Project Development Phase

Date	6 November 2023
Team ID	Team-591965
Project Name	Weather Classification Using Deep Learning
Maximum Marks	15 Marks

Weather Classification Using Deep Learning

Project Description:

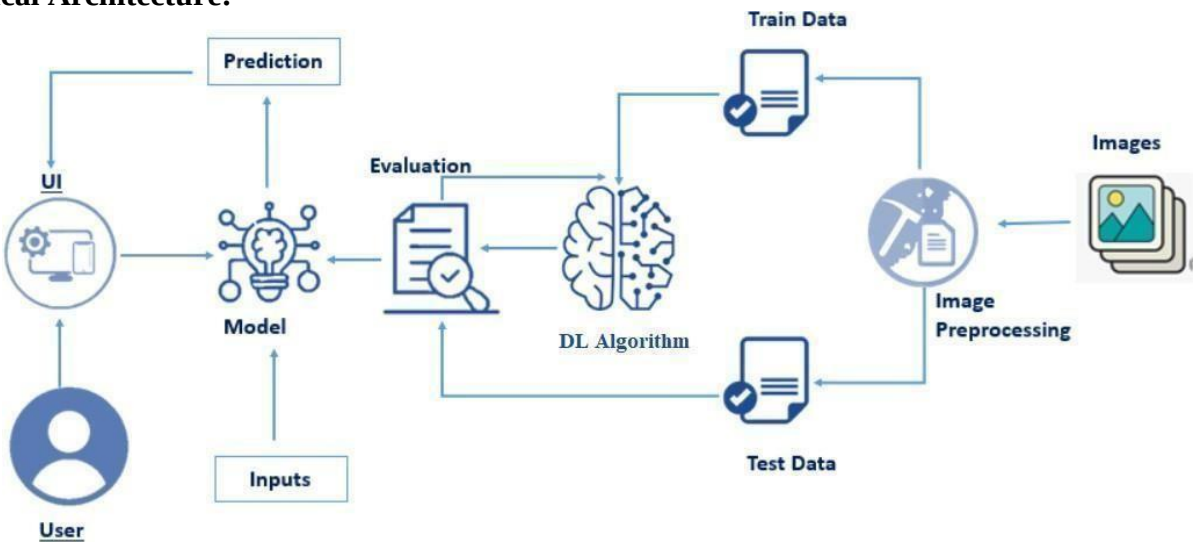
Weather classification is an essential tool for meteorologists and weather forecasters to predict weather patterns and communicate them to the public.Weather phenomenon recognition notably affects many aspects of our daily lives, The analysis of weather phenomenon plays a crucial role in various applications, for example, environmental monitoring, weather forecasting, and the assessment of environmental quality. Besides, different weather phenomena have diverse effects on agriculture. Therefore, accurately distinguishing weather phenomena can improve agricultural planning

The use of a pre-trained model on a new problem is known as transfer learning in machine learning. A machine uses the knowledge learned from a prior assignment to increase prediction about a new task in transfer learning

In this project we are classifying various types of weather. These weathers are majorly classified into 5 categories namely Cloudy, Shine, Rain, Foggy, Sunrise. Deep-learning (DL) methods in artificial intelligence (AI) play a dominant role as high-performance classifiers.

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, VGG19, Xception V3 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 to choose an image.
- The chosen image is processed by a VGG19 deep learning model.
- The VGG19 model is integrated with a Flask application.
- The VGG19 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.
    - Download & extract the dataset.
- 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
    - Checking model accuracy
    - Test the model
- Application Building
    - Create an HTML file

Prior Knowledge:

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

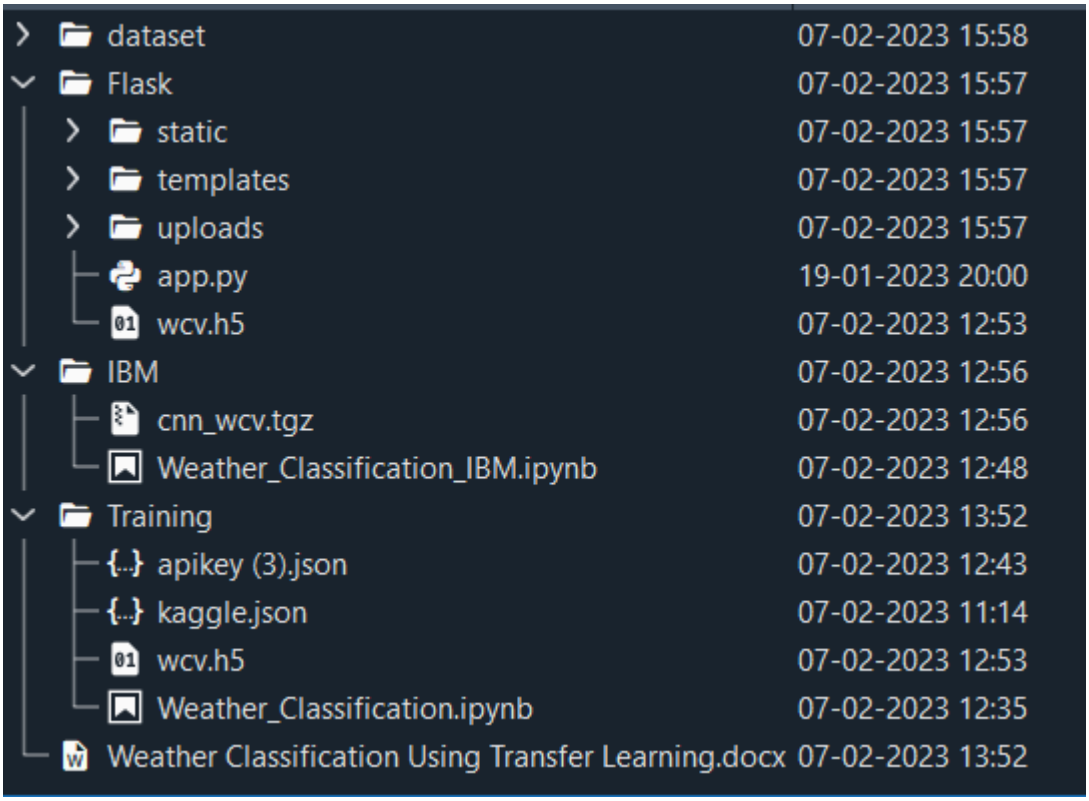
- **Deep Learning Concepts**
  - **CNN**: https://towardsdatascience.com/basics-of-the-classic-cnn-a3dce1225add
  - **VGG19**: VGG16 and VGG19 (keras.io)
  - **ResNet-50**: https://towardsdatascience.com/understanding-and-coding-a-resnet-in-keras-446d7ff84d33
  - **Inception-V3**: https://iq.opengenus.org/inception-v3-model-architecture/
  - **Xception**: https://pyimagesearch.com/2017/03/20/imagenet-vggnet-resnet-inception- xception-keras/
- **Flask:** Flask is a popular Python web framework, meaning it is a third-party Python library used for developing web applications.

  Link: https://www.youtube.com/watch?v=lj4I_CvBnt0

Build Python Code

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 need 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 Weather_Classification.ipynb model training file & wcv.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 & extract the dataset

Collect images of Weather then organize into subdirectories based on their respective names as shown in the project structure. Create folders of types of Weather that need to be recognized.

In this project, we have collected images of 5 types of Weather images like Cloudy, Shine, Rain,

Foggy Sunrise 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:- Weather Classification | Kaggle

Note: For better accuracy train on more images

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

We will be connecting Google 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/multiclass-weather-dataset.zip'
```

```
  inflating: dataset/shine/shine148.jpg
  inflating: dataset/shine/shine149.jpg
  inflating: dataset/shine/shine15.jpg
  inflating: dataset/shine/shine150.jpg
  inflating: dataset/shine/shine151.jpg
  inflating: dataset/shine/shine152.jpg
  inflating: dataset/shine/shine153.jpg
  inflating: dataset/shine/shine154.jpg
  inflating: dataset/shine/shine155.jpg
  inflating: dataset/shine/shine156.jpg
  inflating: dataset/shine/shine157.jpg
  inflating: dataset/shine/shine158.jpg
  inflating: dataset/shine/shine159.jpg
  inflating: dataset/shine/shine16.jpg
  inflating: dataset/shine/shine160.jpg
```

Four different transfer learning models are used in our project and the best model (VGG19) is selected.

The image input size of VGG19 model is 180, 180.

