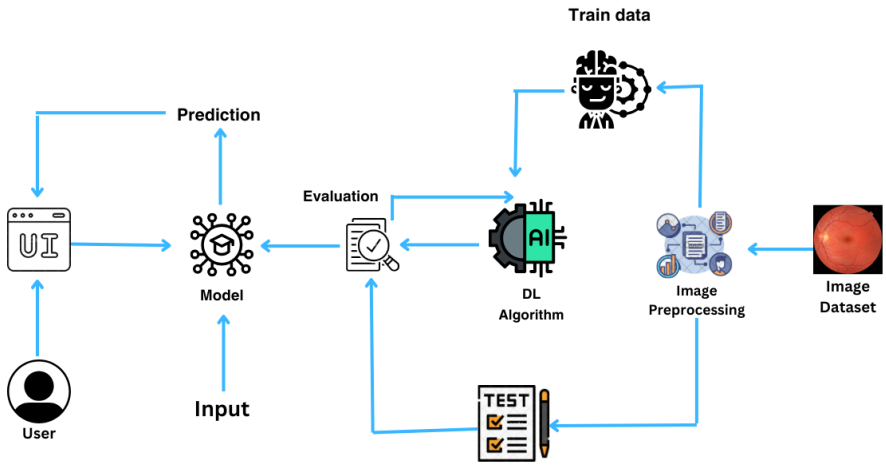
Eye Disease Detection Using Deep Learning

Project Description:

In this project, we are categorizing different kinds of ocular diseases that individuals can develop for a variety of reasons, such as age, diabetes, etc. The four main groups into which these disorders fall are Normal, Diabetic Retinopathy, Cataract, and Glaucoma. In artificial intelligence (AI), deep learning (DL) techniques are key components of high-performance classifiers used in the image-based diagnosis of eye diseases.

One of the most widely used methods that has improved performance in many domains, particularly image analysis and classification, is transfer learning. We employed transfer learning approaches like as Inception V3, VGG19, and Xception V3, which are more popular and very successful in the field of image analysis.

Technical Architecture:



Project Flow:

- The user interacts with the UI (User Interface) to choose the image.
- The chosen image analyzed by the model which is integrated with flask application.
- The VGG19 Model analyzes the image, then the prediction is showcased on the Flask UI.

To accomplish this, we have to complete all the activities and tasks listed below

- Data Collection.
    - Create a Train and Test path.

- Image Pre-processing.
    - Import the required library
    - Configure ImageDataGenerator class
    - Apply ImageDataGenerator functionality to Trainset and Testset
- Model Building
    - Pre-trained CNN model as a Feature Extractor
    - Adding Dense Layer
    - Configure the Learning Process
    - Train the model
    - Save the Model
    - Test the model
- Application Building
    - Create an HTML file

Prior Knowledge:

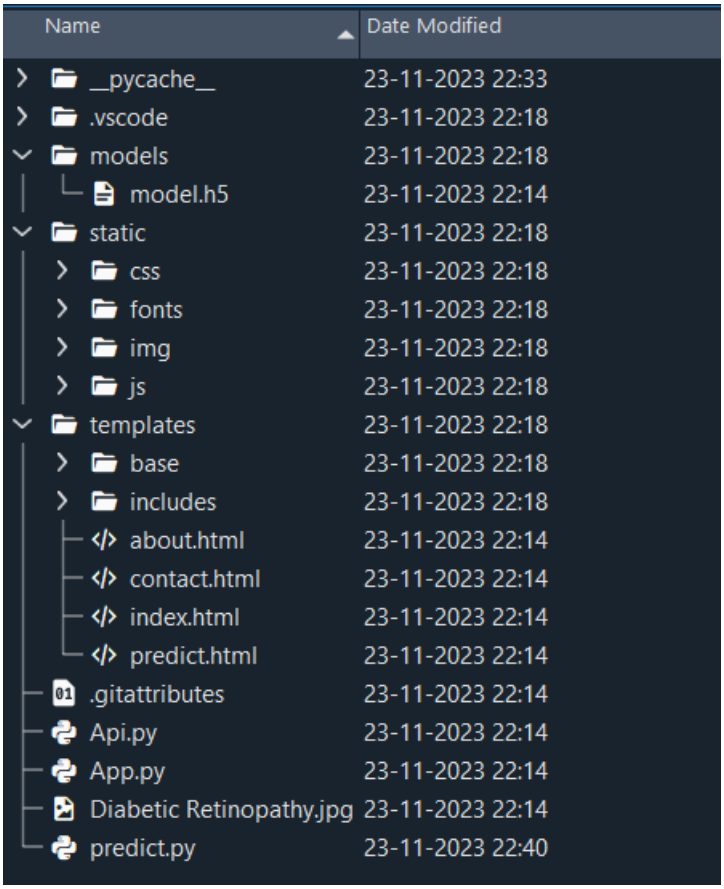
You must have prior knowledge of following topics to complete this project.

- **Deep Learning Concepts**
    - **CNN:** https://towardsdatascience.com/basics-of-the-classic-cnn-a3dce1225add
    - **VGG19: VGG-19 convolutional neural network - MATLAB vgg19 - MathWorks India**
    - **ResNet-50V2: https://towardsdatascience.com/understanding-and-coding-a-resnet-in-keras-446d7ff84d33**
    - **Inception-V3:** https://iq.opengenus.org/inception-v3-model-architecture/
    - **Xception: https://pyimagesearch.com/2017/03/20/imagenet-vggnet-resnet-inception-xception-keras/**
- **Flask:** Flask is a popular Python web framework, meaning it is a third-party Python libraryused for developing web applications.

  Link: **https://www.youtube.com/watch?v=lj4I_CvBnt0**

Project Structure:

Create a Project folder which contains files as shown below



- The Dataset folder contains the training and testing images for training our model.
- For building a Flask Application we needs HTML pages stored in the **templates** folder,CSS for styling the pages stored in the static folder and a python script **app.py** for server side scripting
- The IBM folder consists of a trained model notebook on IBM Cloud.
- Training folder consists of eye_disease_detection.ipynb model training file & adp.h5 is saved model

Milestone 1: Data Collection

There are many popular open sources for collecting the data. Eg: kaggle.com, UCI repository, etc.

Activity 1: Download the dataset

Collect images of Eye Diseases then organize into subdirectories based on their respective names asshown in the project structure. Create folders of types of Eye Diseases that need to be recognized.

