentation_in_folder that applies light augmentations to all PNG images in a directory and its subdirectories. First, we define a set of augmentations, including minor rotations, Gaussian blur, changes in brightness and contrast, and horizontal flipping. Then, using the os.walk function, we recursively access all the files in the specified directory, and for each file with a PNG extension, we call the _process_and_save_image function to apply the augmentations and save the processed image. Finally, we call this function for two separate directories containing "Drowsy" and "Non-Drowsy" images to apply the augmentations simultaneously to both datasets.



```python
import os
from ultralytics import YOLO

def create_yaml_train_and_evaluate(train_dir, val_dir, output_yaml_path, model_path, epochs=1, img_size=128, batch_size=24):
    """
    Creates a YAML configuration file, trains a YOLO model, and evaluates its performance.

    Parameters:
        train_dir (str): Path to the training data directory.
        val_dir (str): Path to the validation data directory.
        output_yaml_path (str): Path to save the YAML configuration file.
        model_path (str): Path to the pre-trained YOLO model file.
        epochs (int): Number of training epochs. Default is 1.
        img_size (int): Image size for training. Default is 128.
        batch_size (int): Batch size for training. Default is 24.
    """
    # Create YAML configuration file
    yaml_content = f"""
path: {os.path.abspath('./data_split')}
train: {os.path.abspath(train_dir)}
val: {os.path.abspath(val_dir)}
nc: 2
names: ['drowsy', 'non_drowsy']
weights: [3.0, 3.0]
"""
    with open(output_yaml_path, 'w') as f:
        f.write(yaml_content)
    print(f"YAML configuration file created at: {output_yaml_path}")
```



```python
    # Train the YOLO model
    model = YOLO(model_path)
    model.train(data=output_yaml_path, epochs=epochs, imgsz=img_size, batch=batch_size)
    print("Model training completed.")

    # Evaluate the model
    metrics = model.val()
    precision = metrics.results_dict['metrics/precision(B)']
    recall = metrics.results_dict['metrics/recall(B)']
    f1_score = 2 * (precision * recall) / (precision + recall + 1e-6)

    # Print evaluation metrics
    print("Evaluation Results:")
    print("F1-Score:", f1_score)
    print("Precision:", precision)
    print("Recall:", recall)

    return f1_score, precision, recall

train_dir = './data_split/train'
val_dir = './data_split/val'
train_data_path = './data_split/data.yaml'
pretrained_model_path = 'yolov8n.pt'

f1_score, precision, recall = create_yaml_train_and_evaluate(
    train_dir, val_dir, train_data_path, pretrained_model_path,
    epochs=1, img_size=128, batch_size=24
)
```

In the code above, we have defined a function called create_yaml_train_and_evaluate whose task is to create a YAML configuration file, train a YOLO model, and evaluate its performance. First, we set the content of the YAML file with the paths to the training and validation directories, the number of classes, and their names, and then save this content to the specified path. After that, we load the YOLO model using the path to the pre-trained model and train it with the data specified in the YAML file. Once training is complete, we evaluate the model and calculate the Precision, Recall, and F1-score metrics. Finally, we print the evaluation results, including the F1-score, Precision, and Recall, and return these values as the output of the function. We call this function for the specified training and validation directories and use the pre-trained YOLO model.