```
IMAGE_SIZE=[180,180]
```

Milestone 2: Image Preprocessing

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

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

Activity 1: Importing the libraries

Import the necessary libraries as shown in the image

```
from tensorflow.keras.applications.vgg19 import VGG19, preprocess_input
from tensorflow.keras.layers import Flatten, Dense
from tensorflow.keras.models import Model
from tensorflow.keras.preprocessing.image import ImageDataGenerator
from PIL import ImageFile
from tensorflow.keras.models import load_model
from tensorflow.keras.preprocessing import image
import numpy as np
```

.

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.
- 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.

An instance of the ImageDataGenerator class can be constructed for train and test.

```
from tensorflow.keras.preprocessing.image import ImageDataGenerator
train_datagen = ImageDataGenerator(rescale = 1./255,
                                   shear_range = 0.2,
                                   zoom_range = [.99, 1.01],
                                   brightness_range= [0.8, 1.2],
                                   data_format= "channels_last",
                                   fill_mode="constant",
                                   horizontal_flip = True)

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

Activity 3: Apply ImageDataGenerator functionality to Train set and Test set

Let us apply ImageDataGenerator functionality to the Train set and Test set by using the following code.

For Training set using flow_from_directory function.

This function will return batches of images from the subdirectories

Arguments:

- directory: Directory where the data is located. If labels are "inferred", it should contain subdirectories, each containing images for a class. Otherwise, the directory structure is ignored.
- batch_size: Size of the batches of data which is 64.
- target_size: Size to resize images after they are read from disk.
- class_mode:
  - ‘int': means that the labels are encoded as integers (e.g. for sparse_categorical_crossentropy loss).
  - 'categorical' means that the labels are encoded as a categorical vector (e.g. for categorical_crossentropy loss).
  - 'binary' means that the labels (there can be only 2) are encoded as float32 scalars with values 0 or 1 (e.g. for binary_crossentropy).
  - None (no labels).

```
training_set = train_datagen.flow_from_directory('/content/train',
                                                 target_size = (180, 180),
                                                 batch_size = 64,
                                                 class_mode = 'categorical')

test_set = test_datagen.flow_from_directory('/content/test',
                                            target_size = (180, 180),
                                            batch_size = 64,
                                            class_mode = 'categorical')
```

Total the dataset is having 1200 train images, 300 test images divided under 5 classes.

Milestone 3: Model Building

Now it's time to build our model. Let's use the pre-trained model which is VGG19, one of the convolution neural net (CNN) architecture which is considered as a very good model for Image classification.

Deep understanding on the VGG19 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 (180,180,3).

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

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

```
VGG19 = VGG19(input_shape=IMAGE_SIZE + [3], weights='imagenet',include_top=False)
```

```
Downloading data from https://storage.googleapis.com/tensorflow/keras-applications/vgg19/vg
80134624/80134624 [==============================] - 1s 0us/step
```

```
for layer in VGG19.layers:
  layer.trainable = False
```

```
x = Flatten()(VGG19.output)
```

Activity 2: Adding Dense Layers

```
x = Flatten()(VGG19.output)
```

```
prediction = Dense(5, activation='softmax')(x)
```

```
model = Model(inputs=VGG19.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 VGG19.input and output as dense layer.

The number of neurons in the Dense layer is the same as the number of classes in the training set.

The neurons in the last Dense layer, use softmax activation to convert their outputs into respective probabilities. Understanding the model is a very important phase to properly use it for training and prediction purposes.

Keras provides a simple method, summary to get the full information about the model and its layers.

```
model.summary()
```

```
Model: "model"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 input_1 (InputLayer)        [(None, 180, 180, 3)]     0

 block1_conv1 (Conv2D)       (None, 180, 180, 64)      1792

 block1_conv2 (Conv2D)       (None, 180, 180, 64)      36928

 block1_pool (MaxPooling2D)  (None, 90, 90, 64)        0

 block2_conv1 (Conv2D)       (None, 90, 90, 128)       73856

 block2_conv2 (Conv2D)       (None, 90, 90, 128)       147584

 block2_pool (MaxPooling2D)  (None, 45, 45, 128)       0

 block3_conv1 (Conv2D)       (None, 45, 45, 256)       295168

 block3_conv2 (Conv2D)       (None, 45, 45, 256)       590080

 block3_conv3 (Conv2D)       (None, 45, 45, 256)       590080

 block3_conv4 (Conv2D)       (None, 45, 45, 256)       590080

 block3_pool (MaxPooling2D)  (None, 22, 22, 256)       0

 block4_conv1 (Conv2D)       (None, 22, 22, 512)       1180160

 block4_conv2 (Conv2D)       (None, 22, 22, 512)       2359808
```

```
 block4_conv3 (Conv2D)       (None, 22, 22, 512)       2359808

 block4_conv4 (Conv2D)       (None, 22, 22, 512)       2359808

 block4_pool (MaxPooling2D)  (None, 11, 11, 512)       0

 block5_conv1 (Conv2D)       (None, 11, 11, 512)       2359808

 block5_conv2 (Conv2D)       (None, 11, 11, 512)       2359808

 block5_conv3 (Conv2D)       (None, 11, 11, 512)       2359808

 block5_conv4 (Conv2D)       (None, 11, 11, 512)       2359808

 block5_pool (MaxPooling2D)  (None, 5, 5, 512)         0

 flatten (Flatten)           (None, 12800)             0

 dense_1 (Dense)             (None, 5)                 64005

=================================================================
Total params: 20,088,389
Trainable params: 64,005
Non-trainable params: 20,024,384
_________________________________________________________________
```

Activity 3: Configure the Learning Process

The compilation is the final step in creating a model. Once the compilation is done, we can move on to The training phase. The loss function is used to find errors or deviations in the learning process. Keras requires a loss function during the model compilation process.

Optimization is an important process that optimizes the input weights by comparing the prediction and the loss function. Here we are using adam optimizer

Metrics are used to evaluate the performance of your model. It is similar to the loss function, but not used in the training process.

```
model.compile(loss='categorical_crossentropy', optimizer='adam', metrics=['accuracy'])
```

Activity 4: Train the model

Now, let us train our model with our image dataset. The model is trained for 50 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 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.

```
r = model.fit(
  training_set,
  validation_data=test_set,
  epochs=50,
  steps_per_epoch=len(training_set),
  validation_steps=len(test_set)
)
```

```
Epoch 35/50
19/19 [==============================] - 22s 1s/step - loss: 0.0492 - accuracy: 0.9933 - val_loss: 0.3652 - val_accuracy: 0.8867
Epoch 36/50
19/19 [==============================] - 21s 1s/step - loss: 0.0459 - accuracy: 0.9950 - val_loss: 0.3861 - val_accuracy: 0.8867
Epoch 37/50
19/19 [==============================] - 22s 1s/step - loss: 0.0474 - accuracy: 0.9933 - val_loss: 0.3280 - val_accuracy: 0.9033
Epoch 38/50
19/19 [==============================] - 22s 1s/step - loss: 0.0522 - accuracy: 0.9933 - val_loss: 0.3704 - val_accuracy: 0.8867
Epoch 39/50
19/19 [==============================] - 22s 1s/step - loss: 0.0464 - accuracy: 0.9942 - val_loss: 0.4042 - val_accuracy: 0.8867
Epoch 40/50
19/19 [==============================] - 22s 1s/step - loss: 0.0499 - accuracy: 0.9925 - val_loss: 0.4149 - val_accuracy: 0.8867
Epoch 41/50
19/19 [==============================] - 21s 1s/step - loss: 0.0447 - accuracy: 0.9958 - val_loss: 0.4006 - val_accuracy: 0.8833
Epoch 42/50
19/19 [==============================] - 22s 1s/step - loss: 0.0452 - accuracy: 0.9942 - val_loss: 0.3110 - val_accuracy: 0.9100
Epoch 43/50
19/19 [==============================] - 22s 1s/step - loss: 0.0466 - accuracy: 0.9967 - val_loss: 0.3272 - val_accuracy: 0.8967
Epoch 44/50
19/19 [==============================] - 22s 1s/step - loss: 0.0408 - accuracy: 0.9917 - val_loss: 0.3340 - val_accuracy: 0.8967
Epoch 45/50
19/19 [==============================] - 21s 1s/step - loss: 0.0405 - accuracy: 0.9950 - val_loss: 0.4396 - val_accuracy: 0.8733
Epoch 46/50
19/19 [==============================] - 22s 1s/step - loss: 0.0438 - accuracy: 0.9933 - val_loss: 0.3572 - val_accuracy: 0.9033
Epoch 47/50
19/19 [==============================] - 22s 1s/step - loss: 0.0416 - accuracy: 0.9925 - val_loss: 0.3708 - val_accuracy: 0.8900
Epoch 48/50
19/19 [==============================] - 21s 1s/step - loss: 0.0415 - accuracy: 0.9908 - val_loss: 0.3959 - val_accuracy: 0.8867
Epoch 49/50
19/19 [==============================] - 22s 1s/step - loss: 0.0390 - accuracy: 0.9942 - val_loss: 0.3136 - val_accuracy: 0.9067
Epoch 50/50
19/19 [==============================] - 21s 1s/step - loss: 0.0321 - accuracy: 0.9942 - val_loss: 0.3324 - val_accuracy: 0.8967
```

From the above run time, we can observe that at 41st epoch the model is giving the best accuracy.

Activity 5: Save the Model

Out of all the models we tried (VGG19, Inception V3 & Xception V3) VGG19 gave us the best accuracy.
We are saving VGG19 as our final model.

The model is saved with .h5 extension as follows

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

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

Activity 6 : Checking model accuracy

```
loss, accuracy = model.evaluate(test_set,
                                steps=11,
                                verbose=2,
                                use_multiprocessing=True,
                                workers=2)
print(f'Model performance on test images:\nAccuracy = {accuracy}\nLoss = {loss}')
```

```
11/11 - 5s - loss: 0.3486 - accuracy: 0.8886 - 5s/epoch - 464ms/step
Model performance on test images:
Accuracy = 0.8885542154312134
Loss = 0.3486107885837555
```

Activity 7: 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.

```
from tensorflow.keras.models import load_model
from tensorflow.keras.preprocessing import image
import numpy as np
model = load_model("/content/wcv.h5")
```

Taking an image as input and checking the results

```
img = image.load_img(r"/content/test/foggy/foggy244.jpg",target_size= (180,180))#loading of the image
x = image.img_to_array(img)#image to array
x = np.expand_dims(x,axis = 0)#changing the shape
preds=model.predict(x)
pred=np.argmax(preds,axis=1)
index=['cloudy','foggy','rainy','shine','sunrise']
result=str(index[pred[0]])
result
```

```
1/1 [==============================] - 1s 775ms/step
'foggy'
```

So, our model has predicted the the class of input correctly as Foggy.

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
- Run the application

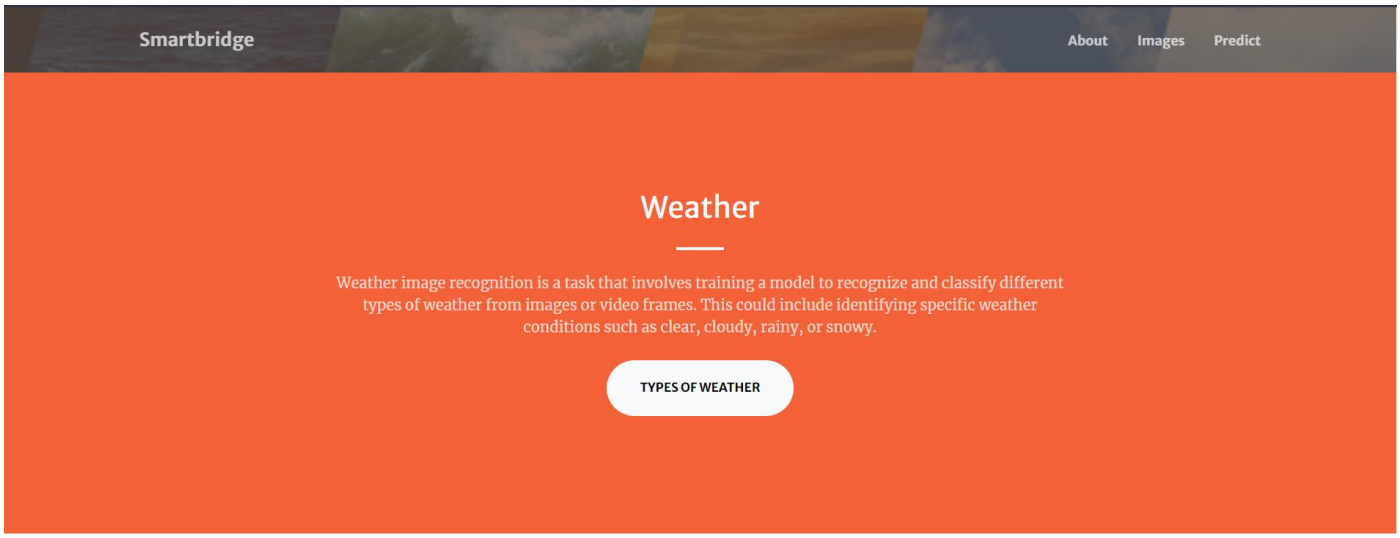
Activity1: Building Html Pages:

For this project create one HTML file namely

- index.html

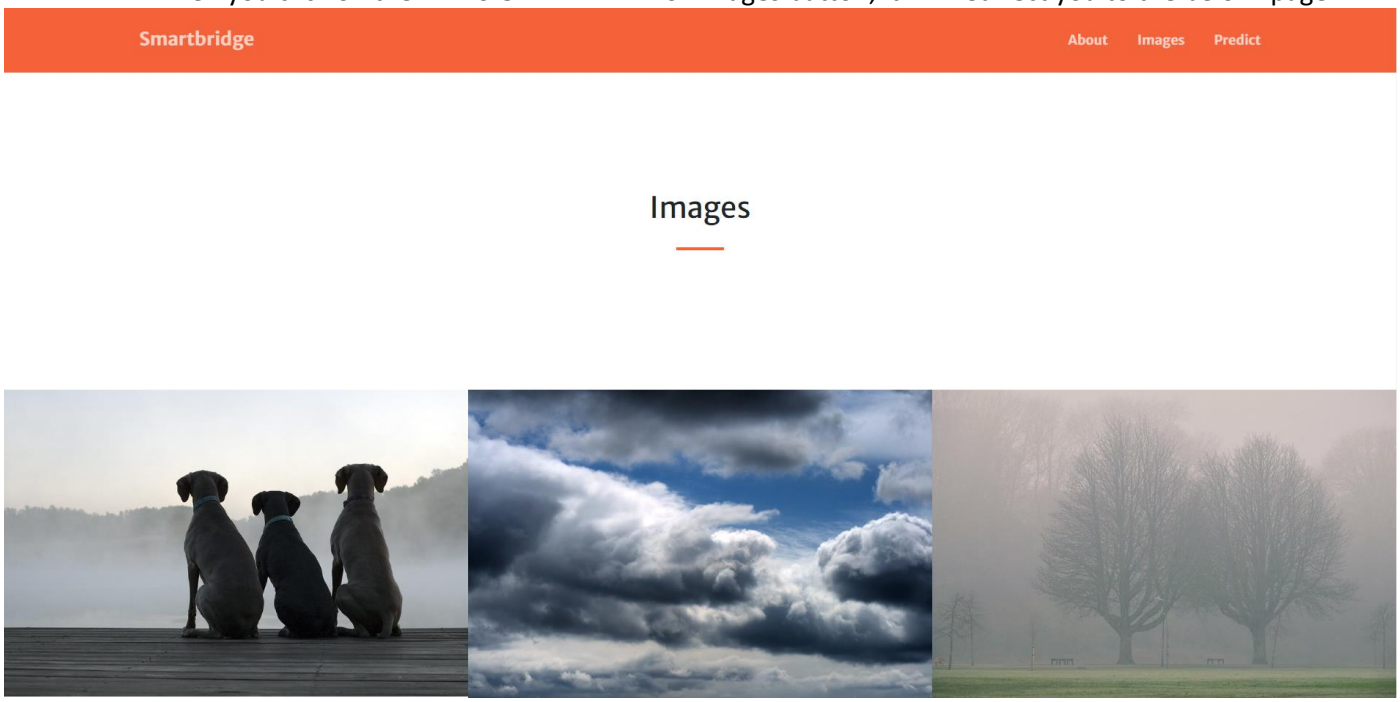
Let's see how our index.html page looks like:

When you click on the FEW DETAILS or About button, you will be redirecting to the following page

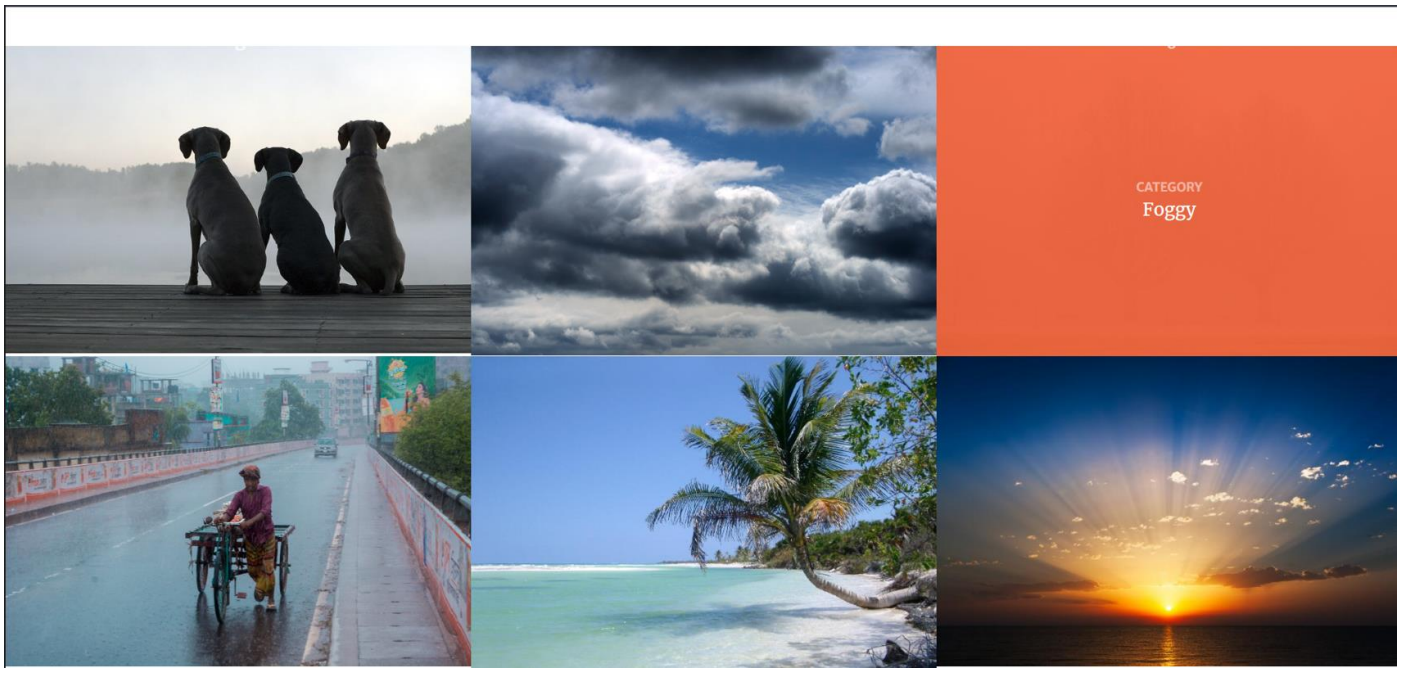

Smartbridge
About
Images
Predict
Weather
Weather image recognition is a task that involves training a model to recognize and classify different types of weather from images or video frames. This could include identifying specific weather conditions such as clear, cloudy, rainy, or snowy.
TYPES OF WEATHER

Images

When you click on the TYPES OF WEATHER or Images button, it will redirect you to the below page



When you scroll down or hover over the images you will be able to see names of various Weather



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.vgg19 import preprocess_input
```

Loading the saved model and initializing the flask app

```
model=load_model(r"wcv.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=['alien_test','cloudy','foggy','rainy','shine','sunrise']

        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 be rendered to the text that we have mentioned in the index.html page earlier.

Main Function:

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

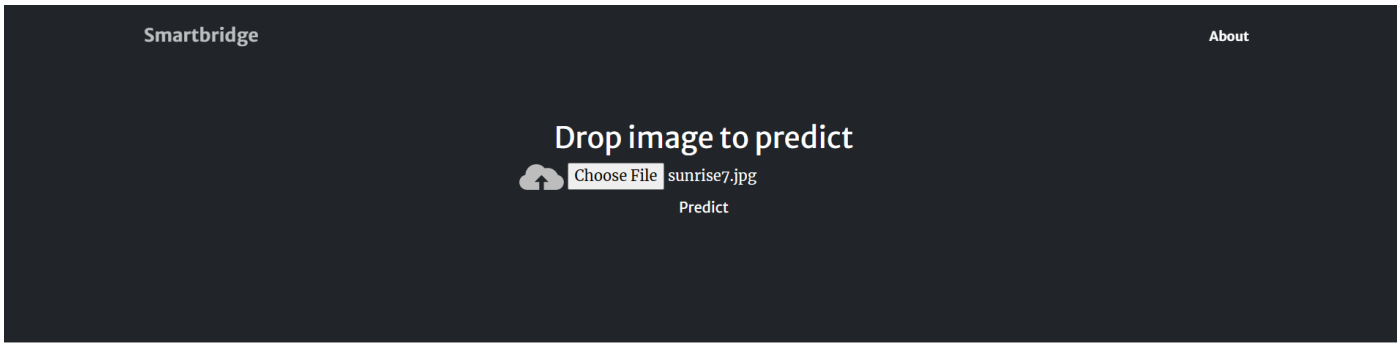
Activity 3: Run the application

- 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

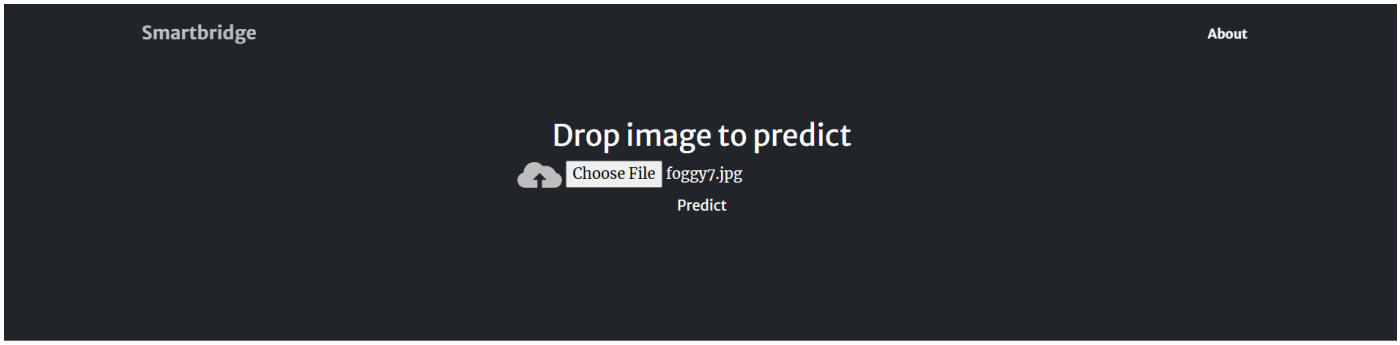
Input 1:



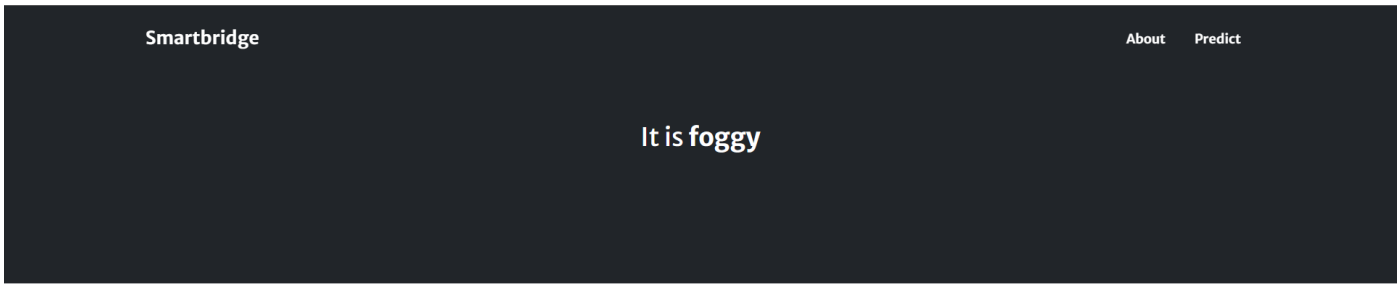
Once you upload the image and click on Predict button, the output will be displayed in the below page.

Output 1:

Input 2:



Output2:


Smartbridge
About
Predict
It is foggy