Alzheimer’s Disease Prediction using Deep learning

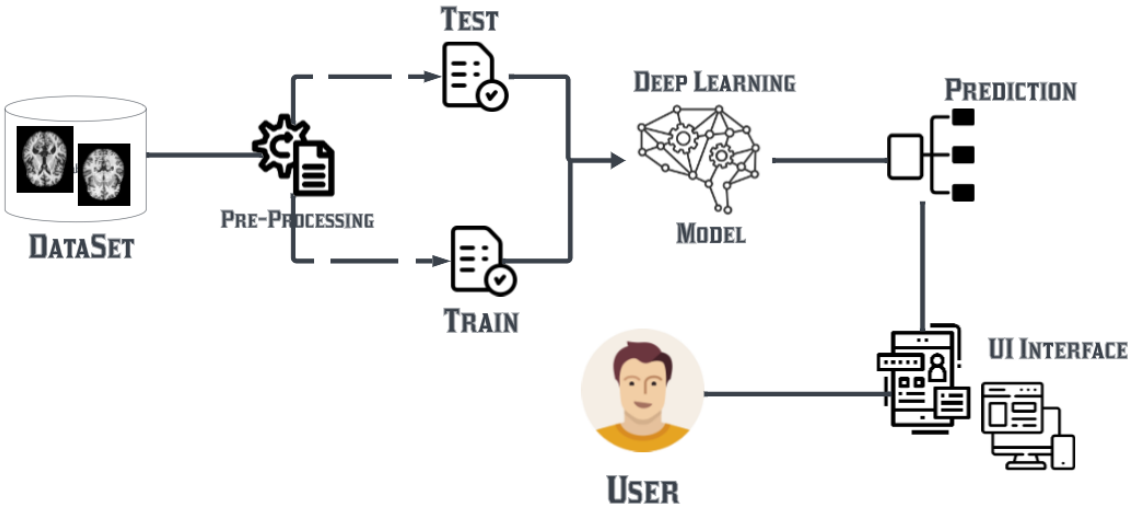
Introduction:

Alzheimer's disease is a progressive neurological disorder characterized by memory loss, cognitive decline, and behavioral changes. It is the most common cause of dementia. The disease is believed to result from a combination of genetic, environmental, and lifestyle factors. Alzheimer's is characterized by the accumulation of abnormal protein deposits in the brain, leading to disruption in brain cell communication. Symptoms develop slowly and worsen over time, including memory loss, difficulty with tasks, language problems, mood changes, and loss of independence. Diagnosis involves a comprehensive assessment, including medical history, cognitive tests, and imaging techniques. While there is no cure, treatments and interventions can help manage symptoms. Research aims to better understand the disease, improve diagnostics, and identify potential treatments. Lifestyle factors, such as exercise and a healthy diet, may contribute to brain health and reduce the risk of developing Alzheimer's.

Imaging techniques, such as magnetic resonance imaging (MRI) or positron emission tomography (PET) scans may be used to assess brain structure and detect any abnormalities or changes associated with Alzheimer's disease. Assessments of daily functioning and activities of daily living (ADLs) are conducted to evaluate the individual's ability to perform basic tasks independently. In recent years, there has been on-going research and clinical trials exploring various approaches aimed at finding a cure or disease-modifying treatments for Alzheimer's. These include strategies targeting beta-amyloid plaques and tau protein tangles, the hallmark pathological features of Alzheimer's disease. Other areas of investigation include inflammation, oxidative stress, and genetic factors.

By leveraging deep learning algorithms such as CNN and Resnet50, researchers and healthcare professionals can gain deeper insights into the disease, enhance early detection, and develop more effective therapeutic interventions.

Technical Architecture:



Project Objectives:

By the end of this project you will:

- Know fundamental concepts and techniques of Convolutional Neural Network and ResNet (Residual Network)50
- Gain a broad understanding of image data.
- Know how to pre-process/clean the data using different data preprocessing techniques.
- Learn to use deep learning models to classify/predict the problems.
- Know how to build a web application using the Flask framework.

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.
- ResNet50 Model analyzes the image, and then prediction is showcased on the Flask UI.

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

- Data Collection.
    - Create Train and Test Folders.
- Data Preprocessing.
    - Import the necessary libraries
    - Import the ImageDataGenerator library
    - Configure ImageDataGenerator class
    - ApplyImageDataGenerator functionality to Trainset and Testset
- Model Building
    - Import the model building Libraries
    - Initializing the model
    - Adding Input Layer
    - Adding Hidden Layer

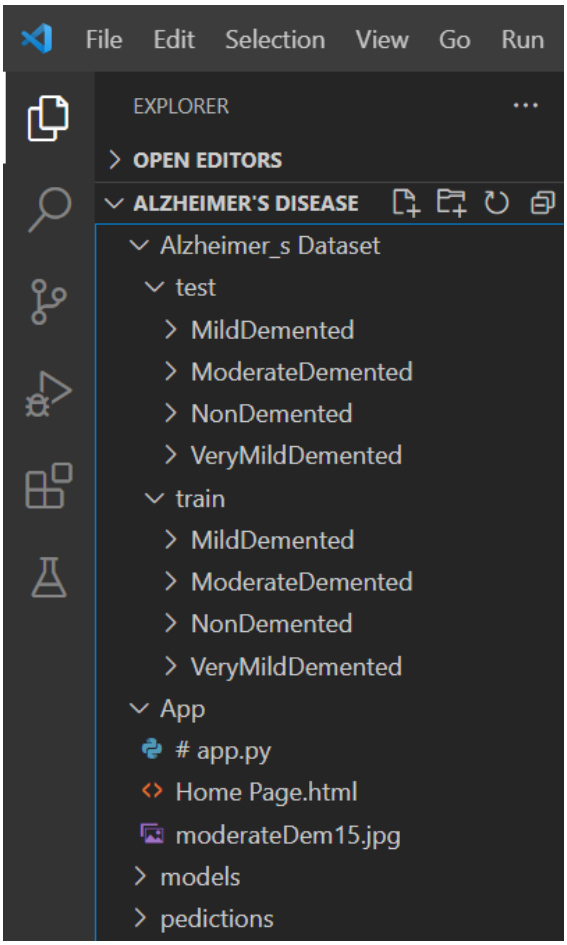
        - Adding Output Layer
        - Configure the Learning Process
        - Training and testing the model
        - Save the Model
    - Application Building
        - Create an HTML file
        - Build Python Code
        - Run the application

