

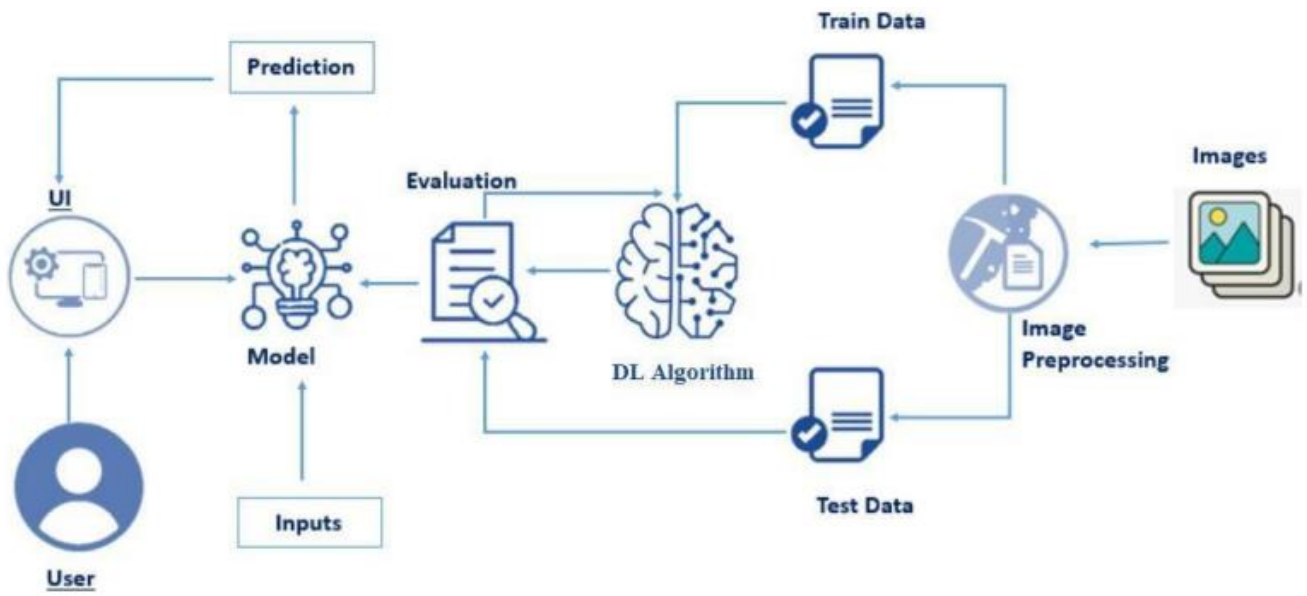
Alzheimer's disease prediction

Project Description:

Alzheimer's disease (AD) is the leading cause of dementia in older adults. There is currently a lot of interest in applying machine learning to find out metabolic diseases like Alzheimer's and Diabetes that affect a large population of people around the world. Their incidence rates are increasing at an alarming rate every year. In Alzheimer's disease, the brain is affected by neurodegenerative changes. As our aging population increases, more and more individuals, their families, and healthcare will experience diseases that affect memory and functioning. These effects will be profound on the social, financial, and economic fronts. In its early stages, Alzheimer's disease is hard to predict. A treatment given at an early stage of AD is more effective, and it causes fewer minor damage than a treatment done at a later stage

A treatment given at an early stage of AD is more effective, and it causes fewer minor damage than a treatment done at a later stage. The purpose here is to build a model in Watson Studio and deploy the model in IBM Watson Machine Learning.

Technical Architecture:



Prerequisites:

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

- **Anaconda navigator and PyCharm / Spyder:**
 - Refer the link below to download anaconda navigator
 - Link (PyCharm) : <https://youtu.be/1ra4zH2G4o0>
 - Link (Spyder) : <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 “pip install keras” and click enter.
 - Type “pip install Flask” and click enter.