In this project, we have collected images of 4 types of Eye Diseases images like Normal, cataract, Diabetic Retinopathy & Glaucoma and they are saved in the respective sub directories with their respective names.

You can download the dataset used in this project using the below link

Dataset:- https://www.kaggle.com/datasets/gunavenkatdoddi/eye-diseases-classification

Note: For better accuracy train on more images

We are going to build our training model on Google colab.

We will be connecting Kaggle with Google Colab because the dataset is too big to import using thefollowing code:

Data collection

```
[ ] !pip install -q kaggle

[ ] !mkdir ~/.kaggle

[ ] !cp kaggle.json ~/.kaggle/

[ ] !kaggle datasets download -d gunavenkatdoddi/eye-diseases-classification

    Warning: Your Kaggle API key is readable by other users on this system! To fix this, you can run 'chmod 600 /root/.kaggle/kaggle.json'
    Downloading eye-diseases-classification.zip to /content
     99% 729M/736M [00:06<00:00, 160MB/s]
    100% 736M/736M [00:06<00:00, 117MB/s]

[ ] !unzip /content/eye-diseases-classification.zip #unzip the dataset

    Archive:  /content/eye-diseases-classification.zip
      inflating: dataset/cataract/0_left.jpg
```

Activity 2: Create training and testing dataset

To build a DL model we have to split training and testing data into two separate folders. But in this project dataset folder training and testing folders are not present. So, in this case we have to separate the data into train & test folders.

```
[ ] !pip install split-folders

    Collecting split-folders
      Downloading split_folders-0.5.1-py3-none-any.whl (8.4 kB)
    Installing collected packages: split-folders
    Successfully installed split-folders-0.5.1

[ ] import splitfolders

[ ] splitfolders.ratio('/content/dataset', output="output",seed=1337, ratio=(0.8,0.2))

    Copying files: 4217 files [00:02, 1627.08 files/s]
```

Four different transfer learning models are used in our project and the best model (VGG19) is selected. The image input size of VGG19 model is 224, 224.

Assigning the input image size

```
[ ]  IMAGE_SIZE = [224,224]
```

Milestone 2: Image Preprocessing
In this milestone we will be improving the image data that suppresses unwilling distortions or enhances some image features important for further processing, although perform some geometric transformations of images like rotation, scaling, translation, etc.

Activity 1: Importing the libraries
Import the necessary libraries as shown in the image

Importing the necessary libraries

```
[ ]  import numpy as np
     import matplotlib.pyplot as plt
     import matplotlib.image as mpimg
     %matplotlib inline
     from tensorflow.keras.preprocessing.image import ImageDataGenerator
     from tensorflow.keras.preprocessing import image
     from PIL import ImageFile
     ImageFile.LOAD_TRUNCATED_IMAGES = True
```

Activity 2: Configure ImageDataGenerator class
ImageDataGenerator class is instantiated and the configuration for the types of data augmentation There are five main types of data augmentation techniques for image data; specifically:

- Image shifts via the width_shift_range and height_shift_range arguments.
- The image flips via the horizontal_flip and vertical_flip arguments.
- Image rotations via the rotation_range argument
- Image brightness via the brightness_range argument.
- Image zoom via the zoom_range argument.

An instance of the ImageDataGenerator class can be constructed for train and test.

Configuring the ImageDataGenerator

```
[ ]  train_datagen = ImageDataGenerator(rescale=1./255,
                                        shear_range = 0.2, zoom_range = 0.2,
                                        horizontal_flip = True)

     test_datagen = ImageDataGenerator(rescale = 1./255)
```

Activity 3: Apply ImageDataGenerator functionality to Train set and Test set

Let us apply ImageDataGenerator functionality to the Train set and Test set by using the following code. For
Training set using flow_from_directory function.
This function will return batches of images from the subdirectories
Arguments:

- directory: Directory where the data is located. If labels are "inferred", it should contain subdirectories, each containing images for a class. Otherwise, the directory structure is ignored.
- batch_size: Size of the batches of data which is 64.
- target_size: Size to resize images after they are read from disk.
- class_mode:

- ‘int': means that the labels are encoded as integers (e.g. for sparse_categorical_crossentropy loss).
- 'categorical' means that the labels are encoded as a categorical vector (e.g. for categorical_crossentropy
loss).
- 'binary' means that the labels (there can be only 2) are encoded as float32 scalars with values 0 or 1 (e.g.
for binary_crossentropy).
- None (no labels).

▾ Splitting the data into train and test set

```
train_data = train_datagen.flow_from_directory(
    '/content/output/train',
    target_size = (224,224),
    class_mode = 'categorical'
)

test_data = test_datagen.flow_from_directory('/content/output/val',
                          target_size = (224,224),
                          batch_size = 64,
                          class_mode = 'categorical')
```

```
Found 3372 images belonging to 4 classes.
Found 845 images belonging to 4 classes.
```

Total the dataset is having 3372 train images, 845 test images divided under 4 classes.

Milestone 3: Model Building

Now it's time to build our model. Let’s use the pre-trained model which is VGG19, one of the convolution neural net (CNN) architecture which is considered as a very good model for Image classification. Deep understanding on the VGG19 model – Link is referred to in the prior knowledge section. Kindly refer to it before starting the model building part.

Activity 1: Pre-trained CNN model as a Feature Extractor

For one of the models, we will use it as a simple feature extractor by freezing all the five convolution blocks to make sure their weights don’t get updated after each epoch as we train our own model.

Here, we have considered images of dimension (224,224,3). Also, we have assigned include_top = False because we are using convolution layer for features extraction and wants to train fully connected layer for our image classification (since it is not the part of Imagenet dataset)
Flatten layer flattens the input. Does not affect the batch size.

Model building

```
[ ] from tensorflow.keras.applications.vgg19 import VGG19, preprocess_input
    from tensorflow.keras.layers import Flatten, Dense
    from tensorflow.keras.models import Model
    from tensorflow.keras.models import load_model
```

```
[ ] VGG19 = VGG19(input_shape=IMAGE_SIZE + [3], weights='imagenet',include_top=False)
```

```
    Downloading data from https://storage.googleapis.com/tensorflow/keras-applications/vgg19/vgg19_weights_tf_dim_ordering_tf_kernels_notop.h5
    80134624/80134624 [==============================] - 0s 0us/step
```

```
[ ] for layers in VGG19.layers:
     layers.trainable = False
```

Activity 2: Adding Dense Layers

```
[ ] x = Flatten()(VGG19.output)
```

```
[ ] prediction = Dense(4, activation='softmax')(x)
```

```
[ ] model = Model(inputs=VGG19.inputs, outputs=prediction)
```

A dense layer is a deeply connected neural network layer. It is the most common and frequently used layer. Let us create a model object named model with inputs as VGG19.input and output as dense layer.
The number of neurons in the Dense layer is the same as the number of classes in the training set. The neurons in the last Dense layer, use softmax activation to convert their outputs into respective probabilities. Understanding the model is a very important phase to properly use it for training and prediction purposes.

Keras provides a simple method, summary to get the full information about the model and its layers.

```
model.summary()
```

```
Model: "model"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 input_1 (InputLayer)        [(None, 224, 224, 3)]     0

 block1_conv1 (Conv2D)       (None, 224, 224, 64)      1792

 block1_conv2 (Conv2D)       (None, 224, 224, 64)      36928

 block1_pool (MaxPooling2D)  (None, 112, 112, 64)      0

 block2_conv1 (Conv2D)       (None, 112, 112, 128)     73856

 block2_conv2 (Conv2D)       (None, 112, 112, 128)     147584

 block2_pool (MaxPooling2D)  (None, 56, 56, 128)       0

 block3_conv1 (Conv2D)       (None, 56, 56, 256)       295168

 block3_conv2 (Conv2D)       (None, 56, 56, 256)       590080

 block3_conv3 (Conv2D)       (None, 56, 56, 256)       590080

 block3_conv4 (Conv2D)       (None, 56, 56, 256)       590080

 block3_pool (MaxPooling2D)  (None, 28, 28, 256)       0

 block4_conv1 (Conv2D)       (None, 28, 28, 512)       1180160

 block4_conv2 (Conv2D)       (None, 28, 28, 512)       2359808

 block4_conv3 (Conv2D)       (None, 28, 28, 512)       2359808

 block4_conv4 (Conv2D)       (None, 28, 28, 512)       2359808

 block4_pool (MaxPooling2D)  (None, 14, 14, 512)       0

 block5_conv1 (Conv2D)       (None, 14, 14, 512)       2359808

 block5_conv2 (Conv2D)       (None, 14, 14, 512)       2359808

 block5_conv3 (Conv2D)       (None, 14, 14, 512)       2359808

 block5_conv4 (Conv2D)       (None, 14, 14, 512)       2359808

 block5_pool (MaxPooling2D)  (None, 7, 7, 512)         0

 flatten (Flatten)           (None, 25088)             0

 dense (Dense)               (None, 4)                 100356

=================================================================
Total params: 20124740 (76.77 MB)
Trainable params: 100356 (392.02 KB)
Non-trainable params: 20024384 (76.39 MB)
_________________________________________________________________
```

Activity 3: Configure the Learning Process

The compilation is the final step in creating a model. Once the compilation is done, we can move on to the training phase. The loss function is used to find errors or deviations in the learning process. Keras requires a loss function during the model compilation process.
Optimization is an important process that optimizes the input weights by comparing the prediction and the loss function. Here we are using adam optimizer.
Metrics are used to evaluate the performance of your model. It is similar to the loss function, but not used in the training process.

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

Activity 4: Train the model

Now, let us train our model with our image dataset. The model is trained for 50 epochs and after every epoch, the current model state is saved if the model has the least loss encountered till that time. We can see that the training loss decreases in almost every epoch and probably there is further scope to improve the model.fit functions used to train a deep learning neural network

Arguments:

- steps_per_epoch: it specifies the total number of steps taken from the generator as soon as one epoch is finished and the next epoch has started. We can calculate the value of steps_per_epoch as the total number of samples in your dataset divided by the batch size.
- Epochs: an integer and number of epochs we want to train our model for.
- validation_data can be either:

- an inputs and targets list
- a generator
- an inputs, targets, and sample_weights list which can be used to
evaluate the loss and metrics for any model after any epoch has ended.

- validation_steps: only if the validation_data is a generator then only this argument can be used. It specifies the total number of steps taken from the generator before it is stopped at every epoch and its value is calculated as the total number of validation data points in your dataset divided by the validation batch size.

```
model.fit(
    train_data,
    validation_data=test_data,
    epochs=60,
    steps_per_epoch=len(train_data),
    validation_steps=len(test_data)
)#fitting the model
```

```
Epoch 1/60
106/106 [==============================] - 72s 676ms/step - loss: 0.4227 - accuracy: 0.8449 - val_loss: 0.4689 - val_accuracy: 0.8462
Epoch 2/60
106/106 [==============================] - 71s 670ms/step - loss: 0.3472 - accuracy: 0.8689 - val_loss: 0.3975 - val_accuracy: 0.8651
Epoch 3/60
106/106 [==============================] - 70s 661ms/step - loss: 0.3710 - accuracy: 0.8597 - val_loss: 0.5094 - val_accuracy: 0.8438
Epoch 4/60
106/106 [==============================] - 70s 660ms/step - loss: 0.3465 - accuracy: 0.8663 - val_loss: 0.4700 - val_accuracy: 0.8450
Epoch 5/60
106/106 [==============================] - 71s 669ms/step - loss: 0.3075 - accuracy: 0.8766 - val_loss: 0.6177 - val_accuracy: 0.8107
Epoch 6/60
106/106 [==============================] - 71s 665ms/step - loss: 0.3346 - accuracy: 0.8775 - val_loss: 0.5336 - val_accuracy: 0.8260
Epoch 7/60
106/106 [==============================] - 70s 666ms/step - loss: 0.3585 - accuracy: 0.8704 - val_loss: 0.4280 - val_accuracy: 0.8402
Epoch 8/60
106/106 [==============================] - 70s 659ms/step - loss: 0.3236 - accuracy: 0.8799 - val_loss: 0.5322 - val_accuracy: 0.8189
Epoch 9/60
106/106 [==============================] - 71s 671ms/step - loss: 0.3498 - accuracy: 0.8639 - val_loss: 0.4711 - val_accuracy: 0.8355
Epoch 10/60
106/106 [==============================] - 70s 665ms/step - loss: 0.3198 - accuracy: 0.8772 - val_loss: 0.4029 - val_accuracy: 0.8568
Epoch 11/60
106/106 [==============================] - 70s 659ms/step - loss: 0.3151 - accuracy: 0.8832 - val_loss: 0.4781 - val_accuracy: 0.8568
Epoch 12/60
106/106 [==============================] - 70s 659ms/step - loss: 0.3189 - accuracy: 0.8855 - val_loss: 0.7017 - val_accuracy: 0.8036
Epoch 13/60
106/106 [==============================] - 72s 674ms/step - loss: 0.3219 - accuracy: 0.8740 - val_loss: 0.3899 - val_accuracy: 0.8805
Epoch 14/60
106/106 [==============================] - 75s 706ms/step - loss: 0.3600 - accuracy: 0.8704 - val_loss: 0.6031 - val_accuracy: 0.8083
Epoch 15/60
106/106 [==============================] - 71s 667ms/step - loss: 0.3802 - accuracy: 0.8597 - val_loss: 0.5647 - val_accuracy: 0.8142
Epoch 16/60
106/106 [==============================] - 70s 659ms/step - loss: 0.3057 - accuracy: 0.8867 - val_loss: 0.5745 - val_accuracy: 0.8083
Epoch 17/60
106/106 [==============================] - 70s 663ms/step - loss: 0.2868 - accuracy: 0.8867 - val_loss: 0.7122 - val_accuracy: 0.7598
Epoch 18/60
106/106 [==============================] - 71s 671ms/step - loss: 0.3039 - accuracy: 0.8861 - val_loss: 0.3279 - val_accuracy: 0.8911
Epoch 19/60
106/106 [==============================] - 70s 663ms/step - loss: 0.2534 - accuracy: 0.9054 - val_loss: 0.5161 - val_accuracy: 0.8178
Epoch 20/60
106/106 [==============================] - 71s 670ms/step - loss: 0.2937 - accuracy: 0.8876 - val_loss: 0.4889 - val_accuracy: 0.8320
Epoch 21/60
106/106 [==============================] - 71s 672ms/step - loss: 0.2963 - accuracy: 0.8858 - val_loss: 0.3723 - val_accuracy: 0.8757
Epoch 22/60
106/106 [==============================] - 72s 678ms/step - loss: 0.2974 - accuracy: 0.8879 - val_loss: 0.4868 - val_accuracy: 0.8379
Epoch 23/60
106/106 [==============================] - 70s 662ms/step - loss: 0.2741 - accuracy: 0.8953 - val_loss: 0.3810 - val_accuracy: 0.8876
Epoch 24/60
106/106 [==============================] - 71s 668ms/step - loss: 0.3248 - accuracy: 0.8772 - val_loss: 0.3212 - val_accuracy: 0.8982
Epoch 25/60
106/106 [==============================] - 70s 664ms/step - loss: 0.3186 - accuracy: 0.8817 - val_loss: 0.6580 - val_accuracy: 0.7846
Epoch 26/60
106/106 [==============================] - 72s 677ms/step - loss: 0.3115 - accuracy: 0.8820 - val_loss: 0.6820 - val_accuracy: 0.7598
Epoch 27/60
106/106 [==============================] - 71s 666ms/step - loss: 0.2752 - accuracy: 0.8974 - val_loss: 0.3571 - val_accuracy: 0.8852
Epoch 28/60
106/106 [==============================] - 71s 670ms/step - loss: 0.3702 - accuracy: 0.8710 - val_loss: 0.3606 - val_accuracy: 0.8757
Epoch 29/60
106/106 [==============================] - 72s 682ms/step - loss: 0.2546 - accuracy: 0.8986 - val_loss: 0.5600 - val_accuracy: 0.8225
```

```
Epoch 33/60
106/106 [==============================] - 71s 672ms/step - loss: 0.2671 - accuracy: 0.8950 - val_loss: 0.5622 - val_accuracy: 0.7964
Epoch 34/60
106/106 [==============================] - 71s 667ms/step - loss: 0.3161 - accuracy: 0.8852 - val_loss: 0.6234 - val_accuracy: 0.8036
Epoch 35/60
106/106 [==============================] - 71s 670ms/step - loss: 0.2863 - accuracy: 0.8906 - val_loss: 0.4846 - val_accuracy: 0.8509
Epoch 36/60
106/106 [==============================] - 73s 687ms/step - loss: 0.2909 - accuracy: 0.8918 - val_loss: 0.7406 - val_accuracy: 0.7740
Epoch 37/60
106/106 [==============================] - 71s 668ms/step - loss: 0.2942 - accuracy: 0.8918 - val_loss: 0.4118 - val_accuracy: 0.8746
Epoch 38/60
106/106 [==============================] - 71s 674ms/step - loss: 0.2701 - accuracy: 0.8947 - val_loss: 0.5307 - val_accuracy: 0.8533
Epoch 39/60
106/106 [==============================] - 70s 663ms/step - loss: 0.2518 - accuracy: 0.9051 - val_loss: 0.6099 - val_accuracy: 0.8095
Epoch 40/60
106/106 [==============================] - 72s 675ms/step - loss: 0.2465 - accuracy: 0.8998 - val_loss: 0.3450 - val_accuracy: 0.8947
Epoch 41/60
106/106 [==============================] - 71s 667ms/step - loss: 0.2295 - accuracy: 0.9081 - val_loss: 0.4681 - val_accuracy: 0.8426
Epoch 42/60
106/106 [==============================] - 71s 669ms/step - loss: 0.2542 - accuracy: 0.8989 - val_loss: 0.5440 - val_accuracy: 0.8284
Epoch 43/60
106/106 [==============================] - 74s 702ms/step - loss: 0.2366 - accuracy: 0.9084 - val_loss: 0.7972 - val_accuracy: 0.7704
Epoch 44/60
106/106 [==============================] - 71s 665ms/step - loss: 0.2451 - accuracy: 0.9012 - val_loss: 0.7485 - val_accuracy: 0.7740
Epoch 45/60
106/106 [==============================] - 74s 700ms/step - loss: 0.2457 - accuracy: 0.9095 - val_loss: 0.3772 - val_accuracy: 0.8734
Epoch 46/60
106/106 [==============================] - 75s 702ms/step - loss: 0.2658 - accuracy: 0.8968 - val_loss: 0.5034 - val_accuracy: 0.8391
Epoch 47/60
106/106 [==============================] - 73s 687ms/step - loss: 0.2424 - accuracy: 0.9039 - val_loss: 0.5629 - val_accuracy: 0.8166
Epoch 48/60
106/106 [==============================] - 71s 674ms/step - loss: 0.2973 - accuracy: 0.8923 - val_loss: 0.4298 - val_accuracy: 0.8793
Epoch 49/60
106/106 [==============================] - 70s 659ms/step - loss: 0.2643 - accuracy: 0.9007 - val_loss: 0.3938 - val_accuracy: 0.8840
Epoch 50/60
106/106 [==============================] - 72s 678ms/step - loss: 0.2386 - accuracy: 0.9119 - val_loss: 0.5152 - val_accuracy: 0.8414
Epoch 51/60
106/106 [==============================] - 73s 691ms/step - loss: 0.2625 - accuracy: 0.8950 - val_loss: 0.3649 - val_accuracy: 0.8935
Epoch 52/60
106/106 [==============================] - 70s 658ms/step - loss: 0.2606 - accuracy: 0.9012 - val_loss: 0.4582 - val_accuracy: 0.8355
Epoch 53/60
106/106 [==============================] - 70s 664ms/step - loss: 0.2500 - accuracy: 0.9015 - val_loss: 0.4878 - val_accuracy: 0.8580
Epoch 54/60
106/106 [==============================] - 71s 670ms/step - loss: 0.2929 - accuracy: 0.8870 - val_loss: 0.4236 - val_accuracy: 0.8722
Epoch 55/60
106/106 [==============================] - 72s 676ms/step - loss: 0.2344 - accuracy: 0.9125 - val_loss: 0.5679 - val_accuracy: 0.8272
Epoch 56/60
106/106 [==============================] - 70s 663ms/step - loss: 0.2506 - accuracy: 0.9018 - val_loss: 0.4533 - val_accuracy: 0.8402
Epoch 57/60
106/106 [==============================] - 71s 673ms/step - loss: 0.2330 - accuracy: 0.9116 - val_loss: 0.3510 - val_accuracy: 0.8852
Epoch 58/60
106/106 [==============================] - 91s 862ms/step - loss: 0.2471 - accuracy: 0.9048 - val_loss: 0.5656 - val_accuracy: 0.8189
Epoch 59/60
106/106 [==============================] - 71s 663ms/step - loss: 0.2429 - accuracy: 0.9057 - val_loss: 0.4903 - val_accuracy: 0.8663
Epoch 60/60
106/106 [==============================] - 76s 719ms/step - loss: 0.2220 - accuracy: 0.9158 - val_loss: 0.5024 - val_accuracy: 0.8615
<keras.src.callbacks.History at 0x78ece16cb100>
```

Activity 5: Save the Model

Out of all the models we tried (CNN, VGG19, Resnet50 V2, Inception V3 & Xception) VGG19 gave us the best accuracy.

```
[ ]  model.save('/content/drive/MyDrive/Eye Disease Detection/model.h5')
```

So we are saving VGG19 as our final mode.
The model is saved with .h5 extension as follows
An H5 file is a data file saved in the Hierarchical Data Format (HDF). It contains multidimensional arrays of scientific data.

Testing the model:

Evaluation is a process during the development of the model to check whether the model is the best fit for the given problem and corresponding data. Load the saved model using load_model.

```
[ ]  from tensorflow.keras.models import load_model
     model = load_model('/content/drive/MyDrive/Eye Disease Detection/model.h5')
```

Taking an image as input and checking the results

```
[12] import numpy as np
     from tensorflow.keras.preprocessing import image
     img = image.load_img(r"/content/drive/MyDrive/Eye Disease Detection/Test images/Glaucoma 4.jpeg",target_size = (224,224)) #loading the model
     x = image.img_to_array(img) #image to array
     x = np.expand_dims(x,axis = 0)#changing the shape
     preds = model.predict(x)
     pred=np.argmax(preds,axis=1)
     index=['cataract','diabetic_retinopathy','glaucoma','normal']
     result=str(index[pred[0]])
     confidence_score=max(preds[0][pred])
     print("Predicted result is ",result)
     print("Confidence Score : ",confidence_score)
```

```
1/1 [==============================] - 0s 26ms/step
Predicted result is  glaucoma
Confidence Score :  1.0
```

So our model has predicted the label correctly as Glaucoma.

Milestone 4: Application Building

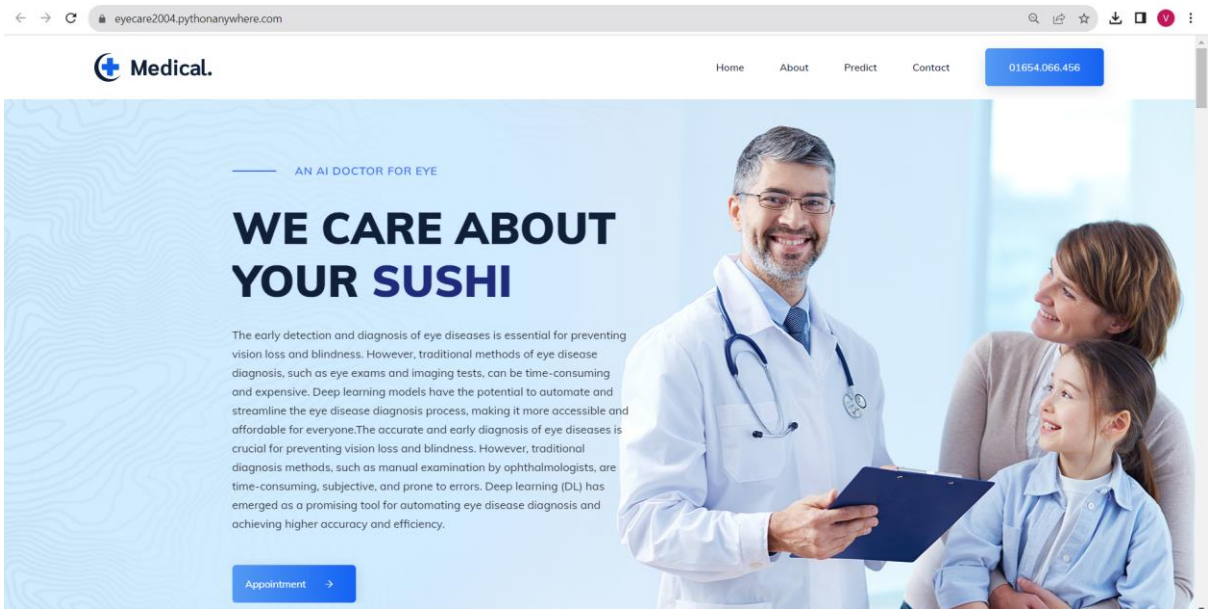
In this section, we will be building a web application that is integrated to the model we built. A UI is provided for the uses where he has to enter the values for predictions. The enter values are given to the
saved model and prediction is showcased on the UI.
This section has the following tasks

