Unveiling the Enigmatic:

Deciphering Monkeypox amidst the Veil of Similar Non-Monkeypox Cases

Project Hand-Out

-Kamya Paliwal

Unveiling the Enigmatic: Deciphering Monkeypox amidst the Veil of Similar Non-Monkeypox Cases

Within the field of infectious illnesses, the difficulty of correctly distinguishing Monkeypox, a rare viral disease, from a plethora of similar non-Monkeypox ailments, has arisen as a confusing riddle. The clinical signs of the monkeypox disease, which is brought on by the monkeypox virus (MPXV), closely match those of other viral disorders including chickenpox and smallpox. The timely detection, efficient containment, and appropriate management of this diagnostic puzzle are severely hampered.

In 1958, monkeypox emerged in Africa and subsequently spread to humans, giving the disease its name. In the past, instances of monkeypox were limited to Central and West Africa. However, recent occurrences outside of these areas have sparked fears all across the world. Further complicating matters, the symptoms' resemblance to those of other viral illnesses frequently results in misdiagnosis, delaying treatment and possibly accelerating the disease's spread.

To address this critical challenge, the utilization of advanced technologies and deep learning methodologies has emerged as a promising solution. Deep learning, a subset of machine learning, harnesses the power of neural networks to automatically learn intricate patterns and derive meaningful insights from complex datasets. By training deep learning models on diverse clinical data, researchers can develop sophisticated algorithms capable of accurately distinguishing Monkeypox from its imitators.

Deep learning has already produced impressive outcomes in the field of infectious illnesses. Deep learning models may identify the minute details and unique patterns that distinguish Monkeypox from related diseases by fusing several data sources, including clinical presentations, laboratory results, and epidemiological data. These models give clinicians the information they need to decide on a course of therapy, a diagnosis, and disease prevention measures.
As we embark on this journey to unravel the mysteries of Monkeypox diagnosis, the integration of deep learning methodologies promises to unlock the answers hidden within the complexities of this puzzling infectious disease.

Aim:

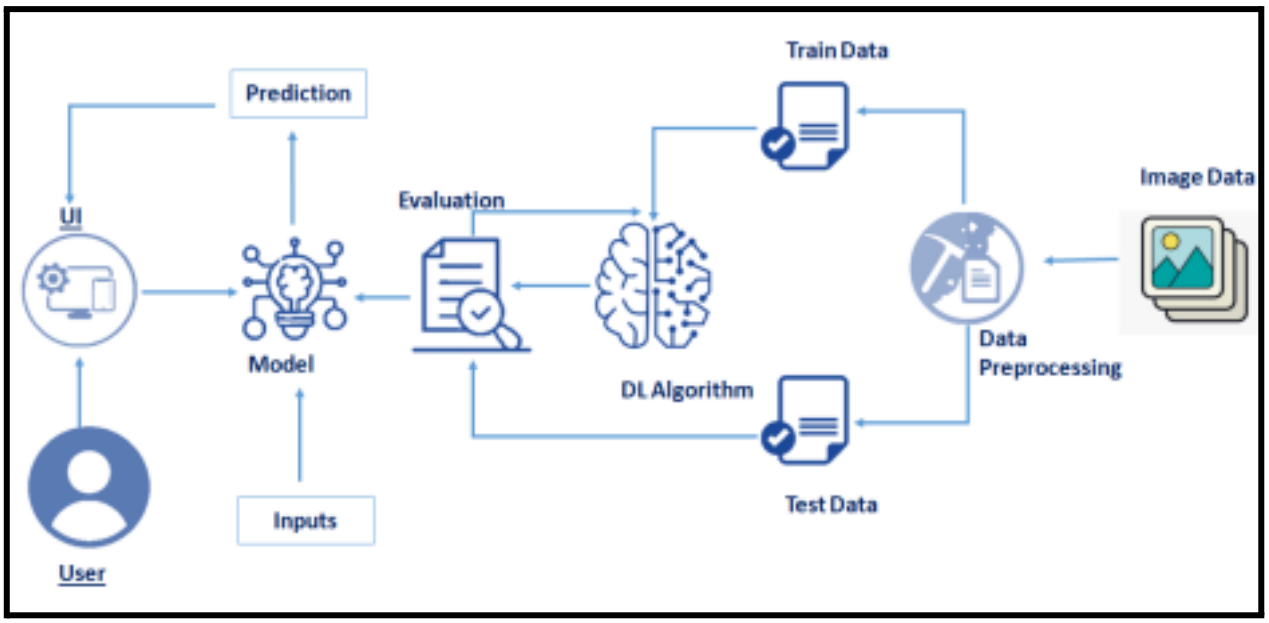
This study aims to investigate the potential of deep learning techniques in accurately distinguishing Monkeypox from similar non-Monkeypox cases. It seeks to develop and evaluate deep learning models that can aid in the timely diagnosis, management, and control of Monkeypox, addressing the challenges posed by its clinical resemblance to other diseases.

Purpose of doing this Project:

This study aims to address the diagnostic difficulties caused by monkeypox by utilizing the capabilities of deep learning algorithms. The research aims to enhance early identification, timely treatment, and efficient management of monkeypox outbreaks by creating and assessing deep learning models capable of properly differentiating monkeypox from comparable non-monkeypox cases. In the end, this study project aims to improve public health interventions and lessen monkeypox's global impact through cutting-edge technical solutions.

Technical Architecture:

The technical framework for this study calls for gathering a wide range of clinical data, laboratory results, and epidemiological data from instances of monkeypox and non-monkeypox. After that, missing values are handled, features are normalized, and categorical variables are encoded as part of the preprocessing of the acquired data. Convolutional neural networks (CNNs) and recurrent neural networks (RNNs) are examples of deep learning models that are developed and trained on preprocessed data to find patterns and distinguishing features. The trained models are put to use for real-time diagnosis after being evaluated using the proper metrics, fine-tuned, and regularised. The models' ability to decode monkeypox and react to changing patterns is continuously monitored and updated, allowing for quicker disease detection and better disease management.



Pre-requisites:

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

- **Anaconda Navigator:**

  - Link: https://www.youtube.com/watch?v=5mDYijMfSzs

- Python packages:

  - Open anaconda prompt as administrator

  - Type "pip install tensorflow" (make sure you are working on Python 64 bit)

  - Type "pip install flask".

- **Deep Learning Concepts**

  - CNN: https://towardsdatascience.com/basics-of-the-classic-cnn-a3dce1225add

  - Flask Basics: https://www.youtube.com/watch?v=lj4I_CvBnt0

Project Objectives:

By the end of this project you will:

- Know fundamental concepts and techniques of Convolutional Neural Network.
- Gain a broad understanding of image data.
- Know how to pre-process/clean the data using different data preprocessing techniques.
- Know how to build a web application using the Flask framework.

Project Flow:

- User interacts with the UI (User Interface) to upload the image as input
- Uploaded image is analyzed by the model which is integrated
- Once the model analyses the uploaded image, the predicted report is showcased on the UI to accomplish this, we have to complete all the activities and tasks listed below

o Data Collection.
    o Collect the dataset or Create the dataset

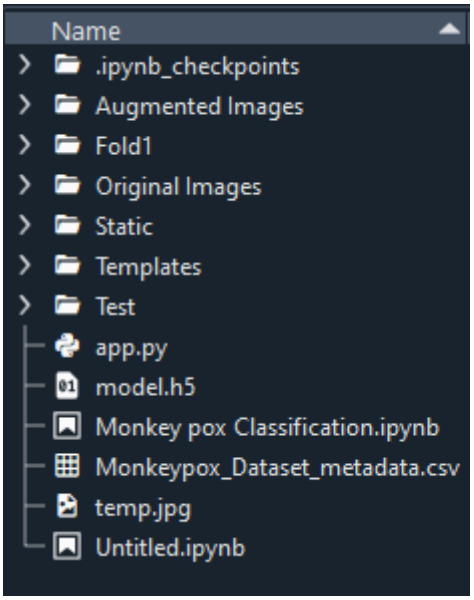
o Data Preprocessing.
    o Import the ImageDataGenerator library
    o Configure ImageDataGenerator class
    o Apply ImageDataGenerator functionality to Trainset and Testset

o Model Building
    o Import the model building Libraries
    o Initializing the model
    o Adding Input Layer
    o Adding Hidden Layer
    o Adding Output Layer
    o Configure the Learning Process
    o Training the model
    o Save the Model
    o Test the Model

o Application Building
    o Create an HTML file
    o Build Python Code

Project Structure:

Create project folder which contains files as shown below:



- The data obtained 2 folders containing 130 images of each for Monkey-Pox cases and Non-Monkey-Pox cases, which is used for training, testing, validation dataset.
- We are building a Flask application that will require the html files to be stored in the templates folder.
- app.py file is used for routing purposes using scripting.
- model.h5 is the saved model. This will further be used in the Flask integration.

Milestone 1: Define Problem/ Problem Understanding

Activity 1: Specify the Business Problem

The project's focus is on the quick and accurate detection of cases of monkeypox in a sea of symptoms that are not related to the disease. Due to the clinical similarities between monkeypox and other viral infections, identification is extremely difficult and frequently incorrect, delaying effective treatment. As a result, there are increased transmission rates, possible dangers to public health, and difficulty managing disease. The project aims to create robust models that can successfully distinguish monkeypox from its imitators using deep learning techniques. This will allow medical professionals to make timely and accurate diagnoses, put effective treatment plans into place, and facilitate targeted disease control measures. This technology will improve patient outcomes overall, lessen the effects of monkeypox epidemics, and boost public health measures.

Activity 2: Business Requirements

1. **Accurate and Timely Diagnosis:** The deep learning solution should accurately differentiate Monkeypox cases from similar non-Monkeypox conditions, enabling healthcare professionals to make precise diagnoses promptly.
2. **Improved Disease Management:** The solution should contribute to enhanced disease management by providing early identification of Monkeypox cases, facilitating timely treatment interventions, and reducing the risk of severe complications.
3. **Effective Public Health Interventions:** The deep learning system should support targeted public health interventions by providing reliable data on Monkeypox outbreaks, aiding in the implementation of appropriate control measures, and minimizing the spread of the disease.
4. **Enhanced Surveillance:** The solution should enable efficient monitoring and surveillance of Monkeypox cases, allowing for early detection of outbreaks, proactive response planning, and effective resource allocation.
5. **Scalability and Adaptability:** The deep learning solution should be scalable to accommodate increasing volumes of data and adaptable to evolving patterns of Monkeypox and its mimics, ensuring its effectiveness in different geographic regions and over extended periods of time.
6. **Trust:** The model should be designed in such a way that it develops trust among the users, especially the advertisers and content creators, who rely on the predicted adviews for their marketing strategies.
7. **User-Friendly Interface:** The deployment using Flask should provide a user-friendly web interface where stakeholders, such as farmers, distributors, and consumers, can easily interact with the system. The interface should allow users to upload mango images, receive classification results, and view the grading category assigned to each mango.
8. **Seamless Deployment:** The deployment using Flask should be smooth and seamless, ensuring that the model is easily accessible and operational. The system should handle multiple requests concurrently and provide consistent and reliable performance.
9. **Scalability:** The solution should be scalable to handle a growing number of mango images and users. As the usage of the system increases, it should be able to accommodate the additional load without compromising on performance or accuracy.
10. **Flexibility for Future Enhancements:** The system should be designed with flexibility in mind, allowing for future enhancements and improvements. This includes the ability to incorporate additional grading categories or expand the dataset to further refine the mango quality assessment capabilities of the model.

Activity 3: Literature Survey

The literature survey for this project aims to explore existing research and studies related to Monkeypox, similar non-Monkeypox conditions, and the application of deep learning in disease diagnosis. It examines published articles, scientific papers, and relevant sources to gather insights into the challenges of distinguishing Monkeypox, the techniques utilized in deep learning for infectious disease identification, and the advancements made in this field. The survey serves as a foundation for understanding the current state of knowledge and identifying research gaps to contribute to the project's objectives.

Activity 4: Social or Business Impact.

Social Impact:

The implementation of deep learning in Monkeypox diagnosis will have significant social implications. It will enable timely identification and containment of outbreaks, reducing the transmission of the disease within communities. Accurate diagnosis will facilitate appropriate treatment, minimizing the potential for severe complications, and ultimately safeguarding public health and well-being.

Business Impact:

The deep learning solution for deciphering Monkeypox will enhance healthcare systems' ability to accurately diagnose and manage the disease. This will improve operational efficiency, reduce healthcare costs associated with misdiagnosis, and contribute to better patient outcomes, thereby strengthening the overall effectiveness and reputation of healthcare organizations.

Milestone 2: Data Collection

Machine Learning & Deep Learning depends heavily on data. It is the most crucial part aspect that makes algorithm training possible. So, this section guides on how to download dataset.

Activity 1: Collect the dataset

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

In this project we have used .csv data. This data is downloaded from kaggle.com. Please refer to the link given below to download the dataset.

Link: https://www.kaggle.com/datasets/nafin59/monkeypox-skin-lesion-dataset

As the dataset is downloaded. Let us read and understand the data properly with the help of some visualisation techniques and some analysing techniques.

Note: There are a number of techniques for understanding the data. But here we have used some of it. In an additional way, you can use multiple techniques.

Milestone 3: Image Preprocessing

Activity 1: Import the ImageDataGenerator library

Image data augmentation is a technique that can be used to artificially expand the size of a training dataset by creating modified versions of images in the dataset.

The Keras deep learning neural network library provides the capability to fit models using image data augmentation via the ImageDataGenerator class.

Let us import the ImageDataGenerator class from keras

```
In [1]: import numpy as np
        import matplotlib.pyplot as plt
        from tensorflow.keras.preprocessing.image import ImageDataGenerator
```

Activity 2: Configure ImageDataGenerator class

The ImageDataGenerator class in Keras is used for real-time data augmentation during model training. It allows the user to perform various image transformations such as rotation, zooming, shifting, and flipping on the input data, which can help prevent overfitting and improve model performance. The class can be configured to apply different types and levels of data augmentation, and it supports both binary and categorical data. The ImageDataGenerator class is a powerful tool for improving the accuracy and robustness of deep learning models, particularly in computer vision applications.

Activity 3 : Data Preparation and Loading for Deep Learning Image Classification

```
train_gen = ImageDataGenerator(rescale=1./255)
valid_gen = ImageDataGenerator(rescale=1./255,validation_split=0.4)
target_img_shape=(64,64)
# Loading Data
train_data = train_gen.flow_from_directory("Augmented Images/Augmented Images",target_size=(64,64),shuffle=True,class_mode='bina
valid_data = valid_gen.flow_from_directory('Original Images/Original Images',target_size=(64,64),shuffle=True,subset='training',
test_data = valid_gen.flow_from_directory('Original Images/Original Images',target_size=(64,64),shuffle=True,subset='validation'
```

```
Found 3192 images belonging to 2 classes.
Found 138 images belonging to 2 classes.
Found 90 images belonging to 2 classes.
```

The above code snippet demonstrates the preparation and loading of image data for training a deep learning model. Using the ImageDataGenerator utility, the code rescales pixel values, sets the desired target image shape, and loads training, validation, and test data from corresponding directories. This ensures that the data is properly preprocessed and ready for training the model.

Activity 4: Training Data Analysis and Class Distribution

```
In [4]: print('Training')
        ids,counts=np.unique(train_data.classes,return_counts=True)
        print(ids)
        print(counts)

        Training
        [0 1]
        [1428 1764]

In [5]: labels=(train_data.class_indices)
        labels=dict((v,k) for k,v in labels.items())
        labels

        for i in ids:
            print('{:>6}={}'.format(labels[i],counts[i]))

        Monkeypox_augmented=1428
        Others_augmented=1764
```

The above code snippet performs an analysis of the training data and examines the class distribution. Firstly, the code prints the header 'Training' to indicate the following analysis. Then, it calculates the unique class IDs and their respective counts using the np.unique() function on train_data.classes. The ids variable stores the unique class IDs, and counts store the corresponding frequency of each class. The code also creates a dictionary, and labels, to map the class indices to their labels. Finally, the code iterates over the unique class IDs and prints the class labels and their corresponding counts using formatted output. This analysis provides insights into the distribution of classes within the training data.

Activity 5: Validation Data Analysis and Class Distribution

```
In [6]: print('Validation')
        ids,counts=np.unique(valid_data.classes,return_counts=True)
        print(ids)
        print(counts)

        Validation
        [0 1]
        [62 76]

In [7]: labels=(valid_data.class_indices)
        labels=dict((v,k) for k,v in labels.items())
        labels

        for i in ids:
            print('{:>6}={}'.format(labels[i],counts[i]))

        Monkey Pox=62
        Others=76
```

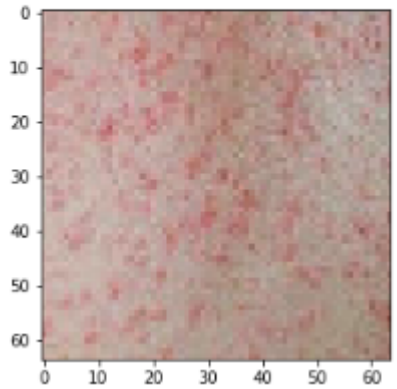
The above code snippet focuses on the analysis of the validation data and examines the class distribution within it. After printing the header 'Validation', the code calculates the unique class IDs and their corresponding counts using np.unique() on valid_data.classes. The resulting ids variable contains the unique class IDs, while counts stores the frequency of each class. The code creates a dictionary, labels, to map the class indices to their respective labels. Then, for each unique class ID, the code prints the class labels and their corresponding counts using formatted output. This analysis offers insights into the distribution of classes within the validation data.

Activity 6: Sample Image Visualization

```
In [9]: print(train_data.class_indices)
        for image_batch, labels_batch in train_data:
            print(image_batch.shape)
            print(labels_batch.shape)
            plt.imshow(image_batch[0])
            print('class',labels_batch[0])
            break
```

The provided code snippet examines the class indices, image and label shapes, and displays the first image from the training data. Firstly, the code prints the dictionary train_data.class_indices, which maps the class labels to their corresponding indices. Then, through a loop iterating over train_data, it retrieves batches of image and label data. For each batch, the code prints the shape of the image batch and the label batch, providing information about the dimensions of the data. It also displays the first image from the batch using plt.imshow(). Additionally, the code prints the class label associated with the first image, extracted from labels_batch[0]. The loop is terminated after processing the first batch using break. This code segment provides insights into the structure and content of the training data.

```
{'Monkeypox_augmented': 0, 'Others_augmented': 1}
(32, 64, 64, 3)
(32,)
class 1.0
```



Milestone 4: Model Building

Activity 1 : Importing the Model Building Libraries

```
In [11]: from tensorflow.keras import Sequential
         from tensorflow.keras.layers import (Dense,Conv2D,AveragePooling2D,Flatten,Dropout, MaxPool2D)
```

Activity 2: CNN Architecture for Image Classification

```
model = Sequential()
model.add(Conv2D(32,(3,3), activation='relu', input_shape=in_shape))
model.add(MaxPool2D((2,2)))

model.add(Conv2D(64,(3,3), activation='relu'))
model.add(MaxPool2D((2,2)))

model.add(Conv2D(128,(3,3), activation='relu'))
model.add(MaxPool2D((2,2)))
model.add(Flatten())

model.add(Dense(128,activation='relu'))
model.add(Dense(1,activation='sigmoid'))
```

The above code snippet showcases the architecture of a Convolutional Neural Network (CNN) for image classification using the Keras framework. The model consists of several layers: Conv2D layers with increasing filter sizes, followed by MaxPool2D layers for downsampling. The final layers include a Flatten layer to convert the 2D feature maps into a 1D vector, Dense layers with ReLU activation for feature extraction, and a final Dense layer with a sigmoid activation for binary classification. This architecture is designed to capture meaningful features from the input images and make predictions based on them.

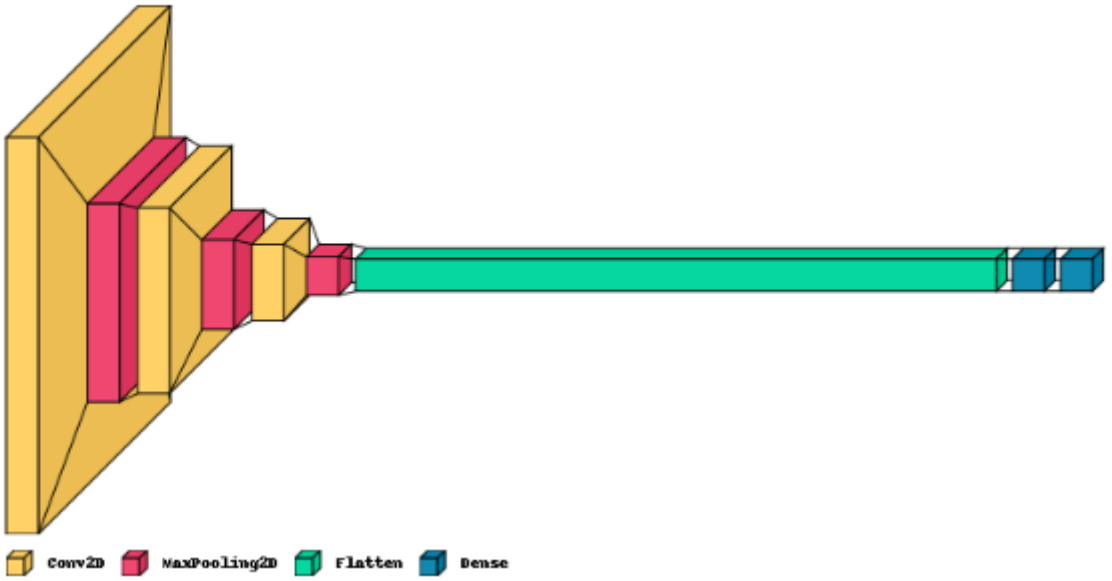
Activity 3: Model Summar

```
In [12]: model.summary()
Model: "sequential"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 conv2d (Conv2D)             (None, 62, 62, 32)        896

 max_pooling2d (MaxPooling2D  (None, 31, 31, 32)       0
 )

 conv2d_1 (Conv2D)           (None, 29, 29, 64)        18496

 max_pooling2d_1 (MaxPooling  (None, 14, 14, 64)       0
 2D)

 conv2d_2 (Conv2D)           (None, 12, 12, 128)       73856

 max_pooling2d_2 (MaxPooling  (None, 6, 6, 128)        0
 2D)

 flatten (Flatten)           (None, 4608)              0

 dense (Dense)               (None, 128)               589952

 dense_1 (Dense)             (None, 1)                 129

=================================================================
Total params: 683,329
Trainable params: 683,329
Non-trainable params: 0
_________________________________________________________________
```

Activity 4: Visualizing the Layered View of the CNN Architecture

```
In [13]: import visualkeras as vk
         vk.layered_view(model,legend=True)
```

Out[13]:



Using the visualkeras library, the code generates a visual representation of the layered view of the CNN architecture, highlighting the different layers and their connectivity.

Activity 5: Model Compilation and Training

```
In [14]: model.compile(loss  = 'binary_crossentropy', optimizer  = 'adam', metrics = ['accuracy'])
```

```
In [15]: import time
         start=time.time()

         history=model.fit(train_data,steps_per_epoch=len(train_data),
                           validation_data=valid_data,
                           epochs=20,verbose=1)
         end=time.time()
         print('Time taken:{:.2f} minutes'.format((end-start)/60))
```

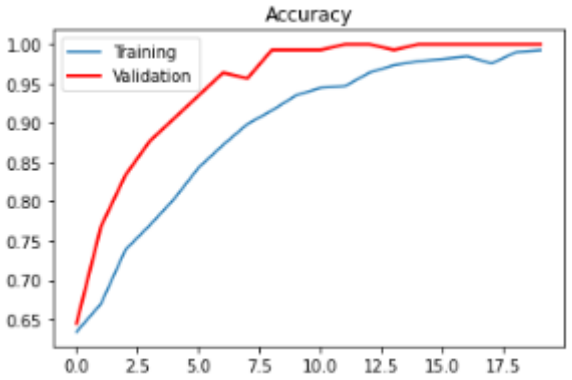
The above code snippet performs several key steps in training the deep learning model. First, the model is compiled using the specified loss function, optimizer, and metrics. Then, the model.fit() function is called to train the model using the training data (train_data) and validate it using the validation data (valid_data). The training process is run for 20 epochs with progress displayed. The time taken for training is measured by calculating the difference between the start and end times and is displayed in minutes.

```
Epoch 1/20
100/100 [==============================] - 64s 630ms/step - loss: 0.6451 - accuracy: 0.6341 - val_loss: 0.6223 - val_accuracy:
0.6449
Epoch 2/20
100/100 [==============================] - 10s 97ms/step - loss: 0.6141 - accuracy: 0.6698 - val_loss: 0.5272 - val_accuracy:
0.7681
Epoch 3/20
100/100 [==============================] - 10s 100ms/step - loss: 0.5407 - accuracy: 0.7384 - val_loss: 0.4079 - val_accuracy:
0.8333
Epoch 4/20
100/100 [==============================] - 11s 106ms/step - loss: 0.4883 - accuracy: 0.7697 - val_loss: 0.3636 - val_accuracy:
0.8768
Epoch 5/20
100/100 [==============================] - 10s 103ms/step - loss: 0.4285 - accuracy: 0.8033 - val_loss: 0.2894 - val_accuracy:
0.9058
Epoch 6/20
100/100 [==============================] - 10s 102ms/step - loss: 0.3555 - accuracy: 0.8430 - val_loss: 0.2044 - val_accuracy:
0.9348
Epoch 7/20
100/100 [==============================] - 10s 104ms/step - loss: 0.2974 - accuracy: 0.8719 - val_loss: 0.1417 - val_accuracy:
0.9638
Epoch 8/20
100/100 [==============================] - 11s 106ms/step - loss: 0.2578 - accuracy: 0.8985 - val_loss: 0.1270 - val_accuracy:
0.9565
Epoch 9/20
100/100 [==============================] - 10s 100ms/step - loss: 0.2076 - accuracy: 0.9157 - val_loss: 0.0668 - val_accuracy:
0.9928
Epoch 10/20
100/100 [==============================] - 10s 101ms/step - loss: 0.1683 - accuracy: 0.9352 - val_loss: 0.0795 - val_accuracy:
0.9928
Epoch 11/20
100/100 [==============================] - 10s 97ms/step - loss: 0.1424 - accuracy: 0.9445 - val_loss: 0.0594 - val_accuracy:
0.9928
Epoch 12/20
100/100 [==============================] - 10s 102ms/step - loss: 0.1419 - accuracy: 0.9467 - val_loss: 0.0339 - val_accuracy:
1.0000
Epoch 13/20
100/100 [==============================] - 10s 102ms/step - loss: 0.0984 - accuracy: 0.9640 - val_loss: 0.0165 - val_accuracy:
1.0000
```

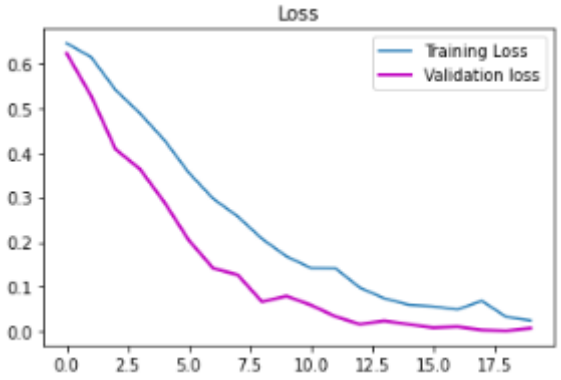
Milestone 5: Visualization of Training and Validation Performance.

In [16]:
```
plt.figure(figsize=(10,3.5))
plt.subplot(1,2,1)
plt.title('Loss')
plt.plot(history.history['loss'],label='Training Loss')
plt.plot(history.history['val_loss'],'m', lw=2, label= 'Validation loss')

plt.legend()
plt.subplot(1,2,2)
plt.title('Accuracy')
plt.plot(history.history['accuracy'],label='Training')
plt.plot(history.history['val_accuracy'],'r' ,lw=2, label='Validation')
plt.legend()
plt.tight_layout()
plt.show()
```

The above code snippet generates a figure with two subplots for visualizing the training and validation loss, as well as the training and validation accuracy. The first subplot displays the loss values over epochs, with the training loss shown in blue and the validation loss in magenta. The second subplot shows the accuracy values, with the training accuracy in blue and the validation accuracy in red. The legend provides labels for the corresponding curves, and plt.tight_layout() ensures proper spacing between subplots. Finally, plt.show() displays the figure.



Model Accuracy



Model Loss

Activity 2: Image Prediction Using Trained Model

In [47]:

```
from tensorflow.keras.preprocessing import image

# Assuming you have a path to the image you want to predict
image_path = "NM63_01.jpg"

# Load the image and resize it to match the input shape of the model
img = image.load_img(image_path, target_size=(64, 64))

# Convert the image to a numpy array
img_array = image.img_to_array(img)

# Expand the dimensions to match the expected input shape of the model
img_array = np.expand_dims(img_array, axis=0)

# Preprocess the image by rescaling its pixel values
img_array = img_array / 255.0

# Make the prediction
prediction = model.predict(img_array)

# Get the predicted class probability
predicted_probability = prediction[0][0]

# Define the class labels
class_labels = ["Monkey Pox", "Other"]

# Set a threshold to classify the image
threshold = 0.5

# Get the predicted class label
if predicted_probability >= threshold:
    predicted_class_label = class_labels[1]  # Monkey Pox
else:
    predicted_class_label = class_labels[0]  # Other

# Print the predicted class label
print("Predicted class:", predicted_class_label)
```

The above code snippet demonstrates the process of predicting the class label for an input image using a trained model. The image is loaded and resized to match the input shape of the model. It is then converted to a numpy array and expanded to include the batch dimension. The pixel values of the image are rescaled, and the prediction is made using the model's predict() function. The predicted class probability is extracted, and a threshold is applied to classify the image as either "Monkey Pox" or "Other." The predicted class label is printed as the final output.

```
1/1 [==============================] - 0s 91ms/step
```

Predicted class: Monkey Pox

Milestone 6 : Performance Testing

Activity 1: Evaluating model on the validation set

This code evaluates the trained model's performance on the validation data. The resulting validation loss and accuracy are extracted from the scores and printed with three decimal places of precision. These metrics provide insights into the model's performance on unseen data, indicating the level of error and the accuracy of predictions.

```
In [26]: scores = model.evaluate(valid_data, steps=len(valid_data), verbose=0)
         validation_loss = scores[0]
         validation_accuracy = scores[1]

         print("Validation loss: {:.3f}".format(validation_loss))
         print("Validation accuracy: {:.3f}".format(validation_accuracy))

         Validation loss: 0.004
         Validation accuracy: 1.000
```

The validation results show a low validation loss value of 0.004, indicating that the model's predictions on the validation data have minimal errors. Additionally, the validation accuracy of 1.000 suggests that the model achieved perfect accuracy in classifying the validation samples, highlighting its strong performance and ability to generalize well to unseen data.

Activity 1: Evaluating model on the test set

In this code snippet evaluates the trained model's performance on the test data. The model.evaluate() function is used to calculate the test loss and test accuracy. The resulting scores are then printed as "Test loss" and "Test accuracy", respectively. This allows us to assess the model's effectiveness in making predictions on unseen data.

```
In [19]: score = model.evaluate(test_data, verbose=0)
         print("Test loss:", score[0])
         print("Test accuracy:", score[1])

         Test loss: 0.0014474054332822561
         Test accuracy: 1.0
```

The test results indicate that the model achieved a very low test loss value of 0.0014, suggesting that it performed well in terms of minimizing prediction errors on the test data. Additionally, the test accuracy of 1.0 implies that the model achieved perfect accuracy in classifying the test samples, indicating its high performance and robustness.

Milestone 7: Model Deployment

Activity 1: Save and load the best model

```
In [48]: # Assuming your model is named 'model'
         model.save("model.h5")
```

```
In [49]: from tensorflow.keras.models import load_model

         # Load the saved model
         model = load_model("model.h5")
```

We save the model into a file named model.h5

Milestone 8: 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

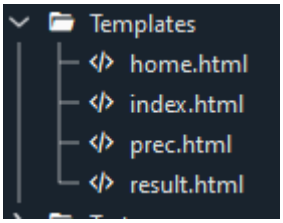
- Building HTML Pages
- Building server-side script
- Run the web application

Activity 2.1: Building HTML pages:

For this project we create two HTML files namely

- Home.html
- Index.html
- Prec.html
- Results.html

And we will save them in the templates folder.



Activity 2.2: Build Python code

Create a new app.py file which will be store in the Flask folder.

- Import the necessary Libraries.

```
from flask import Flask, request, jsonify, render_template
from tensorflow.keras.models import load_model
from tensorflow.keras.preprocessing import image
import numpy as np
import matplotlib.pyplot as plt
import base64
```

- This code loads a pre-trained machine learning model for mango classification, which has been saved in the file 'model.h5'.

```
# Load the trained model
model = load_model('model.h5')
```

- **Image Pre-Processing Function-**

  This function called preprocess_image takes an image file path as input. It loads the image, resizes it to (64, 64) dimensions, converts it to a numpy array, expands the dimensions to match the expected input shape of the model, and then normalizes the pixel values between 0 and 1. The processed image array is returned as output.

```python
# Define a function to preprocess the input image
def preprocess_image(img_path):
    img = image.load_img(img_path, target_size=(64, 64))
    img_array = image.img_to_array(img)
    img_array = np.expand_dims(img_array, axis=0)
    img_array /= 255.
    return img_array
```

- This code creates a new instance of a Flask web application using the Flask class from the Flask library. The __name__ argument specifies the name of the application's module or package.

```python
app = Flask(__name__)
```

- Defining the routes for home.html, prec.htlm, index.html using render_template.

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

@app.route('/prec')
def prec():
    return render_template('prec.html')

@app.route('/predict')
def pred():
    return render_template('index.html')
```

- **Flask Route for Prediction**

  The predict function is a route handler in Flask that is associated with the '/predict' URL route and accepts POST requests. When a user submits an image file through a form, Flask invokes this function. Inside the function, it retrieves the image file from the request, saves it temporarily, and preprocesses the image using the preprocess_image function. Then, it makes a prediction using the trained model. Based on the prediction result, it converts the predicted class to a human-readable string. The image is then loaded and converted to a base64-encoded string for rendering in the HTML template. Finally, it renders the 'result.html' template with the prediction and the image to display the prediction result to the user.

```python
@app.route('/predict', methods=['POST'])
def predict():
    # Get the image file from the request
    file = request.files['file']

    # Save the file to a temporary location
    file_path = 'temp.jpg'
    file.save(file_path)

    # Preprocess the image
    img_array = preprocess_image(file_path)

    # Make a prediction
    prediction = model.predict(img_array)

    # Get the predicted class probability
    predicted_probability = prediction[0][0]

    # Define the class labels
    class_labels = ["It's the case of Monkey Pox", "It's not MonkeyPox"]

    # Set a threshold to classify the image
    threshold = 0.5

    # Get the predicted class label
    if predicted_probability >= threshold:
        prediction_str = class_labels[1]  # Monkey Pox
    else:
        prediction_str = class_labels[0]  # Other


    # Load the image and convert it to a base64-encoded string
    with open(file_path, 'rb') as f:
        img_base64 = base64.b64encode(f.read()).decode()

    # Render the HTML template with the prediction and the image
    return render_template('result.html', prediction=prediction_str, image=img_base64)
```

Main Function:

This code runs the Flask application if the script is being executed directly (i.e. not imported as a module in another script). The if __name__ == '__main__': line checks if the script is the main module being executed, and if so, runs the Flask application using the app.run() method. This method starts the Flask development server, allowing the application to be accessed via a web browser at the appropriate URL.

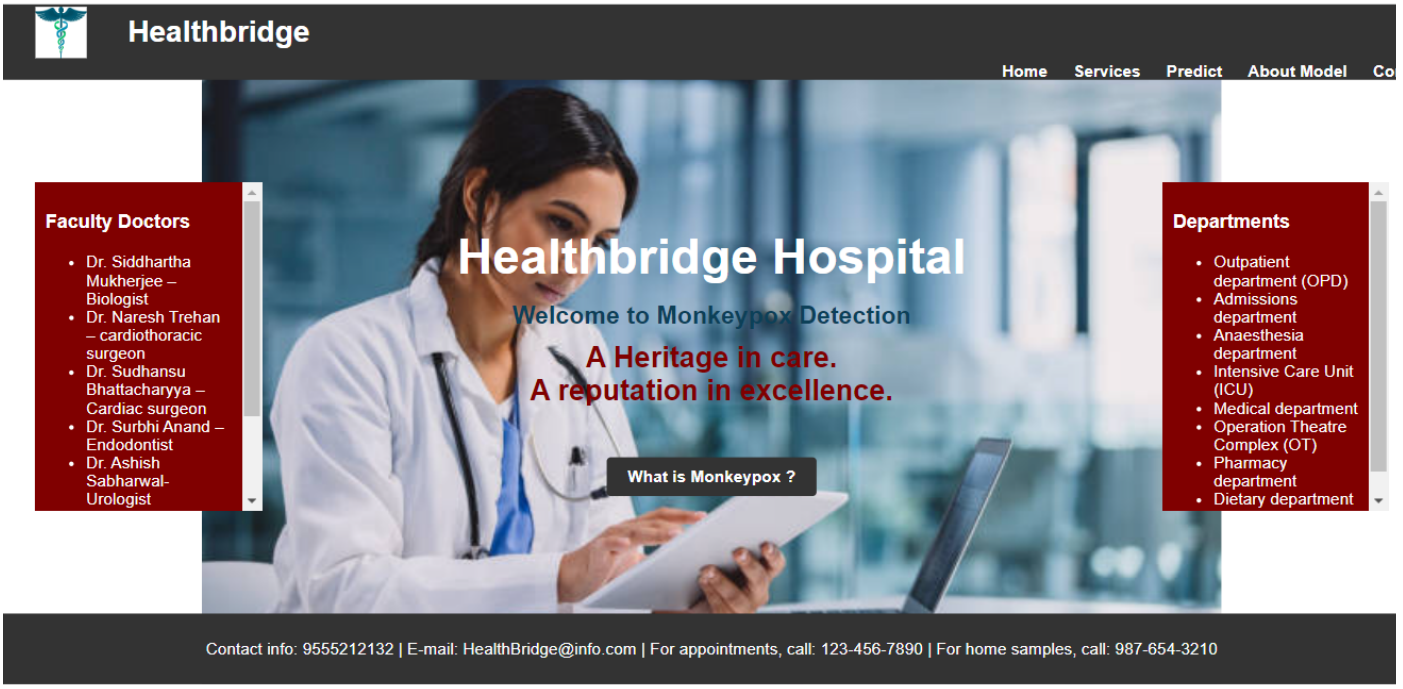
```
if __name__ == '__main__':
    app.run()
```

Activity 2.3: Run the Web Application

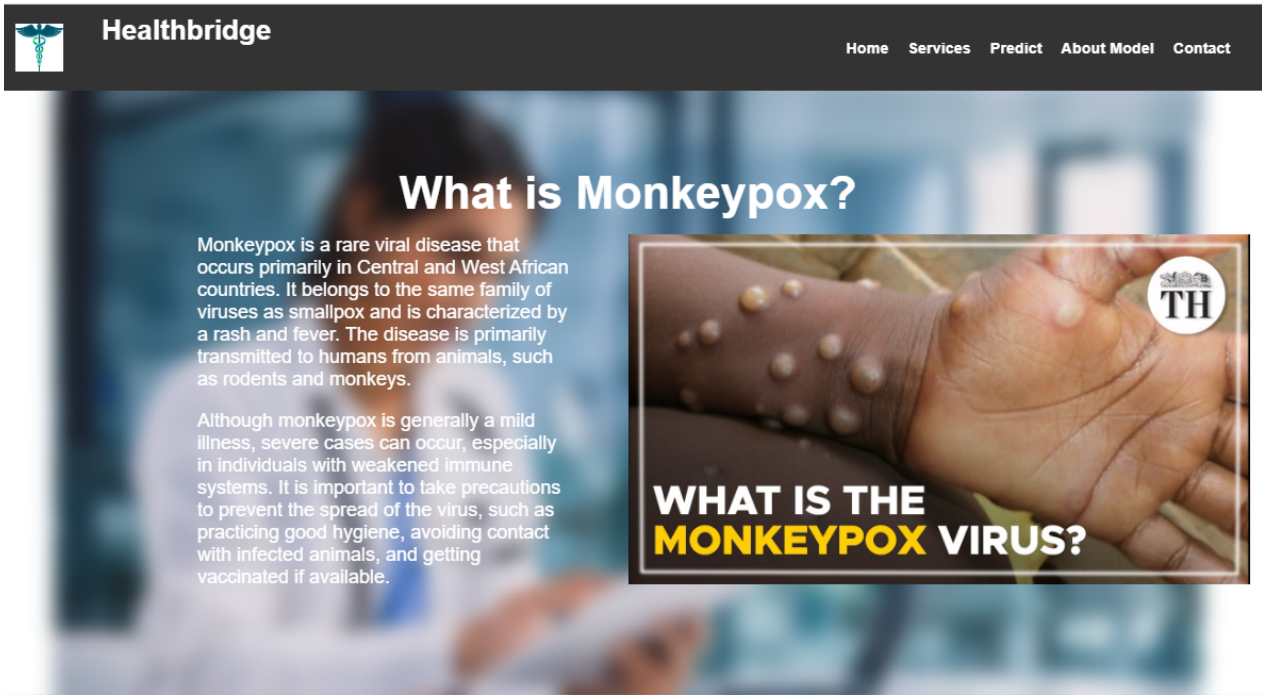
When you run the "app.py" file this window will open in the console or output terminal. Copy the URL given in the form http://127.0.0.1:5000 and paste it in the browser.

```
In [5]: runfile('C:/Users/kamya/OneDrive/Desktop/Monkey_Pox_Detection/
app.py', wdir='C:/Users/kamya/OneDrive/Desktop/Monkey_Pox_Detection')
 * 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)
```

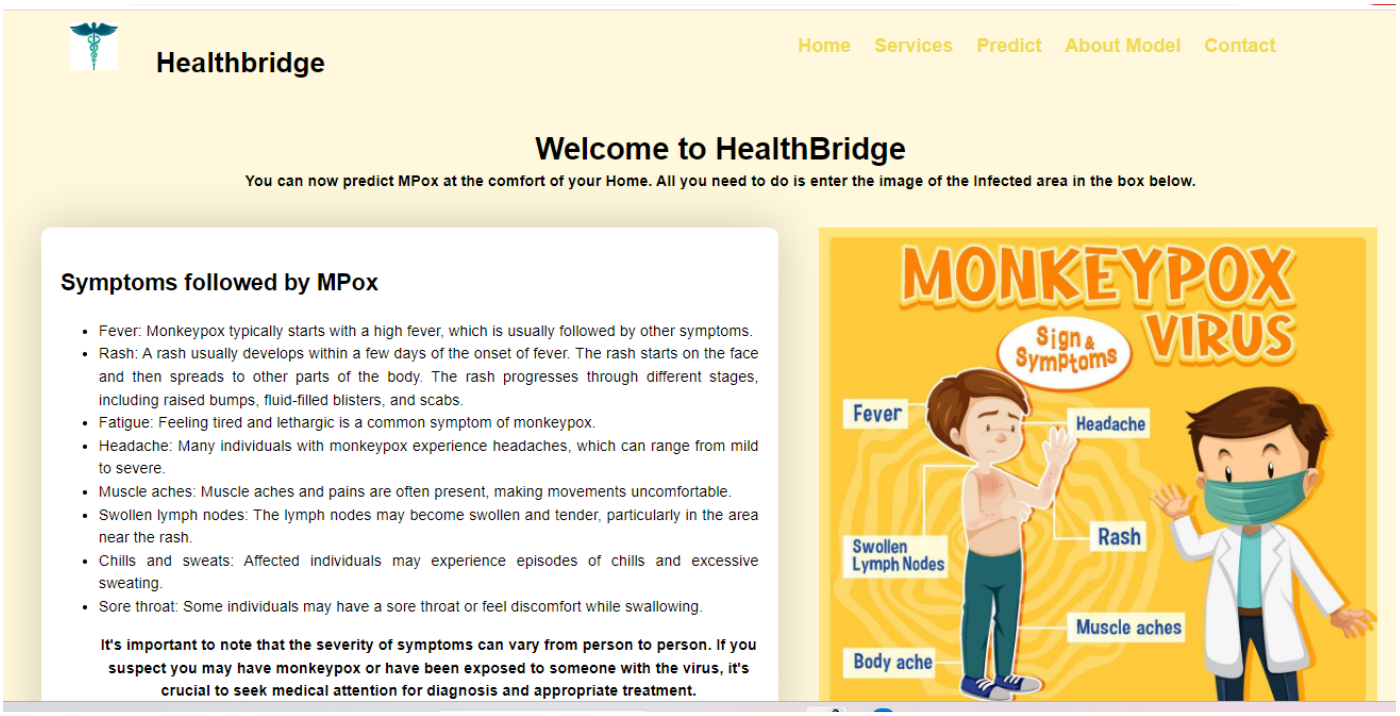
When we paste the URL in a web browser, our home.html page will open. It contains a welcome page of the hosptal named- '**HealthBridge**' and information about - the '**Faculty Dr.**' and the '**Departments**' information. There is navigation button on the top right containing- 'Home', 'Service', 'Predict', 'About Model' and 'Contact'. There is also a button on this webpage called- '**What is Monkeypox**'.



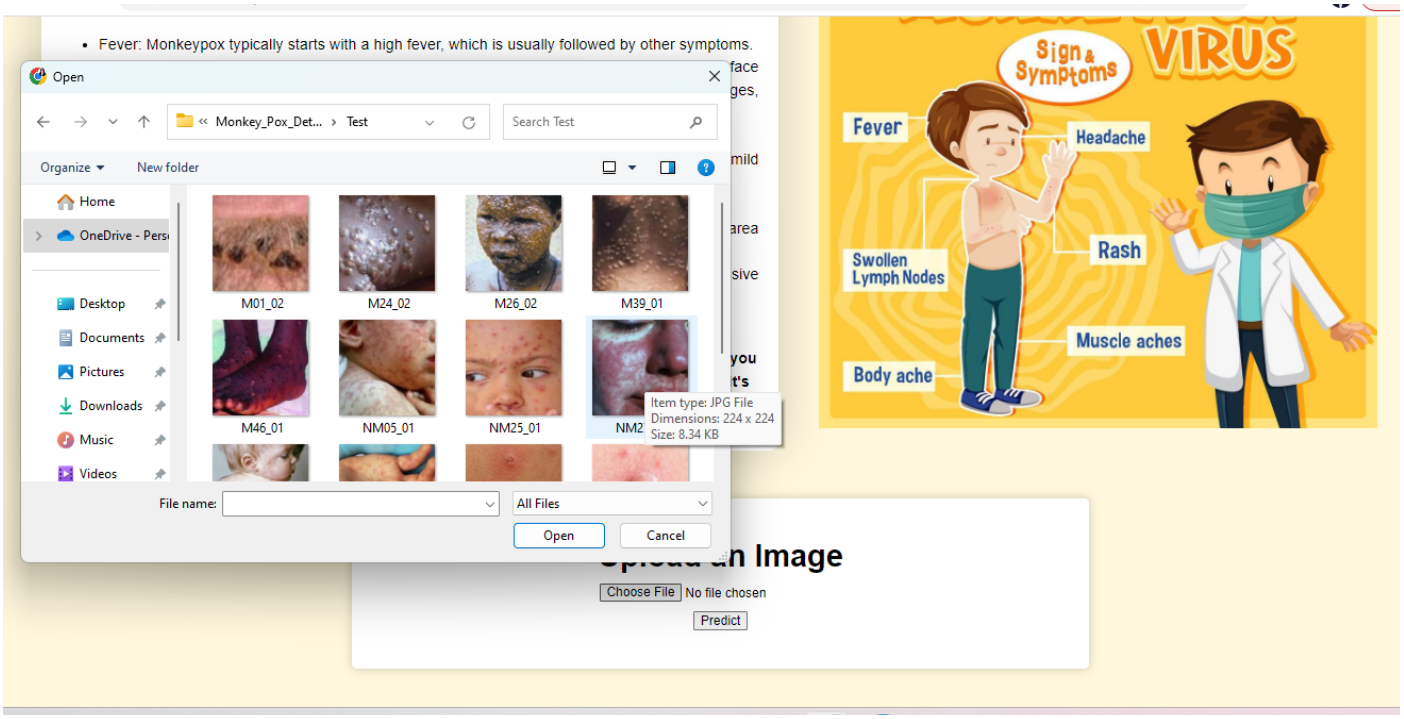
Once you click on the '**What is Monkeypox**' button, you will be redirected to prec.thtml page which contains information about MonkeyPox Virus.

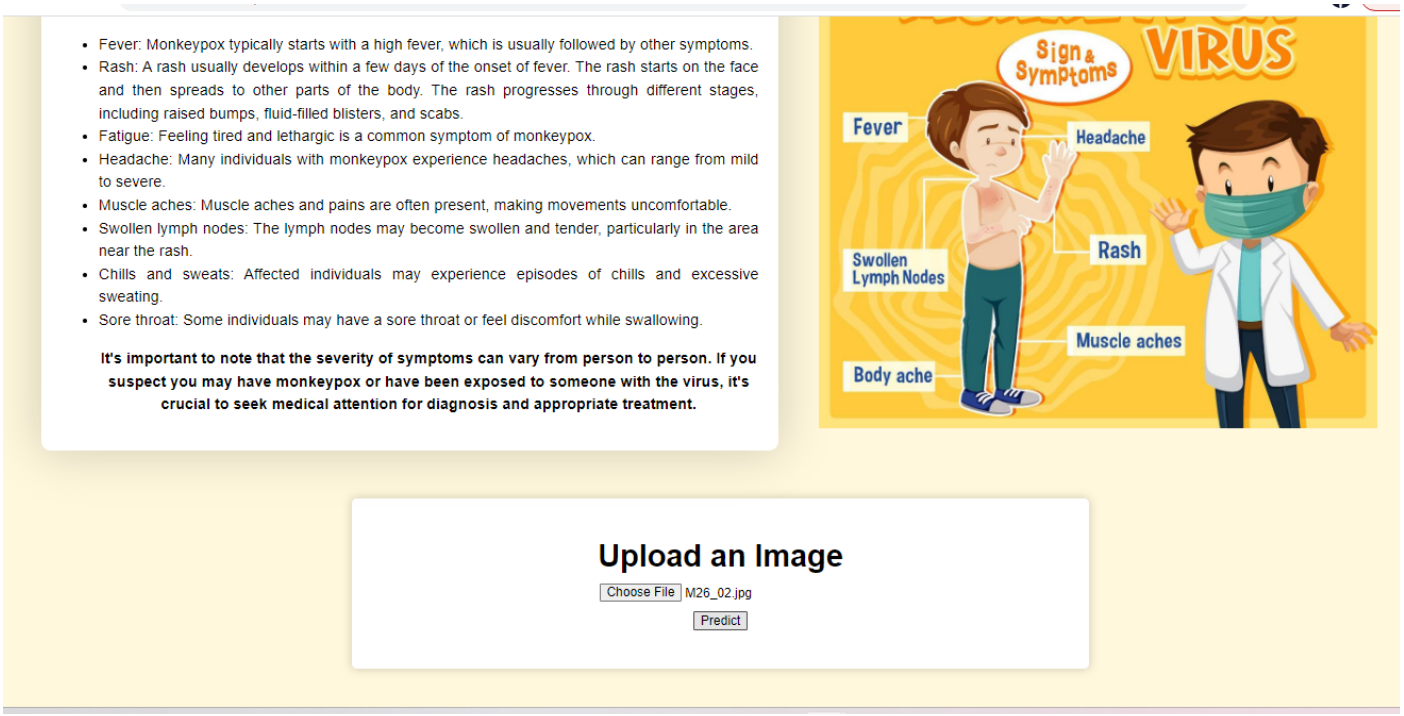


When you click on the '**Predict**' button in the navigation bar, you will be redirected to predict.html page which contains information about- '**Symptoms followed by MPox**'.



When you scroll down you will see the '**Choose file**' option. Click on it and choose the desired image of the infected area. Upload it and hit on the '**Predict**' button.





Now you will be redirected to result.html page which contains your report and also information about- '**Precautionary measures to ensure safety**'.

The uploaded image of the infected area is **'The case of Monkey Pox'**.

To go to the home page again, hit on the 'Home' button in the top right corner and you will be redirected to the home.html page again.

Link to GitHub Repository:

https://github.com/Kamya-Paliwal/Unveiling-the-Enigmatic-Deciphering-Monkeypox-amidst-the-Veil-of-Similar-Non-Monkeypox-Cases