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>
 - **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:** Flask is a popular Python web framework, meaning it is a third-party Python library used for developing web applications.
Link: https://www.youtube.com/watch?v=Ij4I_CvBnt0

Project Objectives:

By the end of this project you'll understand:

- Preprocessing the images.
- Applying Transfer learning algorithms on the dataset.
- How deep neural networks detect the disease.
- You will be able to know how to find the accuracy of the model.
- You will be able to Build web applications 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.
- The VGG16 Model analyzes the image, then the prediction is showcased on the Flask UI.

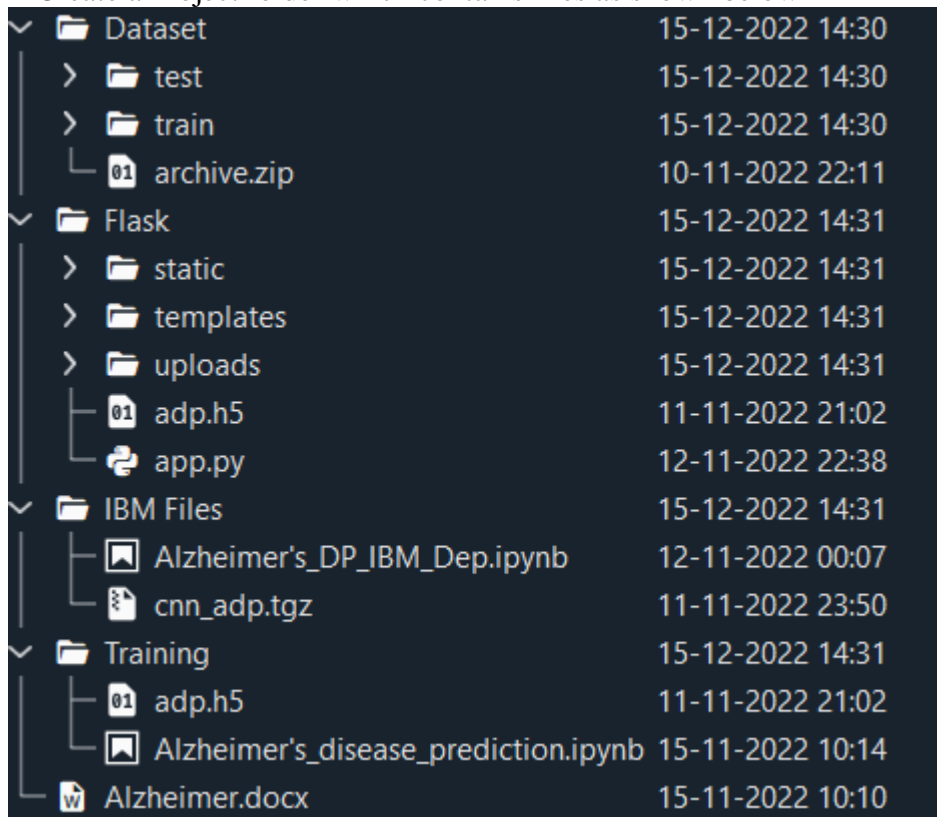
To accomplish this, we have to complete all the activities and tasks listed below

- Data Collection.
 - Create a Train and Test path.
- Data Pre-processing.
 - Import the required library
 - Configure ImageDataGenerator class
 - Apply ImageDataGenerator functionality to Trainset and Testset
- Model Building
 - Adding Dense Layer
 - Configure the Learning Process
 - Train the model
 - Save the Model
 - Test the model
- Application Building
 - Create an HTML file

Build Python Code

Project Structure:

Create a Project folder which contains files as shown below

A screenshot of a file explorer window with a dark background. It shows a tree view of a project structure. The 'Dataset' folder is expanded, showing 'test' and 'train' subfolders, and an 'archive.zip' file. The 'Flask' folder is also expanded, showing 'static', 'templates', and 'uploads' subfolders, and 'adp.h5' and 'app.py' files. The 'IBM Files' folder is expanded, showing 'Alzheimer's_DP_IBM_Dep.ipynb' and 'cnn_adp.tgz' files. The 'Training' folder is expanded, showing 'adp.h5' and 'Alzheimer's_disease_prediction.ipynb' files. At the bottom, the 'Alzheimer.docx' file is listed.

