data_preprocessing

November 1, 2024

1 Getting Dataset Information

Dataset Link : https://www.kaggle.com/datasets/ismailnasri20/driver-drowsiness-dataset-ddd/data

[42]:
```
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns
import cv2
import os
import sys
import gc
import time
from tqdm import tqdm
import uuid

import sklearn
from sklearn.model_selection import train_test_split
from tensorflow.keras.preprocessing.image import ImageDataGenerator
from tensorflow.keras.models import Sequential
from tensorflow.keras.utils import img_to_array, load_img
```

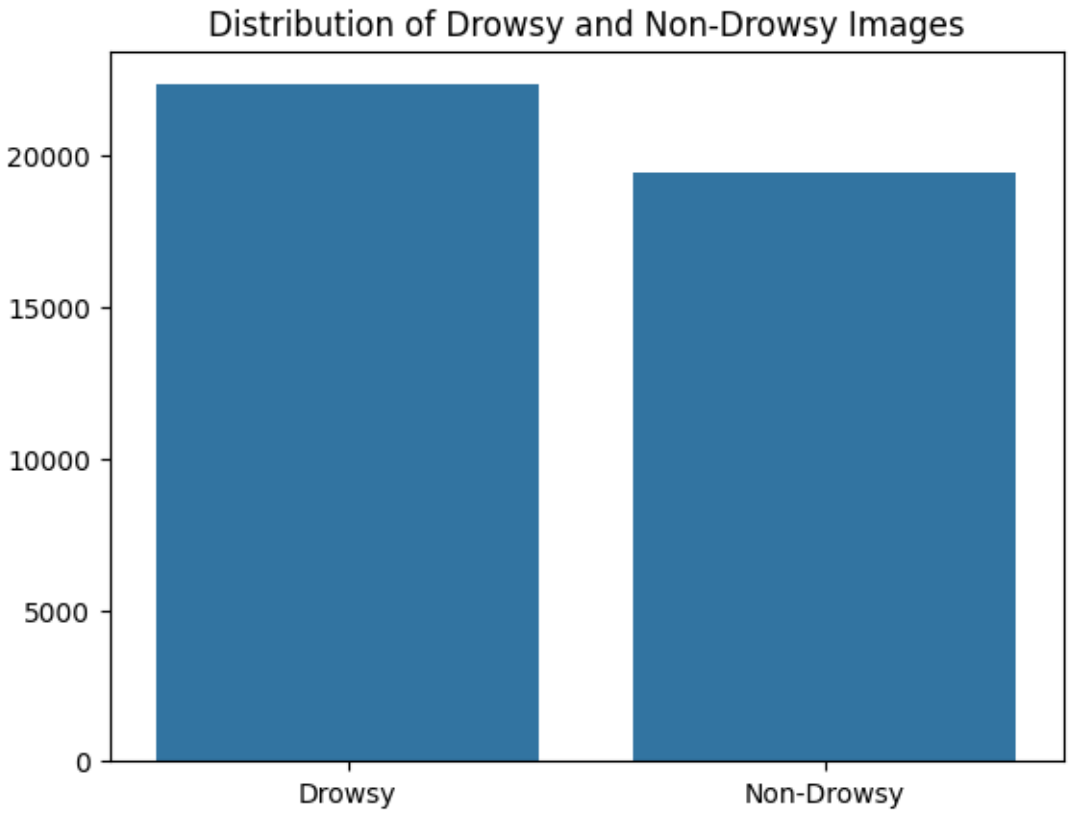
[49]:
```
### General Helper Functions ###

# Counts the number of files within the folder
def count_images_in_folders(base_dir: str, sub_folders: list[str]) ->␣
↪tuple[int]:
    """
    Counts and prints the number of images in each class folder within the base␣
↪directory.

    Parameters:
        base_dir (str): Path to the main directory containing 'Drowsy' and 'Non␣
↪Drowsy' folders.
    """

    out = []
```

```
    for folder in sub_folders:
        dir = os.path.join(base_dir, folder)

        # Count images in each folder
        count = len([f for f in os.listdir(dir) if os.path.isfile(os.path.
 ↪join(dir, f))])
        out.append(count)

        # Print the count
        print(f"Number of images in {folder} folder: {count}")

    return out

# Consolidate image paths from a given directory into a list
def consolidate_image_paths(input_path : str, subfolder_name: str = "") ->␣
 ↪list[str]:
    return [os.path.join(input_path, subfolder_name, p) for p in os.listdir(os.
 ↪path.join(input_path, subfolder_name))]

# Map image paths to labels to a dictionary
def map_image_paths_to_labels(image_paths: list[str], label: int) -> dict:
    return {p: label for p in image_paths}
```