Section 5: Results from Running the Model on Datasets with Preprocessing

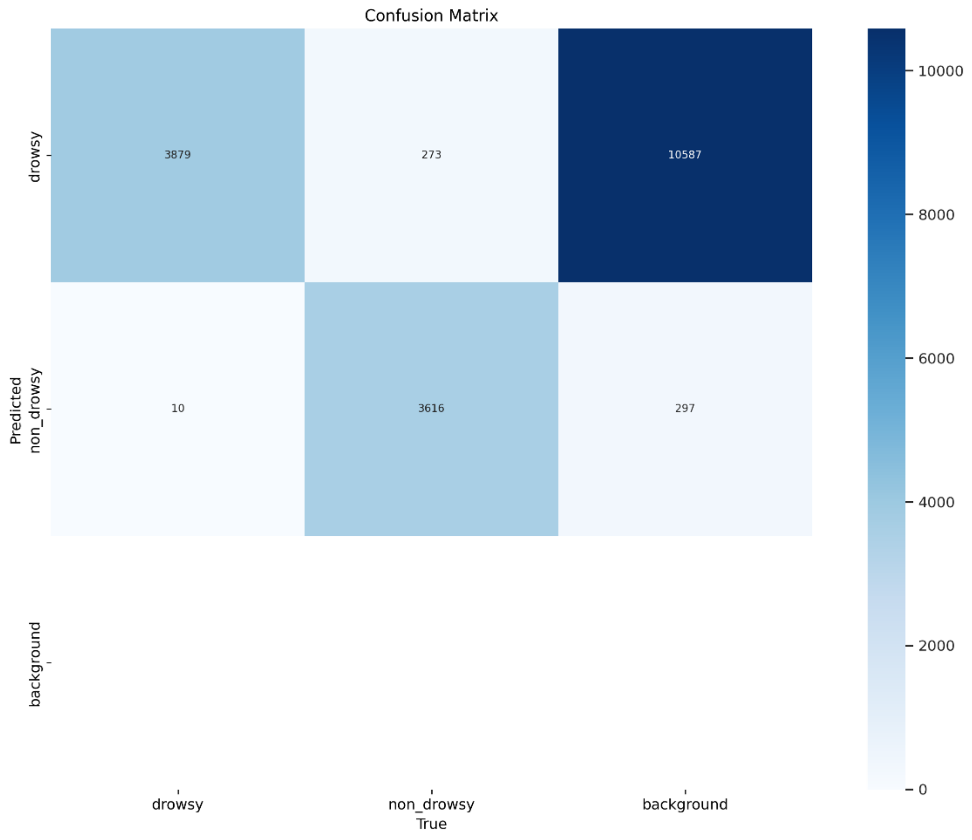
In the table below, the values of F1-score, Precision, and Recall obtained from running the models on the available dataset are displayed. All these values are taken from the test and evaluation section at the end of the models, which can be observed.

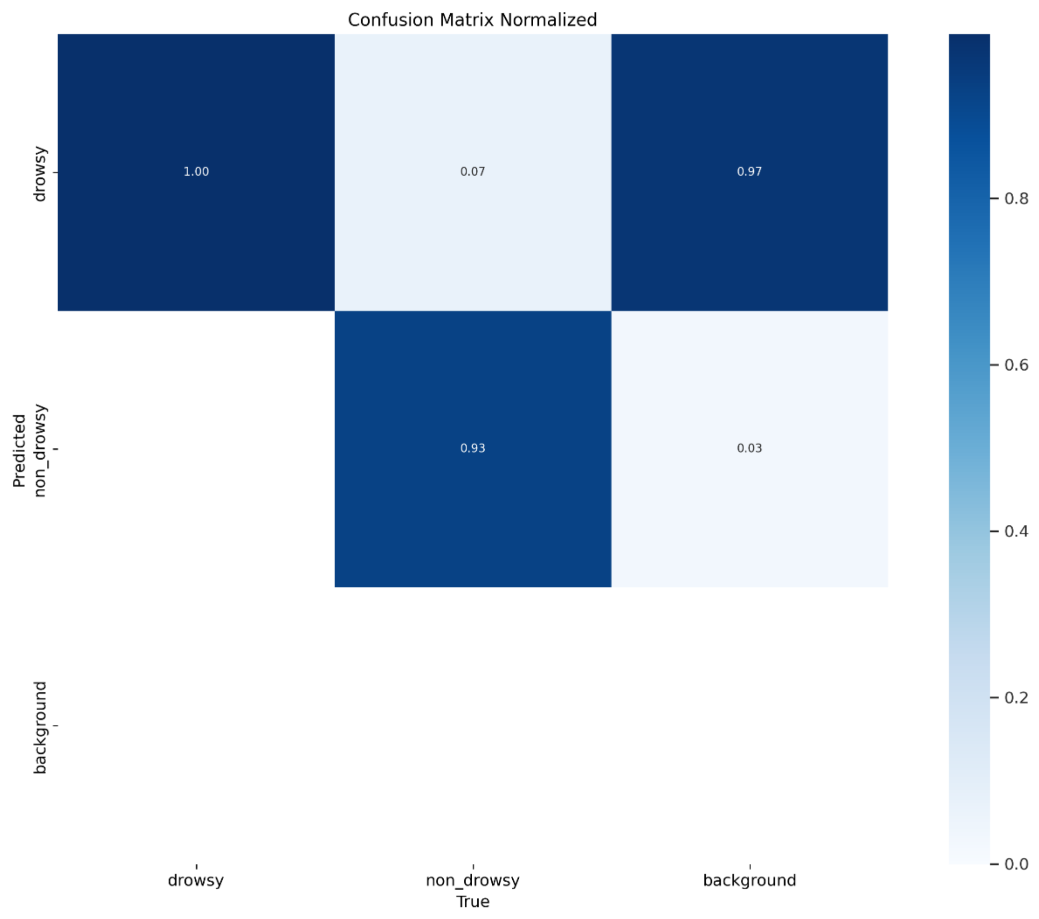
	f1-score	precision	recall
imgsz_128_batch_24	0.985	0.988	0.981
imgsz_128_batch_32	0.969	0.971	0.966
imgsz_224_batch_24	0.991	0.993	0.990
imgsz_224_batch_32	0.934	0.949	0.919

