Potato Classification Using Deep Learning

Introduction:

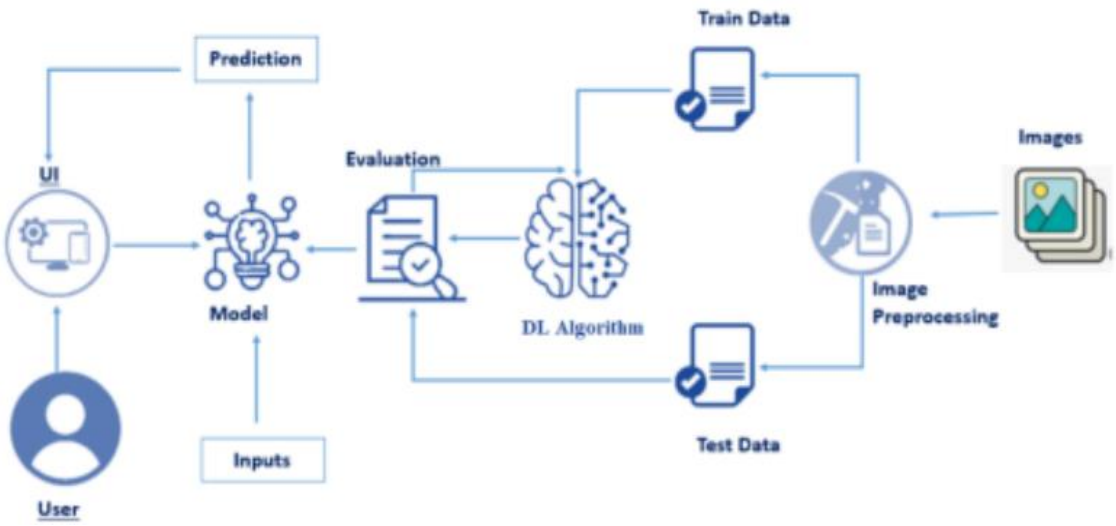
The agricultural industry plays a crucial role in global food production, with plants serving as the backbone of agricultural output. Accurate and efficient potato disease classification is essential for various applications, including crop health monitoring, disease management, and agricultural regulation.

Traditional methods for potato disease classification have often relied on manual inspection, which can be time-consuming and prone to errors. With the advancement of deep learning techniques, there is a significant opportunity to transform potato disease classification by automating the process with high accuracy and speed.

Potato disease classification involves categorizing potato plants into various disease types, such as late blight, early blight, black scurf, and others, based on their symptoms, visual characteristics, and other attributes. Deep learning models like Convolutional Neural Networks (CNNs) have demonstrated their ability to extract fine-grained features from plant images, enabling the development of accurate and efficient potato disease classification systems.

The images in the dataset pertain to various categories of potato diseases, each with distinct symptoms and visual markers. The model used for training is based on a well-established CNN architecture like resnat model, a computer vision model. Pre-trained weights of the model are leveraged and customized to perform potato disease classification. The model is deployed using the Flask framework, making it accessible for practical applications in agriculture.

Technical Architecture:



Prerequisites:

To complete this project, you would need the following software's, concepts and packages.

- Anaconda Navigator: It is a free and open-source distribution of the python and R programming languages for data science and machine learning related applications.

  It can be installed on Windows, Linux and macOS. It also comes with nice tools like JupyterLab, Jupyter, Notebook etc.

To build the Machine learning models you would need the following packages.

- NumPy: It is an open-source numerical Python library. It contains a multidimensional array and matrix data structures and can be used to perform mathematical operations.
- Scikit-learn: It is a free machine learning library for Python. It features various algorithms like support vector machine, random forests, etc. and also supports NumPy.
- Flask: It is a web framework used to build Web Applications.
- Kera's: It is an open-source library that provides a Python interface for artificial neural networks. It also acts as an interface for the TensorFlow library.
- TensorFlow: TensorFlow is an end-to-end open-source platform for machine learning. It has a comprehensive, flexible ecosystem of tools, libraries, and community resources that lets researchers push the state-of-the-art in ML and developers can easily build and deploy ML powered applications.
- Pandas: This is a fast, powerful, flexible, and easy to use open-source data analysis and manipulation tool, built on top of the Python programming language.