[33]:
```
# Base path for the dataset
base_path = "../Datasets/Dataset_2"
base_path_original = os.path.join(base_path, "Original")

### Dataset 2 Initialisation ###

# Adds the relative paths of all the images in the dataset
drowsy_paths = consolidate_image_paths(base_path_original, "Drowsy")
non_drowsy_paths = consolidate_image_paths(base_path_original, "Non Drowsy")

# Combining all the paths
all_paths = drowsy_paths + non_drowsy_paths

# Mapping the image paths to their respective labels
drowsy_labels = map_image_paths_to_labels(drowsy_paths, 1)
non_drowsy_labels = map_image_paths_to_labels(non_drowsy_paths, 0)

# Combining all the labels
all_labels = [drowsy_labels.get(path, non_drowsy_labels.get(path)) for path in␣
 ↪all_paths]

# Verify lengths
print(f"Total Number of Images: {len(all_paths)}")
```

```
print(f"Total Number of Labels: {len(all_labels)}")

# Find difference between no. of drowsy and non-drowsy images
print(f"Difference between Drowsy and Non-Drowsy: {len(drowsy_paths) -␣
 ↪len(non_drowsy_paths)}")

# Displaying the distribution of drowsy and non-drowsy images
bar_distribution = sns.barplot(x = ["Drowsy", "Non-Drowsy"], y =␣
 ↪[len(drowsy_paths), len(non_drowsy_paths)])
plt.title("Distribution of Drowsy and Non-Drowsy Images")
plt.show()
```

```
Total Number of Images: 41793
Total Number of Labels: 41793
Difference between Drowsy and Non-Drowsy: 2903
```



1.0.1 Insights from Dataset 2

- There are 2903 drowsy images than non-drowsy images.
- The dataset is imbalanced.
- To balance the dataset, we can consider several techniques:

  - Oversampling
  - Undersampling
  - Data Augmentation

2 Data Preprocessing

2.0.1 `Steps:`

1. Image Resizing
2. Data Splitting
3. Reshuffling
4. Undersampling (Majority Class)
5. Image Annotation
6. Data Augmentation (for training data)
7. Data Normalization