Below are some charts and tables displayed as a result.

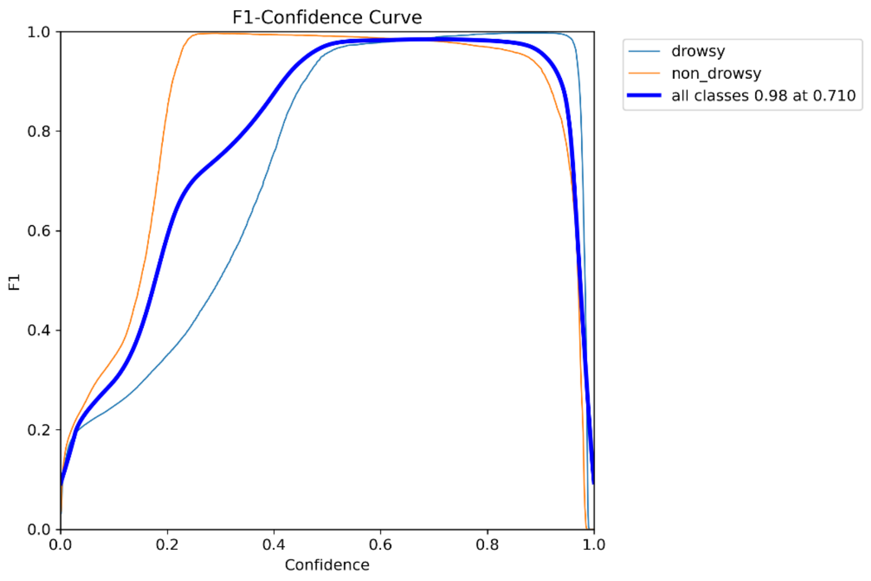
For the model with image size = 128 and batch size = 24:

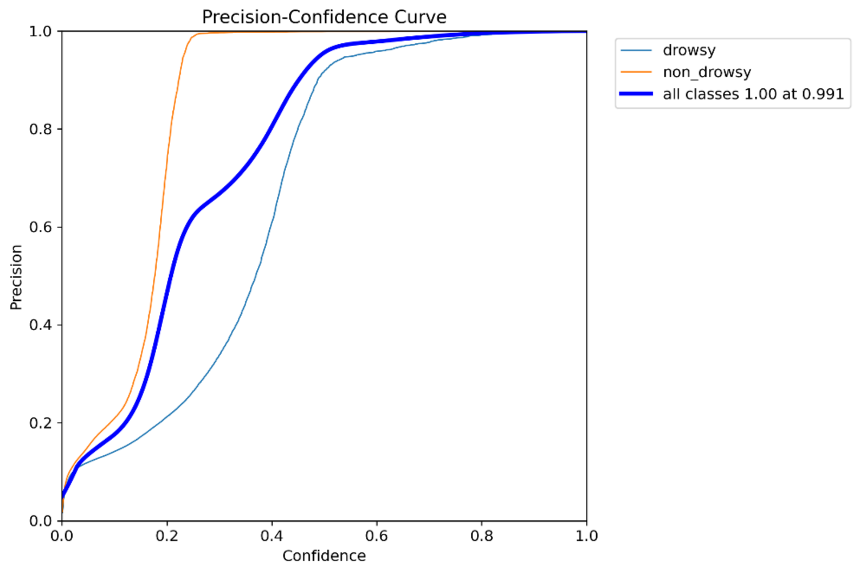
• The confusion matrix along with its normalized version:

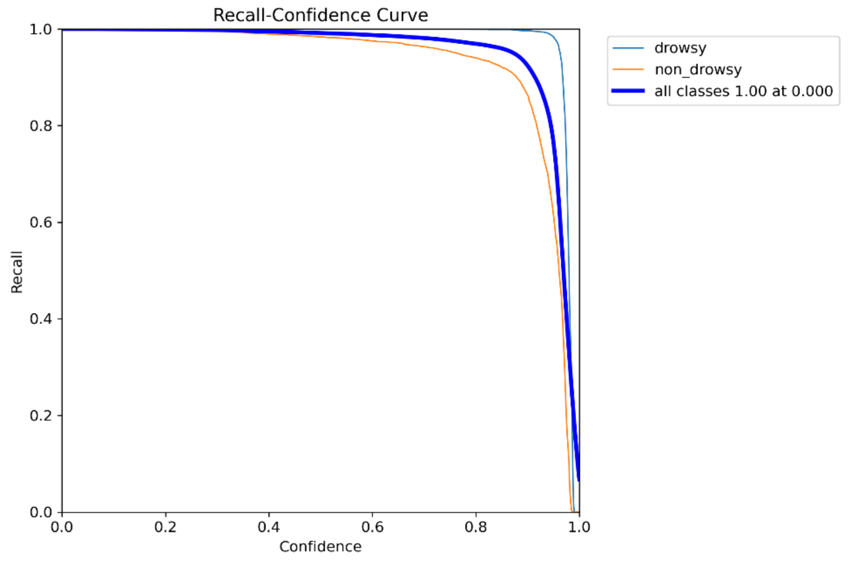




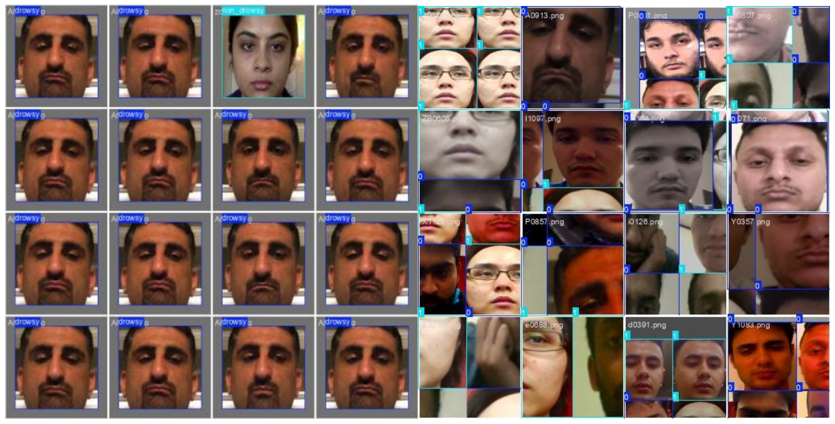
• Curves related to F1-score, Precision, and Recall:







• A sample of the train and validation batches:

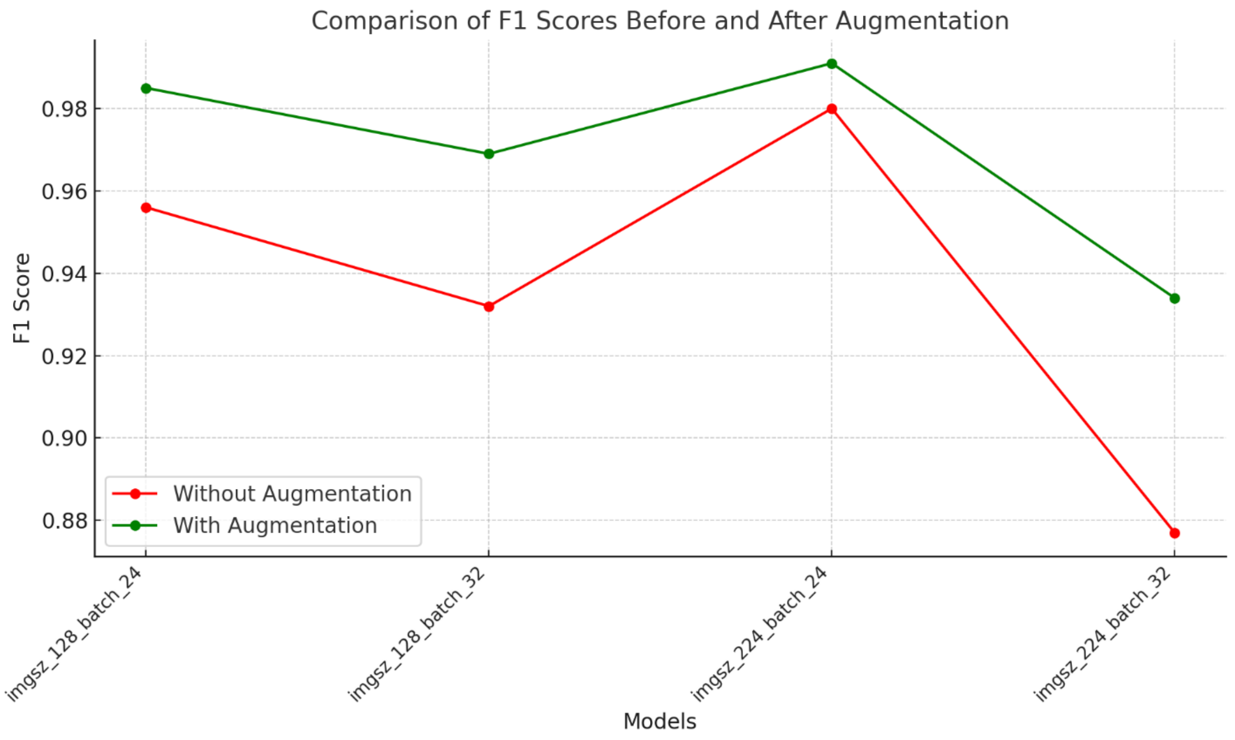


For the other models as well, the confusion matrices, corresponding curves, a sample of the train and validation batches, along with other model data, are available in the project repository.

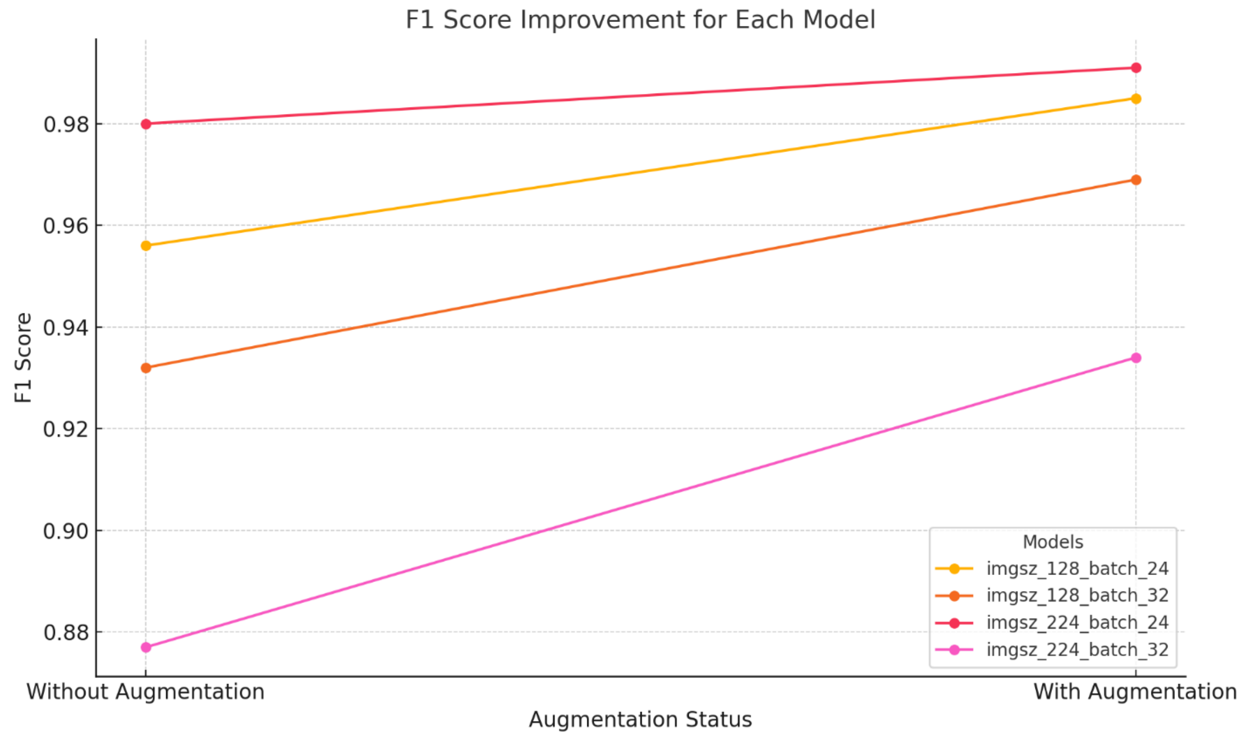
Section 6: Comparison of Results and Progress

	F1_score without_augmentation	F1_score with_augmentation	Amount of increase
imgsz_128_batch_24	0.956	0.985	0.029
imgsz_128_batch_32	0.932	0.969	0.037
imgsz_224_batch_24	0.980	0.991	0.011
imgsz_224_batch_32	0.877	0.934	0.057

As can be seen, the F1-score value increased by 2.9% in the first model, 3.7% in the second model, 1.1% in the third model, and 5.7% in the fourth model.
In the chart below, the progress trend is displayed:



Additionally, in the chart below, the individual progress of each model is displayed:



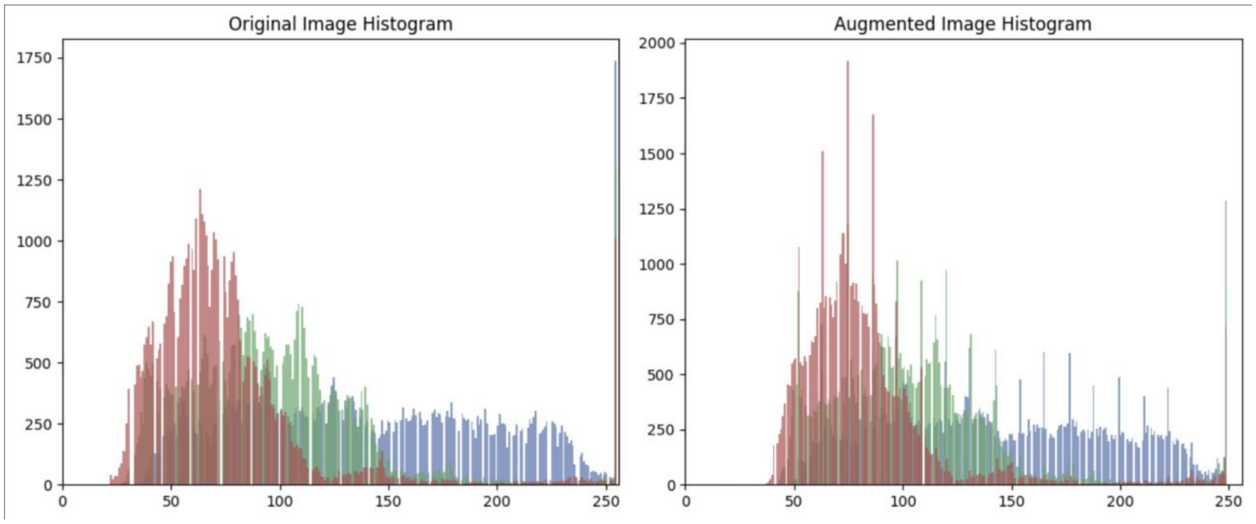
In the table below, the changes in Precision are displayed:

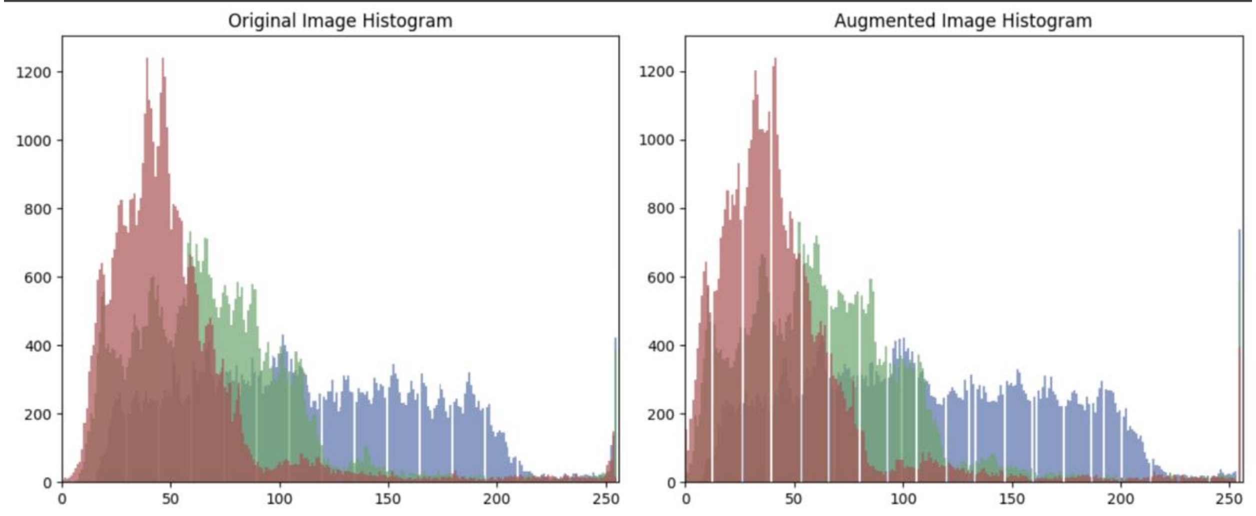
	Precision without_augmentation	Precision with_augmentation	Amount of increase
imgsz_128_batch_24	0.946	0.988	0.042
imgsz_128_batch_32	0.926	0.971	0.045
imgsz_224_batch_24	0.984	0.993	0.009
imgsz_224_batch_32	0.839	0.949	0.011

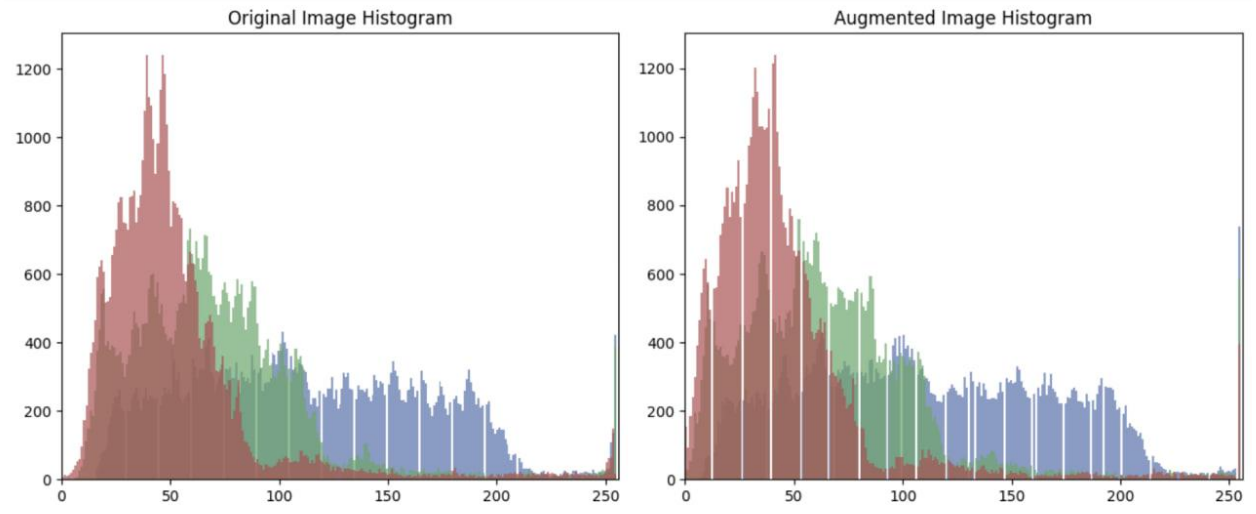
In the table below, the changes in Recall are displayed:

	recall without_augmentation	recall with_augmentation	Amount of increase
imgsz_128_batch_24	0.967	0.981	0.014
imgsz_128_batch_32	0.938	0.966	0.028
imgsz_224_batch_24	0.976	0.990	0.014
imgsz_224_batch_32	0.917	0.919	0.002

Additionally, the following histograms have been randomly generated from images, before and after preprocessing:


Original Image Histogram
Augmented Image Histogram


Original Image Histogram
Augmented Image Histogram


Original Image Histogram
Augmented Image Histogram