| | |
|--------------------------------------|------------------|
| Dataset | 15-12-2022 14:30 |
| > test | 15-12-2022 14:30 |
| > train | 15-12-2022 14:30 |
| 01 archive.zip | 10-11-2022 22:11 |
| Flask | 15-12-2022 14:31 |
| > static | 15-12-2022 14:31 |
| > templates | 15-12-2022 14:31 |
| > uploads | 15-12-2022 14:31 |
| 01 adp.h5 | 11-11-2022 21:02 |
| app.py | 12-11-2022 22:38 |
| IBM Files | 15-12-2022 14:31 |
| Alzheimer's_DP_IBM_Dep.ipynb | 12-11-2022 00:07 |
| cnn_adp.tgz | 11-11-2022 23:50 |
| Training | 15-12-2022 14:31 |
| 01 adp.h5 | 11-11-2022 21:02 |
| Alzheimer's_disease_prediction.ipynb | 15-11-2022 10:14 |
| Alzheimer.docx | 15-11-2022 10:10 |

- Flask folder consists of static, templates and app.py.
- IBM folder consists of trained model notebook.
- Training file consist of Alzheimer's_disease_prediction.ipynb , model training.

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 brain MRI then organize into subdirectories based on their respective names as shown in the project structure. Create folders of types of Alzheimer.

In this project, we have collected images of 4 types of brain MRI images like Mild Demented, Moderate Demented, Non Demented & Very Mild Demented 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:- [Alzheimer's Dataset \(4 class of Images \) | Kaggle](#)

Note: For better accuracy train on more images

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

Upload the dataset into google drive and connect the google colab with drive

```
!unzip '/content/archive.zip'
Archive: /content/archive.zip
```

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 the project dataset folder training and testing folders are presented. So, in this case we just have to assign a variable and pass the folder path to it.

```
imageSize = [224, 224]

trainPath = r"/content/Alzheimer_s Dataset/train"

testPath = r"/content/Alzheimer_s Dataset/test"
```

Different Deep learning models are used in our project and the best model (VGG16) is selected. The image input size of VGG16 model is 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.

Link : <https://thesmartbridge.com/documents/spsaimldocs/CNNprep.pdf>

Activity 1: Importing the libraries

Import the necessary libraries as shown in the image.

```
from tensorflow.keras.layers import Dense, Flatten, Input
from tensorflow.keras.models import Model
from tensorflow.keras.preprocessing import image
from tensorflow.keras.preprocessing.image import ImageDataGenerator, load_img
from tensorflow.keras.applications.vgg16 import VGG16, preprocess_input
from glob import glob
import numpy as np
import matplotlib.pyplot as plt
```

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.

```
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).

```
training_set = train_datagen.flow_from_directory(trainPath,
                                                target_size = (224, 224),
                                                batch_size = 10,
                                                class_mode = 'categorical')

test_set = test_datagen.flow_from_directory(testPath,
                                            target_size = (224, 224),
                                            batch_size = 10,
                                            class_mode = 'categorical')
```

```
Found 5121 images belonging to 4 classes.
Found 1279 images belonging to 4 classes.
```

Total the dataset is having 5121 train images & 1279 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 VGG16, one of the convolution

neural net (CNN) architecture which is considered as a very good model for Image classification.

Deep understanding on the VGG16 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 (224,244,3).

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 images classification(since it is not the part of Imagenet dataset)

Flatten layer flattens the input. Does not affect the batch size.

```
vgg = VGG16(input_shape=imageSize + [3], weights='imagenet',include_top=False)

Downloading data from https://storage.googleapis.com/tensorflow/keras-applications/vgg16/vgg16\_weights\_158889256/58889256 [=====] - 0s 0us/step

for layer in vgg.layers:
    layer.trainable = False

x = Flatten()(vgg.output)
```

Activity 2: Adding Dense Layers

```
x = Flatten()(vgg.output)

prediction = Dense(4, activation='softmax')(x)

model = Model(inputs=vgg.input, 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 resnet.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_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_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_pool (MaxPooling2D) (None, 7, 7, 512)        0
flatten (Flatten)          (None, 25088)             0
dense (Dense)              (None, 4)                 100356

=====
Total params: 14,815,044
Trainable params: 100,356
Non-trainable params: 14,714,688

```

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 25 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.

fit_generator 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.

```
r = model.fit_generator(  
    training_set,  
    validation_data=test_set,  
    epochs=25,  
    steps_per_epoch=50,  
    validation_steps=171//10)
```

/usr/local/lib/python3.7/dist-packages/ipykernel_launcher.py:8: UserWarning: `Model.fit_generator` is deprecated and will be removed in a future version. Please use `Model.fit`, which

```
Epoch 1/25  
50/50 [=====] - 356s 7s/step - loss: 0.8996 - accuracy: 0.6720 - val_loss: 1.6080 - val_accuracy: 0.5941  
Epoch 2/25  
50/50 [=====] - 358s 7s/step - loss: 0.6291 - accuracy: 0.7500 - val_loss: 0.9544 - val_accuracy: 0.6118  
Epoch 3/25  
50/50 [=====] - 357s 7s/step - loss: 0.6558 - accuracy: 0.7400 - val_loss: 1.3245 - val_accuracy: 0.5824  
Epoch 4/25  
50/50 [=====] - 354s 7s/step - loss: 0.8963 - accuracy: 0.6904 - val_loss: 0.9884 - val_accuracy: 0.5765  
Epoch 5/25  
50/50 [=====] - 355s 7s/step - loss: 0.6382 - accuracy: 0.7560 - val_loss: 1.1198 - val_accuracy: 0.5706  
Epoch 6/25  
50/50 [=====] - 360s 7s/step - loss: 1.1938 - accuracy: 0.6000 - val_loss: 1.3052 - val_accuracy: 0.6000  
Epoch 7/25  
50/50 [=====] - 353s 7s/step - loss: 0.7774 - accuracy: 0.6960 - val_loss: 1.8641 - val_accuracy: 0.5471  
Epoch 8/25  
50/50 [=====] - 356s 7s/step - loss: 0.6218 - accuracy: 0.7480 - val_loss: 1.1196 - val_accuracy: 0.5824  
Epoch 9/25  
50/50 [=====] - 353s 7s/step - loss: 0.6710 - accuracy: 0.7640 - val_loss: 0.8414 - val_accuracy: 0.6529  
Epoch 10/25  
50/50 [=====] - 356s 7s/step - loss: 0.6317 - accuracy: 0.7300 - val_loss: 1.6734 - val_accuracy: 0.5471  
Epoch 11/25  
50/50 [=====] - 356s 7s/step - loss: 0.7636 - accuracy: 0.6840 - val_loss: 1.2556 - val_accuracy: 0.6412  
Epoch 12/25  
50/50 [=====] - 359s 7s/step - loss: 0.6638 - accuracy: 0.7420 - val_loss: 1.0322 - val_accuracy: 0.5882
```