Prior Knowledge:

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

- Deep Learning Concepts:
    - ResNet50: ResNet-50, short for "Residual Network with 50 layers," is a specific architecture of a deep convolutional neural network (CNN) And is used as a feature extractor or fine-tuned for various computer vision tasks, including image classification, object detection, and image segmentation.
- Flask: Flask is a popular Python web framework, meaning it is a third-party Python library Used for developing web applications

Project Structure:



- The Dataset folder contains the training and testing MRI Scans for training our model.
- We are building a Flask Application that needs HTML pages stored in the **App** Folder and a python script **app.py** for server side scripting
- We need the model which is saved and the saved model in this content is a **moderateDem15.jpg**
- App folder contains HomePage.html.

Milestone 1:

Data Collection:

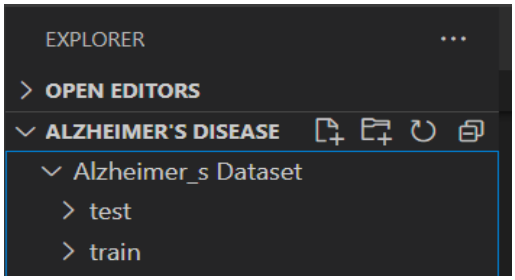
The Alzheimer's disease spectrum is divided into four stages:

- Mild Demented
- Moderate Demented
- Non Demented
- Very Mild Demented

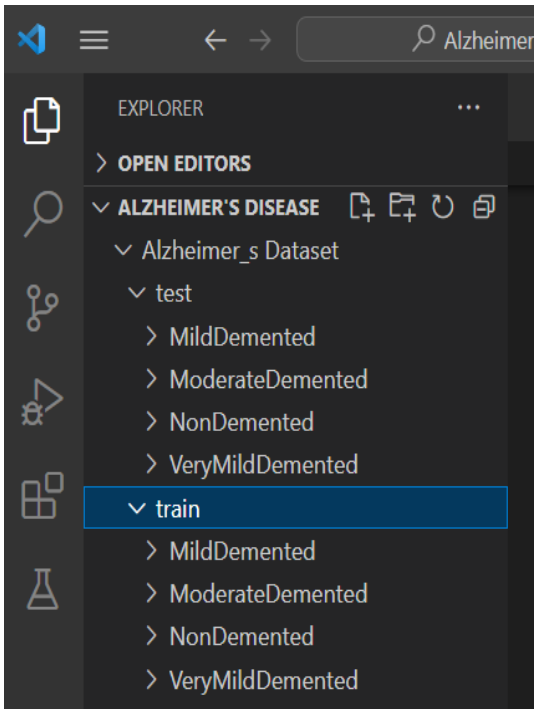
A multi-classification approach is used to classify samples into these four stages. Additionally, separate binary classifications are performed between each pair of classes to further distinguish and classify samples within the Alzheimer's disease spectrum.

Download the Dataset:
https://www.kaggle.com/datasets/tourist55/alzheimers-dataset-4-class-of-images

For building a Deep learning model, we need to divide the dataset into test and train:



The test directory consists of four subfolders, each representing different stages of Alzheimer's disease, and the train dataset (directory) also contains the same structure:



Milestone 2: Image Preprocessing

In this milestone the collected dataset suffers from imbalanced classes, meaning some classes have significantly fewer samples than others. Hence the dataset undergoes several pre-processing steps, including normalization, standardization, resizing, de noising, and format conversion, skull removal, tissue segmentation etc.

Activity 1: Import the ImageDataGenerator library

Image data augmentation in deep learning involves modifying training images with operations like rotation, flipping, scaling, and brightness adjustments to create additional training samples, improving model generalization.
This technique helps prevent overfitting and enhances the model's ability to recognize objects in various orientations and conditions. Popular libraries like TensorFlow and Keras provide tools for implementing image data augmentation.
The Keras deep learning neural network library offers the functionality to train models with image data augmentation using the ImageDataGenerator class.

Let us import the ImageDataGenerator class from tensorflow Keras:

```
from tensorflow.keras.preprocessing.image import ImageDataGenerator
from tensorflow.keras.models import Sequential
```

Activity 2: **Configure ImageDataGenerator class**

The ImageDataGenerator class is initialized, and its settings for various data augmentation techniques are configured.
There are five primary data augmentation techniques for image data, which can be specified using the following arguments:

- For shifting the image horizontally and vertically, you can use the width_shift_range and height_shift_range arguments.
- To apply horizontal and vertical flips to the images, you can use the horizontal_flip and vertical_flip arguments.

- Image rotations can be introduced with the rotation_range argument. Adjusting image brightness can be achieved by setting the brightness_range argument.
- To zoom in or out of the images, you can specify the zoom_range.