If you are using anaconda navigator, do below steps to download required packages:

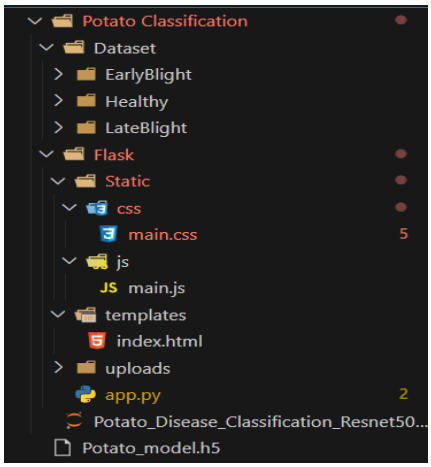
1. Open anaconda prompt.
2. Type "pip install NumPy" and click enter.
3. Type "pip install pandas" and click enter.
4. Type "pip install TensorFlow==2.8.2" and click enter.
5. Type "pip install keras=2.8.0" and click enter.
6. Type "pip install Flask" and click enter.
7. Type "pip install requests" and click enter.

Deep Learning Concepts:

CNN: A convolutional neural network is a class of deep neural networks, most commonly applied to analysing visual imagery.

Flask: Flask is a popular Python web framework, meaning it is a third-party Python library used for developing web applications.

Project Structure:



- The dataset folder contains the images of different conditions of potatoes in their respective folders.
- The flask folder has all necessary files to build the flask application.
- The static folder contains the images and the style sheets needed to customize the web page.
- The templates folder has the HTML pages.
- App.py is the python script for server-side computing.
- Potato Classification.ipynb is the notebook on which the code is executed.

Project Objectives:

By the end of this project, you'll understand:

1. How to pre-process the images.
2. Training resnat model with custom data.
3. How pre-trained models will be useful in object classification.
4. How to evaluate the model.
5. Building a web application using the Flask framework.

Project Flow:

Preparing the data:

- Downloading the dataset
- Categorize the images

Data pre-processing:

- Import ImageDataGenerator Library and Configure it
- Apply ImageDataGenerator functionality to Train and Test set

Building the model:

- Creating and compiling the pre-trained model
- Look at the model summary
- Train the model while monitoring validation loss
- Test the model with custom inputs

Building Flask application:

- Build a flask application
- Build the HTML page and execute

Milestone 1: Preparing the data

The images need to be organized before proceeding with the project.

Downloading the dataset

Collecting the data from the below link:

https://www.kaggle.com/datasets/swastik2004/potato-leaf-diseases

Milestone 2: Data Pre-Processing

The dataset images are to be pre-processed before giving it to the model.

Activity 1: Import ImageDataGenerator Library and Configure it

ImageDataGenerator class is used to augment the images with different modifications like considering the rotation, flipping the image etc.

```
[ ]  from tensorflow.keras.preprocessing.image import ImageDataGenerator

     training_data = keras.preprocessing.image_dataset_from_directory(
         '/content/Dataset',
         batch_size = 70,
         image_size =(240,240),

         shuffle = True,
         seed =123,
         subset ='training',
         validation_split=0.15,
         )
```

Activity 2: Apply ImageDataGenerator functionality to Train and Validation set

Specify the path of both the folders in flow_from_directory method. We are importing the images in 240*240 pixels.

```
    training_data = keras.preprocessing.image_dataset_from_directory(
        '/content/Dataset',
        batch_size = 70,
        image_size =(240,240),

        shuffle = True,
        seed =123,
        subset ='training',
        validation_split=0.15,
        )
    validation_data =keras.preprocessing.image_dataset_from_directory(
        '/content/Dataset',
        batch_size = 70,
        image_size =(240,240),

        shuffle = True,
        seed =123,
        validation_split =0.15,
        subset ='validation',
        )
```

```
Found 4072 files belonging to 3 classes.
Using 3462 files for training.
Found 4072 files belonging to 3 classes.
Using 610 files for validation.
```

Milestone 3: Building the model

We will be creating the pre-trained VGG19 model and ResNet50 model for predicting custom classes and choose the model with the highest accuracy.

Activity 1: Creating and compiling the model

VGG19

Firstly, import the necessary libraries and define a function for creation of the model. Next, call the pre trained model with the parameter include_top=False, as we need to use the model for predicting custom images.

```
from tensorflow.keras.applications import VGG19
from tensorflow.keras.models import Model
from tensorflow.keras.layers import Flatten, Dense
from tensorflow.keras.optimizers import Adam
```

```
base_model = VGG19(weights='imagenet', include_top=False, input_shape=(300, 300, 3))
```

ResNet50

```
resnet_model = Sequential()
pretrained_model= ResNet50(include_top=False,
                   input_shape=(240,240,3),
                   pooling='avg',
                   weights='imagenet')
for layer in pretrained_model.layers:
  layer.trainable=False
```

Next, add dense layers to the top model. Finally, add an output layer with the number of neurons equal to out output classes.

VGG19

```
[ ]  for layer in base_model.layers:
         layer.trainable = False

     x = Flatten()(base_model.output)
     x = Dense(256, activation='relu')(x)
     x = Dense(128, activation='relu')(x)
     output = Dense(3, activation='softmax')(x)  # Replace 'num_classes' with your actual number of classes
```

ResNet50

```
resnet_model.add(pretrained_model)
resnet_model.add(Flatten())
resnet_model.add(BatchNormalization())
resnet_model.add(Dense(512, activation='relu'))
resnet_model.add(BatchNormalization())
resnet_model.add(Dense(4, activation='softmax'))
```

Finally, compile the model.

VGG19

```
[ ] model = Model(inputs=base_model.input, outputs=output)
    model.compile(optimizer=Adam(learning_rate=0.001), loss='categorical_crossentropy', metrics=['accuracy'])
```

ResNet50

```
resnet_model.compile(
    loss='sparse_categorical_crossentropy',
    optimizer='adam',
    metrics=['accuracy']
)
```

Activity 2: Look at the model summary

Call the summary () function to have a look at the model summary:

VGG19

```
model.summary()
```

```
Model: "model"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 input_1 (InputLayer)        [(None, 300, 300, 3)]     0
 block1_conv1 (Conv2D)       (None, 300, 300, 64)      1792
 block1_conv2 (Conv2D)       (None, 300, 300, 64)      36928
 block1_pool (MaxPooling2D)  (None, 150, 150, 64)      0
 block2_conv1 (Conv2D)       (None, 150, 150, 128)     73856
 block2_conv2 (Conv2D)       (None, 150, 150, 128)     147584
 block2_pool (MaxPooling2D)  (None, 75, 75, 128)       0
 block3_conv1 (Conv2D)       (None, 75, 75, 256)       295168
 block3_conv2 (Conv2D)       (None, 75, 75, 256)       590080
 block3_conv3 (Conv2D)       (None, 75, 75, 256)       590080
 block3_conv4 (Conv2D)       (None, 75, 75, 256)       590080
```

```
 block3_pool (MaxPooling2D)  (None, 37, 37, 256)       0
 block4_conv1 (Conv2D)       (None, 37, 37, 512)       1180160
 block4_conv2 (Conv2D)       (None, 37, 37, 512)       2359808
 block4_conv3 (Conv2D)       (None, 37, 37, 512)       2359808
 block4_conv4 (Conv2D)       (None, 37, 37, 512)       2359808
 block4_pool (MaxPooling2D)  (None, 18, 18, 512)       0
 block5_conv1 (Conv2D)       (None, 18, 18, 512)       2359808
 block5_conv2 (Conv2D)       (None, 18, 18, 512)       2359808
 block5_conv3 (Conv2D)       (None, 18, 18, 512)       2359808
 block5_conv4 (Conv2D)       (None, 18, 18, 512)       2359808
 block5_pool (MaxPooling2D)  (None, 9, 9, 512)         0
 flatten (Flatten)           (None, 41472)             0
```

```
 flatten (Flatten)           (None, 41472)             0
 dense (Dense)               (None, 256)               10617088
 dense_1 (Dense)             (None, 128)               32896
 dense_2 (Dense)             (None, 3)                 387
=================================================================
Total params: 30674755 (117.01 MB)
Trainable params: 10650371 (40.63 MB)
Non-trainable params: 20024384 (76.39 MB)
_________________________________________________________________
```

ResNet50

```
resnet_model.summary()
```

```
Model: "sequential"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 resnet50 (Functional)       (None, 2048)              23587712

 flatten (Flatten)           (None, 2048)              0

 batch_normalization (Batch  (None, 2048)              8192
 Normalization)

 dense (Dense)               (None, 512)               1049088

 batch_normalization_1 (Bat  (None, 512)               2048
 chNormalization)

 dense_1 (Dense)             (None, 4)                 2052

=================================================================
Total params: 24649092 (94.03 MB)
Trainable params: 1056260 (4.03 MB)
Non-trainable params: 23592832 (90.00 MB)
_________________________________________________________________
```

Milestone 4: Training and testing the model

Activity 1: Train the model while monitoring validation loss

We will be training the model for 12 epochs using the fit() function:

VGG19

```
history = model.fit(train_batch,
                    steps_per_epoch=len(train_batch),
                    epochs=20,
                    validation_data=valid_batch,
                    validation_steps=len(valid_batch))
```

```
Epoch 1/20
51/51 [==============================] - 137s 2s/step - loss: 1.1687 - accuracy: 0.6563 - val_loss: 0.8760 - val_accuracy: 0.5916
Epoch 2/20
51/51 [==============================] - 96s 2s/step - loss: 0.3884 - accuracy: 0.8460 - val_loss: 0.5634 - val_accuracy: 0.7626
Epoch 3/20
51/51 [==============================] - 97s 2s/step - loss: 0.3373 - accuracy: 0.8678 - val_loss: 0.7003 - val_accuracy: 0.7036
Epoch 4/20
51/51 [==============================] - 96s 2s/step - loss: 0.2695 - accuracy: 0.9006 - val_loss: 0.6073 - val_accuracy: 0.7651
Epoch 5/20
51/51 [==============================] - 95s 2s/step - loss: 0.2752 - accuracy: 0.8978 - val_loss: 0.7050 - val_accuracy: 0.7491
Epoch 6/20
51/51 [==============================] - 96s 2s/step - loss: 0.2926 - accuracy: 0.8972 - val_loss: 0.7410 - val_accuracy: 0.6974
Epoch 7/20
51/51 [==============================] - 98s 2s/step - loss: 0.3115 - accuracy: 0.8819 - val_loss: 1.1235 - val_accuracy: 0.6384
Epoch 8/20
51/51 [==============================] - 97s 2s/step - loss: 0.3296 - accuracy: 0.8751 - val_loss: 0.7098 - val_accuracy: 0.7491
Epoch 9/20
51/51 [==============================] - 95s 2s/step - loss: 0.1918 - accuracy: 0.9276 - val_loss: 0.5112 - val_accuracy: 0.7983
Epoch 10/20
51/51 [==============================] - 99s 2s/step - loss: 0.1811 - accuracy: 0.9310 - val_loss: 0.6497 - val_accuracy: 0.7749
Epoch 11/20
51/51 [==============================] - 99s 2s/step - loss: 0.2209 - accuracy: 0.9159 - val_loss: 0.4945 - val_accuracy: 0.8057
Epoch 12/20
51/51 [==============================] - 97s 2s/step - loss: 0.1491 - accuracy: 0.9506 - val_loss: 0.5268 - val_accuracy: 0.8106
Epoch 13/20
51/51 [==============================] - 97s 2s/step - loss: 0.1335 - accuracy: 0.9509 - val_loss: 0.5203 - val_accuracy: 0.8093
Epoch 14/20
51/51 [==============================] - 98s 2s/step - loss: 0.1890 - accuracy: 0.9270 - val_loss: 0.6613 - val_accuracy: 0.7798
Epoch 15/20
51/51 [==============================] - 97s 2s/step - loss: 0.1250 - accuracy: 0.9555 - val_loss: 0.7076 - val_accuracy: 0.7774
Epoch 16/20
51/51 [==============================] - 98s 2s/step - loss: 0.1852 - accuracy: 0.9310 - val_loss: 0.4994 - val_accuracy: 0.8093
Epoch 17/20
51/51 [==============================] - 97s 2s/step - loss: 0.1498 - accuracy: 0.9488 - val_loss: 0.9279 - val_accuracy: 0.7478
Epoch 18/20
51/51 [==============================] - 96s 2s/step - loss: 0.1551 - accuracy: 0.9423 - val_loss: 0.5196 - val_accuracy: 0.8155
Epoch 19/20
51/51 [==============================] - 97s 2s/step - loss: 0.1776 - accuracy: 0.9307 - val_loss: 0.9083 - val_accuracy: 0.7294
Epoch 20/20
51/51 [==============================] - 98s 2s/step - loss: 0.1404 - accuracy: 0.9481 - val_loss: 0.7183 - val_accuracy: 0.7552
```

ResNet50

```
[ ] history = resnet_model.fit(training_data,
                        steps_per_epoch=len(training_data),
                        epochs=20,
                        validation_data=validation_data,
                        validation_steps=len(validation_data))
```

```
Epoch 1/20
50/50 [==============================] - 32s 358ms/step - loss: 0.3138 - accuracy: 0.9113 - val_loss: 0.2253 - val_accuracy: 0.9426
Epoch 2/20
50/50 [==============================] - 15s 283ms/step - loss: 0.0657 - accuracy: 0.9786 - val_loss: 0.1386 - val_accuracy: 0.9639
Epoch 3/20
50/50 [==============================] - 14s 278ms/step - loss: 0.0279 - accuracy: 0.9916 - val_loss: 0.1251 - val_accuracy: 0.9574
Epoch 4/20
50/50 [==============================] - 15s 286ms/step - loss: 0.0209 - accuracy: 0.9928 - val_loss: 0.1143 - val_accuracy: 0.9590
Epoch 5/20
50/50 [==============================] - 15s 282ms/step - loss: 0.0167 - accuracy: 0.9939 - val_loss: 0.0891 - val_accuracy: 0.9623
Epoch 6/20
50/50 [==============================] - 15s 282ms/step - loss: 0.0093 - accuracy: 0.9974 - val_loss: 0.1041 - val_accuracy: 0.9590
Epoch 7/20
50/50 [==============================] - 15s 283ms/step - loss: 0.0140 - accuracy: 0.9954 - val_loss: 0.0904 - val_accuracy: 0.9639
Epoch 8/20
50/50 [==============================] - 15s 284ms/step - loss: 0.0068 - accuracy: 0.9980 - val_loss: 0.0966 - val_accuracy: 0.9689
Epoch 9/20
50/50 [==============================] - 15s 283ms/step - loss: 0.0060 - accuracy: 0.9991 - val_loss: 0.0915 - val_accuracy: 0.9689
Epoch 10/20
50/50 [==============================] - 15s 286ms/step - loss: 0.0074 - accuracy: 0.9974 - val_loss: 0.0876 - val_accuracy: 0.9689
Epoch 11/20
50/50 [==============================] - 15s 285ms/step - loss: 0.0069 - accuracy: 0.9980 - val_loss: 0.0920 - val_accuracy: 0.9672
Epoch 12/20
50/50 [==============================] - 15s 285ms/step - loss: 0.0043 - accuracy: 0.9997 - val_loss: 0.1078 - val_accuracy: 0.9689
Epoch 13/20
50/50 [==============================] - 15s 300ms/step - loss: 0.0040 - accuracy: 0.9991 - val_loss: 0.1001 - val_accuracy: 0.9623
Epoch 14/20
50/50 [==============================] - 15s 287ms/step - loss: 0.0025 - accuracy: 0.9994 - val_loss: 0.1086 - val_accuracy: 0.9656
```

```
Epoch 15/20
50/50 [==============================] - 15s 288ms/step - loss: 0.0068 - accuracy: 0.9977 - val_loss: 0.1400 - val_accuracy: 0.9557
Epoch 16/20
50/50 [==============================] - 15s 288ms/step - loss: 0.0107 - accuracy: 0.9965 - val_loss: 0.1128 - val_accuracy: 0.9607
Epoch 17/20
50/50 [==============================] - 15s 287ms/step - loss: 0.0095 - accuracy: 0.9965 - val_loss: 0.1217 - val_accuracy: 0.9623
Epoch 18/20
50/50 [==============================] - 15s 286ms/step - loss: 0.0166 - accuracy: 0.9939 - val_loss: 0.1332 - val_accuracy: 0.9574
Epoch 19/20
50/50 [==============================] - 15s 286ms/step - loss: 0.0134 - accuracy: 0.9945 - val_loss: 0.1144 - val_accuracy: 0.9689
Epoch 20/20
50/50 [==============================] - 17s 339ms/step - loss: 0.0082 - accuracy: 0.9965 - val_loss: 0.1230 - val_accuracy: 0.9721
```

Save the model after training it.

VGG19

```
import pickle

# Save the trained model to a file using pickle
with open('trained_model.pkl', 'wb') as model_file:
    pickle.dump(model, model_file)
```

ResNet50

```
[ ] # Save the trained model to a file using pickle
    resnet_model.save("Potato_model.h5")
```

Since the Inception model exhibits the highest level of accuracy, we have decided to select this model for our project.

Activity 2: Test the model with custom inputs

First, specify the path of the image to be tested. Then, pre-process the image and perform predictions.

```
from tensorflow.keras.preprocessing import image
import numpy as np

# Load and preprocess the image
img = image.load_img('/content/Dataset/EarlyBlight/EarlyBlight_1.jpg', target_size=(240, 240))
img_array = image.img_to_array(img)
img_array = np.expand_dims(img_array, axis=0)  # Add a batch dimension

# Make predictions
predictions = resnet_model.predict(img_array)  # Assuming 'model' is your trained VGG19 model
# Get the class with the highest probability
predicted_class_index = np.argmax(predictions)

# Define your class labels
class_labels = ['EarlyBlight', 'Healthy', 'LateBlight']  # Update with your actual class labels

# Get the predicted class label
predicted_class = class_labels[predicted_class_index]

print("Predicted class:", predicted_class)
```

```
1/1 [==============================] - 2s 2s/step
Predicted class: EarlyBlight
```

As we can see the custom images have been predicted successfully.

Milestone 5: Building Flask application

After the model is built, we will be integrating it to a web application so that normal users can also use it. The new users need to initially register in the portal. After registration users can login to browse the images to detect the condition of potatoes.

Activity 1: Build a python application

Step 1: Load the required packages.

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

Step 2: Initialize the flask app, load the model and configure the html pages

Instance of Flask is created and the model is loaded using load_model from keras.

```
app = Flask(__name__)
model = load_model(r"Potato_model.h5",compile = False)

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

@app.route('/predict',methods = ['GET','POST'])
```

Step 3: Pre-process the frame and run

```python
def upload():
    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 =(240,240))
        x = image.img_to_array(img)
        x = np.expand_dims(x,axis = 0)
        pred =np.argmax(model.predict(x),axis=1)
        index =['EarlyBlight','Healthy','LateBlight']
        text="The potato is : "+index[pred[0]]
    return text

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

Run the flask application using the run method. By default, the flask runs on port 5000. If the port is to be changed, an argument can be passed and the port can be modified.

Activity 2: Build the HTML page and execute

Build the UI where a home page will have details about the application and prediction page where a user is allowed to browse an image and get the predictions of images.

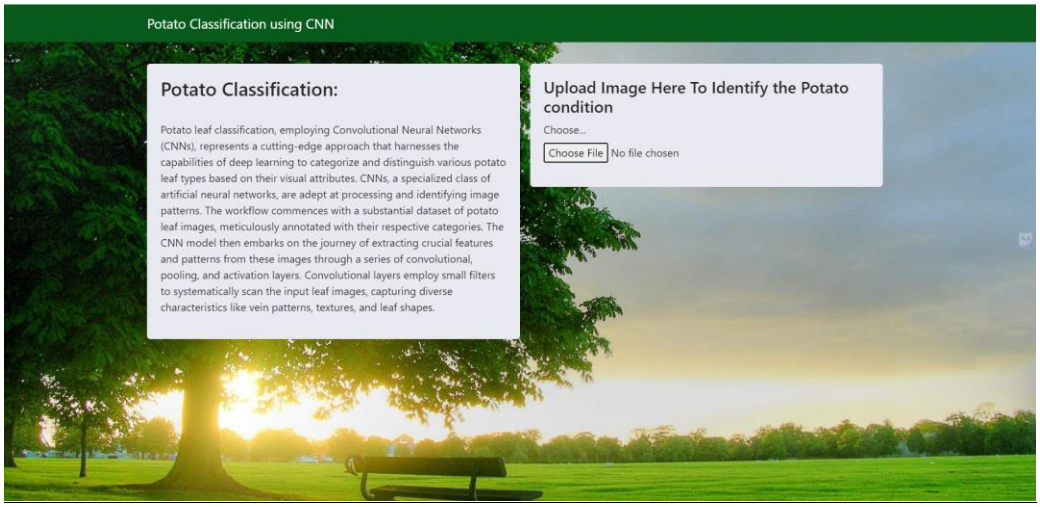
Step 1: Run the application

In the anaconda prompt, navigate to the folder in which the flask app is present. When the python file is executed, the localhost is activated on port 5000 and can be accessed through it.

```
C:\Users\Nidhish\Downloads\Potato Classification (2)> cmd /C "C:\Users\Nidhish\AppData\Local\Programs\Python\Python310\python.exe c:\Users\Nidhish\.vscode\extensions\ms-python.python-2023.20.0\pythonFiles\lib\python\debugpy\adapter/../..\debugpy\launcher 58644 -- "c:\Users\Nidhish\Downloads\Potato Classification (2)\Potato Classification\Flask\app.py" "
2023-11-05 21:20:18.238629: I tensorflow/core/platform/cpu_feature_guard.cc:182] This TensorFlow binary is optimized to use available CPU instructions in performance-critical operations.
To enable the following instructions: SSE SSE2 SSE3 SSE4.1 SSE4.2 AVX AVX2 FMA, in other operations, rebuild TensorFlow with the appropriate compiler flags.
 * Serving Flask app 'app'
 * Debug mode: on
WARNING: This is a development server. Do not use it in a production deployment. Use a production WSGI server instead.
 * Running on http://127.0.0.1:5000
Press CTRL+C to quit
 * Restarting with stat
2023-11-05 21:20:23.584280: I tensorflow/core/platform/cpu_feature_guard.cc:182] This TensorFlow binary is optimized to use available CPU instructions in performance-critical operations.
To enable the following instructions: SSE SSE2 SSE3 SSE4.1 SSE4.2 AVX AVX2 FMA, in other operations, rebuild TensorFlow with the appropriate compiler flags.
 * Debugger is active!
 * Debugger PIN: 267-315-841
```

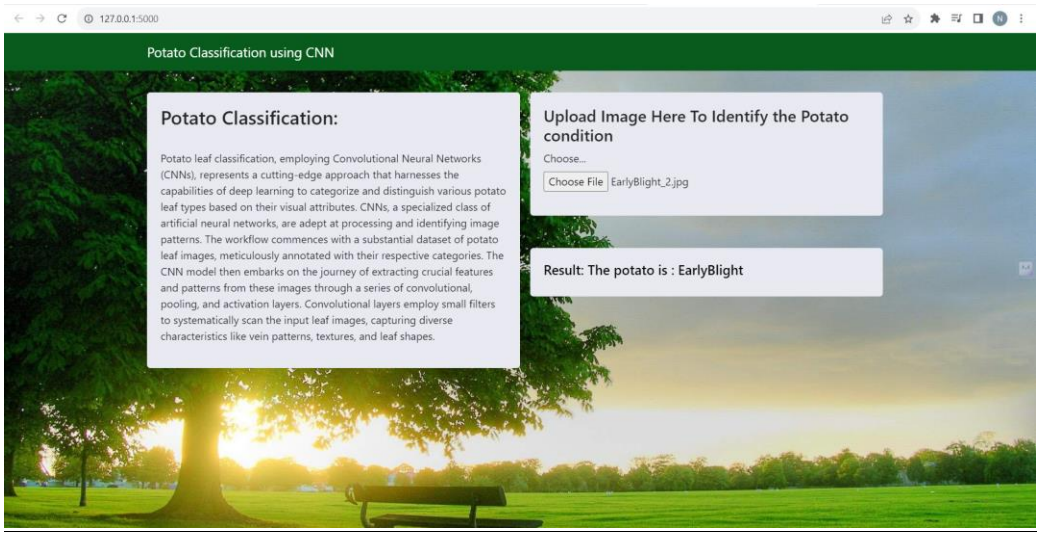
Step 2: Open the browser and navigate to localhost:5000 to check your application

The home page looks like this:

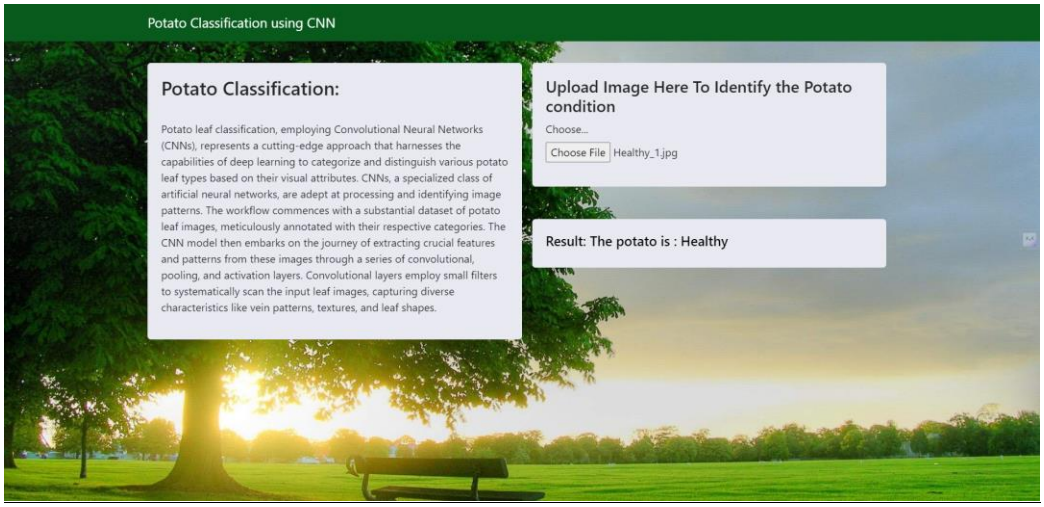


Click on choose the image to upload and select an image to be classified.

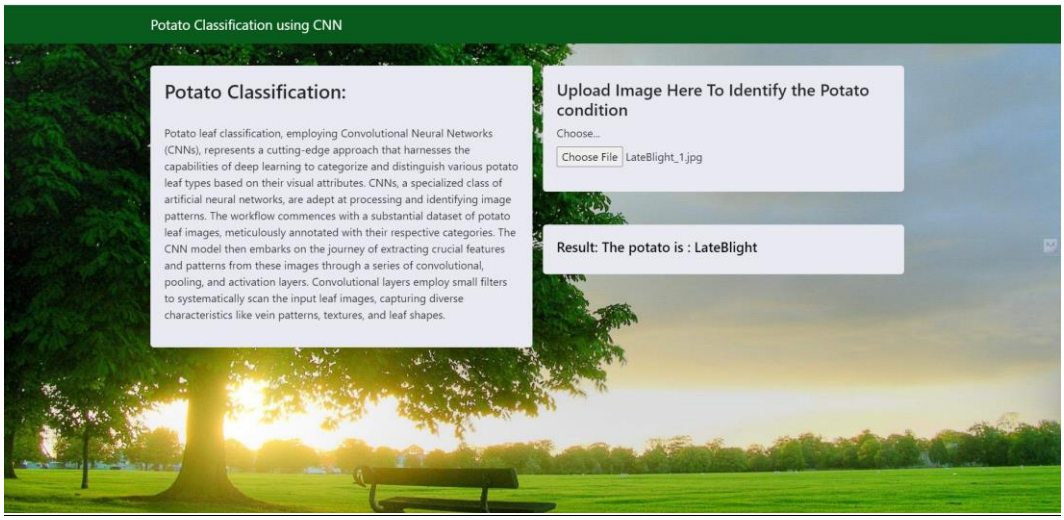
Input-1:



Input-2:



Input-3:



Conclusion: By clicking the predict button we will predict the condition of using our ResNet50 model.