2.0.2 Preprocessing Steps Methodology

1. **Image Resizing**:
   - The images should be resized first to ensure all images are of the same dimensions.
   - The images are resized to `224x224` pixels.
2. **Data Splitting**:
   - Dataset should be split before any form of augmentation or sampling to ensure that the model is evaluated on unseen data.
   - Augmented data can be spilt into the testing and validation sets otherwise.
   - The dataset is split into `70%` training, `15%` validation and `15%` testing sets.
3. **Reshuffling**:
   - The dataset is reshuffled to ensure that the data is not ordered in any way.
   - This helps to prevent the model from learning any patterns in the data that may not be present in real-world scenarios.
4. **Undersampling**:
   - The majority class is undersampled to the number of images in the minority class.
5. **Image Annotation**
   - Detects faces with the Haar Cascade Classifier.
   - The annotated images will be passed on to the YOLO pre-trained model to train on the processed images on our dataset
6. **Data Augmentation**:
   - Data Augmentation is applied to the training set only to increase the variability of the training data.
   - This helps to prevent overfitting and help to contextualise to real-world scenarios.
   - Possible augmentations are:
     - Rotation
     - Horizontal Flip
     - Vertical Flip
     - Increasing the brightness
7. **Data Normalization**:
   - The pixel values are normalized to the range `[0, 1]` by dividing by `255`.

```
[ ]: # Step 1: Image Resizing
def resize_image(image: np.ndarray, size: tuple[int, int] = (224, 224)) -> np.
 ↪ndarray:
    return cv2.resize(image, size)

# Step 2: Data Splitting
def split_data(X: list[str], y: list[int]) -> tuple[list[str], list[str],␣
 ↪list[str], list[int], list[int], list[int]]:
    X_train, X_temp, y_train, y_temp = train_test_split(X, y, test_size=0.3,␣
 ↪random_state=42)
    X_val, X_test, y_val, y_test = train_test_split(X_temp, y_temp, test_size=0.
 ↪5, random_state=42)
    return X_train, X_val, X_test, y_train, y_val, y_test

# Step 3: Shuffle Image Paths by Class
def shuffle_paths(X_train: list[str], y_train: list[int]) ->␣
 ↪tuple[list[tuple[str, int]], list[tuple[str, int]]]:
    drowsy_train = [(path, label) for path, label in zip(X_train, y_train) if␣
 ↪label == 1]
    non_drowsy_train = [(path, label) for path, label in zip(X_train, y_train)␣
 ↪if label == 0]
    np.random.shuffle(drowsy_train)
    np.random.shuffle(non_drowsy_train)
    return drowsy_train, non_drowsy_train

# Step 4: Undersample Majority Class
def undersample_majority_class(drowsy_data: list[tuple], non_drowsy_data:␣
 ↪list[tuple]) -> list[tuple]:
    undersample_size = min(len(drowsy_data), len(non_drowsy_data))
    non_drowsy_data = non_drowsy_data[:undersample_size] if␣
 ↪len(non_drowsy_data) > len(drowsy_data) else non_drowsy_data
    drowsy_data = drowsy_data[:undersample_size] if len(drowsy_data) >␣
 ↪len(non_drowsy_data) else drowsy_data
    balanced_train = drowsy_data + non_drowsy_data
    np.random.shuffle(balanced_train)
    return balanced_train

# Step 5 : Image Annotation using Haar-Cascade Classifier
def generate_face_bounding_box(image: np.ndarray) -> tuple:

    # Load pre-trained haar cascade classifier for face detection
    face_cascade = cv2.CascadeClassifier(cv2.data.haarcascades +␣
 ↪'haarcascade_frontalface_default.xml')

    # Convert img to grayscale -> for faster detection
    gray = cv2.cvtColor(image, cv2.COLOR_BGR2GRAY)
```

```
    # scaleFactor = 1.1 : image is reduced by 10%
    # minNeighbours = 5 : 5 neighbours for each candidate rectangle should have␣
 ↪to retain it
    # minSize = (30,30) : Minimum possible detected object size. Smaller␣
 ↪objects are ignored
    faces = face_cascade.detectMultiScale(gray, scaleFactor=1.1,␣
 ↪minNeighbors=5, minSize=(30, 30))

    if len(faces) > 0:
        (x, y, w, h) = faces[0]  # Taking the first detected face
        img_height, img_width = image.shape[:2]
        x_center = (x + w / 2) / img_width
        y_center = (y + h / 2) / img_height
        width = w / img_width
        height = h / img_height
        return (x_center, y_center, width, height)
    return None  # Return None if no face is detected

# Step 5.1 : Saving annotated image
def save_yolo_annotation(save_dir, image_name, label, bbox):
    annotation_path = os.path.join(save_dir, f"{image_name}.txt")
    with open(annotation_path, "w") as f:
        if bbox:
            f.write(f"{label} {bbox[0]} {bbox[1]} {bbox[2]} {bbox[3]}\n")

# Step 6: Data Augmentation
def augment_image(image: np.ndarray, augment_count: int = 5) -> list[np.
 ↪ndarray]:
    # augment_count = 5 : Generates 5 augmented images for each input image
    # each augmented image is a random combination of augmentations defined in␣
 ↪datagen variable

    datagen = ImageDataGenerator(
        rotation_range=15,
        brightness_range=[0.8, 1.2],
        horizontal_flip=True,
        zoom_range=0.1,
        fill_mode='nearest'
    )
    # Adds an extra dimension to image,
    # since ImageDataGenerator.flow expects a batch of images, even with size 1␣
 ↪batch sizes
    img_array = image.reshape((1,) + image.shape)

    # Loops augment_count times
```

```
    # next() : returns the iterator (augmented image generated)
    # datagen.flow() : creates an iterator that generates batches of augmented␣
 ↪images
    augmented_images = [next(datagen.flow(img_array, batch_size=1))[0].
 ↪astype(np.float32) for _ in range(augment_count)]
    return augmented_images

# Step 7: Data Normalization
def normalize_image(image: np.ndarray) -> np.ndarray:
    return image / 255.0

# Main Pipeline with Reshuffling and Undersampling
def preprocess_pipeline(image_paths: list[str], labels: list[int],␣
 ↪augment_count: int = 5, save_dir: str = "Processed_Images", batch_size: int␣
 ↪= 100):
    os.makedirs(os.path.join(save_dir, "Images", "train"), exist_ok=True)
    os.makedirs(os.path.join(save_dir, "Images", "val"), exist_ok=True)
    os.makedirs(os.path.join(save_dir, "Images", "test"), exist_ok=True)
    os.makedirs(os.path.join(save_dir, "Labels", "train"), exist_ok=True)

    X_train, X_val, X_test, y_train, y_val, y_test = split_data(image_paths,␣
 ↪labels)

    # Shuffle and balance training data
    drowsy_train, non_drowsy_train = shuffle_paths(X_train, y_train)
    balanced_train = undersample_majority_class(drowsy_train, non_drowsy_train)

    # Process Training Data (Annotation + Augmentation + Normalization)
    for img_path, label in tqdm(balanced_train, desc="Processing Training␣
 ↪Images"):
        img = cv2.imread(img_path)
        if img is None:
            print(f"Skipping invalid image: {img_path}")
            continue
        img_rgb = cv2.cvtColor(img, cv2.COLOR_BGR2RGB)

        # Resize the image
        resized_image = resize_image(img_rgb)

        # Face Annotation
        bbox = generate_face_bounding_box(resized_image)
        if not bbox:
            continue

        # Apply augmentation and normalization
        augmented_images = augment_image(resized_image,␣
 ↪augment_count=augment_count)
```

```
        for idx, aug_img in enumerate(augmented_images):
            normalized_img = normalize_image(aug_img)
            save_images_dir = os.path.join(save_dir, "Images", "train")
            class_name = "Drowsy" if label == 1 else "Non Drowsy"

            # Save each processed image
            save_img = cv2.cvtColor((normalized_img * 255).astype(np.uint8),␣
↪cv2.COLOR_RGB2BGR)
            image_name = f"{class_name}_{idx}_{str(uuid.uuid1())}"
            save_filename = f"{image_name}.jpg"
            save_path = os.path.join(save_images_dir, save_filename)
            cv2.imwrite(save_path, save_img)

            # Save image annotation
            annotation_dir = os.path.join(save_dir, "Labels", "train")
            save_yolo_annotation(annotation_dir, image_name, label, bbox)

    # Process Validation and Test Data (Resize + Normalization Only)
    for split_name, (X_split, y_split) in [("val", (X_val, y_val)), ("test",␣
↪(X_test, y_test))]:
        for img_path, label in tqdm(zip(X_split, y_split), desc=f"Processing␣
↪{split_name.capitalize()} Images"):
            img = cv2.imread(img_path)
            if img is None:
                print(f"Skipping invalid image: {img_path}")
                continue
            img_rgb = cv2.cvtColor(img, cv2.COLOR_BGR2RGB)

            # Resize and normalize only
            resized_image = resize_image(img_rgb)
            normalized_img = normalize_image(resized_image)

            save_images_dir = os.path.join(save_dir, "Images", split_name)
            class_name = "Drowsy" if label == 1 else "Non Drowsy"

            # Save the processed image
            save_img = cv2.cvtColor((normalized_img * 255).astype(np.uint8),␣
↪cv2.COLOR_RGB2BGR)
            image_name = f"{class_name}_{os.path.basename(img_path).split('.
↪')[0]}"
            save_filename = f"{image_name}.jpg" # KIV : add uuid.uuid1() to the␣
↪file name?
            save_path = os.path.join(save_images_dir, save_filename)
            cv2.imwrite(save_path, save_img)

            gc.collect()
```

```
    print("Processing complete.")
```