An instance of the ImageDataGenerator class can be created for both training and testing purposes.

Activity 3:Apply ImageDataGenerator functionality to Trainset and Testset

Let us apply ImageDataGenerator functionality to Trainset and Testset by using the following code.

```
# Data Augmentation
train_datagen = ImageDataGenerator(rescale = 1./255,
                                   rotation_range=30,
                                   zoom_range=0.2,
                                   horizontal_flip=True,
                                   vertical_flip=True,
                                   validation_split = 0.2)

valid_datagen = ImageDataGenerator(rescale = 1./255,
                                   validation_split = 0.2)

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

For Training set using flow_from_directory function:

It simplifies the process of loading and preprocessing images from directories for training and validation.

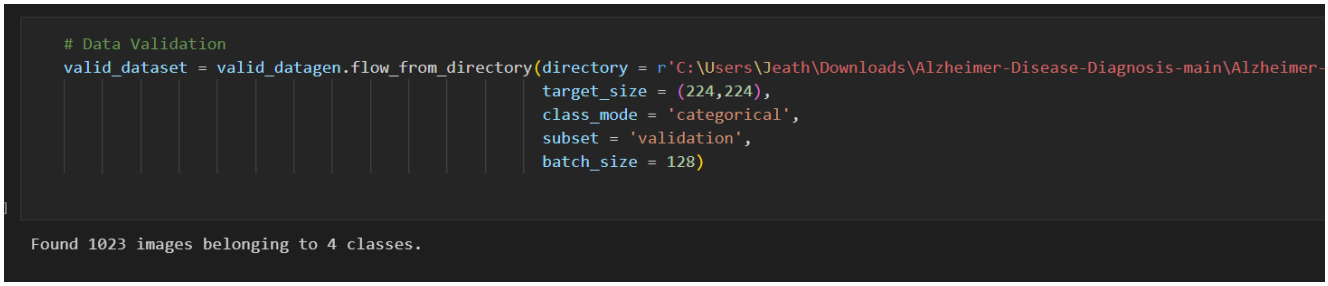
```
# Dataset Training
train_dataset  = train_datagen.flow_from_directory(directory = r'C:\Users\Jeath\Downloads\Alzheimer-Disease-Diagnosis-main
                                                   target_size = (224,224),
                                                   class_mode = 'categorical',
                                                   subset = 'training',
                                                   batch_size = 128)
```

```
Found 4098 images belonging to 4 classes.
```

Arguments:

- directory: This argument specifies the path to the directory containing your training dataset. In this case, the directory path is *'C:\Users\Jeath\Downloads\Alzheimer-Disease-Diagnosis-main\Alzheimer-Disease-Diagnosis-main\Alzheimer_Dataset\train'*.
  The data generator will read images from this directory for training.
- target_size: Sets the image size to 224x224 pixels.
- class_mode: Specifies that it's a multi-class classification problem.
  'Categorical' means that the labels are encoded as a categorical vector
- subset: Indicates it's generating data for the training set.
- batch_size: Defines the batch size for training (128 images per batch).

The training generator is used to feed batches of augmented data into the model for training, while the validation and test generators are used for evaluating the model's performance on validation and test datasets, respectively.



We notice that 1023 images belong to 4 classes.

Milestone 3: Model Building

Now we build our deep learning model:
ResNet-50 consists of 50 layers, including convolutional layers, pooling layers, and fully connected layers. It's a pre-defined architecture with skip connections (residual connections) that helps mitigate the vanishing gradient problem and allows you to train very deep networks.

Activity 1: Importing the Model Building Libraries

Importing the necessary libraries

Import the required Libraries

```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
import skimage.io
import os
import tqdm
import glob
import tensorflow

from tqdm import tqdm
from sklearn.utils import shuffle
from sklearn.model_selection import train_test_split

from skimage.io import imread, imshow
from skimage.transform import resize
from skimage.color import grey2rgb

from tensorflow.keras.preprocessing.image import ImageDataGenerator
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import InputLayer, BatchNormalization, Dropout, Flatten, Dense, Activation, MaxPool2D, Conv2D
from tensorflow.keras.callbacks import EarlyStopping, ModelCheckpoint
from tensorflow.keras.applications.densenet import DenseNet169
from tensorflow.keras.preprocessing.image import load_img, img_to_array
```

Activity 2: Initializing the model

Keras has 2 ways to define a neural network:

- Sequential
- Function API

The Sequential class is used to define linear initializations of network layers which then, collectively, constitute a model. In our example below, we will use the Sequential constructor to create a model, which will then have layers added to it using the add() method.

```
#Inializing the model
from tensorflow.keras.models import Sequential
```

```
# Model Initialization

base_model = ResNet50(input_shape=(224,224,3),
                      include_top=False,
                      weights="imagenet")
```

Activity 3: Adding CNN Layers

model = Sequential(): Initializes a Sequential model, which is a linear stack of layers.

model.add(base_model): Adds the base_model to the model. base_model likely represents a pre-trained model like ResNet-50, which is used as a feature extractor.

model.add(Dropout(0.5)): Adds a dropout layer with a 50% dropout rate to help prevent overfitting. model.add(Flatten()): Flattens the output of the previous layers into a 1D vector for further processing.

model.add(BatchNormalization()): Adds a batch normalization layer to improve training stability and convergence.

model.add(Dense(2048, kernel_initializer='he_uniform'): Adds a dense (fully connected) layer with 2048 neurons and initializes the weights using the He uniform initializer.

model.add(BatchNormalization()):Another batch normalization layer.

model.add(Activation('relu')): Applies the ReLU (Rectified Linear Unit) activation function to introduce non-linearity.

model.add(Dropout(0.5)): Another dropout layer with a 50% dropout rate.

model.add(Dense(1024, kernel_initializer='he_uniform'): Adds another dense layer with 1024 neurons, using the He uniform initializer.

model.add(BatchNormalization()): Another batch normalization layer.

model.add(Activation('relu')): Applies ReLU activation to the previous layer.

model.add(Dropout(0.5)): Another dropout layer.

model.add(Dense(4, activation='softmax'): Adds the output layer with 4 neurons (likely for a classification task) and uses the softmax activation function to obtain class probabilities.

```
# Building Model and adjust the required parameters

model=Sequential()
model.add(base_model)
model.add(Dropout(0.5))
model.add(Flatten())
model.add(BatchNormalization())
model.add(Dense(2048,kernel_initializer='he_uniform'))
model.add(BatchNormalization())
model.add(Activation('relu'))
model.add(Dropout(0.5))
model.add(Dense(1024,kernel_initializer='he_uniform'))
model.add(BatchNormalization())
model.add(Activation('relu'))
model.add(Dropout(0.5))
model.add(Dense(4,activation='softmax'))
```

Activity 5: Adding Dense Layers:

A dense layer is a deeply connected neural network layer. It is the most common and frequently used layer.

```
#Fully connected layers:
#hidden_layer_1
model.add(Dense(2048,kernel_initializer='he_uniform'))

#hidden_laye_2
model.add(Dense(1024,kernel_initializer='he_uniform'))

model.add(Dense(4,activation='softmax'))
```

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: "sequential"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
resnet50 (Functional)        (None, 7, 7, 2048)        23587712
_________________________________________________________________
dropout (Dropout)            (None, 7, 7, 2048)        0
_________________________________________________________________
flatten (Flatten)            (None, 100352)            0
_________________________________________________________________
batch_normalization (BatchNo (None, 100352)            401408
_________________________________________________________________
dense (Dense)                (None, 2048)              205522944
_________________________________________________________________
batch_normalization_1 (Batch (None, 2048)              8192
_________________________________________________________________
activation (Activation)      (None, 2048)              0
_________________________________________________________________
dropout_1 (Dropout)          (None, 2048)              0
_________________________________________________________________
dense_1 (Dense)              (None, 1024)              2098176
_________________________________________________________________
batch_normalization_2 (Batch (None, 1024)              4096
_________________________________________________________________
activation_1 (Activation)    (None, 1024)              0
...
Total params: 231,626,628
Trainable params: 207,832,068
Non-trainable params: 23,794,560
_________________________________________________________________
```

Activity 6: 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.
- OPT = tensorflow.keras.optimizers.Adam(lr=0.001): It defines an Adam optimizer with a learning rate of 0.001. Adam is a popular optimization algorithm used to update the model's weights during training.
- model.compile(): This function prepares the model for training.
    - The loss function is used to find errors or deviations inthe learning process. Keras requires a loss function during the model compilation process: loss='categorical_crossentropy': Sets the loss function to categorical cross-entropy, which is commonly used for multi-class classification tasks.
    - Metrics are used to evaluate the performance of your model. It is similar to the loss function, but not used in the training process metrics=[tensorflow.keras.metrics.AUC(name='auc')]: Specifies that you want to measure the area under the curve (AUC) as a metric during training. AUC is often used to evaluate classification model performance.
    - optimizer=OPT: Specifies the optimizer you defined earlier (Adam with a learning rate of 0.001) for updating the model's weights during training.

```
# Model Compile

OPT     = tensorflow.keras.optimizers.Adam(lr=0.001)

model.compile(loss='categorical_crossentropy',
              metrics=[tensorflow.keras.metrics.AUC(name = 'auc')],
              optimizer=OPT)
```

Activity 7: Train The model

Now, let us train our model with our image dataset. The model is trained for 7 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 till 7 epochs and probably there is further scope to improve the model.

```
# Defining Callbacks

filepath = './best_weights.hdf5'

earlystopping = EarlyStopping(monitor = 'val_auc',
                              mode = 'max' ,
                              patience = 15,
                              verbose = 1)

checkpoint    = ModelCheckpoint(filepath,
                                monitor = 'val_auc',
                                mode='max',
                                save_best_only=True,
                                verbose = 1)


callback_list = [earlystopping, checkpoint]
```

```
# Training the model using number of epochs
model_history=model.fit(train_dataset,
                        validation_data=valid_dataset,
                        epochs = 1, # Number of epochs can be changed
                        callbacks = callback_list,
                        verbose = 1)
```

fit_generator functions used to train a deep learning neural network

Arguments:

- model_history = model.fit(...): Initiates the training process and stores the training history in model_history.
- train_dataset: The dataset containing training samples and labels.
- validation_data = valid_dataset: The dataset used for validation during training to monitor the model's performance on unseen data.
- epochs = 1: Specifies the number of training epochs, which is set to 1 here. You can increase this number to train the model for more epochs.
- callbacks = callback_list: Callbacks are functions or objects that can be called during training to perform various tasks. callback_list likely contains callback functions for tasks like saving model checkpoints, early stopping, or logging training information.
- verbose = 1: Sets the verbosity level of training output. A value of 1 means that training progress will be displayed during training. You can adjust this value to control the amount of training information displayed.

```
model.evaluate(test_dataset)
```

```
10/10 [==============================] - 91s 9s/step - loss: 1.1231 - auc: 0.7804

[1.1231095790863037, 0.780361533164978]
```

```
# Training the model using number of epochs
model_history=model.fit(train_dataset,
                        validation_data=valid_dataset,
                        epochs = 1, # Number of epochs can be changed
                        callbacks = callback_list,
                        verbose = 1)
```

```
33/33 [==============================] - 516s 16s/step - loss: 1.2512 - auc: 0.7773 - val_loss: 1.0808 - val_auc: 0.8090

Epoch 00001: val_auc improved from 0.77528 to 0.80895, saving model to .\best_weights.hdf5
```

```
Epoch 1/30
33/33 [==============================] - ETA: 0s - loss: 1.5367 - auc: 0.7390
Epoch 1: val_auc improved from -inf to 0.58925, saving model to .\best_weights.hdf5
33/33 [==============================] - 336s 10s/step - loss: 1.5367 - auc: 0.7390 - val_loss: 2.9766 - val_auc: 0.5893
Epoch 2/30
33/33 [==============================] - ETA: 0s - loss: 1.2218 - auc: 0.7859
Epoch 2: val_auc improved from 0.58925 to 0.69831, saving model to .\best_weights.hdf5
33/33 [==============================] - 317s 10s/step - loss: 1.2218 - auc: 0.7859 - val_loss: 1.3996 - val_auc: 0.6983
Epoch 3/30
33/33 [==============================] - ETA: 0s - loss: 1.1527 - auc: 0.7999
Epoch 3: val_auc did not improve from 0.69831
33/33 [==============================] - 305s 9s/step - loss: 1.1527 - auc: 0.7999 - val_loss: 1.2900 - val_auc: 0.6431
Epoch 4/30
33/33 [==============================] - ETA: 0s - loss: 1.1105 - auc: 0.7986
Epoch 4: val_auc did not improve from 0.69831
33/33 [==============================] - 304s 9s/step - loss: 1.1105 - auc: 0.7986 - val_loss: 1.3262 - val_auc: 0.6571
Epoch 5/30
33/33 [==============================] - ETA: 0s - loss: 1.0633 - auc: 0.8100
Epoch 5: val_auc improved from 0.69831 to 0.75601, saving model to .\best_weights.hdf5
33/33 [==============================] - 303s 9s/step - loss: 1.0633 - auc: 0.8100 - val_loss: 1.1851 - val_auc: 0.7560
Epoch 6/30
33/33 [==============================] - ETA: 0s - loss: 1.0483 - auc: 0.8123
Epoch 6: val_auc did not improve from 0.75601
33/33 [==============================] - 296s 9s/step - loss: 1.0483 - auc: 0.8123 - val_loss: 1.1452 - val_auc: 0.7508
Epoch 7/30
33/33 [==============================] - ETA: 0s - loss: 1.0214 - auc: 0.8177
Epoch 7: val_auc did not improve from 0.75601
33/33 [==============================] - 294s 9s/step - loss: 1.0214 - auc: 0.8177 - val_loss: 1.1412 - val_auc: 0.7516
```

Activity 8: Save the Model

The model is saved with .h5 extension as follows:
An H5 file is a data file saved in the Hierarchical Data Format (HDF).
It containsmultidimensional arrays of scientific data.

```
saving model to .\best_weights.hdf5
```

Activity 9: Test The model

Evaluation is a step in the model development process that assesses whether the model is well-suited for the specific problem and its associated dataset.

Taking an image as input and checking the results:

```
# Test Case 3: Moderate Demented

dic = test_dataset.class_indices
idc = {k:v for v, k in dic.items()}

img = load_img(r'C:\Users\Jeath\Downloads\Alzheimer-Disease-Diagnosis-main\Alzheimer-Disease-Diagnosis-main\Alzheimer
img = img_to_array(img)
img = img/255
imshow(img)
plt.axis('off')
img = np.expand_dims(img,axis=0)
#answer = model.predict_classes(img)
answer = (model.predict(img) > 0.5).astype("int32")
#probability = round(np.max(model.predict_proba(img)*100),2)

probability = round(np.max(model.predict(img)*100),2)

#predict_classes=np.argmax(predict_prob,axis=1)

print(probability, '% chances are there that the image is Moderate Demented')
```

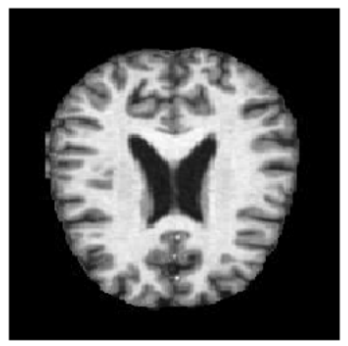
By using the model we are predicting the output for the given input image:

```
72.74 % chances are there that the image is Moderate Demented
```

```
# Test Case No.1: Non-Dementia

dic = test_dataset.class_indices
idc = {k:v for v, k in dic.items()}

img = load_img(r'C:\Users\Jeath\Downloads\Alzheimer-Disease-Diagnosis-main\Alzheimer-Disease-Diagnosis-main\Alz
img = img_to_array(img)
img = img/255
imshow(img)
plt.axis('off')
img = np.expand_dims(img,axis=0)
#answer = model.predict_classes(img)
answer = (model.predict(img) > 0.5).astype("int32")
#probability = round(np.max(model.predict_proba(img)*100),2)

probability = round(np.max(model.predict(img)*100),2)

#predict_classes=np.argmax(predict_prob,axis=1)

print(probability, '% chances are there that the image is Non-Dementia')
```

Output:

```
55.45 % chances are there that the image is Non-Dementia
```



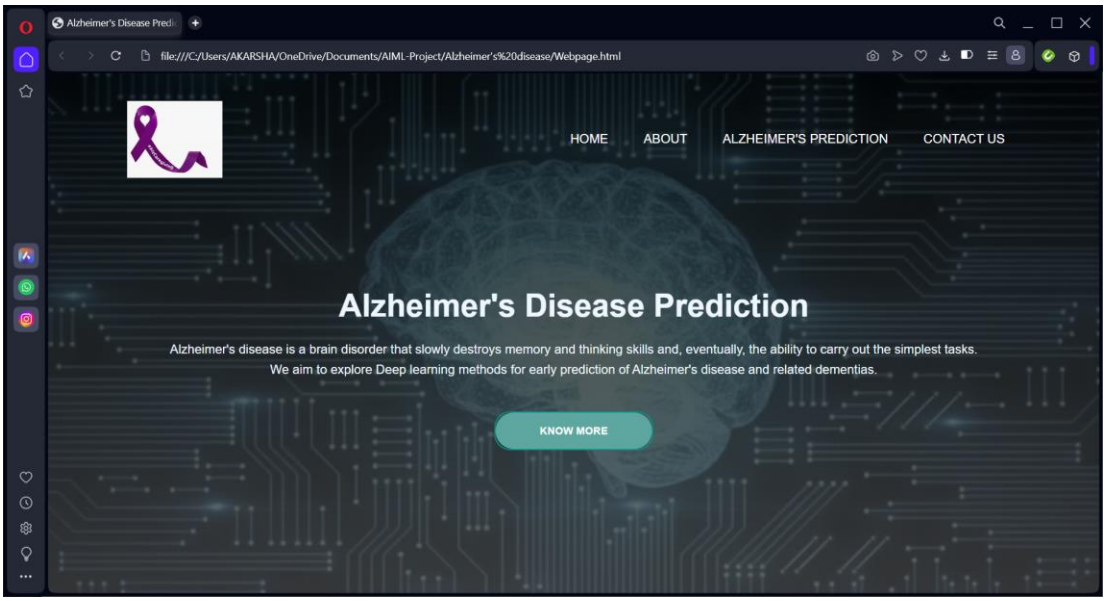
Milestone 4: Application Building

Now that we have trained our model, let us build our flask application which will be running in our local browser with a user interface.

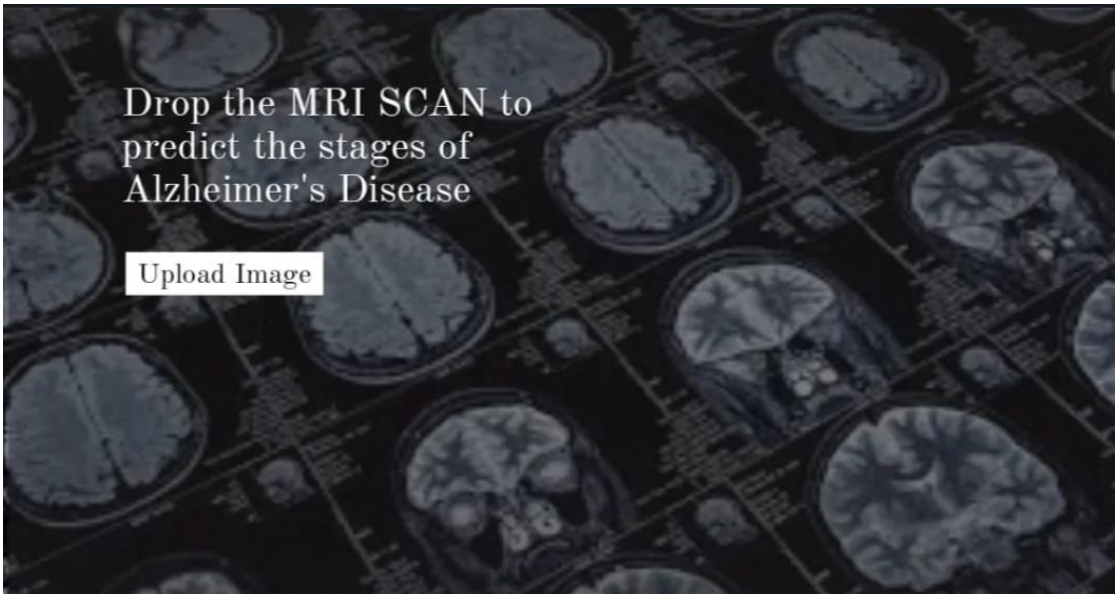
In the flask application, the input parameters are taken from the HTML page these factors are then given to the model to know to predict the stage of Alzheimer's disease and showcased on the HTML page to notify the user. Whenever the user interacts with the UI and selects the "Image" button, the next page is opened where the user chooses the image and predicts the output.

Activity 1 : Create HTML Pages

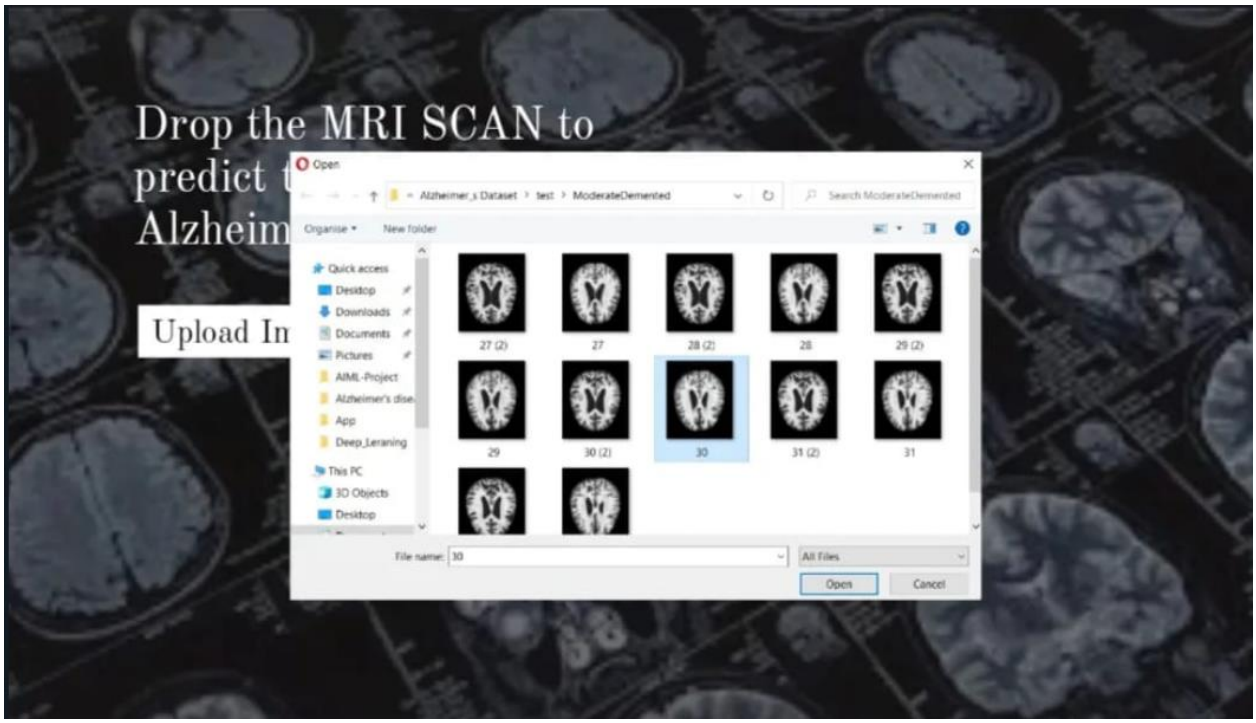
index.html looks like this



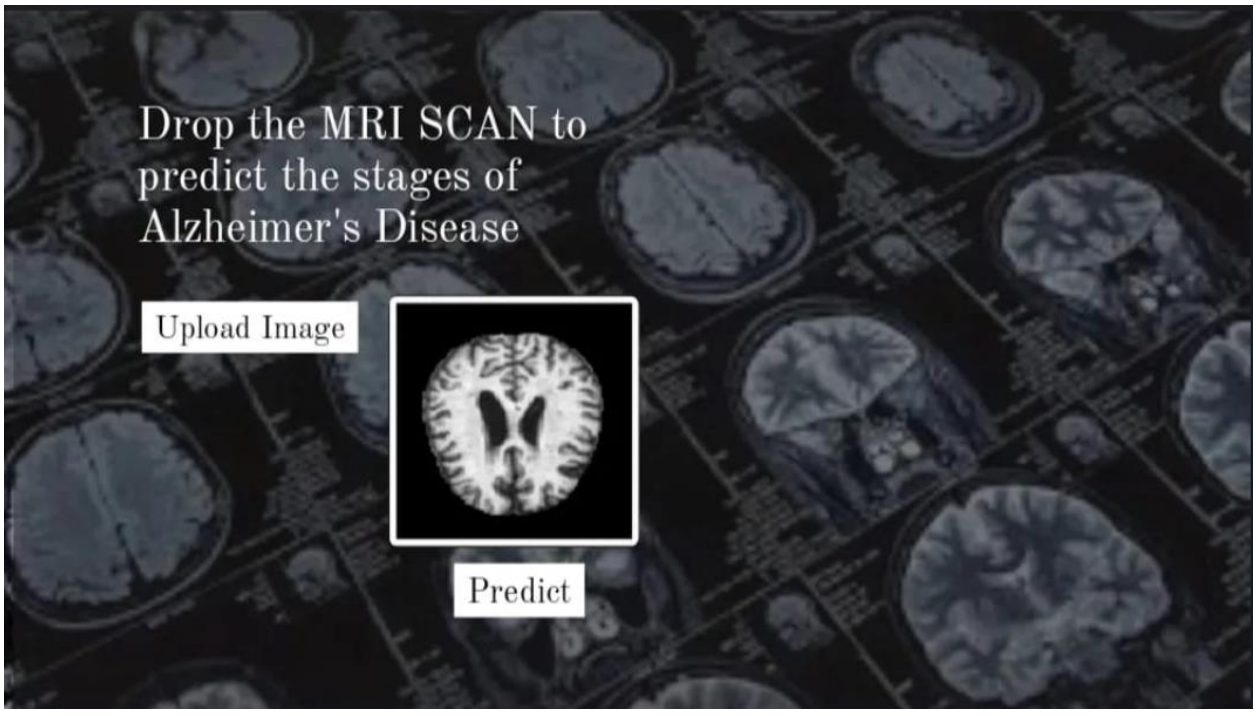
When you click on the Alzheimer's prediction the below page appears:



When you click on the upload image, the files box appears choose the image (MRI SCAN) to upload



After you click on the image:



Activity 2: Build python code

Task 1: Importing Libraries

The first step is usually importing the libraries that will be needed in the program:

```
import graphlib
from future import division, print_function

import sys
import os
import glob
import numpy as np
from tensorflow.keras.preprocessing import image
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense
from tensorflow.keras.layers import Dropout
from tensorflow.keras.applications.imagenet.utils import preprocess_input, decode_predictions
from tensorflow.keras.models import load_model
from tensorflow.keras import backend
from tensorflow.keras import backend
from tensorflow import keras
import tensorflow as tf

from skimage.transform import resize

from flask import Flask, redirect, url_for, request, render_template
from werkzeug.utils import secure_filename
from gevent.pywsgi import WSGIServer
```

Importing the flask module in the project is mandatory. An object of the Flask class is our WSGI application. Flask constructor takes the name of the current module (__name__) as argument Pickle library to load the model file.

Task 2: Creating our flask application and loading our model by using load_model method

```
#Define a flask app
app = Flask(__name__)
#Load the trained model
model=load_model('moderatedementia.h5')
```

Task 3: Routing to the html Page

Here, the previously produced HTML page is accessed using the defined constructor.
The index.html function is connected to the '/' URL in the example above. Thus, the HTML page will be rendered when a web server's home page is viewed in a browser. The POST or GET methods can be used to access an image when browsing an HTML page.

```
#Define a flask app
app = Flask(__name__)
#Load the trained model
model=load_model('moderatedementia.h5')

@app.route('/', methods=['GET'])
def index():
    #Main_Page
    return render_template('index.html')
@app.route('/Image',methods=['POST','GET'])
def prediction():
    #It will direct you to the prediction page
    return render_template('base.html')
```

Showcasing prediction on UI:

Here we are defining a function which requests the browsed file from the html page using the post method. The requested picture file is then saved to the uploads folder in this same directory using OS library.

Using the load image class from Keras library we are retrieving the saved picture from the path declared. We are applying some image processing techniques and then sending that preprocessed image to the model for predicting the class. This returns the numericalvalue of a class (like 0,1 ,2 etc.) which lies in the 0th index of the variable.

This numericalvalue is passed to the index variable declared. This returns the name of the class. This name is rendered to the predict variable used in the html page.

```
@app.route('/', methods=['GET'])
def upload():
    if request.method == 'POST':
        f = request.files['image']
    basepath = os.path.dirname(__file__)
    file_path = os.path.join(basepath, 'uploads', secure_filename(f.filemame))
    f.save(file_path)
    img = image.load_img(file_path, target_size=(224,224))
    x = image.img_to_array(img)
    x = np.expand_dims(x,axis=0)

    with graph.as_default():
        set_session(sess)
        predicition = model.predict(x)[0][0][0]
    print(predicition)
    if predicition==0:
        text = "Mild Demented"
    elif predicition==1:
        text = "Moderate Demented"
    elif predicition==2:
        text = "Non Demented"
    else:
        text = "Very Mild Demented"
    return text
```

Predicting the results

We then proceed to predict the stages of disease in the input image using model.predict function and the result is stored in the result variable.

Finally, Run the application

This is used to run the application in a local host.

```
# running your application
if __name__ == "__main__":
    app.run()
#http://localhost:5000/ or localhost:5000
```

Activity 3:Run the application

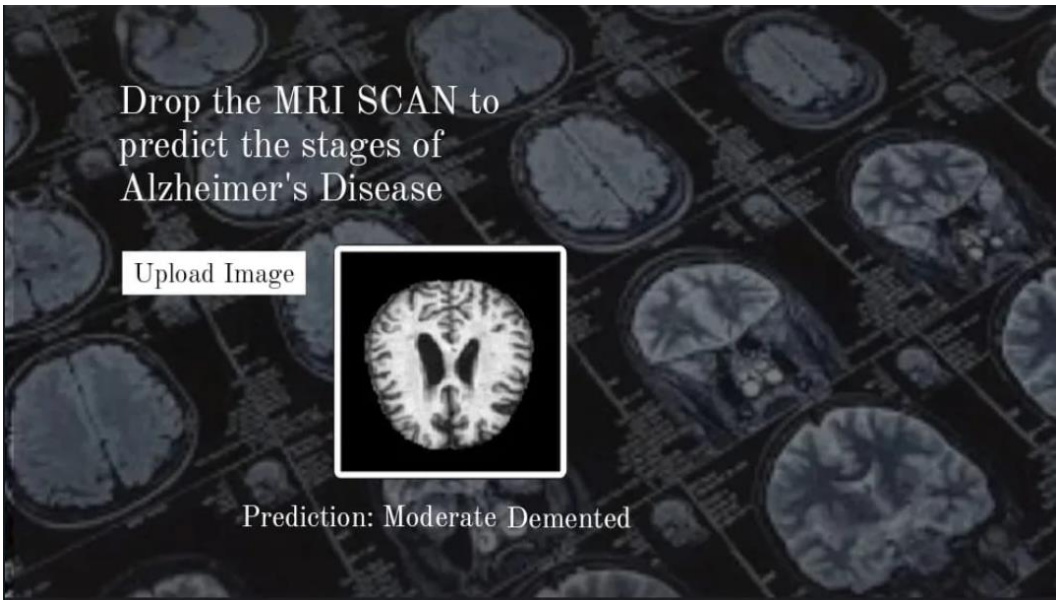
- Open the anaconda prompt from the start menu.
- Navigate to the folder where your app.py resides.
- Now type “python app.py” command.
- It will show the local host where your app is running on **http://127.0.0.1.5000/**
- Copy that local host URL and open that URL in the browser. It does navigate me to where you can view your web page.
- Enter the values, click on the predict button and see the result/prediction on the web page.

Then it will run on localhost:5000

Navigate to the localhost (http://127.0.0.1:5000/)where you can view your web page.

FINAL OUTPUT:

Output 1:



Output 2: