Alzheimer's Disease Prediction

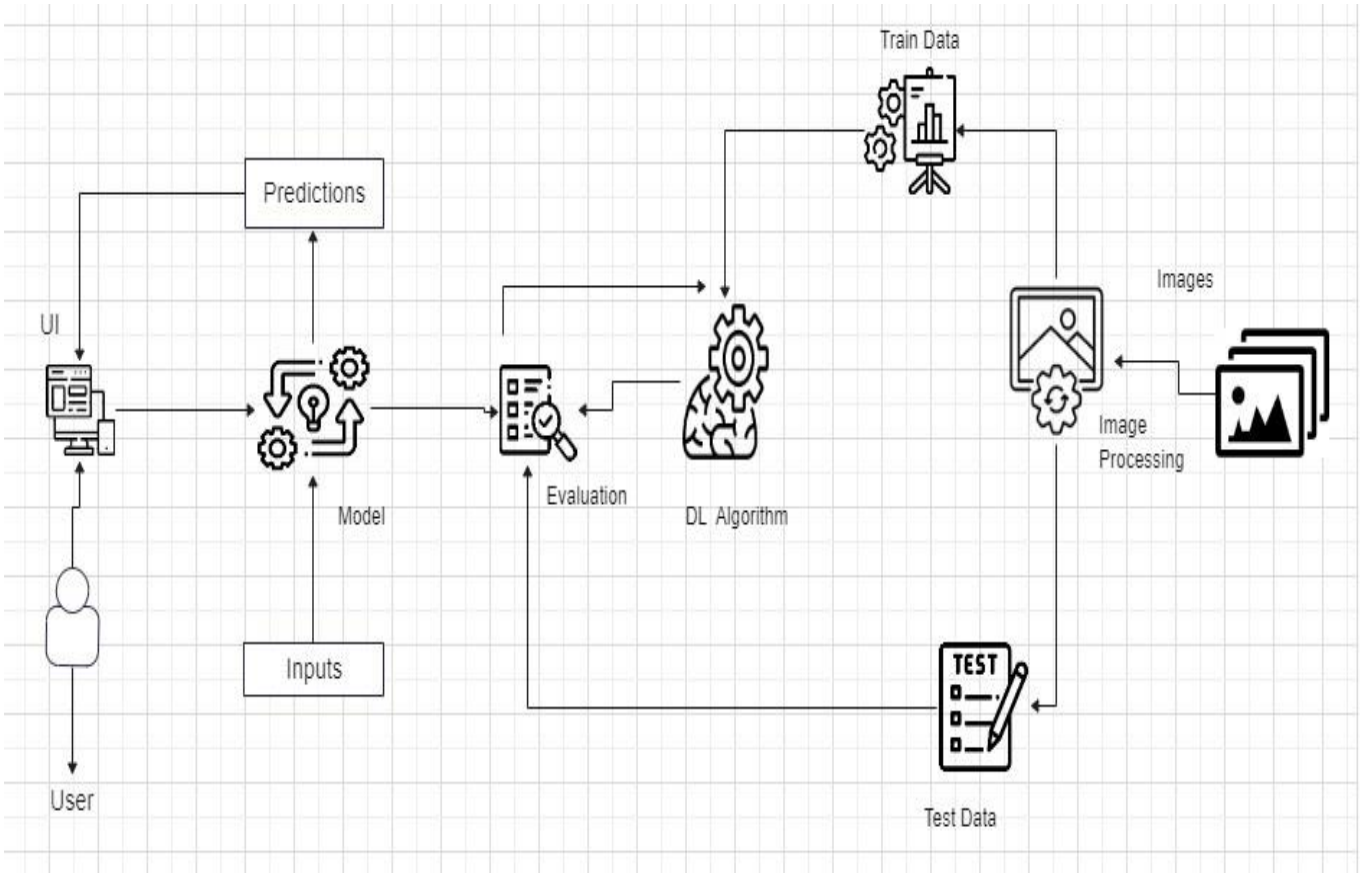
Project Description

Alzheimer's disease (AD) is a progressive and irreversible neurological disorder that affects the brain, leading to memory loss, cognitive impairment, and changes in behavior and personality. It is the most common cause of dementia among older adults and is characterized by the buildup of abnormal protein deposits in the brain, including amyloid plaques and tau tangles.

The exact cause of Alzheimer's disease is not yet fully understood, but it is believed to be influenced by a combination of genetic, environmental, and lifestyle factors. Age is also a significant risk factor, with the risk of developing the disease increasing significantly after the age of 65.The early symptoms of Alzheimer's disease may include mild memory loss, difficulty with problem-solving, and changes in mood or behavior. As the disease progresses, these symptoms become more severe, with individuals experiencing significant memory loss, difficulty communicating, and a loss of the ability to perform daily activities.

By using deep learning models like to analyze medical imaging data, it may be able to identify early signs of Alzheimer's disease before symptoms become severe. This can help healthcare providers to provide early treatment and support for patients and their families, ultimately leading to better outcomes for all involved.

Technical Architecture



Project Flow

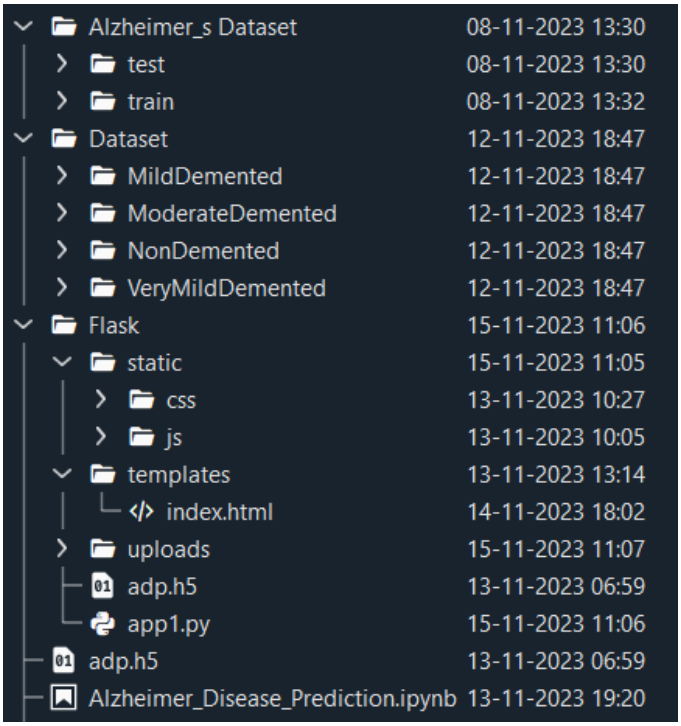
- The user interacts with the UI to choose an image.
- The chosen image is processed by a deep learning model.
- The model is integrated with a Flask application.
- The model analyzes the image and generates predictions.
- The predictions are displayed on the Flask UI for the user to see.
- This process enables users to input an image and receive accurate predictions quickly.

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

- Data Collection.
    - A Kaggle image dataset consisting of scanned images of the brain is used.
    - Create a Train and Test path.
- Image Pre-processing.
    - Import the required libraries
    - Configure Image Data Generator class
    - Handling imbalance data
    - Splitting into train-test split
- Model Building
    - CNN model as a Feature Extractor
    - Creating Sequential layers
    - Configure the Learning Process
    - Train the model
    - Save the Model
    - Test the model
- Application Building
    - Create an HTML file
    - Build Python Flask Code for creating and accessing the prediction website to obtain accurate diagnosis of the Alzheimers disease upon request.
    - Run the application.

Project Structure

Create a Project folder which contains files as shown below



- The Alzheimer_s Dataset folder contains the test and train folders downloaded from Kaggle.
- The Dataset folder contains the rearranged images from the Alzheimer Dataset into 4 folders 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 app1.py for server side scripting
- The folder consists of Alzheimer_Disease_Prediction.ipynb - model training file & adp.h5 – which is the 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 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]

Activity 2: Create a new Dataset

```
import os
import shutil

# Define the source and destination directories
source_dir = r"C:\Users\chris\Downloads\ADP Copy 1\Alzheimer_s Dataset"
destination_dir = r"C:\Users\chris\Downloads\ADP Copy 1\Dataset"

# Create the destination directory if it doesn't exist
if not os.path.exists(destination_dir):
    os.makedirs(destination_dir)

# Define the folder names
folders = ["VeryMildDemented", "ModerateDemented", "NonDemented", "MildDemented"]

# Loop through the folders
for folder in folders:
    # Create the corresponding folder in the destination directory
    destination_folder = os.path.join(destination_dir, folder)
    if not os.path.exists(destination_folder):
        os.makedirs(destination_folder)

    # Loop through the train and test folders
    for subfolder in ["train", "test"]:
        subfolder_path = os.path.join(source_dir, subfolder, folder)

        # Check if the subfolder exists
        if os.path.exists(subfolder_path):
            # Loop through the files in the subfolder
            for file_name in os.listdir(subfolder_path):
                file_path = os.path.join(subfolder_path, file_name)

                # Copy the file to the corresponding folder in the destination directory
                destination_file_path = os.path.join(destination_folder, file_name)
                shutil.copy(file_path, destination_file_path)
```

Milestone 2: Image Processing

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

Activity 1: Import Necessary Libraries

Import important libraries like TensorFlow, keras, NumPy, pandas, os, imblearn, sklearn and so on. Also import the necessary modules.

```
import numpy as np
import pandas as pd
import tensorflow as tf

from keras.preprocessing.image import ImageDataGenerator
from imblearn.over_sampling import SMOTE
from sklearn.model_selection import train_test_split
from sklearn.metrics import classification_report, confusion_matrix

from tensorflow.keras import Sequential,Input
from tensorflow.keras.layers import Conv2D, Dropout
from tensorflow.keras.layers import BatchNormalization, MaxPool2D
from tensorflow.keras.layers import Dense, Flatten
import tensorflow_addons as tfa

from tensorflow.keras.models import load_model
from tensorflow.keras.preprocessing import image
```

Activity 2: Configure ImageDataGenerator class

The ImageDataGenerator class is part of the tensorflow.keras.preprocessing.image module and is used for generating batches of augmented image data for training a neural network. This additional set of modified images provides additional batches of input for training which improves the capability of the model. The image size is set to (176,176) before being augmented. Here's a breakdown of the parameters used in this example:

- rescale=1./255: This normalizes the pixel values of the image to the range of [0, 1], which is a common practice in deep learning models.
- zoom_range=[0.99,1.01]: This applies random zooming transformations to the image, which can help the model learn to be more robust to different scales of objects in the image.
- brightness_range=[0.8,1.2]: We can use it to adjust the brightness_range of any image for Data Augmentation.
- horizontal_flip=True: This applies random horizontal flipping to the image, which can help the model learn to be more invariant to the orientation of objects in the image.
- fill_mode='constant': 'fill_mode' is a parameter used to specify the strategy for filling in newly created pixels that might appear after a transformation. When it is 'constant', any newly created pixels outside the boundaries of the original image are filled by a constant value specified by the 'cval' parameter which is 0 by default.
- data_format = 'channels_last': 'data_format' is a parameter that determines the ordering of the dimensions in the input data. It specifies how the data is structured in multi-dimensional arrays.When it is set as 'channels_last', the order for 2D case is (samples, height, width, channels) and for 3D case is (samples, depth, height, width, channels).
- Also set the batch_size as 6500 and shuffle as False and assign the correct directories for train and test.
- These data augmentation techniques can help increase the diversity and size of the training data, which can improve the performance of the model and reduce overfitting. You can use the train_data_gen object to generate batches of augmented training data on the fly, as needed by the fit() method of a Keras model.

```
#2. Configure image data generator
IMG_SIZE = 176
IMAGE_SIZE = [176,176]
DIM = (IMG_SIZE, IMG_SIZE)

ZOOM = [.99, 1.01]
BRIGHTNESS = [0.8, 1.2]
FILL_MODE = "constant"
DATA_FORMAT = "channels_last"
train_datagen = ImageDataGenerator(rescale = 1./255, brightness_range=BRIGHTNESS, zoom_range=ZOOM, data_format=DATA_FORMAT, fill_
```

```
#3. Apply image data generator functionality to train and test images
x_train = train_datagen.flow_from_directory(r"C:\Users\chris\Downloads\ADP Project\Dataset",target_size=DIM, batch_size=6500, shu
```

```
Found 6400 images belonging to 4 classes.
```

```
train_data, train_labels = x_train.next()
print(train_data.shape, train_labels.shape)
```

```
(6400, 176, 176, 3) (6400, 4)
```

```
print(x_train.class_indices)
```

```
{'MildDemented': 0, 'ModerateDemented': 1, 'NonDemented': 2, 'VeryMildDemented': 3}
```

Activity 3: Handling Imbalance Data (SMOTE)

- The SMOTE() is used to apply the SMOTE oversampling technique to balance the dataset. The random_state value provided is 42.
- The reshape(-1, IMG_SIZE*IMG_SIZE*3) is used to reshape the 2D matrix shape of the input image of (176,176,3) dimension to a 1D array of length 176*176*3.
- The sm.fit_resample technique is used to apply the SMOTE technique on the reshape train_dataset. After the sampling is applied the images are again reshaped back to a 4D array of shape (batch_size, IMG_size, IMG_size,3).
- After oversampling the data which had a shape of (6400, 176, 176, 3) has now doubled to (12,800, 176, 176, 3). The data has been doubled due to oversampling. As we can observe, from 6400 to 12800.

```
#Performing over-sampling of the data, since the classes are imbalanced
sm = SMOTE(random_state=42)
train_data, train_labels = sm.fit_resample(train_data.reshape(-1, IMG_SIZE * IMG_SIZE * 3), train_labels)
train_data = train_data.reshape(-1, IMG_SIZE, IMG_SIZE, 3)
print(train_data.shape, train_labels.shape)
```

```
(12800, 176, 176, 3) (12800, 4)
```

Activity 4: Splitting into train test split

Now we are splitting the labels into train-test split in 80:20 ratio & assigning them to train_data, train_labels & test_data, test_lables.

Validation sets of train_data, val_data, train_labels, val_labels.

```
#Splitting the data into train, test, and validation sets
train_data, test_data, train_labels, test_labels = train_test_split(train_data, train_labels, test_size = 0.2, random_state=42)
train_data, val_data, train_labels, val_labels = train_test_split(train_data, train_labels, test_size = 0.2, random_state=42)
```

Milestone 3: Model Building

Activity 1: Constructing a CNN Architecture

Define Convolution NN and Dense NN block for Sequential CNN model.

```
def conv_block(filters, act='relu'):
    """Defining a Convolutional NN block for a Sequential CNN model. """

    block = Sequential()
    block.add(Conv2D(filters, 3, activation=act, padding='same'))
    block.add(Conv2D(filters, 3, activation=act, padding='same'))
    block.add(BatchNormalization())
    block.add(MaxPool2D())

    return block
```

```
def dense_block(units, dropout_rate, act='relu'):
    """Defining a Dense NN block for a Sequential CNN model. """

    block = Sequential()
    block.add(Dense(units, activation=act))
    block.add(BatchNormalization())
    block.add(Dropout(dropout_rate))

    return block
```

Define a construct_model function block for constructing a Sequential CNN architecture for performing the classification task.

```
def construct_model(act='relu'):
    """Constructing a Sequential CNN architecture for performing the classification task. """

    model = Sequential([
        Input(shape=(*IMAGE_SIZE, 3)),
        Conv2D(16, 3, activation=act, padding='same'),
        Conv2D(16, 3, activation=act, padding='same'),
        MaxPool2D(),
        conv_block(32),
        conv_block(64),
        conv_block(128),
        Dropout(0.2),
        conv_block(256),
        Dropout(0.2),
        Flatten(),
        dense_block(512, 0.7),
        dense_block(128, 0.5),
        dense_block(64, 0.3),
        Dense(4, activation='softmax')
    ], name = "cnn_model")

    return model
```

Activity 2: 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. In this case, we are using the 'adam' optimizer.

Metrics are used to evaluate the performance of your model. They provide insights into how well the model is performing but are not used during the training process. In our code, we have defined a set of custom metrics using TensorFlow's Keras metrics, including Categorical Accuracy, AUC and F1 Score. These metrics will be used to measure various aspects of the model's performance.