```
Epoch 13/25  
50/50 [=====] - 360s 7s/step - loss: 0.6604 - accuracy: 0.7040 - val_loss: 0.8536 - val_accuracy: 0.6529  
Epoch 14/25  
50/50 [=====] - 356s 7s/step - loss: 0.6524 - accuracy: 0.7480 - val_loss: 1.5223 - val_accuracy: 0.5235  
Epoch 15/25  
50/50 [=====] - 357s 7s/step - loss: 0.6880 - accuracy: 0.7340 - val_loss: 0.8914 - val_accuracy: 0.6235  
Epoch 16/25  
50/50 [=====] - 352s 7s/step - loss: 0.6028 - accuracy: 0.7420 - val_loss: 0.9800 - val_accuracy: 0.6471  
Epoch 17/25  
50/50 [=====] - 353s 7s/step - loss: 0.5009 - accuracy: 0.8040 - val_loss: 1.0626 - val_accuracy: 0.6118  
Epoch 18/25  
50/50 [=====] - 354s 7s/step - loss: 0.6210 - accuracy: 0.7540 - val_loss: 1.7021 - val_accuracy: 0.5706  
Epoch 19/25  
50/50 [=====] - 348s 7s/step - loss: 0.8740 - accuracy: 0.6843 - val_loss: 1.0691 - val_accuracy: 0.6588  
Epoch 20/25  
50/50 [=====] - 354s 7s/step - loss: 0.5276 - accuracy: 0.7940 - val_loss: 1.0016 - val_accuracy: 0.6353  
Epoch 21/25  
50/50 [=====] - 352s 7s/step - loss: 0.6114 - accuracy: 0.7580 - val_loss: 1.1657 - val_accuracy: 0.6176  
Epoch 22/25  
50/50 [=====] - 354s 7s/step - loss: 0.5672 - accuracy: 0.7500 - val_loss: 1.5618 - val_accuracy: 0.5706  
Epoch 23/25  
50/50 [=====] - 350s 7s/step - loss: 0.5775 - accuracy: 0.7640 - val_loss: 1.4786 - val_accuracy: 0.5706  
Epoch 24/25  
50/50 [=====] - 349s 7s/step - loss: 0.4979 - accuracy: 0.7960 - val_loss: 1.2209 - val_accuracy: 0.6000  
Epoch 25/25  
50/50 [=====] - 349s 7s/step - loss: 0.5427 - accuracy: 0.7880 - val_loss: 0.8899 - val_accuracy: 0.6471
```

Milestone 4: Save the Model

The model is saved with .h5 extension as follows

```
model.save('adp.h5')
```

An H5 file is a data file saved in the Hierarchical Data Format (HDF). It contains multidimensional arrays of scientific data.

Milestone 5: 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

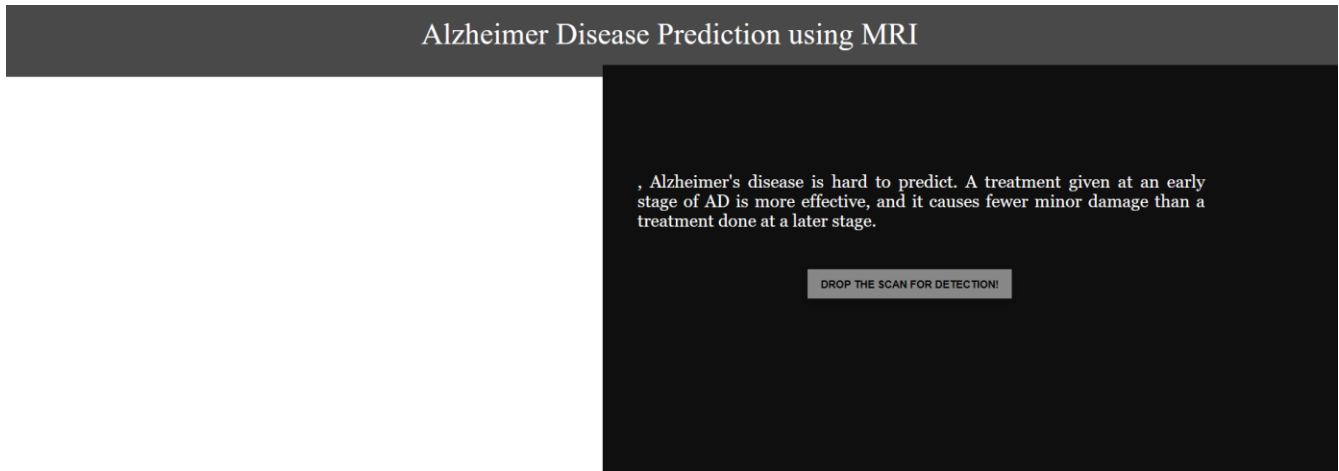
- Building HTML Pages
- Building server side script

Activity1: Building Html Pages:

For this project create one HTML file namely

- alzheimers.html

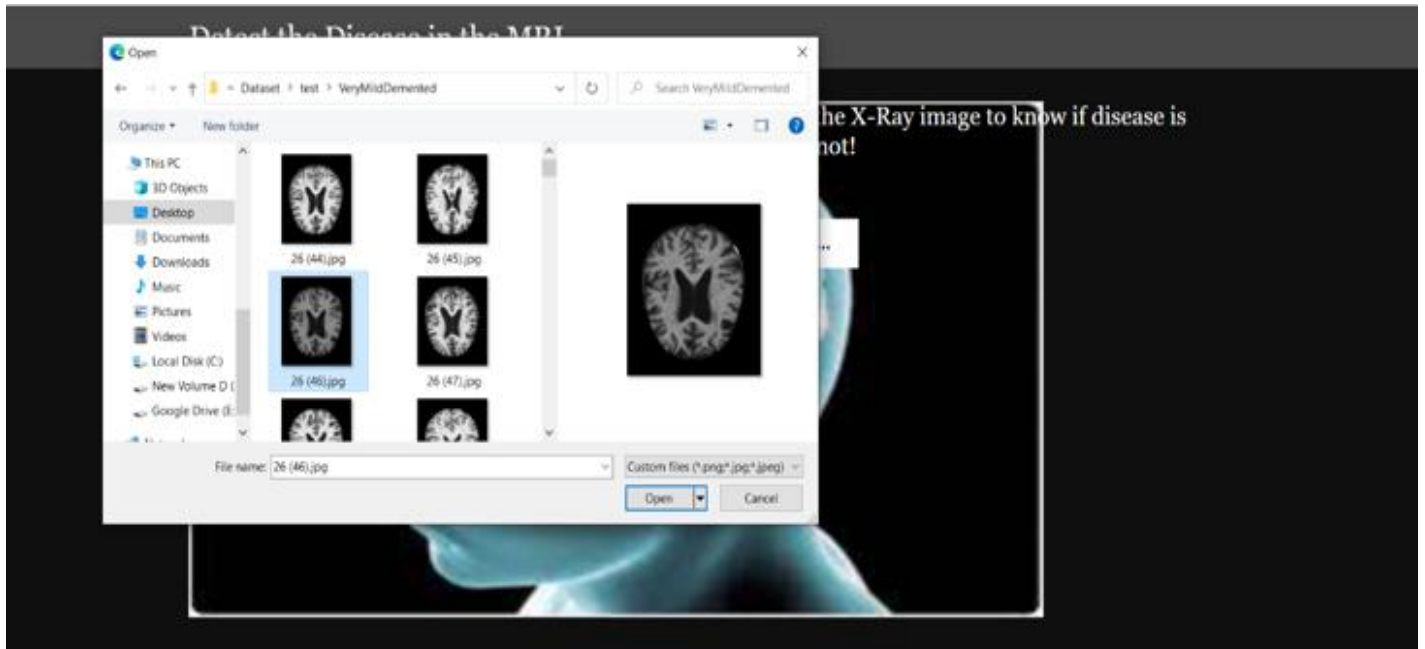
Let's see how our alzheimers.html page looks like:



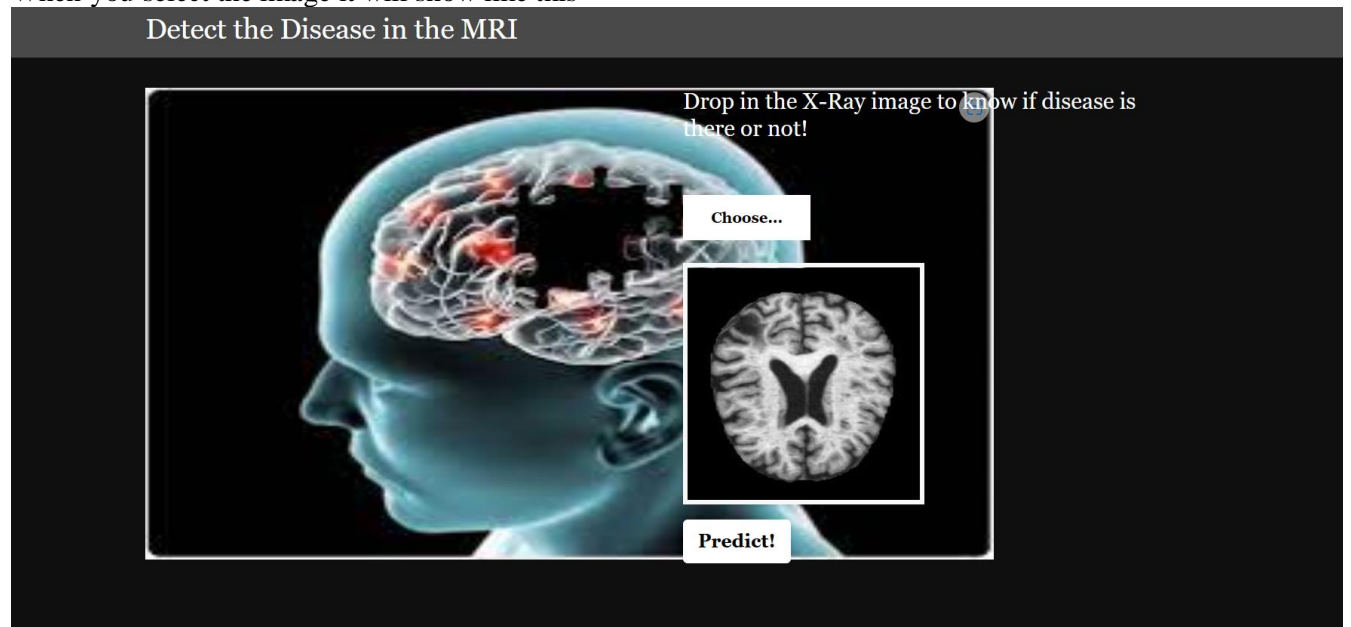
When you click on the Drop the image for detection button, you will be redirecting to the following page



When you click on the Choose button, it will redirect you to the below page



When you select the image it will show like this



Activity 2: Build Python code:

Import the libraries

```
import numpy as np
import os
from keras.preprocessing import image
import pandas as pd
import cv2
import tensorflow as tf
# Flask utils
from flask import Flask, request, render_template
from werkzeug.utils import secure_filename
from tensorflow.python.keras.backend import set_session
from tensorflow.python.keras.models import load_model

global graph
tf.compat.v1.disable_eager_execution()
sess = tf.compat.v1.Session()

import tensorflow.compat.v1 as tf
tf.disable_v2_behavior()
graph=tf.get_default_graph()
```

Loading the saved model and initializing the flask app

```
app = Flask(__name__)
set_session(sess)
# Load your trained model
model = load_model('adp.h5')
```

Render HTML pages:

```

@app.route('/', methods=['GET'])
def index():
    # Main page
    return render_template('alzheimers.html')

@app.route('/predict1', methods=['GET'])
def predict1():
    # Main page
    return render_template('alzpre.html')

```

Once we uploaded the file into the app, then verifying the file uploaded properly or not. 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 alzheimers.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('/predict', methods=['GET', 'POST'])
def upload():
    if request.method == 'POST':
        # Get the file from post request
        f = request.files['image']

        # Save the file to ./uploads
        basepath = os.path.dirname(__file__)
        file_path = os.path.join(
            basepath, 'uploads', secure_filename(f.filename))
        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)
            prediction = model.predict(x)[0][0][0]
        print(prediction)
        if prediction==0:
            text = "Mild Demented"
        elif prediction==1:
            text = "Moderate Demented"
        elif prediction==2:
            text = "Non Demented"
        else:
            text = "Very Mild Demented"

        return text

```

Here we are routing our app to upload 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 be rendered to the text that we have mentioned in the `alzheimers.html` page earlier.

Main Function:

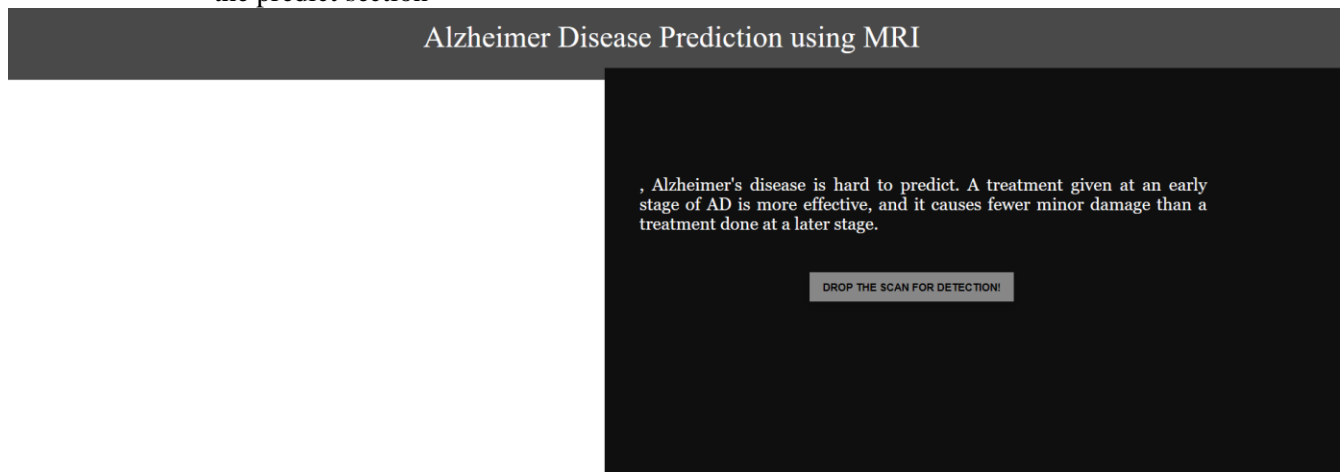
```
if __name__ == "__main__":  
    app.run(debug=False)
```

Activity 3: Run the application

- Open Spyder
- Navigate to the folder where your Python script is.
- Now click on the green play button above.
- Click on the predict button from the top right corner, enter the inputs, click on the Classify button, and see the result/prediction on the web.

```
* Serving Flask app "app" (lazy loading)  
* Environment: production  
  WARNING: This is a development server. Do not use it in a  
production deployment.  
  Use a production WSGI server instead.  
* Debug mode: off  
* Running on http://127.0.0.1:5000/ (Press CTRL+C to quit)
```

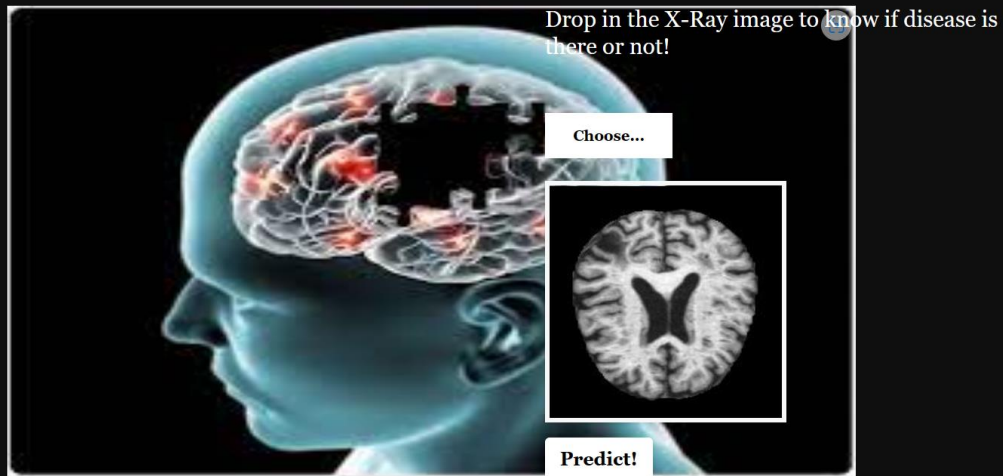
The home page looks like this. When you click on the Predict button, you'll be redirected to the predict section



click on Drop the scan for detection button

Input :

Detect the Disease in the MRI



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

Output:

