Mushroom Classification Using Deep Learning

Project Description:

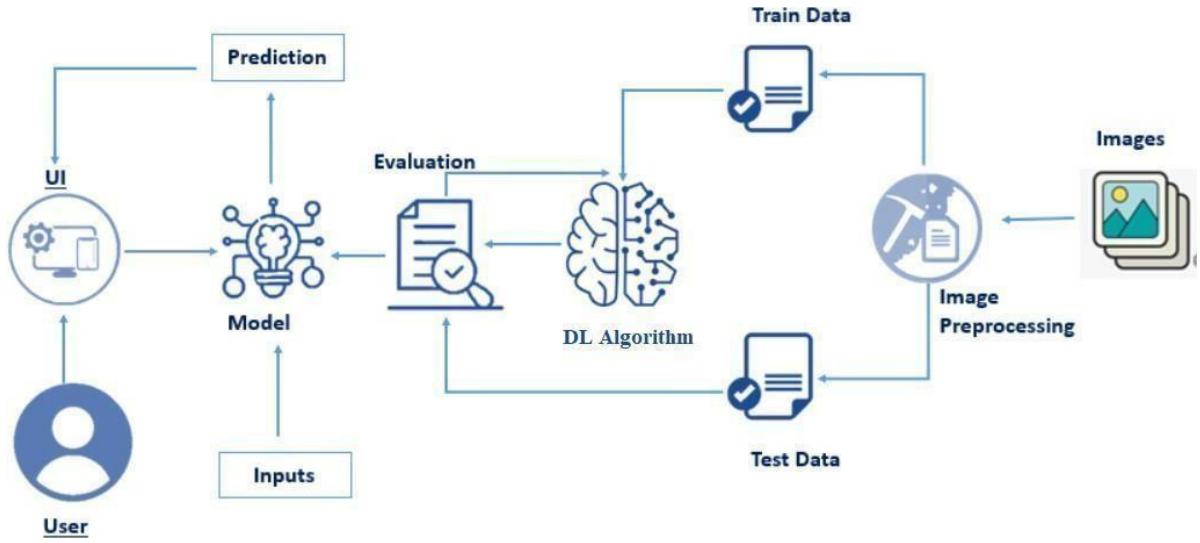
Mushrooms are a type of fungi that grow in a variety of habitats, from forests to fields to decomposing logs. They come in many different shapes, sizes, and colors, and are used for food, medicine, and other purposes.

A mushroom is a fruitful body of fungus which is usually produced above the ground on soil or other nutrients. We only concentrate on mushrooms and do not consider fungus species. our purpose is the optical recognition of species which have cap , gills underside of cap and astern and typically the full body is visible on the image.

In this project we are classifying various types of Mushrooms that are found on various regions of our planet. These Mushrooms are majorly classified into 3 categories namely Boletus, Lactarius & Russula. Deep-learning (DL) methods in artificial intelligence (AI) play a dominant role as high-performance classifiers in the detection of the Mushrooms using images.

Transfer learning has become one of the most common techniques that has achieved better performance in many areas, especially in image analysis and classification. We used Transfer Learning techniques like Inception V3,Resnet50V2, Xception that are more widely used as a transfer learning method in image analysis and they are highly effective.

Technical Architecture:



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 Xception 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.
- Image Pre-processing.
    - Import the required library
    - Configure ImageDataGenerator class
    - Apply ImageDataGenerator functionality to Trainset and Testset
- Model Building
    - Pre-trained CNN model as a Feature Extractor
    - Adding Dense Layer
    - Configure the Learning Process
    - Train the model
    - Save the Model
    - Test the model
- Application Building
    - Building HTML Pages
    - Building Flask Code
    - Run Application

Prior Knowledge:

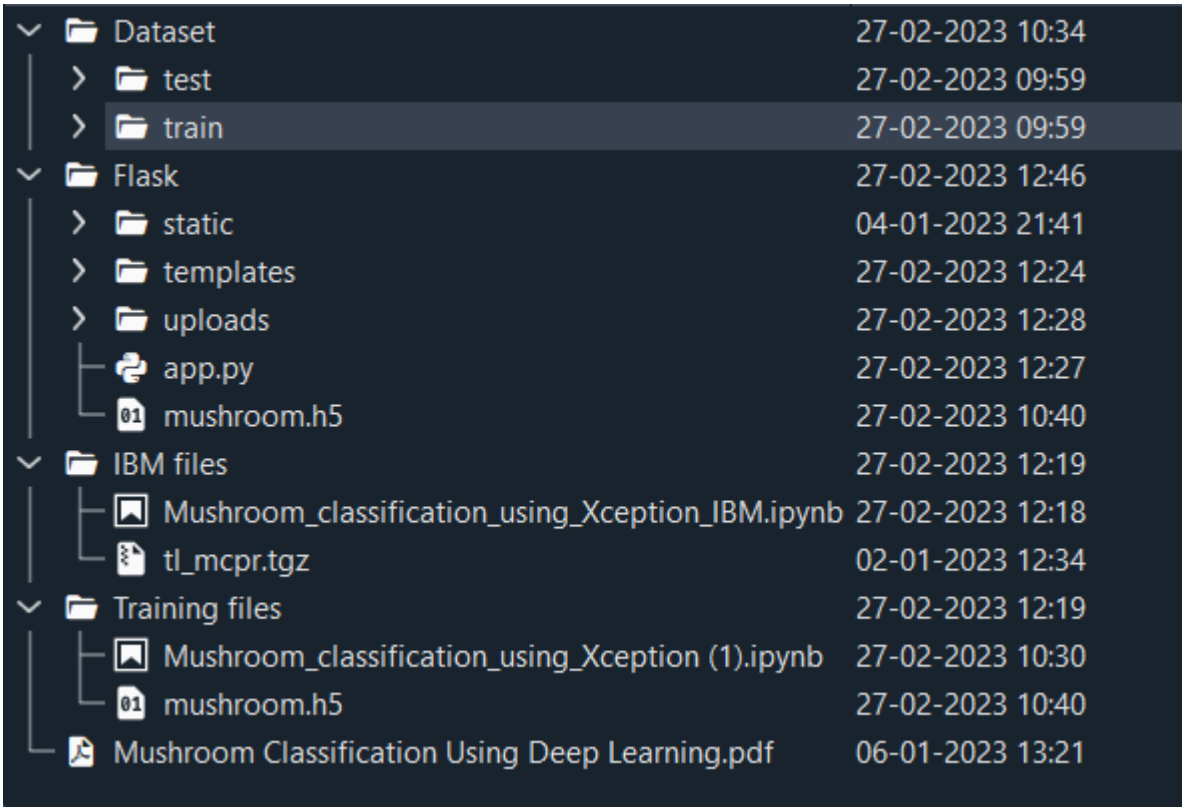
You must have prior knowledge of following topics to complete this project.

- **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-50V2:** 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=lj4I_CvBnt0**

Project Structure:

Create a Project folder which contains files as shown below



- The Dataset folder contains the training and testing images for 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 **app.py** for server side scripting
- The IBM folder consists of a trained model notebook on IBM Cloud.
- Training folder consists of Mushroom_classification_using_Xception (1).ipynb model training file & mushroom.h5 is 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

In this project, we have collected images of 3 types of Mushrooms images like Boletus, Lactarius & Russula 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:- https://www.kaggle.com/datasets/maysee/mushrooms-classification-common-genuss-images

Note: For better accuracy train on more images

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

We will be connecting Drive with Google Colab because the dataset is too big to import, using the following code:

```
from google.colab import drive
drive.mount('/content/drive')
```

```
Mounted at /content/drive
```

