National College of Ireland

Configuration Manual

MSc Research Project
Artificial Intelligence

Aswin Kumar G R
Student ID: x23245778

School of Computing
National College of Ireland

Supervisor: Ms. Sheresh Zahoor

National College of Ireland
Project Submission Sheet
School of Computing

Student Name:	Aswin Kumar G R
Student ID:	x23245778
Programme:	Artificial Intelligence
Year:	2024 - 2025
Module:	MSc. Research Project
Supervisor:	Ms. Sherech Zahoor
Submission Due Date:	12/12/2024
Project Title:	Deep Learning-Based Automated Detection and Classification of Diabetic Retinopathy Using MobileNetV2 and DenseNet201
Word Count:	5990
Page Count:	23

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:	Aswin Kumar G R
Date:	5th December 2024

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

Aswin Kumar G R
x23245778

1 Introduction

This Configuration Manual contains detailed information on configuration of as well as the deployment and operation of the MSc Research Project titled "Deep Learning-Based Automated Detection and Classification of Diabetic Retinopathy Using MobileNetV2 and DenseNet201." The project uses state-of-the-art deep learning approaches to perform automated identification and risk stratification of diabetic retinopathy – a dangerous diabetes-related condition that causes vision impairment. Using MobileNetV2 and DenseNet201, the system provides a high-performance design needed for real-time diagnosis and staging of the disease based on retinal images. This manual is intended for the users of the current project, its developers, and researchers who will need to replicate, maintain, or continue, the project. It outlines system's requirement of the hardware and software, tools and frameworks used and also Preprocessing and analysis of Dataset. Moreover, it explores the deep learning models and training processes, provides the guide for the Flask-based web application system deployment, and presents a comprehensive assessment of the system effectiveness in the form of quantitative and qualitative indices. This manual also contains information on how to work with the system: uploading pictures and choosing models as well as how to analyze the results. Written and illustrated for easy comprehension and ease of use, the manual supports the effective application of the system by the users for maximum advantage in improving the diagnosis of diabetic retinopathy.

2 System Specification (Figure 1)

- Device Name: DESKTOP-2RED1OB
- Processor: 11th Gen Intel(R) Core(TM) i5-1135G7 @ 2.40 GHz (2.42 GHz)
- Installed RAM: 8.00 GB (7.73 GB usable)
- Device ID: 7054E319-8180-418D-BC54-C4D76144954D
- Product ID: 00327-35928-27204-AAOEM
- System Type: 64-bit operating system, x64-based processor
- Pen and Touch: No pen or touch input is available for this display

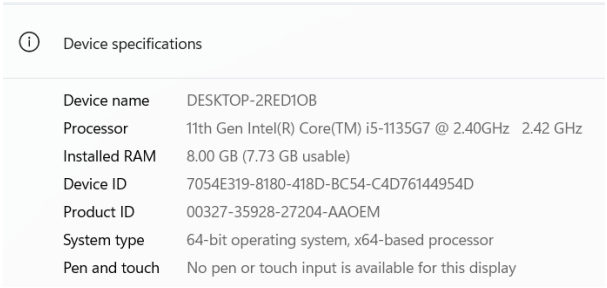


Figure 1: System Specification

3 Software Requirements(Figure 2)

This research work was operationalised using Machine Learning and Deep Learning approaches and, therefore, the project was implemented using Python 3. For creating the web application, Flask was used as the Web Framework, HTML CSS and JavaScript were used for the frontend. The code execution and development was done in Visual Studio Code (VSS) which offers flexibility and an effusive collection of the Libraries needed for the realisation of Deep Learning models.

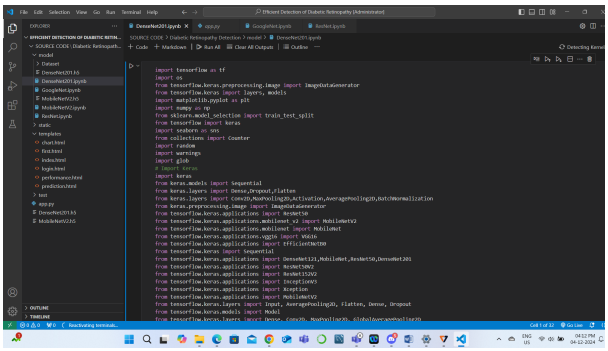


Figure 2: Visual Studio Code (VSS)

4 Data Source(Figure 3)

The dataset applied in this research was obtained from Kaggle, given the title "Retinopathy". This dataset contains a total of 1,812 retinal images, categorized into five distinct classes: No DR (Diabetic Retinopathy) describe images without signs of the disease; Mild DR is characterized by mild signs of DR; Moderate DR distinguishes between mild and moderate symptoms; Severe DR refers to advanced or severe DR; and Proliferative DR describes the most severe form of the disease. This class-wise distribution enables the dataset to provide sufficient data for the training and testing of Machine Learning and Deep Learning algorithm for identification and differentiation of retinopathy stages.

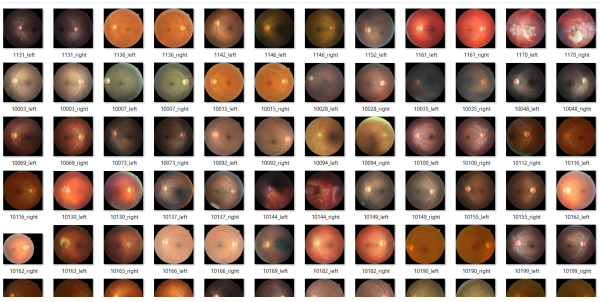


Figure 3: Source Data

5 Data Load and Analysis(Figures 4 - 6)

Firstly, importing, the most necessary libraries for further implementation, such as TensorFlow, Keras, other libraries and modules that were used for data preprocessing, model creating and visualization are presented on the Figure4 and 5. MobileNetV2 and DenseNet201 were used and both models are pretrained models.

- MobileNetV2 was selected due to its small architecture and low computational complexity, the network is effective for real-time tasks and areas with limited computational resources.

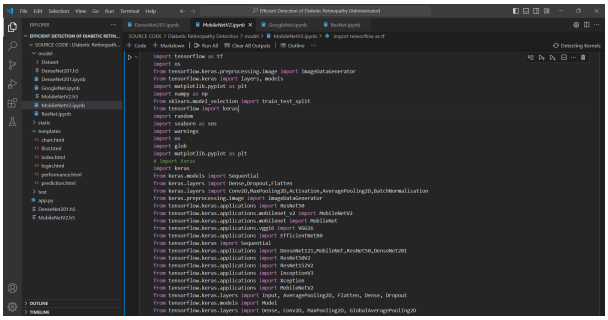


Figure 4: MobileNetV2 libraries

- DenseNet201 was chosen for its dense connectivity concealing high density and overhead map making its parameter high but which we consider an advantage due to possibility of coping with complexity of the dataset well.