[35]:
```
output_folder_path = os.path.join(base_path, "Processed_Images")
```

[44]:
```
preprocess_pipeline(all_paths, all_labels, augment_count=5,␣
 ↪save_dir=output_folder_path)
```

```
Processing Training Images: 100%|      | 27310/27310 [34:34<00:00,
13.16it/s]
Processing Val Images: 6269it [17:55,  5.83it/s]
Processing Test Images: 6269it [17:43,  5.89it/s]

Processing complete.
```

[51]:
```
# Verify total number of processed images in training data

print("Unprocessed Total Images:")
unprocessed_drowsy, unprocessed_non_drowsy =␣
 ↪count_images_in_folders(base_path_original, ["Drowsy", "Non Drowsy"])
print(f"\nUnprocessed Training Images:")
print(f"Number of images in Drowsy folder: {round(unprocessed_drowsy * 0.7)}")
print(f"Number of images in Non Drowsy folder: {round(unprocessed_non_drowsy *␣
 ↪0.7)}\n")
print("Processed Images Folder:")
count_images_in_folders(os.path.join(output_folder_path, "Images"), ["train",␣
 ↪"val", "test"])
```

```
Unprocessed Total Images:
Number of images in Drowsy folder: 22348
Number of images in Non Drowsy folder: 19445

Unprocessed Training Images:
Number of images in Drowsy folder: 15644
Number of images in Non Drowsy folder: 13612

Processed Images Folder:
Number of images in train folder: 126735
Number of images in val folder: 6269
Number of images in test folder: 6269
```

[51]:
```
[126735, 6269, 6269]
```