Deep Learning Fundus Image Analysis For Early Detection Of Diabetic Retinopathy

Project description:

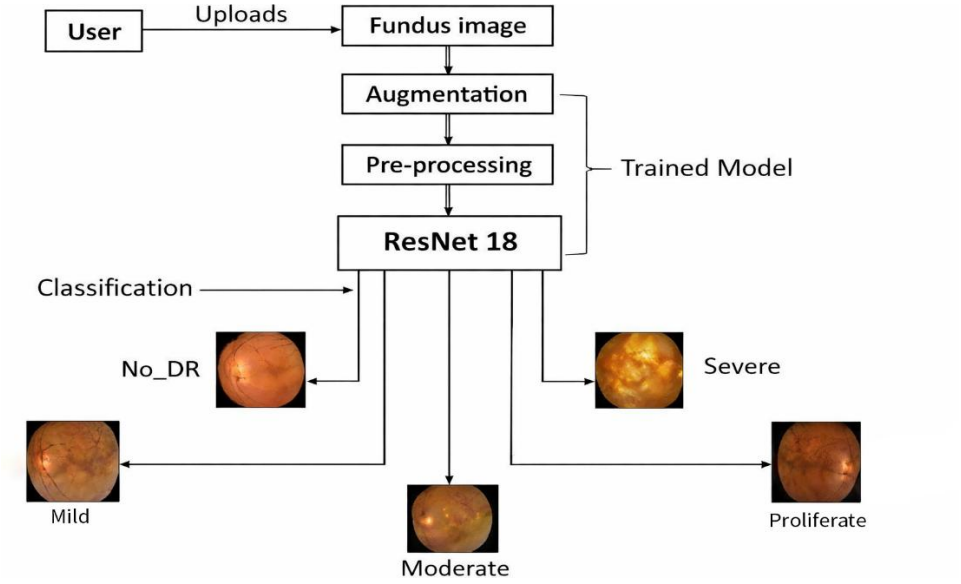
One of the most critical challenges in modern healthcare is the early detection of Diabetic Retinopathy (DR) - a diabetes-related eye condition that affects the retina and is a leading cause of blindness worldwide. As diabetes prevalence continues to rise globally, DR has become a major public health concern requiring scalable screening solutions.

"The manual screening process by ophthalmologists is time-consuming, expensive, and prone to human error - especially in early stages where subtle retinal changes like microaneurysms are barely visible. Early detection through automated analysis can prevent 90% of vision loss cases."

This project develops an AI-powered screening system using transfer learning with Xception CNN architecture to automatically detect and classify 5 stages of Diabetic Retinopathy from fundus images. From model training to production deployment, this end-to-end solution transforms DR screening from manual expert analysis to automated, accessible healthcare technology - enabling early intervention that saves vision and reduces healthcare costs dramatically.

We will be using transfer learning with Xception CNN, ImageDataGenerator for augmentation, Flask web framework for deployment, and SQLite for user management. The trained model is saved in .h5 format and integrated with Flask application for real-time predictions. We train and test the model on a comprehensive dataset containing training and test images across all DR severity levels.

Technical Architecture:



Pre-Requisites:

To complete this project, you must required following software's, concepts and packages

- **Python 3.8+** - Programming language
- **Anaconda Navigator** - Python distribution and environment manager
- **PyCharm or Spyder** - IDE for development
    - PyCharm Installation: https://youtu.be/1ra4zH2G400
    - Spyder Installation: https://youtu.be/5mDYijMfSzs
- **Python packages:**
    - Open anaconda prompt as administrator
    - Type "pip install numpy" and click enter.
    - Type "pip install pandas" and click enter.
    - Type "pip install tensorflow" and click enter.
    - Type "2.3.2pip install keras 2.3.1" and click enter.
    - Type "pip install flask" and click enter.
    - Type "pip install werkzeug" and click enter.
    - Type "pip install opencv-python" and click enter.
    - Type "pip install matplotlib" and click enter.
    - Type "pip install scikit-learn" and click enter.

Prior Knowledge :

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

Deep Learning Concepts: https://towardsdatascience.com/basics-of-the-classic-cnn-a3dce1225add

- **CNN Architecture**: Understanding convolutional neural networks
- **Transfer Learning Models**:
    - VGG16: https://medium.com/@mygreatlearning/what-is-vgg16-introduction-to-vgg16-f2d63849f615
    - ResNet-50: 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 Web Framework**: https://www.youtube.com/watch?v=lj4I_CvBnt0
- **Image Processing**: Basic knowledge of image formats and preprocessing
- **Database Concepts**: SQL and SQLite basics

Project Objectives:

By the end of this project, you will:

- Understand fundamental concepts and techniques of transfer learning using pre-trained models like Xception
- Gain proficiency in image data preprocessing and augmentation for deep learning
- Master data loading and preparation for medical imaging datasets
- Build and train deep learning models using TensorFlow/Keras
- Integrate trained models with Flask web applications
- Deploy a complete end-to-end machine learning web application

Project Flow:

The complete project execution follows these steps:

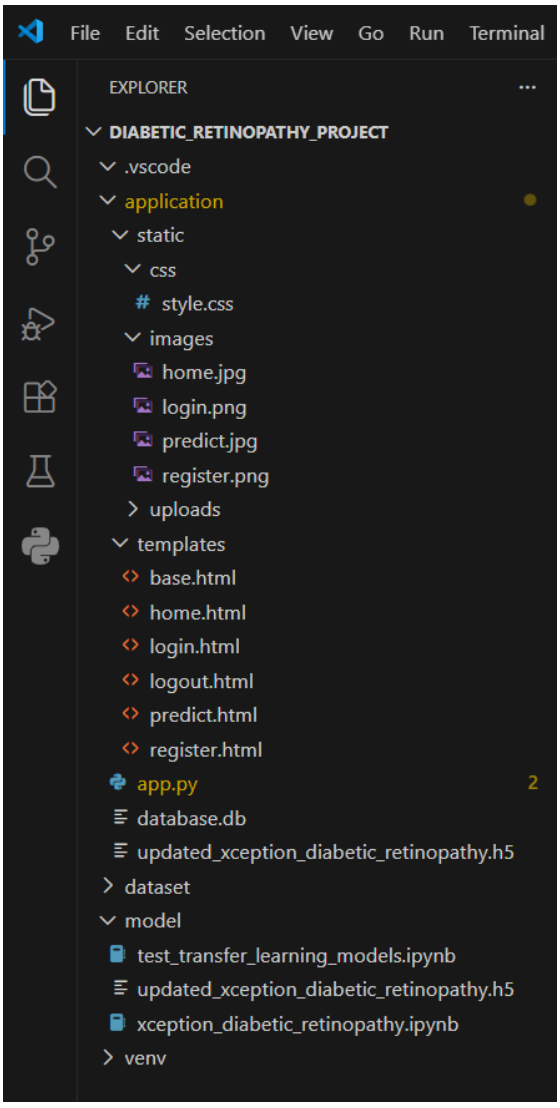
1. **User Interaction:** User accesses the Flask web application and registers/logs in
2. **Image Upload:** User uploads a fundus retinal image
3. **Image Preprocessing:** Image is resized, normalized, and prepared for model input
4. **Model Analysis:** Xception model analyzes the preprocessed image
5. **Classification:** Model predicts the DR stage
6. **Result Display:** Prediction results are displayed on the user interface

Project Workflow Tasks:

- Data Collection and Preparation
- Dataset Organization (Training/Testing Split)
- Image Preprocessing and Augmentation
- Import Required Libraries
- Configure ImageDataGenerator
- Apply Data Augmentation to Training and Testing Sets
- Model Building with Xception
- Transfer Learning Feature Extraction
- Adding Dense Layers for Classification
- Configure Learning Process (Optimizer, Loss, Metrics)
- Train the Model
- Evaluate Model Performance

- Save the Trained Model
- Application Building with Flask
- Create HTML Templates
- Build Python Flask Application
- User Authentication (Registration and Login)
- Integrate Prediction Model
- Deploy and Test Application

Project Structure:



Folder Descriptions:

- **templates/:** Contains HTML files for the Flask web interface (login, registration, prediction pages)

- **static/:** Contains CSS styling, images, and upload folder for user-submitted images
- **model/:** Contains Jupyter notebooks for model training and the saved Xception model (.h5 file)
- **dataset/:** Contains preprocessed training and testing dataset folders
- **app.py:** Main Flask application script
- **database.db:** SQLite database for user credentials and data storage
- **venv/:** Python virtual environment

Diabetic Retinopathy Classification Stages:

The model classifies fundus images into 5 different stages of Diabetic Retinopathy:

Classification	Description
No disease visible	Healthy retina with no signs of diabetic retinopathy
Mild NPDR	Localized swelling of small blood vessels in the retina (microaneurysms)
Moderate NPDR	Mild NPDR plus small bleeds (dot and blot hemorrhages), leaks (hard exudates) or closure (cotton wool spots) of small blood vessels
Severe NPDR	Moderate NPDR plus further damage to blood vessels (interretinal hemorrhages, venous beading, intraretinal microvascular abnormalities)
Proliferative DR	New vessel formation or vitreous/preretinal hemorrhage or tractional retinal detachment

Table 1: DR Classification Categories

Milestone 1: Data Collection

Activity 1: Download the dataset

The first activity of this milestone involved collecting a suitable dataset required for training the deep learning model. The dataset used in this project is the ***Diabetic Retinopathy Level Detection*** dataset obtained from Kaggle.

Kaggle is a widely recognized platform that provides high-quality datasets for machine learning projects. The dataset contains preprocessed fundus retinal images categorized into five severity levels of Diabetic Retinopathy.

The dataset was downloaded in compressed format and extracted into the project directory. After extraction, the folder structure was examined carefully to ensure that images were

properly classified according to their respective labels. Any corrupted or incomplete image files were identified and removed to maintain data integrity.

This activity ensured that a clean and structured dataset was ready for preprocessing and training.

Please refer to the link given below to download the dataset.

Link:

https://www.kaggle.com/datasets/arbethi/diabetic-retinopathy-level-detection

Milestone 2: Data Preprocessing and Image Augmentation

Activity 1: Importing the libraries

The first step in preprocessing was importing all necessary Python libraries required for image processing, model building, and data handling.

Libraries such as NumPy and Pandas were used for numerical and data manipulation tasks. TensorFlow and Keras were imported for building and training the deep learning model. OpenCV and Matplotlib were used for image visualization and preprocessing.

The ImageDataGenerator module from Keras was imported specifically for performing data augmentation. Proper library setup ensured smooth execution of all subsequent steps.

Import all necessary libraries for data processing and model building:

```python
import tensorflow as tf
from tensorflow.keras.preprocessing.image import ImageDataGenerator
from tensorflow.keras.applications import Xception
from tensorflow.keras.layers import Dense, Flatten, Dropout
from tensorflow.keras.models import Model
from tensorflow.keras.optimizers import Adam
```

Python

Activity 2:Image Data Generator Configuration

In this activity, the ImageDataGenerator was configured to perform both preprocessing and augmentation tasks. Since medical images may vary in brightness, angle, and orientation, augmentation techniques were applied to improve the model's ability to generalize.

The images were rescaled to normalize pixel values between 0 and 1. Additional augmentation techniques such as rotation, zooming, width shifting, height shifting, shearing, and horizontal flipping were applied. These transformations generate multiple variations of the same image, effectively increasing the size and diversity of the dataset. This reduces overfitting and enhances model robustness. This reduces overfitting and enhances model robustness.

```python
train_gen = ImageDataGenerator(
    rescale=1./255,
    rotation_range=20,
    zoom_range=0.2,
    horizontal_flip=True
)

test_gen = ImageDataGenerator(rescale=1./255)

train_data = train_gen.flow_from_directory(
    train_path,
    target_size=(299,299),
    batch_size=32,
    class_mode='categorical'
)

test_data = test_gen.flow_from_directory(
    test_path,
    target_size=(299,299),
    batch_size=32,
    class_mode='categorical'
)
```

Python

Milestone 3: Model Building

Model Architecture

The project uses the Xception pre-trained model as a feature extractor. Xception is a state-of-the-art CNN architecture that has been pre-trained on the ImageNet dataset.

Activity 1: Xception Feature Extractor

In this activity, the Xception model was loaded as the base architecture for transfer learning. Xception is a powerful convolutional neural network that has been pre-trained on the ImageNet dataset. The top classification layer of Xception was removed so that it could function as a feature extractor.