Dataset Paths for the training and testing images are defined for both DenseNet201 and MobileNetV2 models. train_dir variable allows to set the path to the directory containing training dataset which is a set of images divided by classes in subdirectories. Likewise, the test_dir variable is used to represent the testing dataset directory in order to validate the model files demos. These paths are defined as raw strings (r") so that the backslashes appearing in the path names in Windows style are not treated as escape characters by Python. This step is very important with loading and arranging the data properly for other preprocessing steps and also in building the deep learning model.

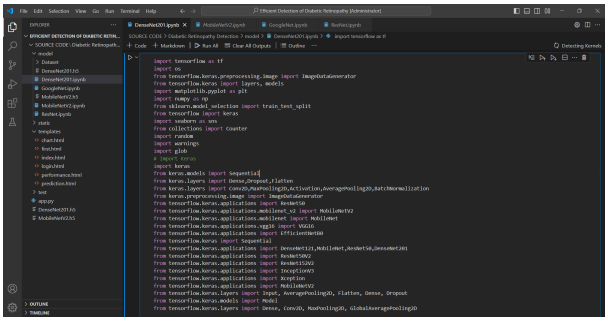


Figure 5: DenseNet201 libraries

```
train_dir=r'C:\Aswin Thesis\Efficient Detection of Diabetic Retinopathy\SOURCE CODE\Diabetic Retinopathy Detection\model\Dataset\training'
test_dir=r'C:\Aswin Thesis\Efficient Detection of Diabetic Retinopathy\SOURCE CODE\Diabetic Retinopathy Detection\model\Dataset\testing'
```

Figure 6: Dataset Paths

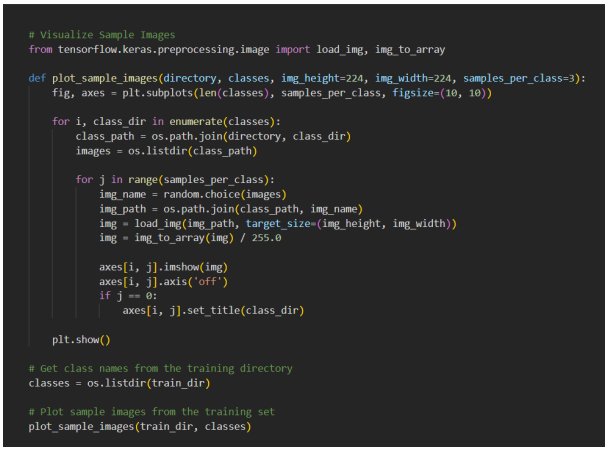
6 Exploratory Data Analysis (EDA)(Figures 7 - 16)

To generalize the results of DenseNet201 and MobileNetV2 the exploratory data analysis (EDA) is carried out to find out more about the given data set. Using the count_images_in_directory function we first determine the number of images in each class in the training and the testing datasets. This step evaluates that both models have a balanced data distribution, or identify any distribution issue that may impact the models. The function plot_class_distribution helps to display the number of instances in each class for the training and testing datasets: bar plots that will allow to evaluate the distribution of the dataset and its heterogeneity. Further, in the plot_sample_images function, few im-

```
# EDA Section

# Function to count number of images per class
def count_images_in_directory(directory):
    class_counts = {}
    for class_dir in os.listdir(directory):
        class_path = os.path.join(directory, class_dir)
        if os.path.isdir(class_path):
            class_counts[class_dir] = len(os.listdir(class_path))
    return class_counts

# Count images in the training and test sets
train_class_counts = count_images_in_directory(train_dir)
test_class_counts = count_images_in_directory(test_dir)

# Plot the class distributions
def plot_class_distribution(class_counts, dataset_type):
    classes = list(class_counts.keys())
    counts = list(class_counts.values())

    plt.figure(figsize=(10,6))
    sns.barplot(x=classes, y=counts)
    plt.title(f'Class Distribution in {dataset_type} Dataset')
    plt.xlabel('Classes')
    plt.ylabel('Number of Samples')
    plt.xticks(rotation=45)
    plt.show()

# Plot for training and test sets
plot_class_distribution(train_class_counts, 'Training')
plot_class_distribution(test_class_counts, 'Test')
```

Figure 7: Count and Plot the Class Distribution

ages randomly from each class are selected using the montage widget. This visualization aids in, not only, the quality and variety of the input data to be supplied to both the DenseNet201 and the MobileNetV2 models but also the appearance. These steps lead the preprocessing decisions and the selection of models such that both models are trained based on the well understood data. Hence the visualization and statistical comprehension of the given dataset enable a fine-tuning or adjustment of both models for better performance of the detection of diabetic retinopathy.



```python
# Visualize Sample Images
from tensorflow.keras.preprocessing.image import load_img, img_to_array

def plot_sample_images(directory, classes, img_height=224, img_width=224, samples_per_class=3):
    fig, axes = plt.subplots(len(classes), samples_per_class, figsize=(10, 10))

    for i, class_dir in enumerate(classes):
        class_path = os.path.join(directory, class_dir)
        images = os.listdir(class_path)

        for j in range(samples_per_class):
            img_name = random.choice(images)
            img_path = os.path.join(class_path, img_name)
            img = load_img(img_path, target_size=(img_height, img_width))
            img = img_to_array(img) / 255.0

            axes[i, j].imshow(img)
            axes[i, j].axis('off')
            if j == 0:
                axes[i, j].set_title(class_dir)

    plt.show()

# Get class names from the training directory
classes = os.listdir(train_dir)

# Plot sample images from the training set
plot_sample_images(train_dir, classes)
```

Figure 8: Visualize Sample Images

In order to determine the dimensions of the images within the dataset, image shapes and aspect ratios are examined before preparing the data for both DenseNet201 and MobileNetV2 models. The analyze_image_shapes function just loops through all files in a given set and gather file width and height data. This information assists in deciding whether the data set includes images of the fixed dimension or if the images should be resized to fit the model's input dimensions. The dense net 201 as well as mobile net v2 expect the images to be resized to 224*224 pixels as their expected input shape. These

```python
# Analyze Image Shapes and Aspect Ratios
def analyze_image_shapes(directory, classes):
    image_shapes = []

    for class_dir in classes:
        class_path = os.path.join(directory, class_dir)
        images = os.listdir(class_path)

        for img_name in images:
            img_path = os.path.join(class_path, img_name)
            img = load_img(img_path)
            image_shapes.append(img.size)  # (width, height)

    return image_shapes

image_shapes = analyze_image_shapes(train_dir, classes)
image_shapes = np.array(image_shapes)

# Plot aspect ratios
aspect_ratios = image_shapes[:, 0] / image_shapes[:, 1]
plt.hist(aspect_ratios, bins=50, color='blue', alpha=0.7)
plt.title('Aspect Ratios of Images in the Dataset')
plt.xlabel('Aspect Ratio (Width / Height)')
plt.ylabel('Frequency')
plt.show()
```