Unzipping the folder:

```
! unzip '/content/drive/MyDrive/Colab Notebooks/Mushroom/Dataset/Dataset-20230227T042828Z-001.zip'
```

```
Archive:  /content/drive/MyDrive/Colab Notebooks/Mushroom/Dataset/Dataset-20230227T042828Z-001.zip
  inflating: Dataset/test/Lactarius/0099_DXVPtL-oMIU.jpg
  inflating: Dataset/test/Lactarius/0085_m5xg2ioUV4k.jpg
  inflating: Dataset/test/Lactarius/0205_TasOn6XYRJA.jpg
  inflating: Dataset/test/Lactarius/0150_rvcxmA_JX-I.jpg
  inflating: Dataset/test/Lactarius/0212_6r0AFhCvTk4.jpg
  inflating: Dataset/test/Lactarius/0140_zbOJpzuaGMw.jpg
  inflating: Dataset/test/Lactarius/0047_d6T94oP5tMY.jpg
  inflating: Dataset/test/Lactarius/011_LoMY4qu_wxo.jpg
  inflating: Dataset/test/Lactarius/0278_xasbi9piTJU.jpg
  inflating: Dataset/test/Lactarius/0162_DiWMr6h-ujY.jpg
  inflating: Dataset/test/Lactarius/0171_YjPamqUNzvk.jpg
  inflating: Dataset/test/Lactarius/0260_w-5dZ5ZCMsg.jpg
  inflating: Dataset/test/Lactarius/0250_WqScI51egU0.jpg
  inflating: Dataset/test/Lactarius/0194_lq_Mh0iwUk8.jpg
  inflating: Dataset/test/Lactarius/0132_phTDi_QE6ew.jpg
  inflating: Dataset/test/Lactarius/0040_8wmTdjhnq64.jpg
  inflating: Dataset/test/Lactarius/0156_xCwDrPY72us.jpg
  inflating: Dataset/test/Lactarius/0319_PQ7CYFw-qTk.jpg
  inflating: Dataset/test/Lactarius/0181_gQu_ll_-SEY.jpg
  inflating: Dataset/test/Lactarius/0019_ecX_lrLyoMY.jpg
  inflating: Dataset/test/Lactarius/0301_egpSMo3VaDg.jpg
  inflating: Dataset/test/Lactarius/0237_zZdLcl0mkD0.jpg
  inflating: Dataset/test/Lactarius/0030_i2ZcdNrZWwM.jpg
  inflating: Dataset/test/Lactarius/018_mCR7ztzhcXo.jpg
  inflating: Dataset/test/Lactarius/0413_S0sUGVa_N_M.jpg
```

Activity 2: Create training and testing dataset

To build a DL model we have to split training and testing data into two separate folders. In this project dataset folder training and testing folders are present. So, in this case we have to give the path to train & test folders.

Four different transfer learning models are used in our project and the best model (Xception) is selected.

```
imageSize = [224, 224]

trainPath = r"/content/Dataset/train"

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

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 performing some geometric transformations of images like rotation, scaling, translation, etc.

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.xception import Xception, preprocess_input
from glob import glob
import numpy as np
import matplotlib.pyplot as plt
```

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 & for that purpose we use **ImageDataGenerator**

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

Dense layer is a deeply connected neural network layer

Flatten layer is used to convert multi dimensional matrix to a vector.

image is used to read the image.

load_img is used for reading image from the system.

preprocess_input meant to adequate your image to the format the model requires

Xception model is a deep convolutional neural network architecture to improve the efficiency and performance of large-scale image recognition tasks.

Numpy is for performing mathematical functions.

Matplotlib is for visualization purpose.

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 32.
- 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 encodedas 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 = 32,
                                                 class_mode = 'categorical')

test_set = test_datagen.flow_from_directory(testPath,
                                            target_size = (224,224),
                                            batch_size = 32,
                                            class_mode = 'categorical')
```

```
Found 911 images belonging to 3 classes.
Found 306 images belonging to 3 classes.
```

Total the dataset is having 911 train images & 306 test images divided under 3 classes.

Milestone 3: Model Building

Now it's time to build our model. Let's use the pre-trained model which is Xception, one of the convolution neural net (CNN) architecture which is considered as a very good model for Image classification.

Deep understanding on the Xception 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 (224,224,3).

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.

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

```
Downloading data from https://storage.googleapis.com/tensorflow/keras-applications/xcepti
83683744/83683744 [==============================] - 0s 0us/step
```

```
# don't train existing weights
for layer in xception.layers:
  layer.trainable = False
```

```
# our layers - you can add more if you want
x = Flatten()(xception.output)
```

Activity 2: Adding Dense Layers

```
# our layers - you can add more if you want
x = Flatten()(xception.output)
```

```
prediction = Dense(3, activation='softmax')(x)
```

```
# create a model object
model = Model(inputs=xception.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 Xception.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.

```
# view the structure of the model
model.summary()
```

```
batch_normalization_3 (BatchNo  (None, 7, 7, 1024)  4096        ['conv2d_3[0][0]']
rmalization)

add_11 (Add)                   (None, 7, 7, 1024)   0           ['block13_pool[0][0]',
                                                                 'batch_normalization_3[0][0]']

block14_sepconv1 (SeparableCon  (None, 7, 7, 1536)  1582080     ['add_11[0][0]']
v2D)

block14_sepconv1_bn (BatchNorm  (None, 7, 7, 1536)  6144        ['block14_sepconv1[0][0]']
alization)

block14_sepconv1_act (Activati  (None, 7, 7, 1536)  0           ['block14_sepconv1_bn[0][0]']
on)

block14_sepconv2 (SeparableCon  (None, 7, 7, 2048)  3159552     ['block14_sepconv1_act[0][0]']
v2D)

block14_sepconv2_bn (BatchNorm  (None, 7, 7, 2048)  8192        ['block14_sepconv2[0][0]']
alization)

block14_sepconv2_act (Activati  (None, 7, 7, 2048)  0           ['block14_sepconv2_bn[0][0]']
on)

flatten (Flatten)              (None, 100352)       0           ['block14_sepconv2_act[0][0]']

dense (Dense)                  (None, 3)            301059      ['flatten[0][0]']

==================================================================================================
Total params: 21,162,539
Trainable params: 301,059
Non-trainable params: 20,861,480
__________________________________________________________________________________________________
```

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

```
# tell the model what cost and optimization method to use
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 20 epochs and after every epoch, the current model state is saved if the model has the least loss encountered till that time. We can see that the training loss decreases in almost every epoch till 10 epochs and probably there is further scope to improve the model.

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.

```
# fit the model
r = model.fit_generator(
  training_set,
  validation_data=test_set,
  epochs=20,
  steps_per_epoch=len(training_set)//5,
  validation_steps=len(test_set)//5
)
```

```
Epoch 4/20
5/5 [==============================] - 4s 873ms/step - loss: 1.7211 - accuracy: 0.7688 - val_loss: 2.4075 - val_accuracy: 0.7344
Epoch 5/20
5/5 [==============================] - 4s 954ms/step - loss: 1.7201 - accuracy: 0.7563 - val_loss: 0.5145 - val_accuracy: 0.8750
Epoch 6/20
5/5 [==============================] - 5s 875ms/step - loss: 1.0205 - accuracy: 0.7875 - val_loss: 1.7563 - val_accuracy: 0.7656
Epoch 7/20
5/5 [==============================] - 4s 838ms/step - loss: 1.2439 - accuracy: 0.8000 - val_loss: 0.9134 - val_accuracy: 0.8281
Epoch 8/20
5/5 [==============================] - 5s 1s/step - loss: 0.8242 - accuracy: 0.8687 - val_loss: 1.5532 - val_accuracy: 0.7188
Epoch 9/20
5/5 [==============================] - 4s 875ms/step - loss: 0.7589 - accuracy: 0.8875 - val_loss: 1.4784 - val_accuracy: 0.7969
Epoch 10/20
5/5 [==============================] - 5s 1s/step - loss: 1.0895 - accuracy: 0.8313 - val_loss: 2.1636 - val_accuracy: 0.6875
Epoch 11/20
5/5 [==============================] - 4s 809ms/step - loss: 1.4165 - accuracy: 0.7937 - val_loss: 1.3776 - val_accuracy: 0.7812
Epoch 12/20
5/5 [==============================] - 4s 867ms/step - loss: 1.2759 - accuracy: 0.7937 - val_loss: 1.0087 - val_accuracy: 0.8438
Epoch 13/20
5/5 [==============================] - 5s 1s/step - loss: 1.0568 - accuracy: 0.8375 - val_loss: 1.5242 - val_accuracy: 0.7344
Epoch 14/20
5/5 [==============================] - 4s 907ms/step - loss: 0.7603 - accuracy: 0.8671 - val_loss: 1.0103 - val_accuracy: 0.8750
Epoch 15/20
5/5 [==============================] - 5s 995ms/step - loss: 0.8661 - accuracy: 0.8625 - val_loss: 0.6626 - val_accuracy: 0.8594
Epoch 16/20
5/5 [==============================] - 5s 901ms/step - loss: 1.0268 - accuracy: 0.8500 - val_loss: 0.8959 - val_accuracy: 0.8750
Epoch 17/20
5/5 [==============================] - 4s 861ms/step - loss: 0.9261 - accuracy: 0.8938 - val_loss: 1.0083 - val_accuracy: 0.7969
Epoch 18/20
5/5 [==============================] - 6s 1s/step - loss: 0.8228 - accuracy: 0.8438 - val_loss: 0.4911 - val_accuracy: 0.8594
Epoch 19/20
5/5 [==============================] - 4s 815ms/step - loss: 0.7387 - accuracy: 0.8811 - val_loss: 1.0043 - val_accuracy: 0.8594
Epoch 20/20
5/5 [==============================] - 4s 872ms/step - loss: 1.3756 - accuracy: 0.7750 - val_loss: 0.4722 - val_accuracy: 0.9062
```

From the above run time, we can easily observe that at 20th epoch the model is giving the best validation accuracy.

Activity 5: Save the Model

The model is saved with .h5 extension as follows

Among all the models we tested (CNN, VGG16, RestNet, Inception & Xception) Xception gave us the best accuracy so we have saved our model using Xception.

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

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

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. Load the saved model using load_model.

```
training_set.class_indices
```

```
{'Boletus': 0, 'Lactarius': 1, 'Russula': 2}
```

```
#load one random image from local system
img=image.load_img(r"/content/Dataset/test/Lactarius/0047_d6T94oP5tMY.jpg",target_size=(224,224))
```

```
#convert image to array format
x=image.img_to_array(img)
```

```
x.shape
```

```
(224, 224, 3)
```

```
import numpy as np
x=np.expand_dims(x,axis=0)
img_data=preprocess_input(x)
img_data.shape
```

```
(1, 224, 224, 3)
```

```
output=np.argmax(model.predict(img_data), axis=1)
output
```

```
1/1 [==============================] - 1s 1s/step
array([1])
```

So our model will give in index position of the label. In this case 1st index is for Lactarius, 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

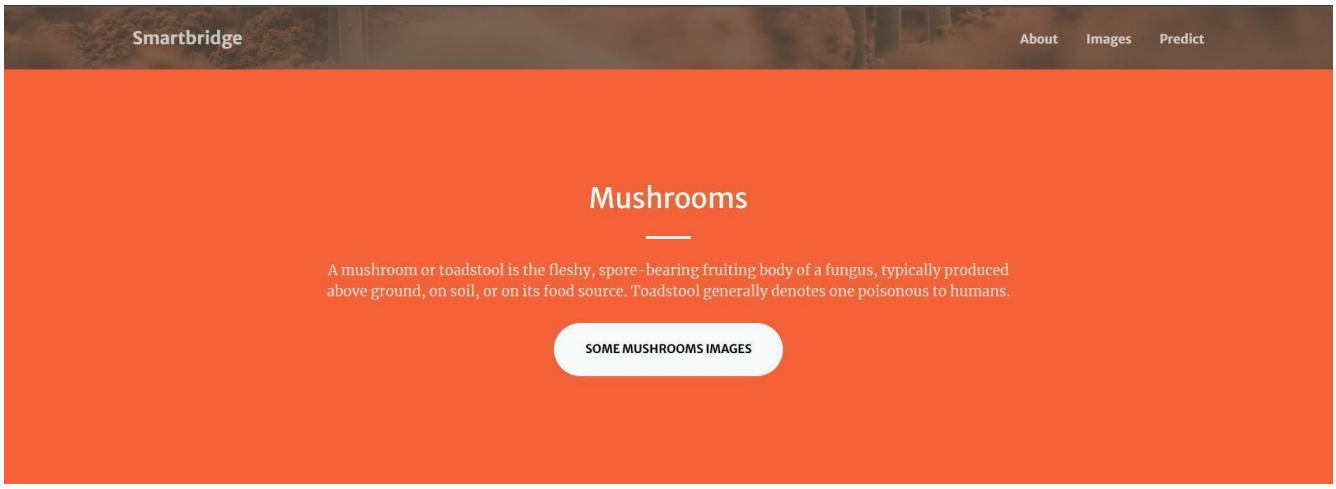
Activity1: Building Html Pages:

For this project create one HTML file namely

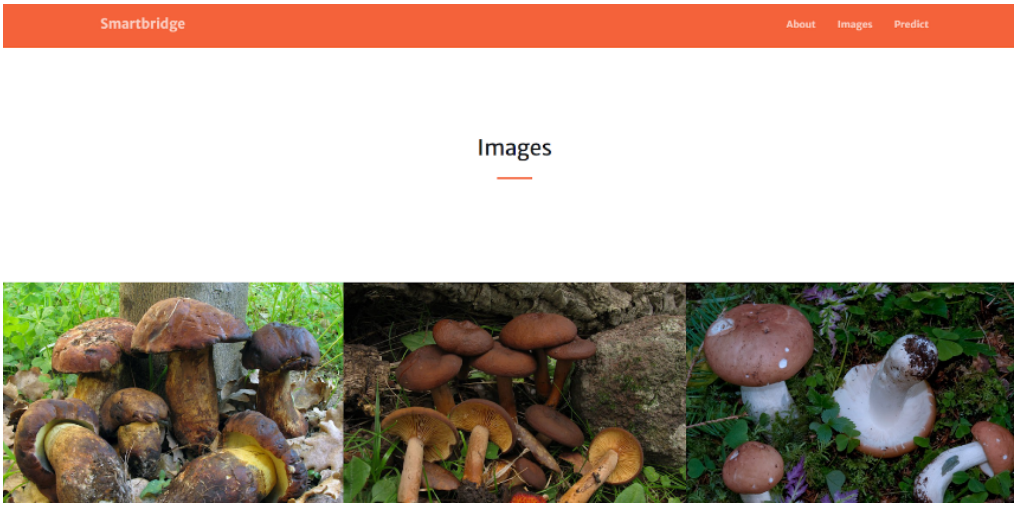
- index.html

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

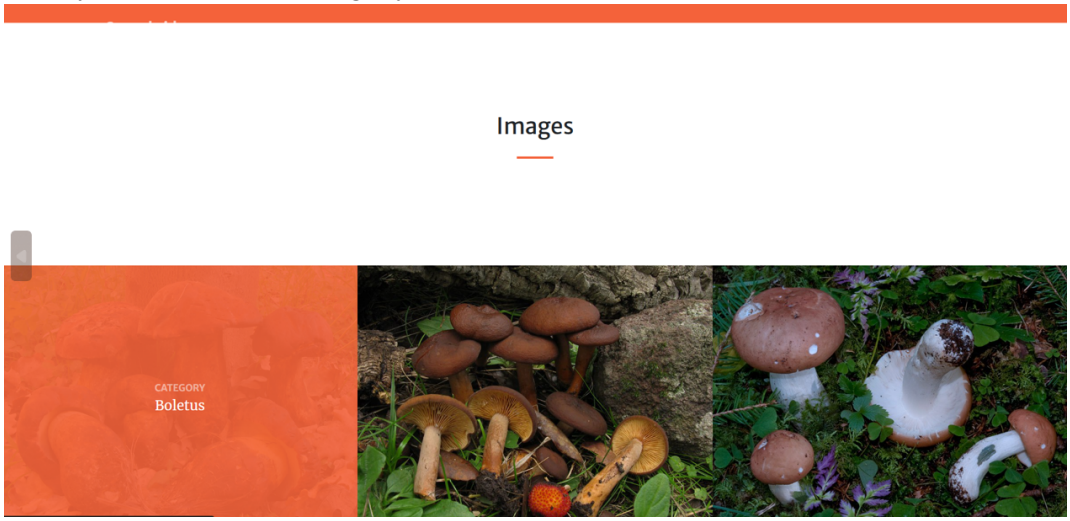
When you click on the ABOUT MUSHROOMS or About button, you will be redirecting to the following page



When you click on the SOME MUSHROOM IMAGES or Images button, it will redirect you to the below page



When you hover over the images you will be able to see names of various Mushrooms



Activity 2: Build Python code:

Import the libraries

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

Loading the saved model and initializing the flask app

```
model=load_model(r"mushroom.h5")

app=Flask(__name__)
```

Render HTML pages:

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


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

@app.route('/input')
def input1():
    return render_template("input.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 index.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 res():
    if request.method=="POST":
        f=request.files['image']
        basepath=os.path.dirname(__file__)
        filepath=os.path.join(basepath,'uploads',f.filename)
        f.save(filepath)

        img=image.load_img(filepath,target_size=(224,224,3))
        x=image.img_to_array(img)
        x=np.expand_dims(x,axis=0)

        img_data=preprocess_input(x)
        prediction=np.argmax(model.predict(img_data), axis=1)

        index=['Boletus','Lactarius','Russula']

        result=str(index[prediction[0]])
        print(result)
        return render_template('output.html', prediction=result)
```

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

Main Function:

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

Activity 3: Run the application

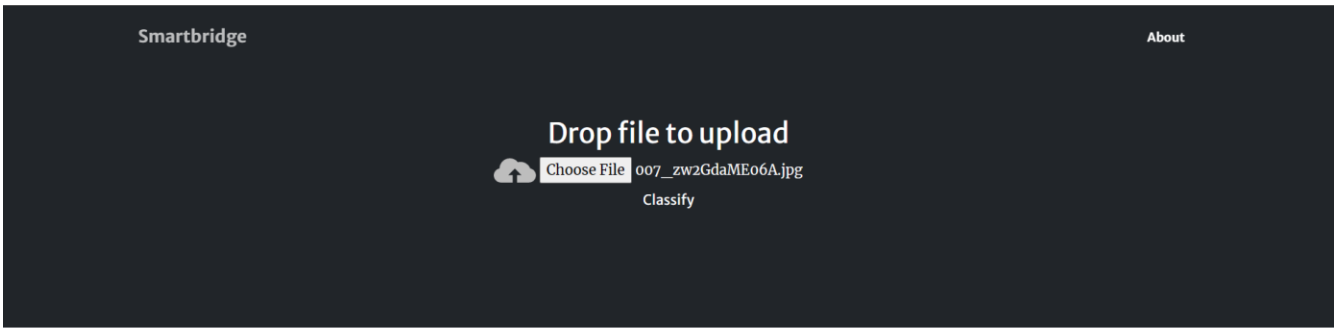
- Open Spyder
- Navigate to the folder where your Python script is.
- Now click on the green play button above.
- Click on the predict button from the top right corner, enter the inputs, click on the Classify button, and see the result/prediction on the web.

```
* Serving Flask app "app" (lazy loading)
* Environment: production
  WARNING: This is a development server. Do not use it in a
roduction 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 testbutton

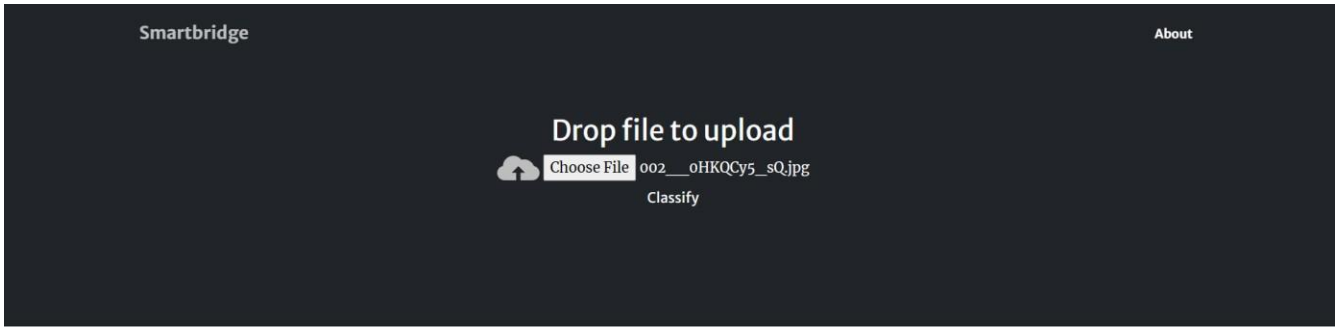
Input 1:



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

Output:1

Input:2



Output:2