The convolutional layers were frozen initially to preserve the pre-learned features such as edges, shapes, and textures. This approach significantly reduces training time and improves performance, especially when working with medical imaging datasets that may not be extremely large.

```python
base_model = Xception(
    weights='imagenet',
    include_top=False,
    input_shape=(299,299,3)
)

for layer in base_model.layers:
    layer.trainable = False
```

Activity 2: Adding Custom Layers

After setting up the Xception base model, custom classification layers were added on top of it. A Global Average Pooling layer was used to reduce the spatial dimensions of the

feature maps. Then, fully connected Dense layers were added to learn complex patterns specific to Diabetic Retinopathy detection. A Dropout layer was included to randomly deactivate neurons during training, which helps prevent overfitting.

Finally, a Dense output layer with Softmax activation was added to classify images into five DR stages. This customization adapts the general Xception model to the specific requirements of this medical classification task.

```
x = Flatten()(base_model.output)
x = Dense(256, activation='relu')(x)
x = Dropout(0.5)(x)
output = Dense(5, activation='softmax')(x)

model = Model(inputs=base_model.input, outputs=output)
```

Activity 3: Compiling the Model

Once the architecture was finalized, the model was compiled. The Adam optimizer was selected because of its efficiency in handling sparse gradients and adaptive learning rates. The categorical cross-entropy loss function was used since the problem involves multi-class classification. Accuracy was selected as the evaluation metric to measure the percentage of correctly classified images. Compiling the model prepared it for the training phase.

```
model.compile(
    optimizer=Adam(learning_rate=0.0001),
    loss='categorical_crossentropy',
    metrics=['accuracy']
)
```

Milestone 4: Model Training

Train the model using the prepared training and testing datasets:

Activity 1:Train the Model

The prepared model was trained using the augmented training dataset for 30 epochs. During training, the model learned to adjust its weights to minimize the loss function. Both training accuracy and validation accuracy were monitored in each epoch. The training process showed steady improvement, with accuracy increasing and loss decreasing over time.

The validation dataset was used to evaluate how well the model generalized to unseen data. By the final epoch, the model achieved approximately 77% training accuracy and around 78% validation accuracy, indicating good learning performance.

```
history = model.fit(
    train_data,
    epochs=30,
    validation_data=test_data
)
```

Training Results

The model was trained for 30 epochs on the training dataset with validation on the testing dataset.

Training Performance Summary:

Epoch	Loss	Accuracy	Val Loss	Val Accuracy
1	1.2845	0.5132	1.0234	0.5876
10	0.6760	0.7425	0.6101	0.7892
20	0.6245	0.7534	0.5674	0.7956
30	0.5905	0.7755	0.5871	0.7834

Table 2: Model Training Performance Across Epochs

Activity 2: Saving the Model

After completing training and evaluation, the model was saved in .h5 format. Saving the model allows it to be reused later without retraining. The saved model file was stored inside the project's model directory. This activity ensures that the trained model can be integrated into the Flask web application for real-time predictions.

```
model.save("updated_xception_diabetic_retinopathy.h5")
```
Python

Milestone 5: Application Building with Flask

Activity 1: Building Html pages:

- home.html
- register.html
- login.html
- predict.html
- logout.html

In this activity, multiple frontend web pages were developed using HTML and CSS. The home page provides an overview of the project and navigation options. The register page allows new users to create accounts. The login page enables authentication using stored credentials. The prediction page provides an interface for uploading retinal images and

displaying prediction results. The logout page allows users to securely exit the session. These pages collectively create a user-friendly interface for interacting with the model.

The home page provides an overview of the project and navigation options:

Diabetic Retinopathy Classification

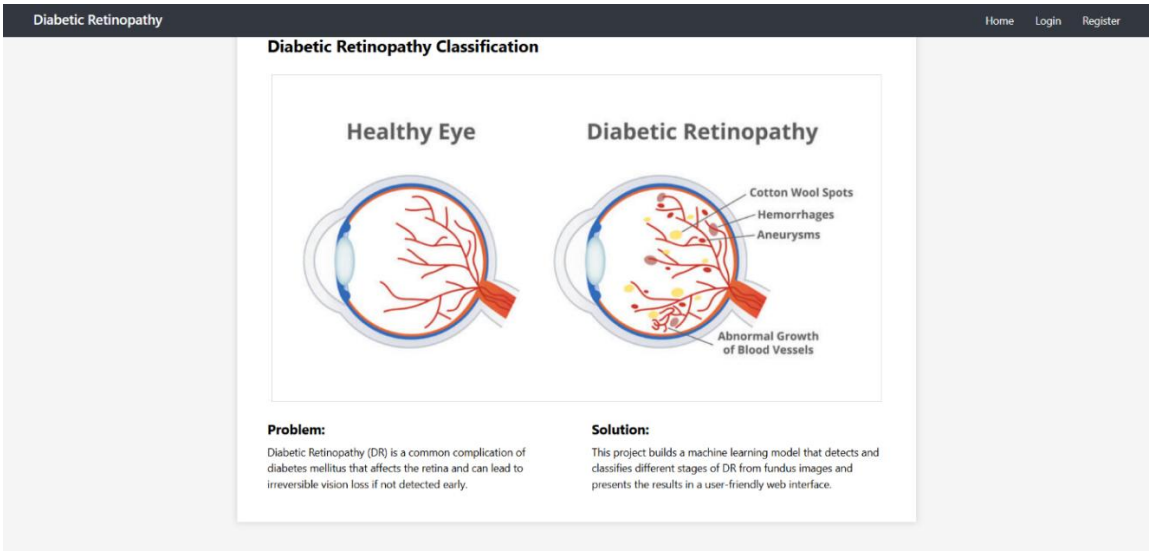
Home

Login

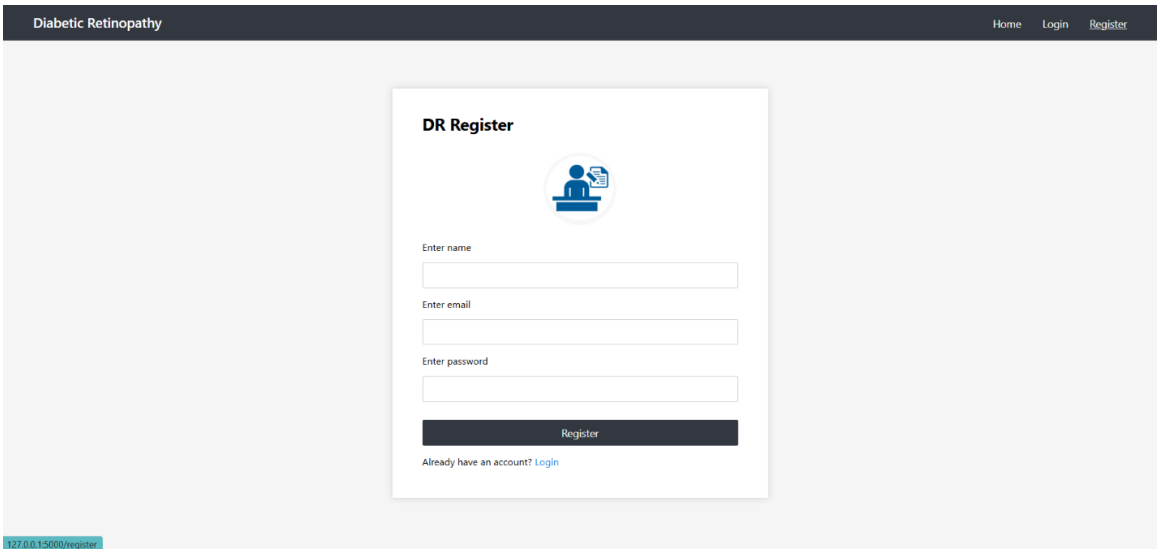
Register

Prediction

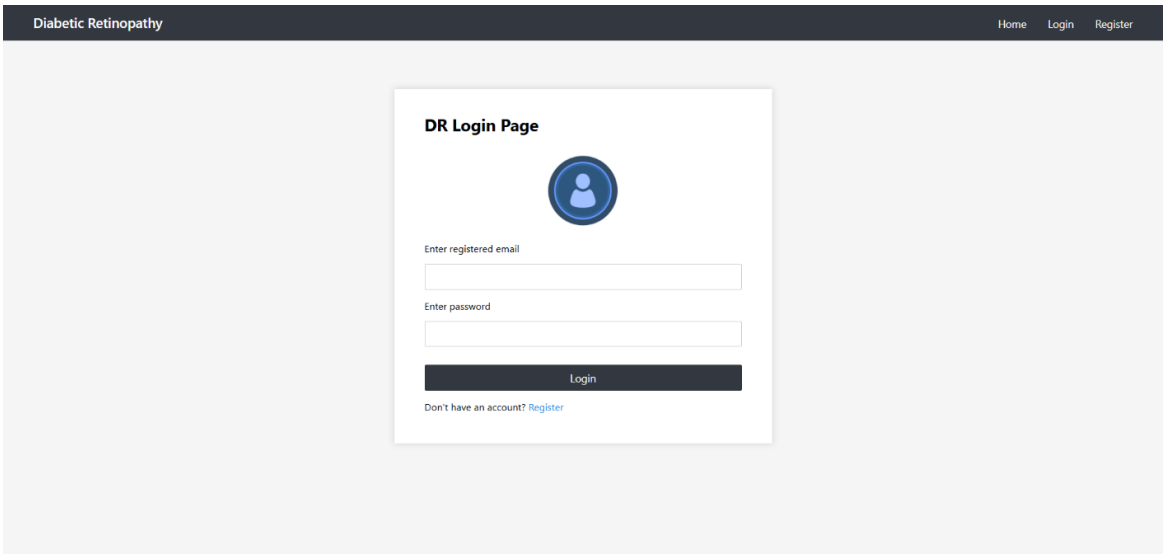
Let's see how our home.html looks like:



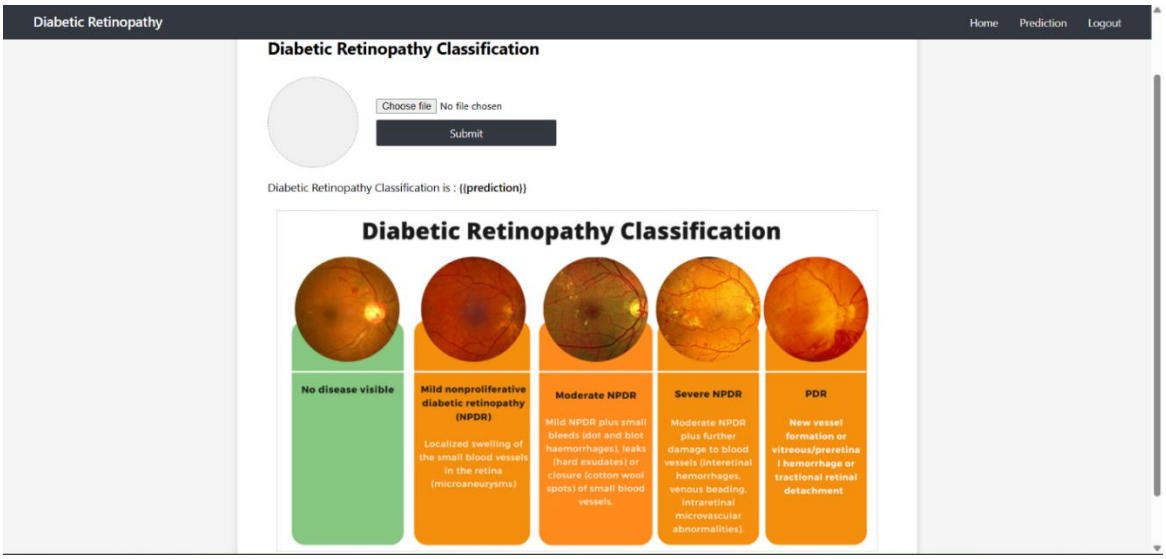
Now when you click on register button from top right corner you will get redirected to registered.html



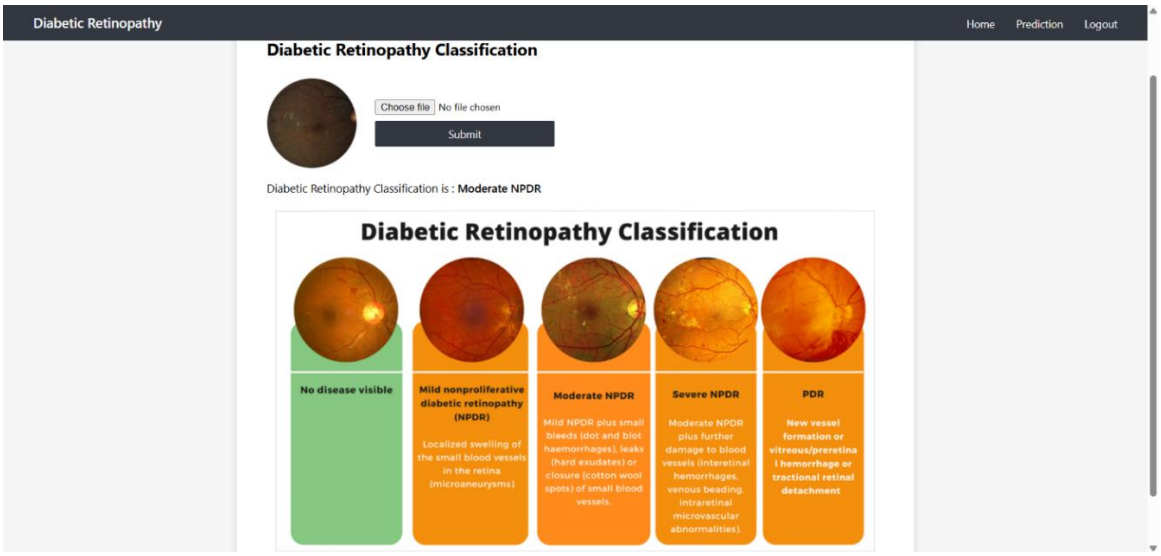
After registering in the register page, use those credentials to login.



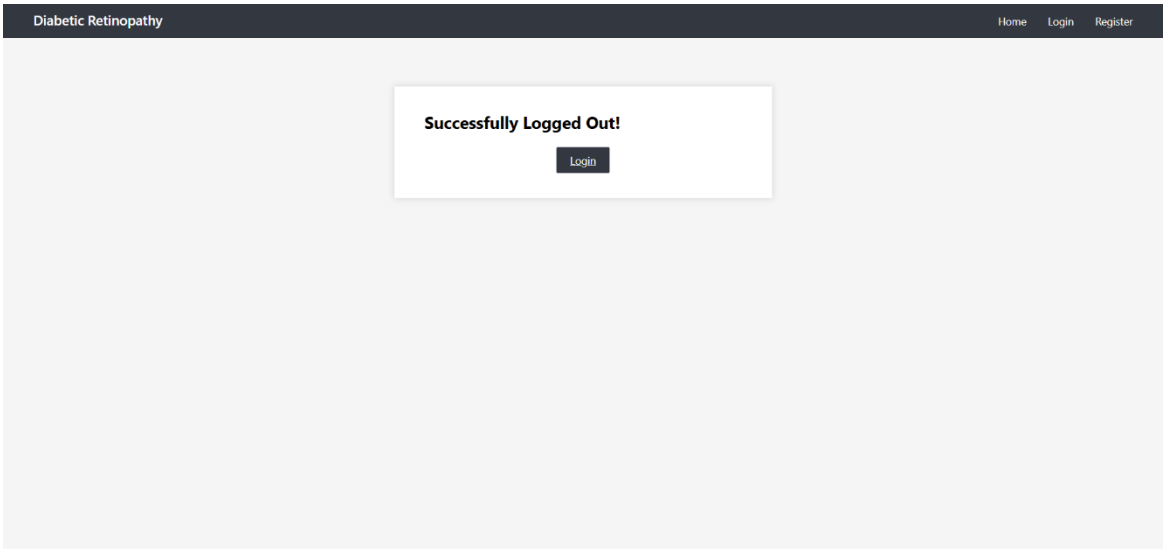
Lets look how our predict.html looks like:



In the prediction page we select the image of the retina by using choose file button and after selecting image then click submit to predict.



Then the image of the predicted image is displayed beside submit button and the prediction of the selected image is displayed below the submit button.



Finally after you logged out this is how the logout page looks like.

Activity 2: Build python code:

The backend logic was implemented using Flask in the app.py file. The saved Xception model was loaded at the beginning of the application. Flask routes were created to handle different URLs such as home, register, login, predict, and logout.

When a user uploads an image on the prediction page, the image is preprocessed (resized and normalized) before being passed to the model for prediction. The model returns a probability score for each class, and the class with the highest probability is selected as the final prediction. The result is then displayed on the webpage.

Import the libraries

```
from flask import Flask, render_template, request, redirect, url_for, session, send_from_directory
from werkzeug.security import generate_password_hash, check_password_hash
from werkzeug.utils import secure_filename
import os
import sqlite3
from tensorflow.keras.models import load_model
from tensorflow.keras.preprocessing.image import load_img, img_to_array
import numpy as np
```

Load the saved model. Importing flask module in the project is mandatory. An object of Flask class is our WSGI application. Flask constructor takes the name of the current module (__name__) as argument.

```
app = Flask(__name__)
app.secret_key = "change_this_secret_to_secure_key"

BASE_DIR = os.path.abspath(os.path.dirname(__file__))
UPLOAD_FOLDER = os.path.join(BASE_DIR, "static", "uploads")
os.makedirs(UPLOAD_FOLDER, exist_ok=True)
app.config["UPLOAD_FOLDER"] = UPLOAD_FOLDER

DB_PATH = os.path.join(BASE_DIR, "database.db")
MODEL_PATH = os.path.join(BASE_DIR, "updated_xception_diabetic_retinopathy.h5")
model = load_model(MODEL_PATH)
```

Render HTML page:

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

Here we will be using declared constructor to route to the HTML page which we have created earlier.

In the above example, '/' URL is bound with home.html function. Hence, when the home page of the web server is opened in browser, the html page will be rendered. Whenever you enter the values from the html page the values can be retrieved using POST Method.

```
@app.route("/register", methods=["GET", "POST"])
def register():
    if request.method == "POST":
        name = request.form.get("name", "").strip()
        email = request.form.get("email", "").strip().lower()
        password = request.form.get("password", "")

        if not name or not email or not password:
            return render_template("register.html", error="All fields are required")

        hashed = generate_password_hash(password)
        try:
            conn = sqlite3.connect(DB_PATH)
            cur = conn.cursor()
            cur.execute(
                "INSERT INTO users (name, email, password) VALUES (?, ?, ?)",
                (name, email, hashed)
            )
            conn.commit()
            conn.close()
            return redirect(url_for("login"))
        except sqlite3.IntegrityError:
            return render_template("register.html", error="Email already registered")
    return render_template("register.html", error=None)
```

Here we are storing the details of the user and giving access.

```python
@app.route("/login", methods=["GET", "POST"])
def login():
    if request.method == "POST":
        email = request.form.get("email", "").strip().lower()
        password = request.form.get("password", "")
        conn = sqlite3.connect(DB_PATH)
        cur = conn.cursor()
        cur.execute("SELECT id, name, password FROM users WHERE email = ?", (email,))
        row = cur.fetchone()
        conn.close()

        if row and check_password_hash(row[2], password):
            session["user"] = {"id": row[0], "name": row[1], "email": email}
            return redirect(url_for("predict"))
        else:
            return render_template("login.html", error="Invalid email or password")
    return render_template("login.html", error=None)
```

And here the user who is registered only have access to login.

```python
@app.route("/predict", methods=["GET", "POST"])
def predict():
    if "user" not in session:
        return redirect(url_for("login"))

    prediction_text = None
    uploaded_filename = None

    if request.method == "POST":
        file = request.files.get("image")
        if file and file.filename:
            filename = secure_filename(file.filename)
            path = os.path.join(app.config["UPLOAD_FOLDER"], filename)
            file.save(path)

            uploaded_filename = filename
            img = load_img(path, target_size=(299, 299))
            arr = img_to_array(img) / 255.0
            arr = np.expand_dims(arr, axis=0)
            preds = model.predict(arr)
            class_index = int(np.argmax(preds, axis=1)[0])
            prediction_text = CLASS_LABELS[class_index]

    return render_template(
        "predict.html",
        prediction=prediction_text,
        uploaded_image=uploaded_filename
    )
```

Here we are routing our app to predict() function. This function retrieves all the values from the HTML page using Post request. That is stored in an array. This array is passed to the model.predict() function. This function returns the prediction. And this prediction value will rendered to the text that we have mentioned in the submit.html page earlier.

Main Function:

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

Activity 3: Running the application

The final activity involved running and testing the web application. The command python app.py was executed in the terminal. The application was accessed through the local host address in a web browser. All functionalities, including registration, login, image upload, prediction display, and logout, were tested successfully. This ensured that the entire system worked as an end-to-end solution.

- Open Vscode
- Navigate to the folder where the python script is and open terminal.
- Now type “python app.py” command
- Navigate to the localhost where you can view your web page.
- Click on the register button and register, then login with those credentials. Enter the inputs, click on the submit button, and see the prediction on the web.

```
ppropriate compiler flags.
 * Serving Flask app 'app'
 * Debug mode: on
WARNING: This is a development server. Do not use it in a production deployment. Use a production WSGI server instead.
 * Running on http://127.0.0.1:5000
Press CTRL+C to quit
```