Figure 9: Analyze Image Shapes and Aspect Ratios

aspect ratios are obtained by dividing the width of the image with the height, and plotted in a histogram. It also allows to determine whether the dataset has predominantly landscapes or portraits for example, which might influence decisions on padding or cropping. Also, for the quantitative distribution of images, a scatter plot is used to display the various dimensions that are available in the given set. In these analyses we verify compatibility of input data with both models and identify possible preprocessing steps necessary to train these models effectively. Distributions of pixel intensities and RGB channels are examined; this analysis helps to determine certain distinctive features of the image data sets with reference to both the DenseNet201 and the MobileNetV2 mod-

```python
# Plot image sizes
plt.scatter(image_shapes[:, 0], image_shapes[:, 1], alpha=0.5)
plt.title('Image Sizes in the Dataset')
plt.xlabel('Width')
plt.ylabel('Height')
plt.show()
```

Figure 10: Image Sizes

els. The plot_pixel_intensity_distribution function is dedicated to the grayscale images through getting pixel values, scaling it for the range [0;1] and plotting histogram. This aids to determine if predominantly the input dataset has bright or dark pixel values or some mixture of the two which is very essential to provide similar inputs to the two models at the time of training. In case of RGB images the plot_rgb_channel_distribution function

```python
def plot_pixel_intensity_distribution(directory, classes, num_images=50):
    pixel_values = []

    for class_dir in classes:
        class_path = os.path.join(directory, class_dir)
        images = os.listdir(class_path)

        # Collect pixel intensities from a sample of images
        for img_name in images[:num_images]:  # Sample a few images
            img_path = os.path.join(class_path, img_name)
            img = load_img(img_path, target_size=(img_height, img_width), color_mode='grayscale')
            img = img_to_array(img).flatten() / 255.0  # Normalize pixel values
            pixel_values.extend(img)

    # Plot the histogram
    plt.figure(figsize=(10,6))
    plt.hist(pixel_values, bins=50, color='blue', alpha=0.7)
    plt.title('Pixel Intensity Distribution Across Images')
    plt.xlabel('Pixel Intensity (Normalized)')
    plt.ylabel('Frequency')
    plt.show()

# Analyze pixel intensities for training images
plot_pixel_intensity_distribution(train_dir, classes)
```

Figure 11: Analyze Pixel Intensity

extracts the images into red, green, and blue channel. It standardizes each channel's pixel and creates two different histograms to represent the intensity of the given channels. This step allows for checking dataset for bias with regards to color channels, guaranteeing the color information for the models to learn from is balanced. The foundations for both

```python
def plot_rgb_channel_distribution(directory, classes, num_images=50):
    red_channel = []
    green_channel = []
    blue_channel = []

    for class_dir in classes:
        class_path = os.path.join(directory, class_dir)
        images = os.listdir(class_path)

        for img_name in images[:num_images]:  # Sample a few images
            img_path = os.path.join(class_path, img_name)
            img = load_img(img_path, target_size=(img_height, img_width))
            img_array = img_to_array(img).astype('float32') / 255.0  # Normalize pixel values
            red_channel.extend(img_array[:, :, 0].flatten())  # Red channel
            green_channel.extend(img_array[:, :, 1].flatten())  # Green channel
            blue_channel.extend(img_array[:, :, 2].flatten())  # Blue channel

    # Plot histograms for each color channel
    plt.figure(figsize=(15,5))
    plt.subplot(1, 3, 1)
    plt.hist(red_channel, bins=50, color='red', alpha=0.7)
    plt.title('Red Channel Intensity Distribution')

    plt.subplot(1, 3, 2)
    plt.hist(green_channel, bins=50, color='green', alpha=0.7)
    plt.title('Green Channel Intensity Distribution')

    plt.subplot(1, 3, 3)
    plt.hist(blue_channel, bins=50, color='blue', alpha=0.7)
    plt.title('Blue Channel Intensity Distribution')

    plt.show()

# Analyze RGB channels for training images
plot_rgb_channel_distribution(train_dir, classes)
```

Figure 12: Analyze RGB Channels

DenseNet201 and MobileNetV2 are on the ImageNet weights, where this assumption of normalized image input holds. This way you guarantee that by next preprocessing segment it adheres with above requirements, which in turn aids both models employed in

diabetic retinopathy detection to perform their best. The mean and standard deviation

```python
def calculate_mean_std_per_channel(directory, classes):
    means = np.zeros(3)
    stds = np.zeros(3)
    num_images = 0

    for class_dir in classes:
        class_path = os.path.join(directory, class_dir)
        images = os.listdir(class_path)

        for img_name in images:
            img_path = os.path.join(class_path, img_name)
            img = load_img(img_path, target_size=(img_height, img_width))
            img = img_to_array(img) / 255.0

            means += np.mean(img, axis=(0, 1))
            stds += np.std(img, axis=(0, 1))
            num_images += 1

    means /= num_images
    stds /= num_images

    print(f"Mean per channel: {means}")
    print(f"Standard deviation per channel: {stds}")

# Calculate the mean and standard deviation for the training images
calculate_mean_std_per_channel(train_dir, classes)
```

Figure 13: Calculate Mean and Standard Deviation for Each RGB Channel

of pixel intensities are computed, and the distribution of pixel intensities in images is assessed utilizing the correponding histograms to verify whether the content of the dataset corresponds to the training of the models of DenseNet201 and MobileNetV2. The function calculate_mean_std_per_channel calculates the statistics, mean, and standard deviation of all images 's pixel intensities channel wise, meaning three channels- Red, Green, and Blue respectively. This step helps to get the intensity and contrast of each channel and strike a balance for each channel of color in the frame to avoid one or more channels being dominant of the others. Next, they calculate the mean and variance of these statistics and depict such in bar plots alongside others to expose any bias that may hinder the learning of the models.

```python
import numpy as np
import matplotlib.pyplot as plt

# Example values for the mean and standard deviation (replace with your calculated values)
means = np.array([0.37779536, 0.26045484, 0.18353424])
stds = np.array([0.22850083, 0.16039101, 0.11263551])

# Channel labels (R, G, B)
channels = ['Red', 'Green', 'Blue']

# Create a bar plot for mean and standard deviation
fig, ax = plt.subplots(1, 2, figsize=(12, 6))

# Plot Mean per channel
ax[0].bar(channels, means, color=['r', 'g', 'b'])
ax[0].set_title('Mean per Channel')
ax[0].set_ylabel('Mean Value')
ax[0].set_ylim(0, 1)  # Since the pixel values are normalized to [0,1]

# Plot Standard Deviation per channel
ax[1].bar(channels, stds, color=['r', 'g', 'b'])
ax[1].set_title('Standard Deviation per Channel')
ax[1].set_ylabel('Standard Deviation Value')
ax[1].set_ylim(0, 1)  # Since standard deviation is also normalized to [0,1]

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

Figure 14: Calculate Mean and Standard Deviation for Each RGB Channel

The calculate_overall_stats function calculates the overall mean, median and the standard deviation of pixel intensity average over all images and all channels. These metrics give a simple yet comprehensive picture of how bright and variable the entire dataset is. These overall statistics are presented in a bar plot to get an idea of the global features of the data set.

As it will be recalled both DenseNet201 and MobileNetV2 use pre-trained weights on ImageNet where input data is normalized to specific distributions. By doing these

```python
def calculate_overall_stats(directory, classes):
    all_pixels = []

    for class_dir in classes:
        class_path = os.path.join(directory, class_dir)
        images = os.listdir(class_path)

        for img_name in images:
            img_path = os.path.join(class_path, img_name)
            img = load_img(img_path, target_size=(img_height, img_width))
            img = img_to_array(img) / 255.0

            # Flatten the image array and append to the list of all pixels
            all_pixels.append(img.reshape(-1, 3))  # Reshape to (num_pixels, 3) for R, G, B channels

    all_pixels = np.vstack(all_pixels)  # Stack all pixel arrays into one

    # Calculate overall statistics
    overall_mean = np.mean(all_pixels)
    overall_median = np.median(all_pixels)
    overall_std = np.std(all_pixels)

    print(f"Overall Mean: {overall_mean}")
    print(f"Overall Median: {overall_median}")
    print(f"Overall Standard Deviation: {overall_std}")

# Calculate the overall mean, median, and standard deviation for the training images
calculate_overall_stats(train_dir, classes)
```

Figure 15: Calculate Overall Pixel Intensity Statistics

```python
mean = 0.2739
median = 0.2510
std_dev = 0.2235

# Labels and values for the bar plot
labels = ['Mean', 'Median', 'Standard Deviation']
values = [mean, median, std_dev]

# Create the bar plot
plt.figure(figsize=(8, 6))
plt.bar(labels, values, color=['blue', 'green', 'orange'])

# Add titles and labels
plt.title('Overall Mean, Median, and Standard Deviation of Pixel Intensities')
plt.ylabel('Values')

# Display the plot
plt.show()
```

Figure 16: Calculate Overall Pixel Intensity Statistics

statistic, you make sure the input format of the models will be matched with the dataset, resulting into higher training and better prediction of diabetic retinopathy disease. All the above-stated preprocessing steps are very important to obtain higher results and to overcome the problem of unequal or imbalanced pixel distribution.

7 Preprocess Dataset for Training (Figures 17 - 18)

```python
batch_size = 1
epochs = 5
img_height = 224
img_width = 224
```

Figure 17: Image Size

This involves pre-processing of the dataset in readiness to train DenseNet201 and the MobileNetV2 model. An ImageDataGenerator is then created with rescale parameter to scale the pixel values from range [0, 255] to [0, 1]. This appears to normalise the data as a similar pre-process was performed for both inputs based on the models' expectations of normalised input data as used in ImageNet datasets. Two generator models

train_data_gen and val_data_gen are built for the training and validation data set. These generators read images from the given directories, resize them to the standard input size of 224 x 224 pixels for both models, normalize them and arrange them into batches.

During training, the model seems to learn in sequence, and the training data generator fixes this by using random shuffling to minimize this issue. The validation generator, in the same way, brings images for model assessment while not permutating, to make outcomes well-arranged. Part of the optimization parameters include using a batch size equal to 32, which is memory efficient during training yet fast for computations. In preprocessing the dataset in this way, both DenseNet201 and MobileNetV2 are fed with correctly formatted and standardized data which ensures the best results are obtained from the networks in the detection of Diabetic Retinopathy.

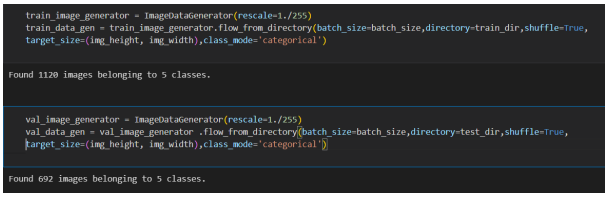


Figure 18: Creating Data Generators for Training and Validation

8 Define and Train the Model

8.1 MobileNetV2 (Figures 19 - 23)

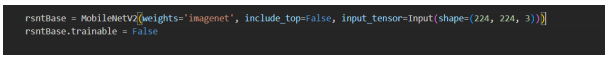


Figure 19: Pre-trained MobileNetV2

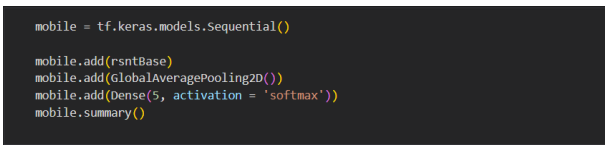


Figure 20: Custom Layers

MobileNetV2 architecture is customized, trained, and then saved for the diabetic retinopathy detection problem. MobileNetV2 which is improved and pre-trained on ImageNet dataset is selected as the base model for transfer learning. Same as in the previous example the convolutional base is loaded with include_top=False and its weights are frozen. Additional layers are custom layers, which are GlobalAveragePooling2D, applied before the final fully connected Dense layer which has 5 nodes to classify the diabetic retinopathy into the five classes. In the output layer a softmax nonlinearity is applied to give probabilities for each class of the input. The model is compiled using the Adam optimizer for an efficient adaptive learning and categorical crossentropy loss for multi-class classification problem. Training is done to 60 epochs with the help of the previously

```
mobile.compile(optimizer= 'adam' , loss= 'categorical_crossentropy', metrics=['accuracy'])
```

Figure 21: Model Compilation

```
history = mobile.fit(train_data_gen, epochs=60,
validation_data= val_data_gen,)
```

Figure 22: Model Training

created data generators, the training generator generates the set of normalized image batches and the validation generator tracks performance on unseen images. After training the model is stored as MobileNetV2.h5 so that the model can be returned to for testing or used in deployment without having to be trained again. This approach effectively enables transfer learning while guaranteeing a solid model well suited to diabetic retinopathy classification.

```
mobile.save('MobileNetV2.h5')
```

Figure 23: Saving the Model

8.2 DenseNet201 (Figures 24 - 27)

```
rsntBase = DenseNet201(weights='imagenet', include_top=False, input_tensor=Input(shape=(224, 224, 3)))
rsntBase.trainable = False
#rsntBase.summary()
```

Figure 24: Pre-trained DenseNet201

```
resnet = tf.keras.models.Sequential()

resnet.add(rsntBase)
resnet.add(GlobalAveragePooling2D())
resnet.add(Dense(5, activation = 'softmax'))
resnet.summary()
```

Figure 25: Custom Layers

The DenseNet201 model is configured, trained and then saved with respect to the task of diabetic retinopathy classification. ImageNet pre-trained Convolutional Neural Network DenseNet201 is used as a transfer learning basis model. The top layers of the base model that does not include the top classification layers are not included (include_top=False) and the weights of the model are frozen in order to use them for feature extraction only. The base model is modified with additional layers; GlobalAveragePooling2D layer to transform the feature maps into a single vector, the latter is followed by a Dense layer containing 5 neurons, which will classify the images into the stages of diabetic retinopathy. In the final layer, the softmax activation function is applied so as to produce the probability of each class. The model used Adam optimizer, this optimizer optimizes the learning rate automatically to improve training efficiency, categorical cross entropy

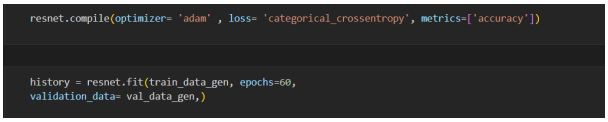


Figure 26: Model Compilation and Training

was the loss function used for multi class classification, the model used accuracy to measure its performance. Developed for 60 epochs using the data generators processed earlier. The training generator supplies normalized and augmented images for learning and the validation generator then assesses the performance of the model on data that it has not seen before. The trained model is then saved in h5 format as DenseNet201.h5 format so that the reuse is accomplished for evaluation, fine-tuning or deployment. This approach takes advantage of feature extraction of DenseNet201 to provide reliable performance in identifying diabetic retinopathy.

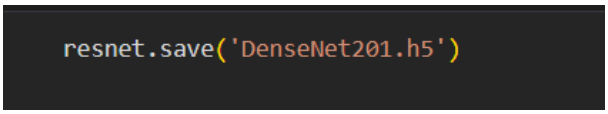


Figure 27: Saving the Model

9 Model Evaluation (Figure 28)

The trained models are DenseNet201 and MobileNetV2, the model's performance is assessed on the validation data set finding the true labels and proposed labels from the models. The actual class labels are obtained from val_data_gen which generates batches of images and their one hot encoded equivalents. These labels are combined and join into a new array and then the code convert all one hot encoded form into simple numeric form using np.argmax. The trained models are then used in the manner of predicting

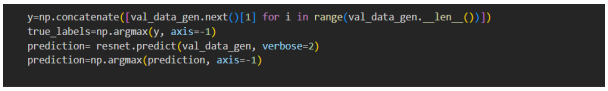


Figure 28: Model Prediction

the classes of all validation images. These predictions are produced as likelihoods of the five classes of diabetic retinopathy. Applying the function np.argmax to these probabilities, we find the predicted class indices which are most likely to belong to the images. This makes the process same for both models to obtain similar results of the evaluation process. The true and predicted labels serve as a basis for other performance analysis including construction of confusion matrix and determining accuracy, precision and recall and F1-measure. This step enables a comparison to be made quantitatively of how well DenseNet201 and MobileNetV2 are suited for the diabetic retinopathy classification task.

```
def plot_confusion_matrix(cm, classes,
                          normalize=False,
                          title='Confusion matrix',
                          cmap=plt.cm.Blues):
    """
    This function prints and plots the confusion matrix.
    Normalization can be applied by setting `normalize=True`.
    """
    plt.imshow(cm, interpolation='nearest', cmap=cmap)
    plt.title(title)
    plt.colorbar()
    tick_marks = np.arange(len(classes))
    plt.xticks(tick_marks, classes, rotation=45)
    plt.yticks(tick_marks, classes)

    if normalize:
        cm = cm.astype('float') / cm.sum(axis=1)[:, np.newaxis]
        print("Normalized confusion matrix")
    else:
        print('Confusion matrix, without normalization')

    print(cm)

    thresh = cm.max() / 2.
    for i, j in itertools.product(range(cm.shape[0]), range(cm.shape[1])):
        plt.text(j, i, cm[i, j],
            horizontalalignment="center",
            color="white" if cm[i, j] > thresh else "black")

    plt.tight_layout()
    plt.ylabel('True label')
    plt.xlabel('Predicted label')
```

Figure 29: Confusion Matrix

10 Confusion Matrix (Figures 29 - 30)

Therefore, a confusion matrix is constructed and displayed to assess the DenseNet201 and MobileNetV2 on the classification of diabetic retinopathy. The confusion matrix shows the truths value of the validation set with the model's found prediction which gives clearly classification of accuracy ratio for each class. The confusion matrix has actual classes at the row side, and the predicted classes are on the column side. The actual classifications are formed within the diagonal whereas misclassifications are displayed at other locations in the grid.

The plot_confusion_matrix function can be used to plot a heatmap, in which Y-axis represents True Label and X-axis represents Predicted label and the color intensity actually shows how many samples fall in that cell. The function can also optionally scale the values to expressions of percentages as well to make further more relative contrasts between model performance on the imbalanced datasets easier to observe . To make the generated matrix interpretable, classes that are used for labeling are class names including 'No_DR,' 'Mild,' 'Moderate,' 'Severe,' and 'Proliferative_DR.' By visualizing confusion matrices for

```
from sklearn.metrics import confusion_matrix
import itertools
import matplotlib.pyplot as plt
cm = confusion_matrix(y_true=true_labels, y_pred=prediction)


cm_plot_labels = ['No_DR','Mild','Moderate','Severe','Proliferate_DR']


plot_confusion_matrix(cm=cm, classes=cm_plot_labels, title='Confusion Matrix')
```

Figure 30: Visualization

DenseNet201 and MobileNetV2, we realize specific classes that each model is good at or bad at. For instance, it can tell the extent to which the models are overlapping between severe and moderate cases or if the rare classes are not being identified. It is important with this step in order to obtain perspectives for further improvements of the models where the pros and cons are outlined.

11 Model Metrics (Figure 31)

Performance measures of the DenseNet201 and MobileNetV2 for the classification of DR are computed and assessed. Specifically, we calculate four evaluation scores: accuracy, precision, recall, and F1 score. Accuracy determines the total percentage of preciseness by dividing the number of samples that the models have predicted correctly, to the total samples available. Accuracy measures the ratio of the total accurately predicted positive samples to the total number of these types of samples thereby illuminating just how accurate the models are when making their predictions. Recall computes the percentage of actual positive samples correctly remembered by the models as it focuses on accurate identification of true positives. The F1-score computes the harmony mean of both precision and recall to giving a single figure for checking the two measures concurrently. Both these metrics are computed through the true_labels (labels obtained from the valid-

Figure 31: Model Metrics

ation data sets) and predicted_labels (predictions generated by the models). In the case of multi-class classification, the metrics are averaged for all classes, by using the micro method. From both DenseNet201 and MobileNetV2 model, it becomes possible to define which model has a higher or lower overall accuracy, class level of precision, sensitivity and specificity, and F1 score. It helps in deciding whether to choose or improve the best option for the model in the application for detecting diabetic retinopathy.

12 Explanation of app.py (Figures 32 - 42)

This Flask application serves as a web-based interface to predict diabetic retinopathy classification using two deep learning models: DenseNet201 and MobileNetV2. The code enables users to upload an image and choose between the described models and get the prediction. Below is a detailed explanation of each part of the code:

12.1 Import Required Libraries

The first part of the application requires importation of several libraries for building and running the Flask web application, pre-processing images and generate predictions with the help of pre-existing machine learning models. Flask is imported as the core web interface builder tool as well as for manoeuvring user interactions and for routing the

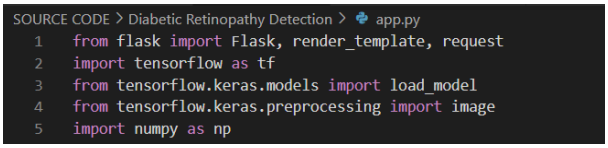


Figure 32: Import Required Libraries

requests and last but not least for HTML templating. As for the libraries, TensorFlow and Keras are imported to load the pre-trained deep learning models such as DenseNet201 and MobileNetV2 as well as for performing the transformation of images for making the prediction. In particular, load_model in the Keras utilities is applied to load the saved models, and the image module that helps to preprocess the uploaded images (resizing, normalizing and reshaping) corresponding to the input format of the model. Last, numpy lib is imported to perform array manipulations and data transformation tasks which may be required in preprocessing or during the stage of making predictions. Combined these libraries are the fundamental core that is employed when aiming to create a viable—but basic—machine learning web application.

12.2 Initialize Flask Application

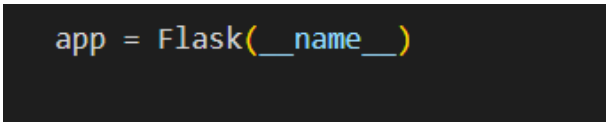


Figure 33: Initialize Flask Application

The second step involves allowing Flask to handle the application hence creating an instance of the Flask class and saving it in variable app. This initialization is used to configure to Flask, the web framework of choice for this application. By default, the flask instance is the central figure and through it one can define routes, handle HTTP requests and templates. Calling Flask(__name__) tells the app where it is defined and Flask then understands where static and template files are relative to the app directory. This initialization step is necessary for making routes and for setting up the fundamental application's functionality and what it will do, for instance, handle inputs from users and generating HTML pages on the fly.

12.3 Load Models

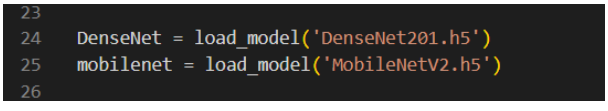
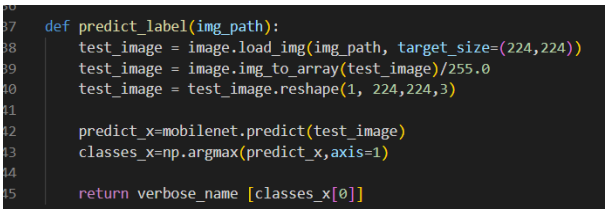


Figure 34: Model Loading

In the third step, the DenseNet201 and MobileNetV2 were imports into memory space using load_model function in Keras. The models stored in .h5 format hold the wt. value, which was learned firearm analysis procedure or architecture during the training session and can be used to make the predictions on the new data. The DenseNet201.h5

file contains DenseNet201 model fine-tuned on ImageNet for the diabetic retinopathy classification as well as for the MobileNetV2 model optimized for the same task. The deployed models are made easily accessible for use during inference using the variables DenseNet and mobilenet when loaded into the application. They are designed to be ready to take preprocessed input images and spit out predictions to form the ML capability of the envisaged application.

12.4 MobileNetV2 Prediction



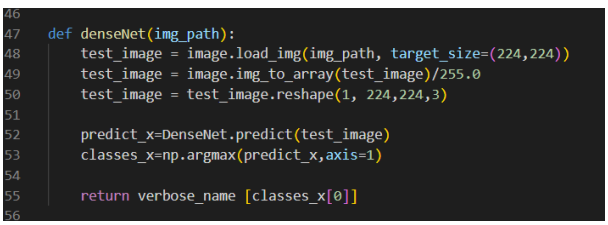
```
def predict_label(img_path):
    test_image = image.load_img(img_path, target_size=(224,224))
    test_image = image.img_to_array(test_image)/255.0
    test_image = test_image.reshape(1, 224,224,3)

    predict_x=mobilenet.predict(test_image)
    classes_x=np.argmax(predict_x,axis=1)

    return verbose_name [classes_x[0]]
```

Figure 35: MobileNetV2 Prediction

The predict_label function was also designed entailing the generation of the predictions using the MobileNetV2 model. It accepts the path of an image file img_path as the input and then it properly prepares the given image for the feeding to the model. First, to represent the image in the model, it is loaded using the load_img function from Keras and immediately resized to the standard size for the chosen model: 224 x 224. Subsequently, the input image is converted to a NumPy array format and then further standardize by the division the pixel value through 255.0 to make it into a range of 0 to 1. The image array is then resized to a 4-D tensor with the batch size of 1 (As the inputs to the network must be in the form 1 * height * width * channel).

Preprocessed image is then used to pass to the MobileNetV2 model for prediction using the predict method which gives the probabilities of each class. The np.argmax is used to find out the index corresponding to the class having maximum probability, which is the predicted class for the model. Lastly, the function employs the verbose_name dictionary to replace the class index with the label that would be easier to understand such as "No_DR," "Mild," and the like and then return the label as the prediction. This function bows up the entire work flow for preprocessing and predicting an image using MobileNetV2.

12.5 DenseNet201 Prediction



```
def denseNet(img_path):
    test_image = image.load_img(img_path, target_size=(224,224))
    test_image = image.img_to_array(test_image)/255.0
    test_image = test_image.reshape(1, 224,224,3)

    predict_x=DenseNet.predict(test_image)
    classes_x=np.argmax(predict_x,axis=1)

    return verbose_name [classes_x[0]]
```

Figure 36: DenseNet201 Prediction

The denseNet function makes predictions applying the DenseNet201 model in the same way as the function predict_label. The function takes the image path, img_path as input and processes the image in order to meet the standard of DenseNet201. Firstly, the image is built in and then resized through Keras's load_img to 224 x 224 pixels. Next, the obtained image is converted into the NumPy array where its pixel values are divided by 255.0 in order to get range [0, 1]. The normalized image array is rescaled into the DenseNet201 model acceptable 4D tensor format with a batch size of 1 (shape (1,224,224,3)).

The generated preprocessed image is inputted to the DenseNet201 model, using predict method, to get the probabilities of each class. The argmax function is used in numpy called np.argmax whence we obtain the index of the class with the highest probability which gives the output class. Last but not the least, the function converts this class index to its human-readable label by checking the verbose_name dictionary and returns the label. They represent prefect tools for preprocessing and prediction, and this function is designed to encapsulate the DenseNet201 model directly into the application's workflow.

12.6 Home and Navigation Routes

```
56
57  @app.route("/")
58  @app.route("/first")
59  def first():
60      return render_template('first.html')
61
62  @app.route("/login")
63  def login():
64      return render_template('login.html')
65
66  @app.route("/index", methods=['GET', 'POST'])
67  def index():
68      return render_template("index.html")
69
```

Figure 37: Home and Navigation Routes

The home and navigation routes determine the layout of the web application because they map URLs to their correspondent HTML templates. The @app.route("/") and @app.route("/first") decorators define the root route and the /first, the first.html template is probably the home page, or, the first page the user visits upon their interaction with the application. Likewise, the /login is created using @app.route("/login") decorator and returns the 'templates/login.html' which can offer the interface for the user login. The /index route, was created using the BeautifulSoup and is defined with the @app.route("/index", methods = ['GET', 'POST']) decorator, it will respond to both the GET and POST methods and pass the input to the index.html template. This page could act as the homepage or the base or the mode of entrance into/operation of the application. Combined, they provide an easy way to move from one section of the web app to another, creating the framework of the user interface.

12.7 Image Upload and Prediction

The /submit route deals with image uploading and prediction, the only way in which users can in some way engage with the application. It supports GET and POST methods, but the main function is processed during the POST request. The users upload an

Figure 38: Image Upload and Prediction

image using a form (my_image) and choose between two models, DenseNet201 or MobileNetV2. When the image has been uploaded, it is given a file name and saved in the static/tests/ folder for ease of processing. It then examines the chosen model and executes the correct prediction function either density prediction for DenseNet201 or label prediction for MobileNetV2 to analyze the image, make the prediction, and identify the type of diabetic retinopathy.

The prediction result is then passed alongside the image path and the model name to the template for prediction.html. It looks also intersects with the previous one and it is a simple HTML template that enables the identification of the uploaded photo and the prediction result to facilitate model-based classification. This route adopts the work of combining image input, model selection/processing, and result display which is the significant function of this web application.

12.8 Chart and Performance

Figure 39: Chart and Performance

The /chart and the /performance routes can be used as end points to display structures and statistics inside the application. The /chart route is defined having the @app.route("/chart") decorator and this function returns the chart.html template which presumably contains charts or graphical representations regarding diabetic retinopathy classification or trends, distribution and so on. Likewise the /performance route defined with @app.route("/performance") as its decorator, would notify the performance.html template, which might present metrics like accuracy, precision, recall, or F1-score as well as compare the performance of DenseNet201 against MobileNetV2 . These routes enrich the application by presenting analytical visualisations and dynamic status updates while increasing comprehensibility and refine decision making abilities.

```
11
12    verbose_name = {
13
14
15    0: 'No_DR',
16    1: 'Mild',
17    2: 'Moderate',
18    3: 'Severe',
19    4: 'Proliferate_DR'
20
21
22    }
23
```

Figure 40: Defining Prediction Labels

12.9 Defining Prediction Labels

The verbose_name dictionary shown is a general form used to address a situation where the models assign numerical class indices as their labels and relates them to their human equivalent. These indices, such as 0, 1, 2, 3, and 4, represent the five categories of diabetic retinopathy: Normal no DR (No Diabetic Retinopathy), Mild, Moderate, Severe, and Proliferative DR (Proliferative Diabetic Retinopathy). Since DenseNet201 and MobileNetV2 provide outputs in the form of numerical values this dictionary helps the application translate these figures into results which are more easily interpretable by a user. This way the keys of the dictionary give an accurate and informative understandable picture to the user as a result of the model predictions which are associated with these labels. This mapping is something that would form a part of the user experience since it offers a translation of the technical output to diagnostic type.

12.10 Running the Flask App

The final step in the application involves running the Flask development server by including the statement if __name__ == '__main__':. This helps to ensure that, if it is being run directly, it will run without considering it a module of some other program. Flask app development requires the use of a command line whereby the app.run(debug=True) command will start the Flask server locally, making the developed application accessible through a web browser is done. Debug=True puts the web application in Debug mode which will give more detailed errors messages and will also restart each time that changes are made. This is useful particularly during development as the various problems that characterize this process are easily diagnosable and can be easily tested. By running this block, the application is now up and fully ready to process client interactions, prediction, as well as other routings within the code.

12.11 Flask Application on Development Server

Checking the terminal, the message that the Flask Application is running on local server in development mode gives the green light that the app was successfully created and

```
19
20    if __name__ =='__main__':
21        app.run(debug = True)
22
```

Figure 41: Running the Flask App

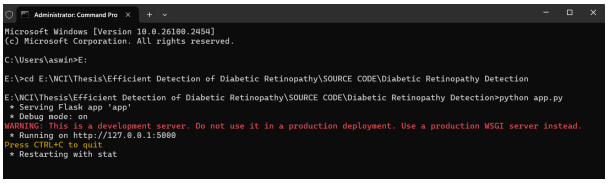


Figure 42: Flask Application on Development Server

deployed. The line * Serving Flask app 'app' means that now a web application that is launched in the Python script with the same name – app, is ready to serve to requests. Debugging mode is enabled, as shown by the line * Debug mode: on, which means that the server is continuously reloading every time some changes are made on the code and gives the best errors whenever a problem occurs. The application is accessible at http:For example, go to in the address bar of your web browser and type http://127.0.0.1:5000, where 127.0.0.1 represents the localhost (the machine you are currently using), and 5000 is the default port for Flask. Other users can open the given URL in web browser in order to interact with it and observe main goals of the application.

12.11.1 Landing Page: Efficient Detection of Diabetic Retinopathy (Figures 43 - 48)

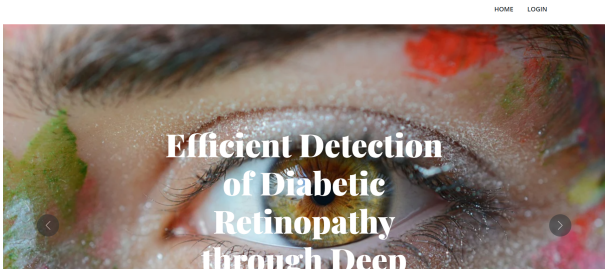


Figure 43: Landing Page: Efficient Detection of Diabetic Retinopathy

In Figure 43, the visitor of the web application of the work titled "Efficient Detection of Diabetic Retinopathy through Deep Learning" can only see the basic landing page. At the top of the page, there is a beautiful large picture of an eye, which tells the visitor that this site is about retinal health. Large letters are positioned over the image and state the goal of the application: to detect diabetic retinopathy using deep learning algorithms such as DenseNet201 and MobileNetV2.

The primary horizontal bar contains the hyperlinks "Home" and "Login" allowing people to go to the main page and get to the webpage that requires authorization. The general layout is clear and leaves a businesslike impression with target users in a specific niche

who are searching for the application's capabilities. It is a landing page of the application or a website encouraging the users to learn more about the specific application.

12.11.2 Login Page: Access with Admin Credentials

The Login Page gives the user a secure way of getting to the application for the detection of Diabetic Retinopathy. In order to continue, the users should enter their credentials at stages Username / Password . For demonstration purposes, the default credentials are set as:

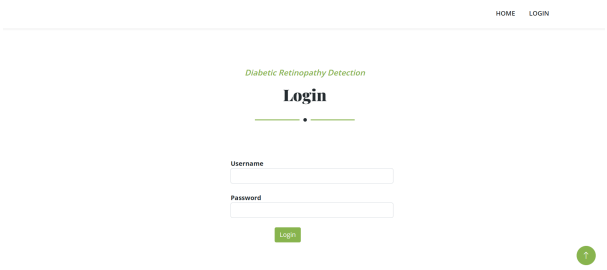


Figure 44: Login Page: Access with Admin Credentials

- Username: admin
- Password: admin

Once the correct credentials have been inputted into the spaces provided, hitting the "Login' button sends the entered data for validation. This simple login capability serves as a way of protecting the application from unauthorized users only. BYOD capability has been integrated to enhance usage of the system, with a clean interface and great organizational format as the vital elements that give users easy control for secure and restricted access to the application.

12.11.3 Image Upload and Model Selection Page

The following completes the figure shows the Image Upload and Prediction Interface of diabetic retinopathy detection application. Due to the fact that this system is the main functional page of the system, users can upload images and then submit the same for analysis and classification. At the very beginning of the page, a title of "Diabetic Retinopathy Detection" is present and after it there is a large and aggressive heading "Preview" which points out that the goal of the page is to present the user with the preview of the offer. In the interface, users are offered an input field which is labeled "Upload Image," and through it people can choose the image file that is stored in their local device for further processing. Under the uploader field, there is a drop down list which has a label model on it, where the user have to select the deep learning models for prediction like MobileNetV2 or DenseNet201 etc. The green-colored "Submit" button is located at the bottom of the page, and submitting the uploaded image along with the selected model will cause the backend to generate and display the classification outcomes. The style used is clean with no complicated graphics allowing its users to use the system with relative ease as well as effectively utilizing the prediction facility.

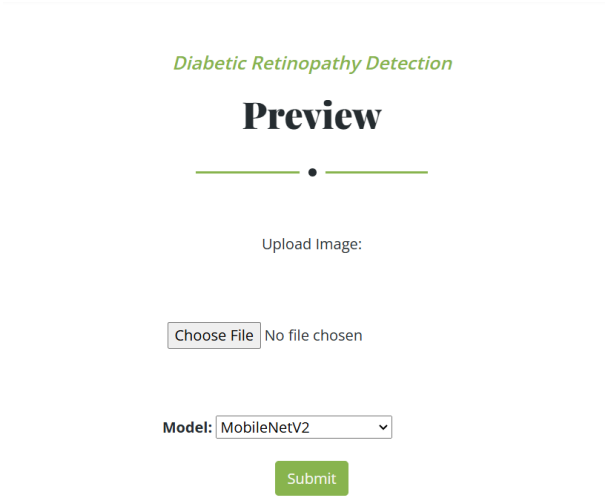


Figure 45: Image Upload and Model Selection Page

12.11.4 Prediction Result and Selected Model

Figure 46 shows the output generated by the diabetic retinopathy detection application once a picture has been uploaded, and the "Submit" button is clicked. On the page, there is the retinal image that was uploaded by the user, the prediction result, and finally the model selected for classification. In this case according to the Prediction Result the uploaded retinal image is described as "Mild" which denotes mild stage of the DR. The Model Selected label is clearly shown as MobileNetV2 to provide an indication that this deep learning model was employed to analyze the image and produce the prediction. In

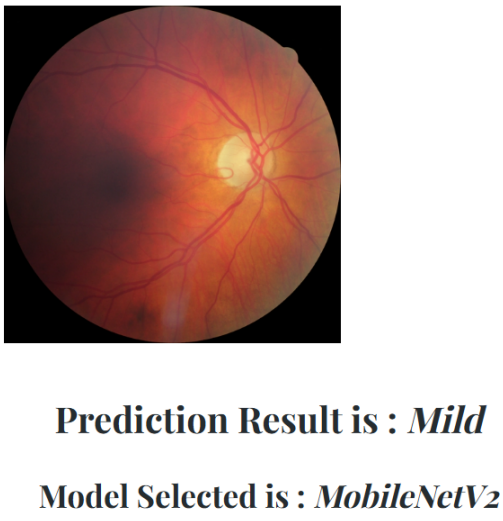


Figure 46: Prediction Result and Selected Model

this simple result page, the input image is displayed and there is an easy-to-read result of the application's diagnostic. Since the uploaded Image is shown to the users and the prediction is printed along with it, the end users can audit the input and its output. Ensuring the selected model is also included adds to the clarity recurrent in analyses using similar models, particularly when other models are used as a benchmark. As a

consequence of the present work, this output refers to the effectiveness of the proposed application for producing accurate and user-friendly results when it comes to diabetic retinopathy detection.

12.11.5 Performance Analysis Page

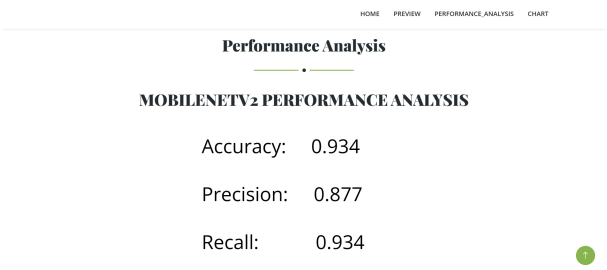


Figure 47: Performance Analysis Page

The Performance Analysis Page gives a comparison of the specific results of both the MobileNetV2 and DenseNet201 for classifying DR. A variety of performance indicators, including Accuracy, Precision, Recall and F1-Score are provided for the two models, providing important data for comparison. Accuracy is a measure of how much of the total predictions got it right independent of the category while Precision looks into how well the models do in reducing the number of false positives or in other words looking at the actual positives among the total number of positives that were predicted. Recall defines the ability of distinguishing between actual positive cases and potential false negatives. Precision is a measure of how many of the identified entities are genuinely negative, and Recall which is a measure of how many of the negative entities in the actual dataset are successfully flagged as such by the models In order to address both Precision and Recall values in managing the models' imbalanced datasets, a balance is passed in the form of a harmonic mean called F1-Score.

The page also contains the Confusion Matrix for each model where a visual demonstration is provided for the classification in all the classes like "No_DR" and "Proliferate_DR". This matrix enables users to find out about the strengths and weaknesses of the models by pointing out areas of right and wrong classifications or misclassifications. The structure of the design supports end users to understand the metrics and the visualization forms in an easy manner. For this reason, users are infused with equal performance results of both models; MobileNetV2 and DenseNet201 making a decision on what is well suited for diabetic retinopathy detection and identifying potential improvements to be made.

12.11.6 Chart Page

The Figure 48 gives the Chart Page of the developed application related to the detection of Diabetic Retinopathy that shows the training and validation accuracy of the two DL models; MobileNetV2 and DenseNet201. The chart uses a bar graph where each model or step is described in blue bar indicating the training accuracy and red bar for the validation accuracy.The aims of this side by comparison is to give an uncomplicated feel of how each of these models trains and performs against unseen validation data. From the chart, the users can tell which model gives a higher accuracy and how much

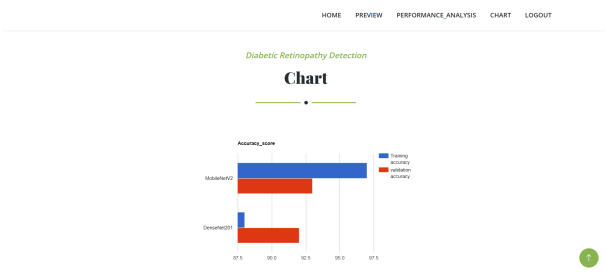


Figure 48: Chart Page

difference exists between the training accuracy and validation accuracy meaning that the model is overfitting or underfitting.The name of the page is Chart with the subtitle Diabetic Retinopathy Detection which indicates that presented charts are concerning the application's performance. With such a comparison shown graphically, an application allows users to assess and compare the efficacies of MobileNetV2 and DenseNet201 and select between the two models most effectively.