We have also defined a custom callback function to stop training our model when accuracy goes above 99%.

```
model = construct_model()
```

```
METRICS = [tf.keras.metrics.CategoricalAccuracy(name='acc'),tf.keras.metrics.AUC(name='auc'),tfa.metrics.F1Score(num_classes=4)]
model.compile(optimizer='adam',loss=tf.losses.CategoricalCrossentropy(),metrics=METRICS)
```

```
#Defining a custom callback function to stop training our model when accuracy goes above 99%

class MyCallback(tf.keras.callbacks.Callback):
    def on_epoch_end(self, epoch, logs={}):
        if logs.get('val_acc') > 0.99:
            print("\nReached accuracy threshold! Terminating training.")
            self.model.stop_training = True

my_callback = MyCallback()
```

Activity 3: Train the Model

Now, we will train our model with our image dataset. The model is trained for 100 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 training accuracy increases. The fit function is used to train a deep learning neural network.

```
#Fit the training data to the model and validate it using the validation data
model.fit(train_data, train_labels, validation_data=(val_data, val_labels),callbacks=my_callback,epochs=100)
```

```
Epoch 1/100
256/256 [==============================] - 145s 562ms/step - loss: 1.6891 - acc: 0.2765 - auc: 0.5282 - f1_score:
0.2753 - val_loss: 1.9973 - val_acc: 0.2422 - val_auc: 0.5185 - val_f1_score: 0.0975
Epoch 2/100
256/256 [==============================] - 145s 567ms/step - loss: 1.4286 - acc: 0.3192 - auc: 0.5847 - f1_score:
0.3145 - val_loss: 1.5013 - val_acc: 0.2393 - val_auc: 0.5012 - val_f1_score: 0.0987
Epoch 3/100
256/256 [==============================] - 816s 3s/step - loss: 0.9941 - acc: 0.5436 - auc: 0.8183 - f1_score: 0.52
02 - val_loss: 1.8700 - val_acc: 0.4146 - val_auc: 0.6506 - val_f1_score: 0.3658
Epoch 4/100
256/256 [==============================] - 153s 595ms/step - loss: 0.7979 - acc: 0.6385 - auc: 0.8830 - f1_score:
0.6302 - val_loss: 2.1478 - val_acc: 0.4775 - val_auc: 0.7073 - val_f1_score: 0.3594
Epoch 5/100
256/256 [==============================] - 155s 605ms/step - loss: 0.6883 - acc: 0.6904 - auc: 0.9125 - f1_score:
0.6841 - val_loss: 1.1839 - val_acc: 0.4956 - val_auc: 0.8323 - val_f1_score: 0.3769
Epoch 6/100
256/256 [==============================] - 233s 912ms/step - loss: 0.6539 - acc: 0.7031 - auc: 0.9213 - f1_score:
0.6997 - val_loss: 1.0413 - val_acc: 0.6060 - val_auc: 0.8701 - val_f1_score: 0.5719
```

```
Epoch 7/100
256/256 [==============================] - 216s 845ms/step - loss: 0.6184 - acc: 0.7233 - auc: 0.9297 - f1_score:
0.7227 - val_loss: 0.6159 - val_acc: 0.6958 - val_auc: 0.9264 - val_f1_score: 0.6964
Epoch 8/100
256/256 [==============================] - 226s 884ms/step - loss: 0.5485 - acc: 0.7570 - auc: 0.9445 - f1_score:
0.7564 - val_loss: 0.6865 - val_acc: 0.6328 - val_auc: 0.9048 - val_f1_score: 0.6020
Epoch 9/100
256/256 [==============================] - 251s 981ms/step - loss: 0.5481 - acc: 0.7615 - auc: 0.9447 - f1_score:
0.7612 - val_loss: 0.8633 - val_acc: 0.6333 - val_auc: 0.8720 - val_f1_score: 0.6429
Epoch 10/100
256/256 [==============================] - 223s 871ms/step - loss: 0.4852 - acc: 0.7883 - auc: 0.9562 - f1_score:
0.7888 - val_loss: 0.6715 - val_acc: 0.6777 - val_auc: 0.9150 - val_f1_score: 0.6657
Epoch 11/100
256/256 [==============================] - 213s 834ms/step - loss: 0.4496 - acc: 0.8059 - auc: 0.9627 - f1_score:
0.8060 - val_loss: 0.4677 - val_acc: 0.7920 - val_auc: 0.9592 - val_f1_score: 0.7865
Epoch 12/100
256/256 [==============================] - 219s 854ms/step - loss: 0.4074 - acc: 0.8285 - auc: 0.9689 - f1_score:
0.8288 - val_loss: 0.8827 - val_acc: 0.6479 - val_auc: 0.8891 - val_f1_score: 0.5976
```

```
Epoch 13/100
256/256 [==============================] - 218s 850ms/step - loss: 0.4339 - acc: 0.8164 - auc: 0.9654 - f1_score:
0.8166 - val_loss: 0.5375 - val_acc: 0.7305 - val_auc: 0.9532 - val_f1_score: 0.6629
Epoch 14/100
256/256 [==============================] - 219s 855ms/step - loss: 0.3637 - acc: 0.8467 - auc: 0.9752 - f1_score:
0.8462 - val_loss: 1.6647 - val_acc: 0.5508 - val_auc: 0.8326 - val_f1_score: 0.4944
Epoch 15/100
256/256 [==============================] - 213s 832ms/step - loss: 0.3477 - acc: 0.8601 - auc: 0.9774 - f1_score:
0.8601 - val_loss: 1.2978 - val_acc: 0.5728 - val_auc: 0.8516 - val_f1_score: 0.5450
Epoch 16/100
256/256 [==============================] - 214s 836ms/step - loss: 0.3018 - acc: 0.8799 - auc: 0.9827 - f1_score:
0.8797 - val_loss: 0.4002 - val_acc: 0.8330 - val_auc: 0.9729 - val_f1_score: 0.8344
Epoch 17/100
256/256 [==============================] - 233s 910ms/step - loss: 0.2894 - acc: 0.8827 - auc: 0.9841 - f1_score:
0.8825 - val_loss: 0.6789 - val_acc: 0.7505 - val_auc: 0.9537 - val_f1_score: 0.7156
Epoch 18/100
256/256 [==============================] - 246s 962ms/step - loss: 0.2828 - acc: 0.8903 - auc: 0.9848 - f1_score:
0.8901 - val_loss: 0.3758 - val_acc: 0.8521 - val_auc: 0.9766 - val_f1_score: 0.8523
```

```
Epoch 19/100
256/256 [==============================] - 252s 986ms/step - loss: 0.2316 - acc: 0.9147 - auc: 0.9893 - f1_score:
0.9145 - val_loss: 0.5368 - val_acc: 0.7900 - val_auc: 0.9605 - val_f1_score: 0.7639
Epoch 20/100
256/256 [==============================] - 241s 941ms/step - loss: 0.1983 - acc: 0.9264 - auc: 0.9922 - f1_score:
0.9264 - val_loss: 0.8691 - val_acc: 0.7163 - val_auc: 0.9227 - val_f1_score: 0.7108
Epoch 21/100
256/256 [==============================] - 240s 939ms/step - loss: 0.1885 - acc: 0.9327 - auc: 0.9928 - f1_score:
0.9326 - val_loss: 0.6804 - val_acc: 0.7798 - val_auc: 0.9473 - val_f1_score: 0.7735
Epoch 22/100
256/256 [==============================] - 239s 934ms/step - loss: 0.1591 - acc: 0.9409 - auc: 0.9947 - f1_score:
0.9407 - val_loss: 0.2856 - val_acc: 0.8896 - val_auc: 0.9861 - val_f1_score: 0.8877
Epoch 23/100
256/256 [==============================] - 248s 967ms/step - loss: 0.1460 - acc: 0.9482 - auc: 0.9951 - f1_score:
0.9482 - val_loss: 1.2427 - val_acc: 0.6699 - val_auc: 0.8628 - val_f1_score: 0.6741
Epoch 24/100
256/256 [==============================] - 243s 948ms/step - loss: 0.1459 - acc: 0.9476 - auc: 0.9951 - f1_score:
0.9476 - val_loss: 0.4146 - val_acc: 0.8491 - val_auc: 0.9790 - val_f1_score: 0.8425
```

```
Epoch 25/100
256/256 [==============================] - 292s 1s/step - loss: 0.1275 - acc: 0.9585 - auc: 0.9959 - f1_score: 0.95
84 - val_loss: 0.1938 - val_acc: 0.9277 - val_auc: 0.9928 - val_f1_score: 0.9274
Epoch 26/100
256/256 [==============================] - 273s 1s/step - loss: 0.1140 - acc: 0.9590 - auc: 0.9972 - f1_score: 0.95
89 - val_loss: 0.3242 - val_acc: 0.8940 - val_auc: 0.9834 - val_f1_score: 0.8938
Epoch 27/100
256/256 [==============================] - 251s 980ms/step - loss: 0.1071 - acc: 0.9626 - auc: 0.9971 - f1_score:
0.9626 - val_loss: 0.2522 - val_acc: 0.9077 - val_auc: 0.9902 - val_f1_score: 0.9056
Epoch 28/100
256/256 [==============================] - 233s 909ms/step - loss: 0.0856 - acc: 0.9739 - auc: 0.9978 - f1_score:
0.9739 - val_loss: 0.3217 - val_acc: 0.8882 - val_auc: 0.9831 - val_f1_score: 0.8898
Epoch 29/100
256/256 [==============================] - 235s 918ms/step - loss: 0.0934 - acc: 0.9691 - auc: 0.9975 - f1_score:
0.9691 - val_loss: 0.3955 - val_acc: 0.8828 - val_auc: 0.9802 - val_f1_score: 0.8757
Epoch 30/100
256/256 [==============================] - 247s 965ms/step - loss: 0.0907 - acc: 0.9697 - auc: 0.9975 - f1_score:
0.9697 - val_loss: 0.3866 - val_acc: 0.8745 - val_auc: 0.9816 - val_f1_score: 0.8702
```

```
Epoch 31/100
256/256 [==============================] - 237s 926ms/step - loss: 0.0759 - acc: 0.9764 - auc: 0.9983 - f1_score:
0.9764 - val_loss: 0.2729 - val_acc: 0.9116 - val_auc: 0.9872 - val_f1_score: 0.9101
Epoch 32/100
256/256 [==============================] - 211s 825ms/step - loss: 0.0983 - acc: 0.9657 - auc: 0.9976 - f1_score:
0.9657 - val_loss: 0.2741 - val_acc: 0.9043 - val_auc: 0.9880 - val_f1_score: 0.9059
Epoch 33/100
256/256 [==============================] - 219s 855ms/step - loss: 0.0694 - acc: 0.9766 - auc: 0.9988 - f1_score:
0.9766 - val_loss: 0.2249 - val_acc: 0.9268 - val_auc: 0.9907 - val_f1_score: 0.9258
Epoch 34/100
256/256 [==============================] - 213s 833ms/step - loss: 0.0746 - acc: 0.9761 - auc: 0.9979 - f1_score:
0.9761 - val_loss: 0.5122 - val_acc: 0.8633 - val_auc: 0.9691 - val_f1_score: 0.8593
Epoch 35/100
256/256 [==============================] - 213s 832ms/step - loss: 0.0665 - acc: 0.9795 - auc: 0.9982 - f1_score:
0.9795 - val_loss: 0.2069 - val_acc: 0.9331 - val_auc: 0.9917 - val_f1_score: 0.9327
Epoch 36/100
256/256 [==============================] - 212s 826ms/step - loss: 0.0673 - acc: 0.9797 - auc: 0.9982 - f1_score:
0.9797 - val_loss: 0.3220 - val_acc: 0.9004 - val_auc: 0.9843 - val_f1_score: 0.8989
```

```
Epoch 37/100
256/256 [==============================] - 213s 832ms/step - loss: 0.0702 - acc: 0.9773 - auc: 0.9985 - f1_score:
0.9773 - val_loss: 0.2536 - val_acc: 0.9238 - val_auc: 0.9869 - val_f1_score: 0.9242
Epoch 38/100
256/256 [==============================] - 218s 853ms/step - loss: 0.0454 - acc: 0.9861 - auc: 0.9993 - f1_score:
0.9861 - val_loss: 0.2527 - val_acc: 0.9331 - val_auc: 0.9845 - val_f1_score: 0.9324
Epoch 39/100
256/256 [==============================] - 214s 836ms/step - loss: 0.0564 - acc: 0.9816 - auc: 0.9986 - f1_score:
0.9816 - val_loss: 0.3924 - val_acc: 0.8921 - val_auc: 0.9762 - val_f1_score: 0.8933
```

```
Epoch 40/100
256/256 [==============================] - 207s 810ms/step - loss: 0.0514 - acc: 0.9829 - auc: 0.9990 - f1_score:
0.9829 - val_loss: 1.1440 - val_acc: 0.7554 - val_auc: 0.9148 - val_f1_score: 0.7481
Epoch 41/100
256/256 [==============================] - 203s 792ms/step - loss: 0.0538 - acc: 0.9832 - auc: 0.9990 - f1_score:
0.9832 - val_loss: 0.2511 - val_acc: 0.9253 - val_auc: 0.9899 - val_f1_score: 0.9243
Epoch 42/100
256/256 [==============================] - 215s 840ms/step - loss: 0.0508 - acc: 0.9839 - auc: 0.9991 - f1_score:
0.9839 - val_loss: 0.5120 - val_acc: 0.8608 - val_auc: 0.9717 - val_f1_score: 0.8501
```

```
Epoch 43/100
256/256 [==============================] - 201s 784ms/step - loss: 0.0578 - acc: 0.9822 - auc: 0.9986 - f1_score:
0.9822 - val_loss: 0.2956 - val_acc: 0.9048 - val_auc: 0.9865 - val_f1_score: 0.9033
Epoch 44/100
256/256 [==============================] - 200s 781ms/step - loss: 0.0605 - acc: 0.9799 - auc: 0.9983 - f1_score:
0.9798 - val_loss: 0.1963 - val_acc: 0.9390 - val_auc: 0.9920 - val_f1_score: 0.9388
Epoch 45/100
256/256 [==============================] - 203s 793ms/step - loss: 0.0465 - acc: 0.9854 - auc: 0.9992 - f1_score:
0.9853 - val_loss: 0.1754 - val_acc: 0.9512 - val_auc: 0.9921 - val_f1_score: 0.9507
Epoch 46/100
256/256 [==============================] - 204s 797ms/step - loss: 0.0302 - acc: 0.9910 - auc: 0.9996 - f1_score:
0.9910 - val_loss: 0.6197 - val_acc: 0.8555 - val_auc: 0.9593 - val_f1_score: 0.8549
Epoch 47/100
256/256 [==============================] - 197s 769ms/step - loss: 0.0794 - acc: 0.9725 - auc: 0.9979 - f1_score:
0.9725 - val_loss: 0.2373 - val_acc: 0.9297 - val_auc: 0.9894 - val_f1_score: 0.9285
Epoch 48/100
256/256 [==============================] - 195s 761ms/step - loss: 0.0374 - acc: 0.9879 - auc: 0.9995 - f1_score:
0.9879 - val_loss: 0.2649 - val_acc: 0.9292 - val_auc: 0.9846 - val_f1_score: 0.9293

Epoch 49/100
256/256 [==============================] - 196s 766ms/step - loss: 0.0338 - acc: 0.9890 - auc: 0.9993 - f1_score:
0.9890 - val_loss: 0.1791 - val_acc: 0.9473 - val_auc: 0.9923 - val_f1_score: 0.9468
Epoch 50/100
256/256 [==============================] - 209s 815ms/step - loss: 0.0491 - acc: 0.9847 - auc: 0.9989 - f1_score:
0.9847 - val_loss: 0.1678 - val_acc: 0.9521 - val_auc: 0.9918 - val_f1_score: 0.9522
Epoch 51/100
256/256 [==============================] - 179s 700ms/step - loss: 0.0329 - acc: 0.9907 - auc: 0.9995 - f1_score:
0.9907 - val_loss: 0.2764 - val_acc: 0.9258 - val_auc: 0.9859 - val_f1_score: 0.9241
Epoch 52/100
256/256 [==============================] - 179s 700ms/step - loss: 0.0335 - acc: 0.9894 - auc: 0.9995 - f1_score:
0.9894 - val_loss: 0.2164 - val_acc: 0.9414 - val_auc: 0.9895 - val_f1_score: 0.9406
Epoch 53/100
256/256 [==============================] - 187s 729ms/step - loss: 0.0410 - acc: 0.9869 - auc: 0.9993 - f1_score:
0.9869 - val_loss: 0.5101 - val_acc: 0.8667 - val_auc: 0.9631 - val_f1_score: 0.8676
Epoch 54/100
256/256 [==============================] - 193s 755ms/step - loss: 0.0475 - acc: 0.9856 - auc: 0.9988 - f1_score:
0.9856 - val_loss: 0.3262 - val_acc: 0.9165 - val_auc: 0.9787 - val_f1_score: 0.9156

Epoch 55/100
256/256 [==============================] - 195s 761ms/step - loss: 0.0360 - acc: 0.9882 - auc: 0.9995 - f1_score:
0.9882 - val_loss: 0.2790 - val_acc: 0.9263 - val_auc: 0.9854 - val_f1_score: 0.9256
Epoch 56/100
256/256 [==============================] - 191s 747ms/step - loss: 0.0228 - acc: 0.9938 - auc: 0.9997 - f1_score:
0.9938 - val_loss: 0.3429 - val_acc: 0.9233 - val_auc: 0.9771 - val_f1_score: 0.9233
Epoch 57/100
256/256 [==============================] - 190s 742ms/step - loss: 0.0303 - acc: 0.9899 - auc: 0.9994 - f1_score:
0.9899 - val_loss: 0.2657 - val_acc: 0.9307 - val_auc: 0.9845 - val_f1_score: 0.9297
Epoch 58/100
256/256 [==============================] - 182s 710ms/step - loss: 0.0430 - acc: 0.9862 - auc: 0.9990 - f1_score:
0.9862 - val_loss: 0.1925 - val_acc: 0.9458 - val_auc: 0.9917 - val_f1_score: 0.9459
Epoch 59/100
256/256 [==============================] - 183s 715ms/step - loss: 0.0351 - acc: 0.9888 - auc: 0.9994 - f1_score:
0.9888 - val_loss: 0.8623 - val_acc: 0.7896 - val_auc: 0.9312 - val_f1_score: 0.7827
Epoch 60/100
256/256 [==============================] - 183s 715ms/step - loss: 0.0373 - acc: 0.9880 - auc: 0.9994 - f1_score:
0.9880 - val_loss: 0.2484 - val_acc: 0.9331 - val_auc: 0.9877 - val_f1_score: 0.9322

Epoch 61/100
256/256 [==============================] - 184s 719ms/step - loss: 0.0360 - acc: 0.9901 - auc: 0.9991 - f1_score:
0.9901 - val_loss: 0.1849 - val_acc: 0.9507 - val_auc: 0.9916 - val_f1_score: 0.9501
Epoch 62/100
256/256 [==============================] - 197s 770ms/step - loss: 0.0193 - acc: 0.9941 - auc: 0.9997 - f1_score:
0.9941 - val_loss: 0.4036 - val_acc: 0.9077 - val_auc: 0.9757 - val_f1_score: 0.9070
Epoch 63/100
256/256 [==============================] - 201s 785ms/step - loss: 0.0340 - acc: 0.9901 - auc: 0.9992 - f1_score:
0.9901 - val_loss: 0.2027 - val_acc: 0.9448 - val_auc: 0.9896 - val_f1_score: 0.9450
Epoch 64/100
256/256 [==============================] - 200s 783ms/step - loss: 0.0439 - acc: 0.9861 - auc: 0.9990 - f1_score:
0.9861 - val_loss: 0.1760 - val_acc: 0.9521 - val_auc: 0.9913 - val_f1_score: 0.9522
Epoch 65/100
256/256 [==============================] - 201s 784ms/step - loss: 0.0369 - acc: 0.9897 - auc: 0.9994 - f1_score:
0.9897 - val_loss: 0.1864 - val_acc: 0.9473 - val_auc: 0.9909 - val_f1_score: 0.9472
Epoch 66/100
256/256 [==============================] - 203s 792ms/step - loss: 0.0375 - acc: 0.9877 - auc: 0.9990 - f1_score:
0.9877 - val_loss: 0.1647 - val_acc: 0.9541 - val_auc: 0.9925 - val_f1_score: 0.9539
```

```
Epoch 67/100
256/256 [==============================] - 202s 789ms/step - loss: 0.0201 - acc: 0.9940 - auc: 0.9997 - f1_score:
0.9940 - val_loss: 0.2174 - val_acc: 0.9453 - val_auc: 0.9893 - val_f1_score: 0.9452
Epoch 68/100
256/256 [==============================] - 204s 797ms/step - loss: 0.0295 - acc: 0.9911 - auc: 0.9994 - f1_score:
0.9911 - val_loss: 0.1949 - val_acc: 0.9492 - val_auc: 0.9909 - val_f1_score: 0.9489
Epoch 69/100
256/256 [==============================] - 204s 797ms/step - loss: 0.0307 - acc: 0.9908 - auc: 0.9994 - f1_score:
0.9908 - val_loss: 0.1685 - val_acc: 0.9507 - val_auc: 0.9924 - val_f1_score: 0.9504
Epoch 70/100
256/256 [==============================] - 204s 798ms/step - loss: 0.0330 - acc: 0.9891 - auc: 0.9991 - f1_score:
0.9891 - val_loss: 0.1688 - val_acc: 0.9536 - val_auc: 0.9922 - val_f1_score: 0.9538
Epoch 71/100
256/256 [==============================] - 205s 802ms/step - loss: 0.0260 - acc: 0.9927 - auc: 0.9995 - f1_score:
0.9927 - val_loss: 0.1847 - val_acc: 0.9507 - val_auc: 0.9917 - val_f1_score: 0.9502
Epoch 72/100
256/256 [==============================] - 204s 799ms/step - loss: 0.0182 - acc: 0.9946 - auc: 0.9996 - f1_score:
0.9946 - val_loss: 0.2271 - val_acc: 0.9443 - val_auc: 0.9869 - val_f1_score: 0.9447

Epoch 73/100
256/256 [==============================] - 205s 800ms/step - loss: 0.0256 - acc: 0.9932 - auc: 0.9993 - f1_score:
0.9932 - val_loss: 0.1973 - val_acc: 0.9463 - val_auc: 0.9900 - val_f1_score: 0.9459
Epoch 74/100
256/256 [==============================] - 206s 803ms/step - loss: 0.0250 - acc: 0.9919 - auc: 0.9996 - f1_score:
0.9919 - val_loss: 0.1898 - val_acc: 0.9497 - val_auc: 0.9896 - val_f1_score: 0.9499
Epoch 75/100
256/256 [==============================] - 206s 805ms/step - loss: 0.0200 - acc: 0.9930 - auc: 0.9998 - f1_score:
0.9930 - val_loss: 0.2569 - val_acc: 0.9419 - val_auc: 0.9854 - val_f1_score: 0.9422
Epoch 76/100
256/256 [==============================] - 207s 808ms/step - loss: 0.0175 - acc: 0.9950 - auc: 0.9996 - f1_score:
0.9950 - val_loss: 0.3128 - val_acc: 0.9287 - val_auc: 0.9804 - val_f1_score: 0.9287
Epoch 77/100
256/256 [==============================] - 207s 810ms/step - loss: 0.0396 - acc: 0.9886 - auc: 0.9987 - f1_score:
0.9886 - val_loss: 0.1571 - val_acc: 0.9541 - val_auc: 0.9944 - val_f1_score: 0.9538
Epoch 78/100
256/256 [==============================] - 208s 813ms/step - loss: 0.0163 - acc: 0.9957 - auc: 0.9997 - f1_score:
0.9957 - val_loss: 0.2138 - val_acc: 0.9482 - val_auc: 0.9885 - val_f1_score: 0.9479

Epoch 79/100
256/256 [==============================] - 206s 806ms/step - loss: 0.0170 - acc: 0.9946 - auc: 0.9997 - f1_score:
0.9946 - val_loss: 0.2035 - val_acc: 0.9507 - val_auc: 0.9880 - val_f1_score: 0.9506
Epoch 80/100
256/256 [==============================] - 205s 802ms/step - loss: 0.0230 - acc: 0.9937 - auc: 0.9996 - f1_score:
0.9937 - val_loss: 0.2214 - val_acc: 0.9414 - val_auc: 0.9883 - val_f1_score: 0.9405
Epoch 81/100
256/256 [==============================] - 208s 811ms/step - loss: 0.0277 - acc: 0.9922 - auc: 0.9994 - f1_score:
0.9922 - val_loss: 0.2643 - val_acc: 0.9287 - val_auc: 0.9857 - val_f1_score: 0.9292
Epoch 82/100
256/256 [==============================] - 205s 802ms/step - loss: 0.0252 - acc: 0.9918 - auc: 0.9994 - f1_score:
0.9918 - val_loss: 0.2049 - val_acc: 0.9507 - val_auc: 0.9875 - val_f1_score: 0.9509
Epoch 83/100
256/256 [==============================] - 205s 800ms/step - loss: 0.0206 - acc: 0.9938 - auc: 0.9998 - f1_score:
0.9938 - val_loss: 0.2573 - val_acc: 0.9370 - val_auc: 0.9828 - val_f1_score: 0.9374
Epoch 84/100
256/256 [==============================] - 205s 801ms/step - loss: 0.0151 - acc: 0.9956 - auc: 0.9998 - f1_score:
0.9956 - val_loss: 0.1761 - val_acc: 0.9526 - val_auc: 0.9912 - val_f1_score: 0.9525

Epoch 85/100
256/256 [==============================] - 206s 804ms/step - loss: 0.0232 - acc: 0.9933 - auc: 0.9995 - f1_score:
0.9933 - val_loss: 0.1840 - val_acc: 0.9502 - val_auc: 0.9918 - val_f1_score: 0.9501
Epoch 86/100
256/256 [==============================] - 211s 826ms/step - loss: 0.0170 - acc: 0.9944 - auc: 0.9997 - f1_score:
0.9944 - val_loss: 0.2619 - val_acc: 0.9443 - val_auc: 0.9863 - val_f1_score: 0.9441
Epoch 87/100
256/256 [==============================] - 207s 808ms/step - loss: 0.0157 - acc: 0.9958 - auc: 0.9996 - f1_score:
0.9958 - val_loss: 0.2902 - val_acc: 0.9365 - val_auc: 0.9817 - val_f1_score: 0.9361
Epoch 88/100
256/256 [==============================] - 206s 806ms/step - loss: 0.0236 - acc: 0.9927 - auc: 0.9996 - f1_score:
0.9927 - val_loss: 0.1584 - val_acc: 0.9590 - val_auc: 0.9924 - val_f1_score: 0.9589
Epoch 89/100
256/256 [==============================] - 205s 801ms/step - loss: 0.0222 - acc: 0.9932 - auc: 0.9996 - f1_score:
0.9932 - val_loss: 1.0883 - val_acc: 0.7988 - val_auc: 0.9231 - val_f1_score: 0.7939
Epoch 90/100
256/256 [==============================] - 204s 798ms/step - loss: 0.0312 - acc: 0.9904 - auc: 0.9994 - f1_score:
0.9904 - val_loss: 0.1729 - val_acc: 0.9561 - val_auc: 0.9897 - val_f1_score: 0.9562
```

```
Epoch 91/100
256/256 [==============================] - 205s 801ms/step - loss: 0.0188 - acc: 0.9943 - auc: 0.9996 - f1_score:
0.9943 - val_loss: 0.1855 - val_acc: 0.9541 - val_auc: 0.9916 - val_f1_score: 0.9540
Epoch 92/100
256/256 [==============================] - 209s 816ms/step - loss: 0.0124 - acc: 0.9963 - auc: 0.9999 - f1_score:
0.9963 - val_loss: 0.2140 - val_acc: 0.9521 - val_auc: 0.9888 - val_f1_score: 0.9523
Epoch 93/100
256/256 [==============================] - 206s 807ms/step - loss: 0.0123 - acc: 0.9973 - auc: 0.9997 - f1_score:
0.9973 - val_loss: 0.3036 - val_acc: 0.9375 - val_auc: 0.9874 - val_f1_score: 0.9367
Epoch 94/100
256/256 [==============================] - 208s 812ms/step - loss: 0.0185 - acc: 0.9944 - auc: 0.9996 - f1_score:
0.9944 - val_loss: 0.3299 - val_acc: 0.9375 - val_auc: 0.9854 - val_f1_score: 0.9366
Epoch 95/100
256/256 [==============================] - 210s 820ms/step - loss: 0.0392 - acc: 0.9884 - auc: 0.9988 - f1_score:
0.9884 - val_loss: 0.1867 - val_acc: 0.9497 - val_auc: 0.9911 - val_f1_score: 0.9492
Epoch 96/100
256/256 [==============================] - 215s 839ms/step - loss: 0.0120 - acc: 0.9965 - auc: 0.9999 - f1_score:
0.9965 - val_loss: 0.2038 - val_acc: 0.9521 - val_auc: 0.9881 - val_f1_score: 0.9522

Epoch 97/100
256/256 [==============================] - 212s 830ms/step - loss: 0.0150 - acc: 0.9957 - auc: 0.9997 - f1_score:
0.9957 - val_loss: 0.2031 - val_acc: 0.9521 - val_auc: 0.9885 - val_f1_score: 0.9518
Epoch 98/100
256/256 [==============================] - 205s 802ms/step - loss: 0.0188 - acc: 0.9945 - auc: 0.9996 - f1_score:
0.9945 - val_loss: 0.1918 - val_acc: 0.9546 - val_auc: 0.9887 - val_f1_score: 0.9546
Epoch 99/100
256/256 [==============================] - 204s 795ms/step - loss: 0.0188 - acc: 0.9943 - auc: 0.9998 - f1_score:
0.9943 - val_loss: 0.2093 - val_acc: 0.9521 - val_auc: 0.9873 - val_f1_score: 0.9516
Epoch 100/100
256/256 [==============================] - 204s 796ms/step - loss: 0.0089 - acc: 0.9972 - auc: 1.0000 - f1_score:
0.9972 - val_loss: 0.2315 - val_acc: 0.9551 - val_auc: 0.9872 - val_f1_score: 0.9548

<keras.src.callbacks.History at 0x28a0286d0>
```

```
model.summary()
```

```
Model: "cnn_model"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 conv2d (Conv2D)             (None, 176, 176, 16)      448

 conv2d_1 (Conv2D)           (None, 176, 176, 16)      2320

 max_pooling2d (MaxPooling2  (None, 88, 88, 16)        0
 D)

 sequential (Sequential)     (None, 44, 44, 32)        14016

 sequential_1 (Sequential)   (None, 22, 22, 64)        55680

 sequential_2 (Sequential)   (None, 11, 11, 128)       221952

 dropout (Dropout)           (None, 11, 11, 128)       0

 sequential_3 (Sequential)   (None, 5, 5, 256)         886272

 dropout_1 (Dropout)         (None, 5, 5, 256)         0

 flatten (Flatten)           (None, 6400)              0

 sequential_4 (Sequential)   (None, 512)               3279360

 sequential_5 (Sequential)   (None, 128)               66176

 sequential_6 (Sequential)   (None, 64)                8512

 dense_3 (Dense)             (None, 4)                 260

=================================================================
Total params: 4534996 (17.30 MB)
Trainable params: 4532628 (17.29 MB)
Non-trainable params: 2368 (9.25 KB)
_________________________________________________________________
```

Activity 4: Evaluate the Model

```
# Evaluating the model on the data

train_scores = model.evaluate(train_data, train_labels)
val_scores = model.evaluate(val_data, val_labels)
test_scores = model.evaluate(test_data, test_labels)

print("Training Accuracy: %.2f%%"%(train_scores[1] * 100))
print("Validation Accuracy: %.2f%%"%(val_scores[1] * 100))
print("Testing Accuracy: %.2f%%"%(test_scores[1] * 100))
```

```
256/256 [==============================] - 52s 203ms/step - loss: 0.0019 - acc: 0.9993 - auc: 1.0000 - f1_score: 0.
9993
64/64 [==============================] - 13s 200ms/step - loss: 0.2315 - acc: 0.9551 - auc: 0.9872 - f1_score: 0.95
48
80/80 [==============================] - 16s 200ms/step - loss: 0.2358 - acc: 0.9484 - auc: 0.9868 - f1_score: 0.94
82
Training Accuracy: 99.93%
Validation Accuracy: 95.51%
Testing Accuracy: 94.84%
```

Activity 5: Save the Model

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

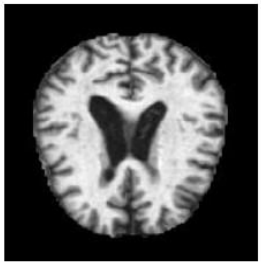
The model is saved with .h5 extension. A .h5 file is a data file saved in the Hierarchical Data Format (HDF). It contains multidimensional arrays of scientific data

Activity 6: 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. Use an image to test the model and augment the image.

```
model = load_model(r"C:\Users\chris\Downloads\ADP Project\adp.h5",compile=False)
```

```
img = image.load_img(r"C:\Users\chris\Downloads\ADP Project\Dataset\ModerateDemented\moderateDem49.jpg",target_size=(176,176))
img
```



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

```
pred = model.predict(x)
pred_class = np.argmax(pred,axis=1)
index = ['MildDemented','ModerateDemented','NonDemented','VeryMildDemented']
result = str(index[pred_class[0]])
result
```

```
1/1 [==============================] - 0s 253ms/step

'ModerateDemented'
```

So, our model will give the index position of the label. In this case, the 1st index is for Moderate Demented, which has been predicted correctly.

Milestone 4: Application Building

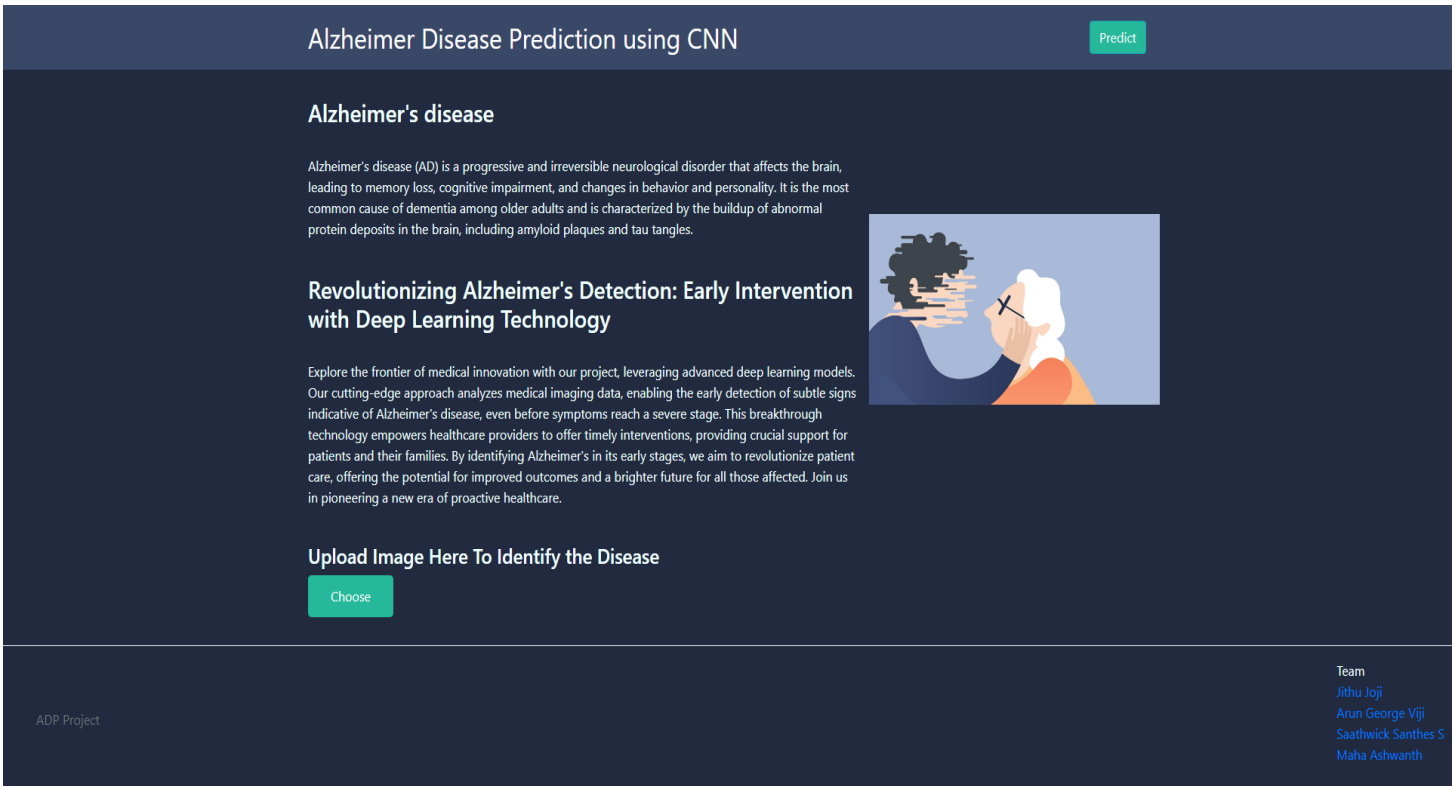
In this section, we will be building a web application that is integrated to the model we built. A UI is provided for the uses where he has to enter the values for predictions. The enter values are given to the saved model and prediction is showcased on the UI.

This section has the following tasks

- Building HTML Pages
- Building python code

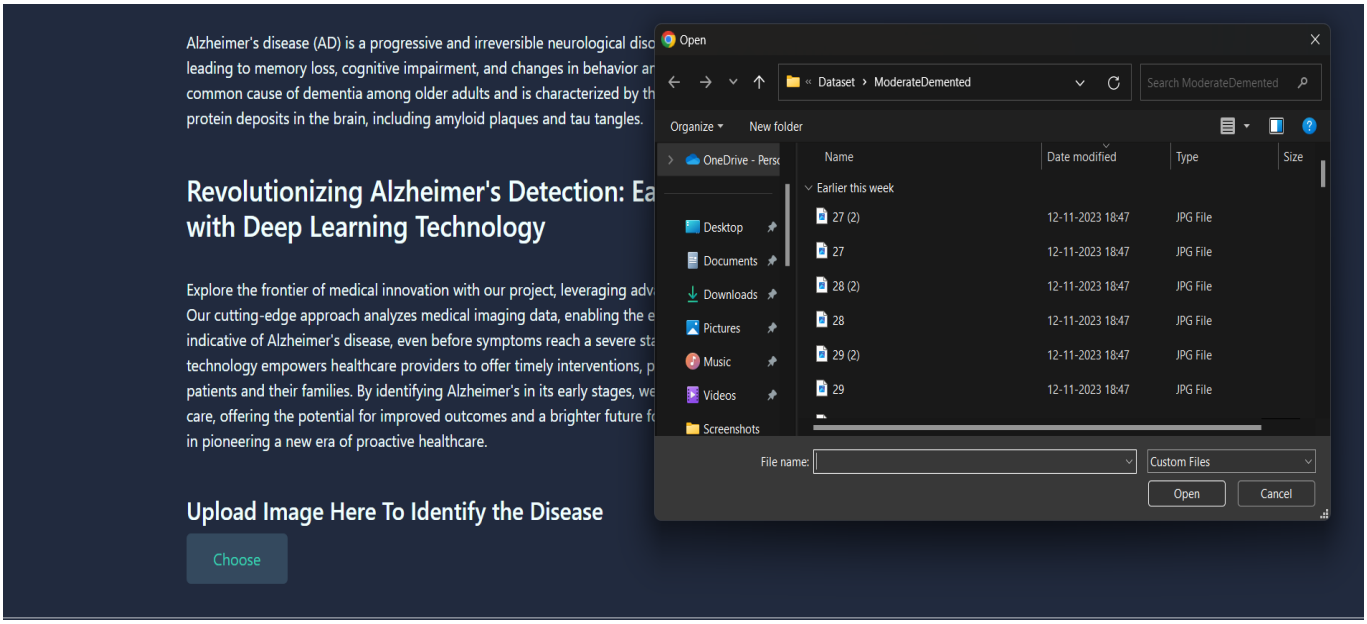
Activity 1: Building HTML Pages

For this project create one HTML file, index.html. The index.html page look as given below

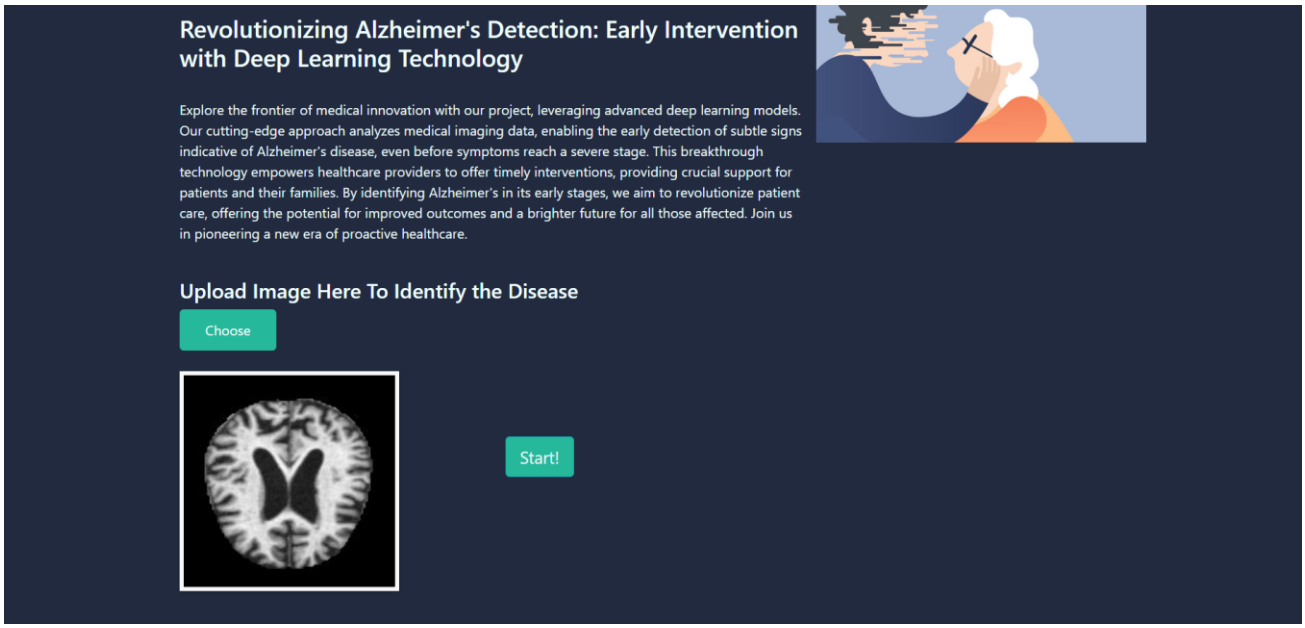


There is a predict button to redirect us to the upload image section.

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



When you select the image, it will display as given below



Activity 2: Build Python code

Import the libraries

```
import numpy as np
import os
from tensorflow.keras.models import load_model
from tensorflow.keras.preprocessing import image
from flask import Flask , request, render_template
```

Initializing Flask App and loading the saved model

```
app = Flask(__name__)

model = load_model("adp.h5",compile=False)
```

Render HTML Page

```
@app.route('/')
def index():
    return render_template('index.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. 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':
        f = request.files['image']
        print("current path")
        basepath = os.path.dirname(__file__)
        print("current path", basepath)
        filepath = os.path.join(basepath,'uploads',f.filename)
        print("upload folder is ", filepath)
        f.save(filepath)

        img = image.load_img(filepath,target_size = (176,176))
        x = image.img_to_array(img)
        print(x)
        x = np.expand_dims(x,axis =0)
        print(x)
        y=model.predict(x)
        preds=np.argmax(y, axis=1)
        print("Prediction",preds)
        index = ['Mild Demented','Moderate Demented','Non Demented','Very Mild Demented']
        text = "The person is " + str(index[preds[0]])
    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 rendered to the text that we have mentioned in the alzheimers.html page earlier.

Main Function

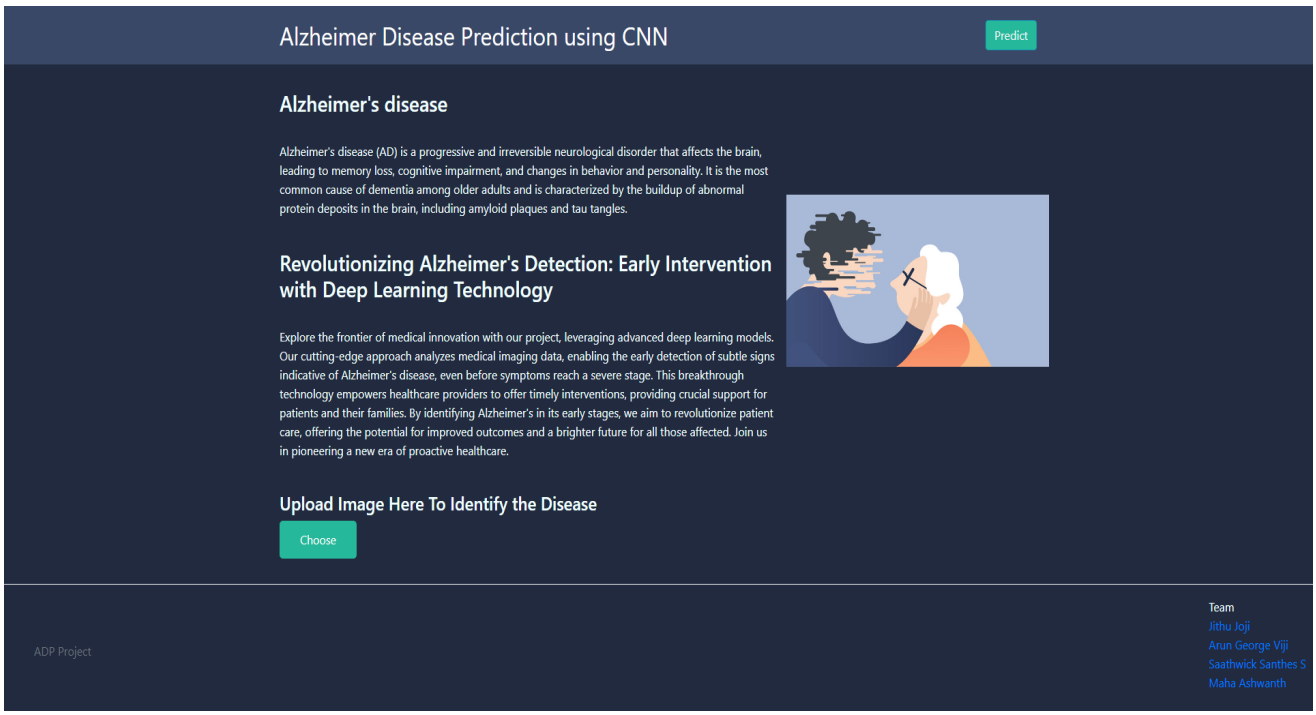
```
if __name__ == '__main__':
    app.run(debug = False, threaded = False)
```

Activity 3: Run the Application

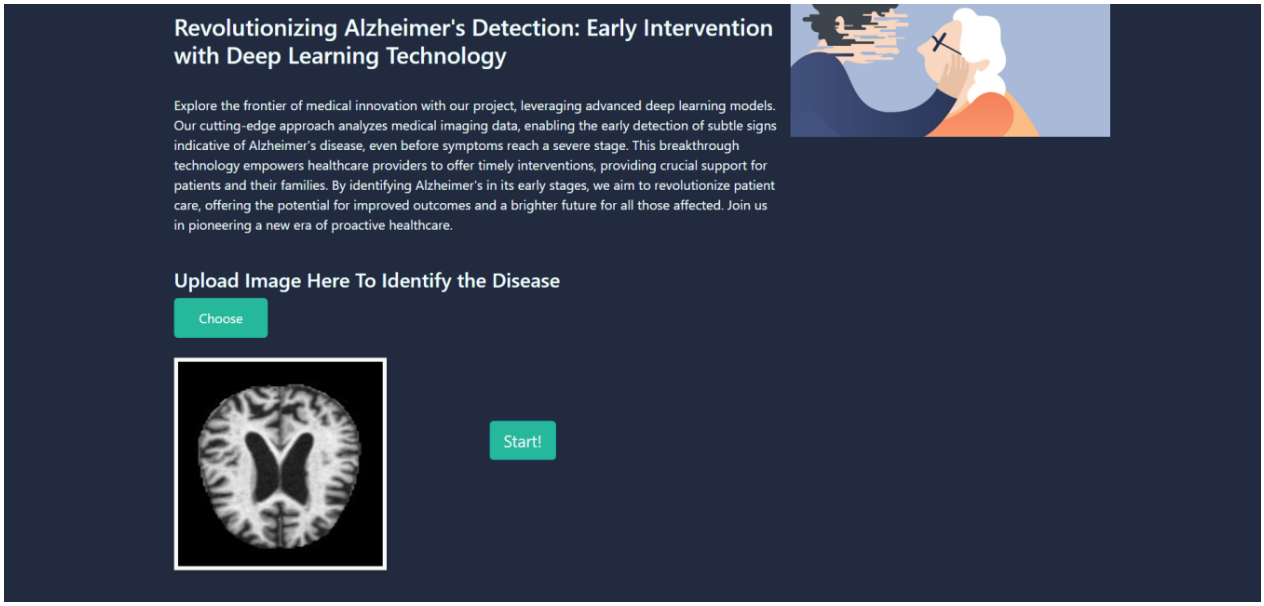
- Open Spyder
- Navigate to the folder where your Python script is.
- Now click on Run button above.
- Click on the predict button from the top right corner, enter the inputs, click on the Start button, and see the result/prediction on the web.

```
 * Serving Flask app 'app1'
 * Debug mode: off
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
```

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



Input :



Once you upload the image and click on Start button, the output will be displayed as below.

Output :