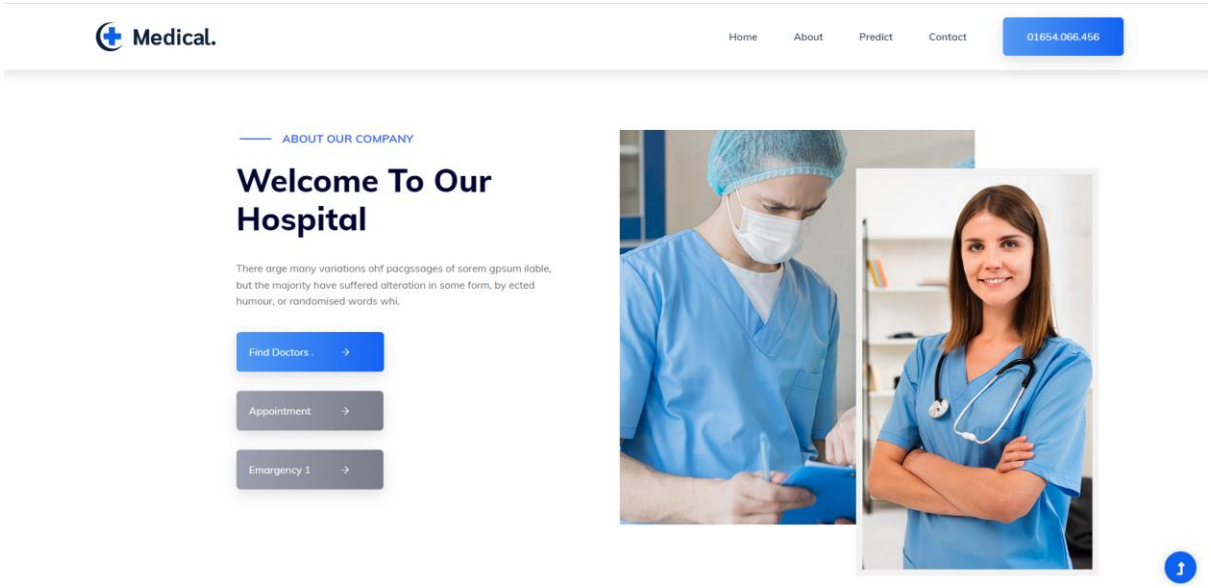
- Building HTML Pages
- Building python code
- Run the programme

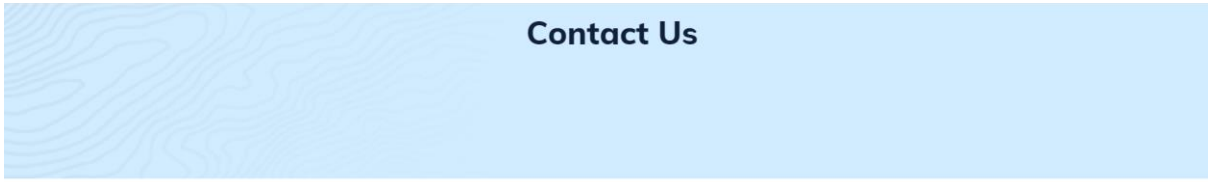
Activity1: Building HTML Pages:
For this project create one HTML file namely

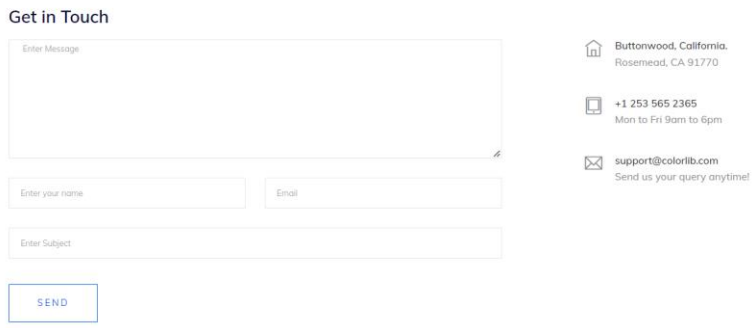
- index.html

Let's see how our index.html page looks like:
predict section









Activity 2: Build Python code:

Import the libraries

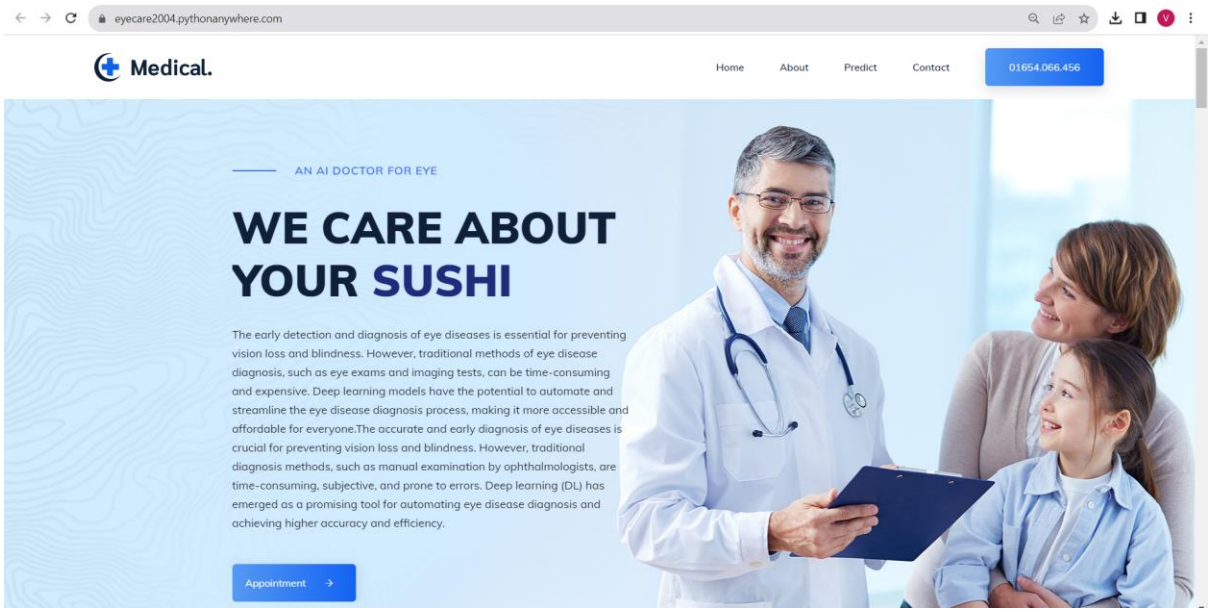
```python
from flask_restx import Api, Namespace, Resource
from predict import classify_using_bytes
from flask import request

api = Api(
    version="1.0",
    title="Eye Disease Predictor",
    description="This api help to predict the eye disease of an infected eye using its image",
    doc="/api",
    validate=True
)

requestCtrl = Namespace("Prediction", "Send an image file in post to predict.....!!!", path="/a

@requestCtrl.route("/")
class RequestController(Resource):
    def get(self):
        return {'hello' : "world"}

    def post(self):
        image = request.files.get("image")
        result = classify_using_bytes(image.read(), "models/model.h5")
        return result

api.add_namespace(requestCtrl)
```

```python
from flask import Flask, render_template

app = Flask(__name__)

@app.route("/")
def index():
    return render_template("index.html")

@app.route("/about")
def about():
    return render_template("about.html")

@app.route("/predict")
def predict():
    return render_template("predict.html")

@app.route("/contact")
def contact():
    return render_template("contact.html")

from Api import api
api.init_app(app)

if __name__ == "__main__":
    app.run(debug=True)
```

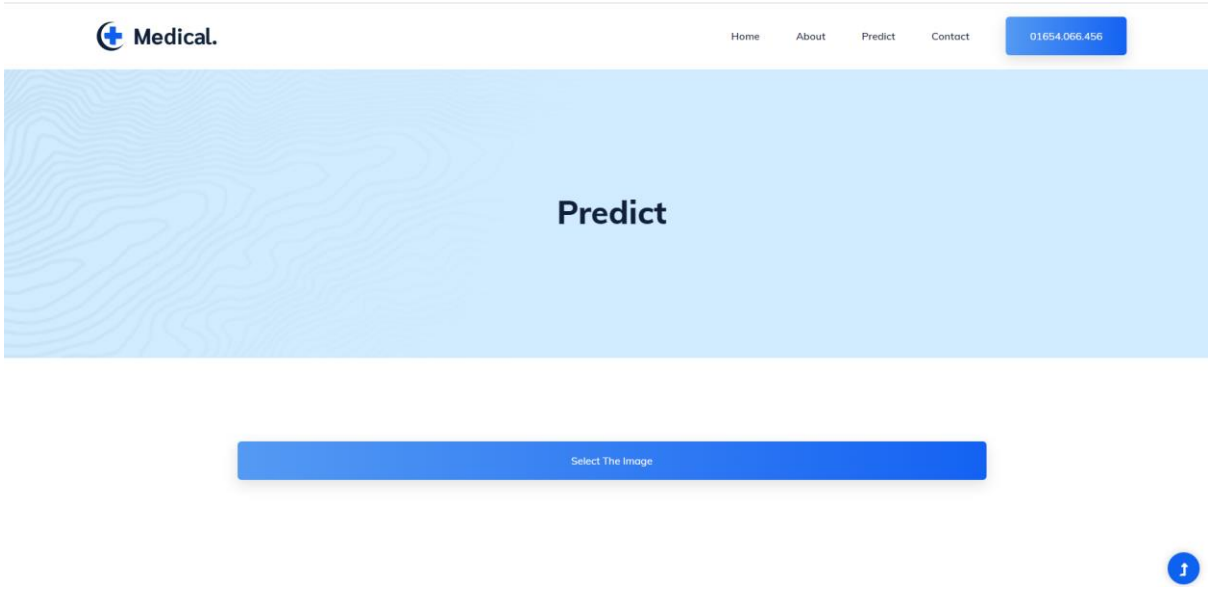
```python
from tensorflow.keras.preprocessing import image
from tensorflow.keras.models import load_model
from PIL import Image, ImageOps
import tensorflow as tf
import numpy as np

class_names=['cataract','diabetic_retinopathy','glaucoma','normal']

def predict_disease(img_path, model_path, class_names=class_names):
    model = load_model(model_path, compile=False)
    model.compile(loss='categorical_crossentropy',
                optimizer='adam',
                metrics=['accuracy'])

    img = image.load_img(img_path,target_size = (224,224))

    x = image.img_to_array(img)
    x = np.expand_dims(x,axis = 0)

    prediction = model.predict(x)
    pred = np.argmax(prediction,axis = 1)

    disease = str(class_names[pred[0]])
    confidence_score = max(prediction[0][pred])

    print("Predicted result : ",disease)
    print("Confidence Score : ",confidence_score)

    result = {
        'disease' : disease,
        'score' : confidence_score
    }

    return result
```

Activity 3: Run the application

The home page looks like this. When you click on the Predict button, you'll be redirected to the predict section

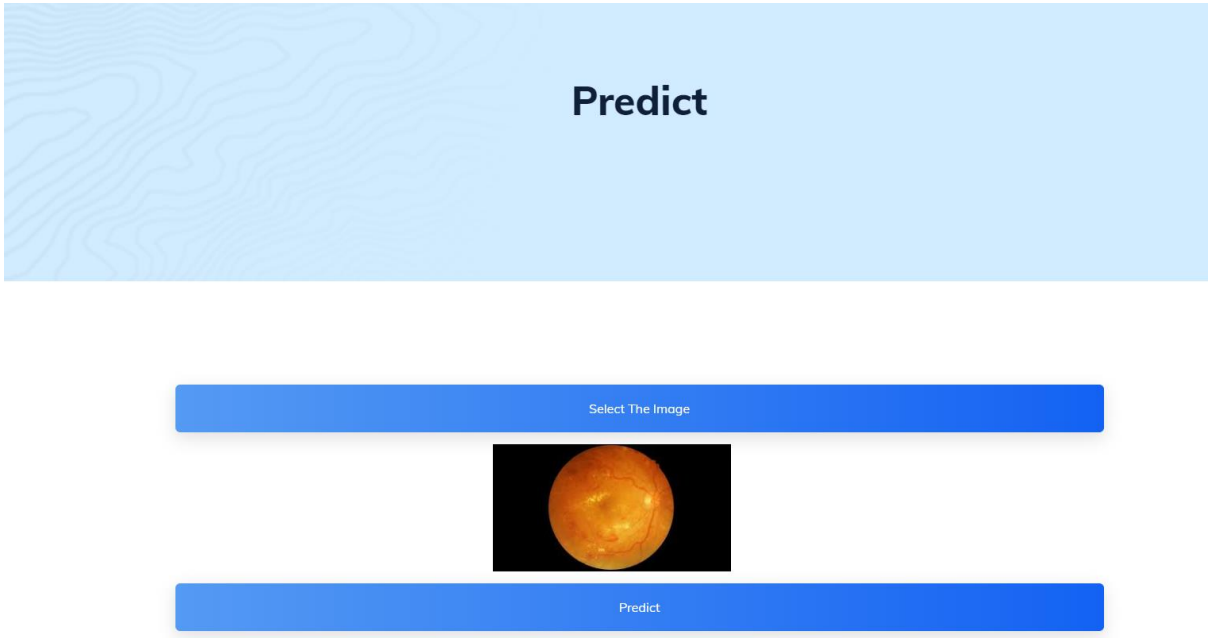


Prediction page:

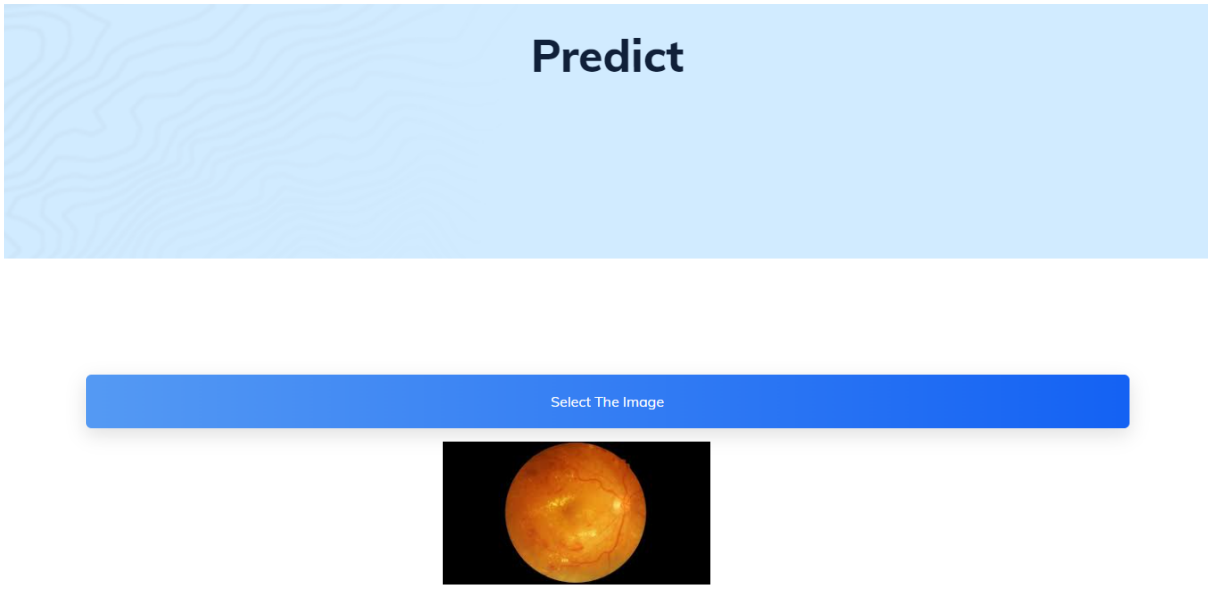


Once you upload the image and click on Predict button, the output will be displayed in the below page.

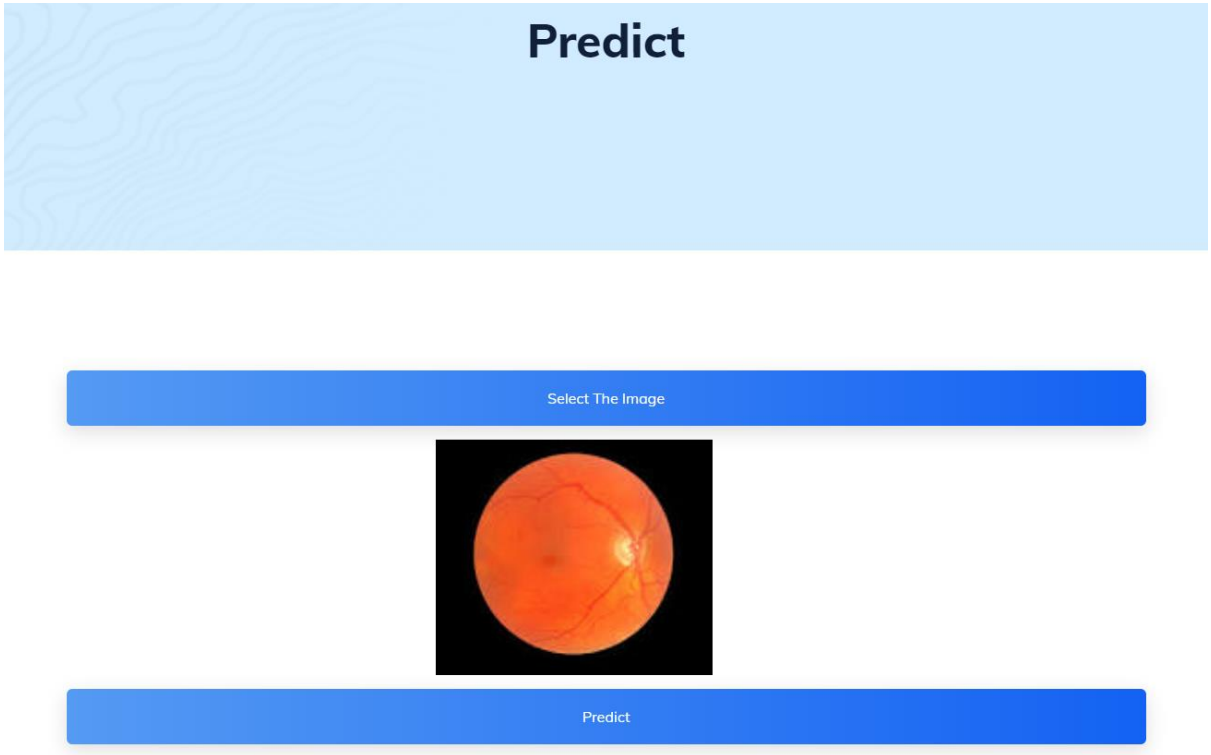
Input 1:



Output 1:



Input 2:



Output 2: