

# Table of Contents

<b>List of Figures.....</b>	<b>1</b>
<b>1. Problem statement.....</b>	<b>2</b>
<b>1.1 Aim .....</b>	<b>2</b>
<b>1.2 Objectives.....</b>	<b>2</b>
<b>1.3 Solution .....</b>	<b>2</b>
<b>1.4 Dataset description.....</b>	<b>4</b>
<b>1.5 Feature extraction.....</b>	<b>5</b>
<b>1.6 Dataset division .....</b>	<b>7</b>
<b>1.7 Extreme learning machine.....</b>	<b>7</b>
<b>2 Experimental setup .....</b>	<b>8</b>
<b>2.1 Performance metrics.....</b>	<b>9</b>
<b>2.1.1 Accuracy.....</b>	<b>9</b>
<b>2.1.2 Precision.....</b>	<b>9</b>
<b>2.1.3 Recall.....</b>	<b>9</b>
<b>2.1.4 F1 score.....</b>	<b>9</b>
<b>2.2 Results and Analysis.....</b>	<b>10</b>
<b>2.3 Discussion and Recommendations.....</b>	<b>11</b>

## List of Figures

Fig .1	Startup environment of kaggle .....	2
Fig .2	Importing dataset to the kernel .....	3
Fig .3	Notebook editor with initial configurations.....	3
Fig .4	Necessary imported libraries for the underlying problem.....	4
Fig .5	Visualization of Plant disease image dataset.....	5
Fig .6	Declaring and analyzing ResNet-50 architecture.....	5
Fig .7	ResNet-50 architecture parameters and output feature vector .....	6
Fig .8	Feature extraction using ResNet-50 model.....	6
Fig .9	Data partition using sklearn library .....	7
Fig .10	Determining input to hidden layer, number of neurons and random weights.....	8
Fig .11	Dot product between inputs and its corresponding weights.....	8
Fig .12	Dot product between output of hidden layer and its corresponding weights.....	8
Fig .13	Time taken for training the model and the output weights shape.....	8
Fig .14	Classification results for the overall system performance.....	10
Fig .15	Classification results for the number of neurons set to 50 .....	10
Fig .16	Confusion matrix of the system for its highest performance .....	12

## 1 Problem statement

Train the extreme learning machines ELM classifier with deep features extracted from ResNet-50 on PlantVillage dataset to classify the different plant diseases.

### 1.1 Aim:

To classify the plant disease with extreme learning machine ELM classifier using ResNet-50 features

### 1.2 Objectives :

- a) To process the plant village dataset and identify the different categories
- b) To extract features from the images using ResNet-50 pre-trained model
- c) To develop an architecture of extreme learning machines ELM that will process and classify the input image data
- d) To validate the model using hold out technique and determine system performance using different evaluation metrics

### 1.3 Solution :

The dataset for the underlying problem was acquired from the kaggle website an online platform (<https://www.kaggle.com/emmarex/plantdisease>) for machine learning and data science. Kaggle provides python jupyter notebook environment and also the user access to its resources such as CPU, GPU and TPU for the data simulations. Since the current problem requires the implementation of deep neural networks which requires high computational resources I choose the kaggle platform for the code simulation. The notebook file/kernel is created by selecting “New Notebook” as shown in the Fig .1. Once the kernel is loaded we can load the data by choosing “+Add data” and search for the desired dataset (i-e plant disease) as shown in the Fig .2

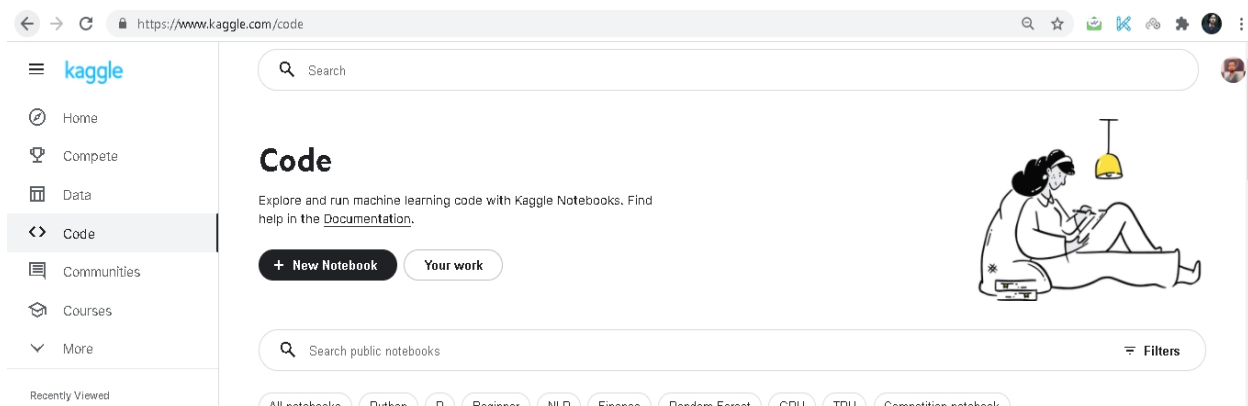


Fig. 1 Startup environment of kaggle

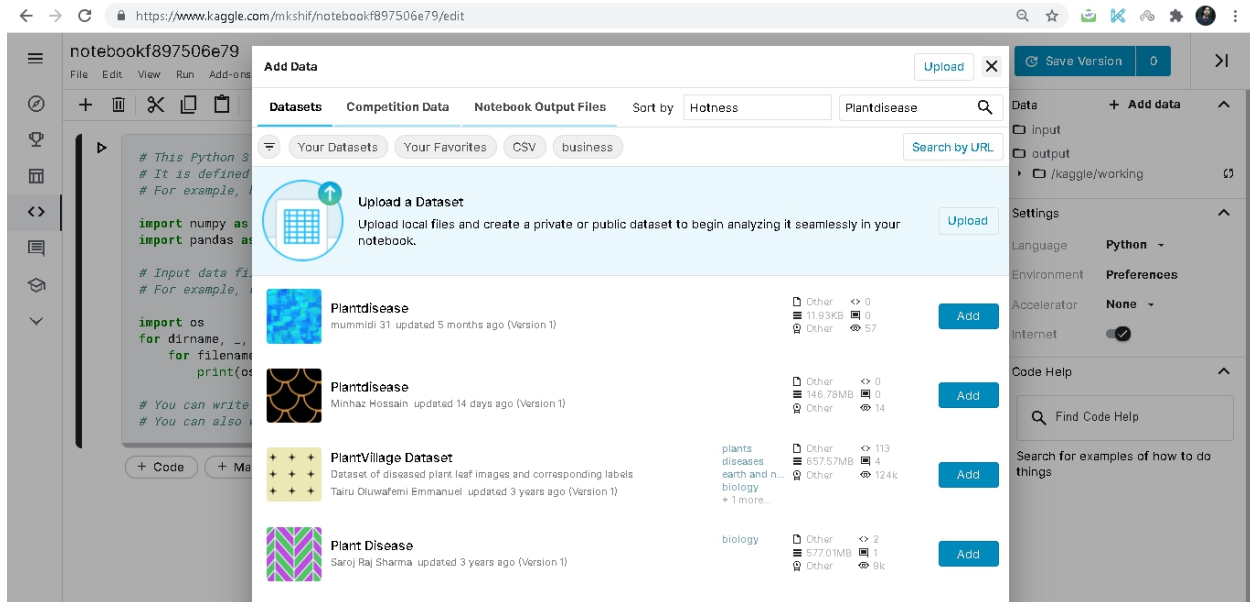


Fig .2 Importing dataset to the kernel

and also adding it to our kernel. Once the data is loaded it appears at the top right window under the input directory. The data is loaded and we can now load it to our kernel by specifying input path as shown in the Fig .3 and also assign name to the kernel. After that the next step is to

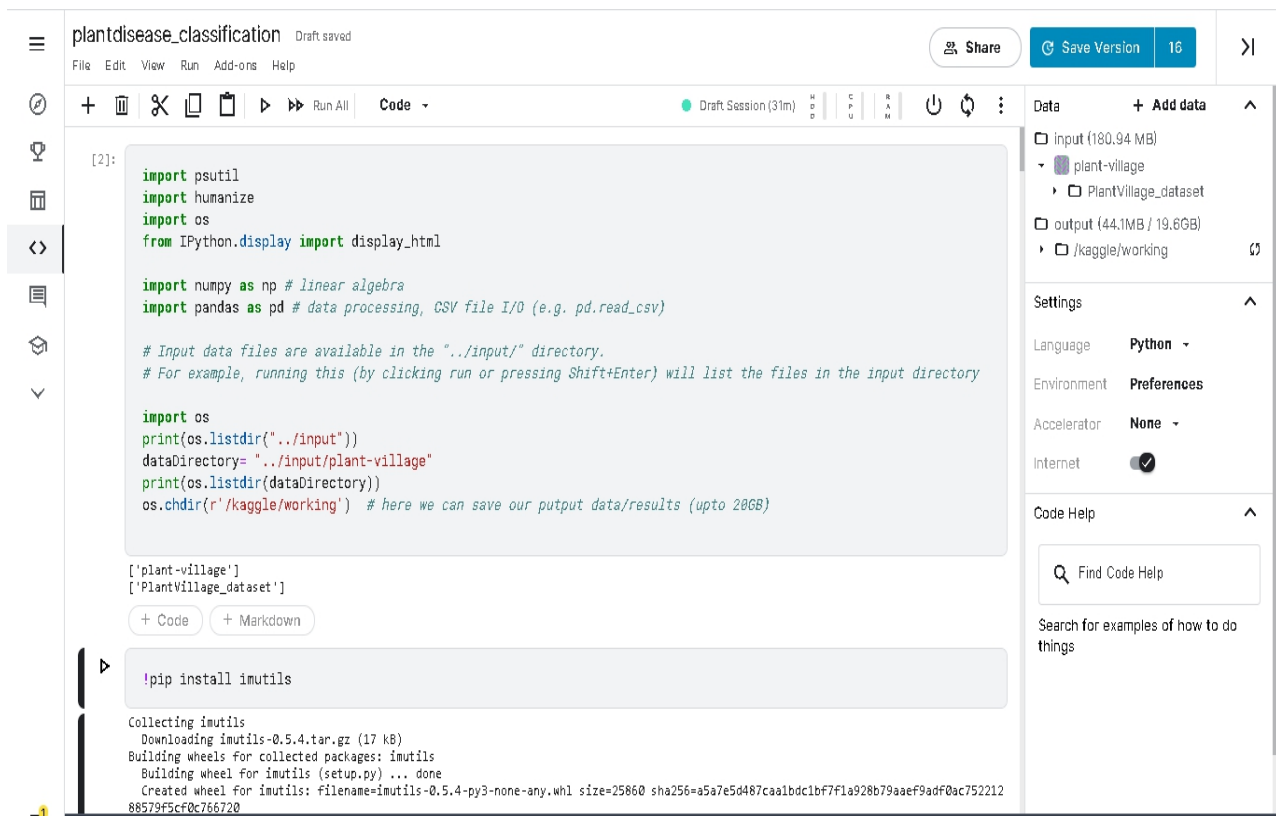


Fig .3 Notebook editor with initial configurations

import all the necessary libraries as shown in the Fig .4. The numpy array is used for mathematical manipulation of arrays. Keras were imported to obtain ResNet-50 pre-trained model and also to manipulate the deep neural network architecture. Keras uses tensor flow at the backend and provides high level API to the users to speed up the experimentation and ease the usage and implementation of deep networks. Furthermore, OpenCV library were used for image read and write. The scikit-learn libraries were imported for performance metrics and label categorizations.

```

import numpy as np
import keras
import cv2
from time import time
from keras import backend as K
from keras.models import Sequential
from keras.models import Model
from keras.layers import Activation
from keras.layers.core import Flatten
from keras.preprocessing.image import ImageDataGenerator
from keras.layers import Flatten
from keras.applications.resnet50 import ResNet50
from keras.preprocessing import image
from keras.applications.resnet50 import preprocess_input, decode_predictions
from sklearn.utils import class_weight
from tensorflow.keras.utils import to_categorical
from sklearn.preprocessing import LabelEncoder
from sklearn.preprocessing import LabelBinarizer
from sklearn.model_selection import train_test_split
from sklearn.metrics import confusion_matrix
from sklearn.metrics import average_precision_score
from sklearn.metrics import recall_score
from sklearn.metrics import precision_score
from sklearn.metrics import accuracy_score
from sklearn.metrics import classification_report
from imutils import paths
import itertools
import matplotlib.pyplot as plt
import time
import pandas as pd
%matplotlib inline

```

Fig .4 Necessary imported libraries for the underlying problem

## 1.4 Dataset description:

The dataset consists of 15 plant categories with each category having 3-channel images of resolution 256x256. These images were categorized based upon the specific plant disease. There are also some healthy plant images classes. Some of the dataset images were visualized using keras image generator library as shown in the Fig .4. The title of the images are the labels of each category which were extracted from their respective folder names.

## 1.5 Feature extraction:

According to the problem requirements, the feature extraction from the dataset was carried out using ResNet-50 model. It is a 50 layer deep convolutional neural network. Further it is trained

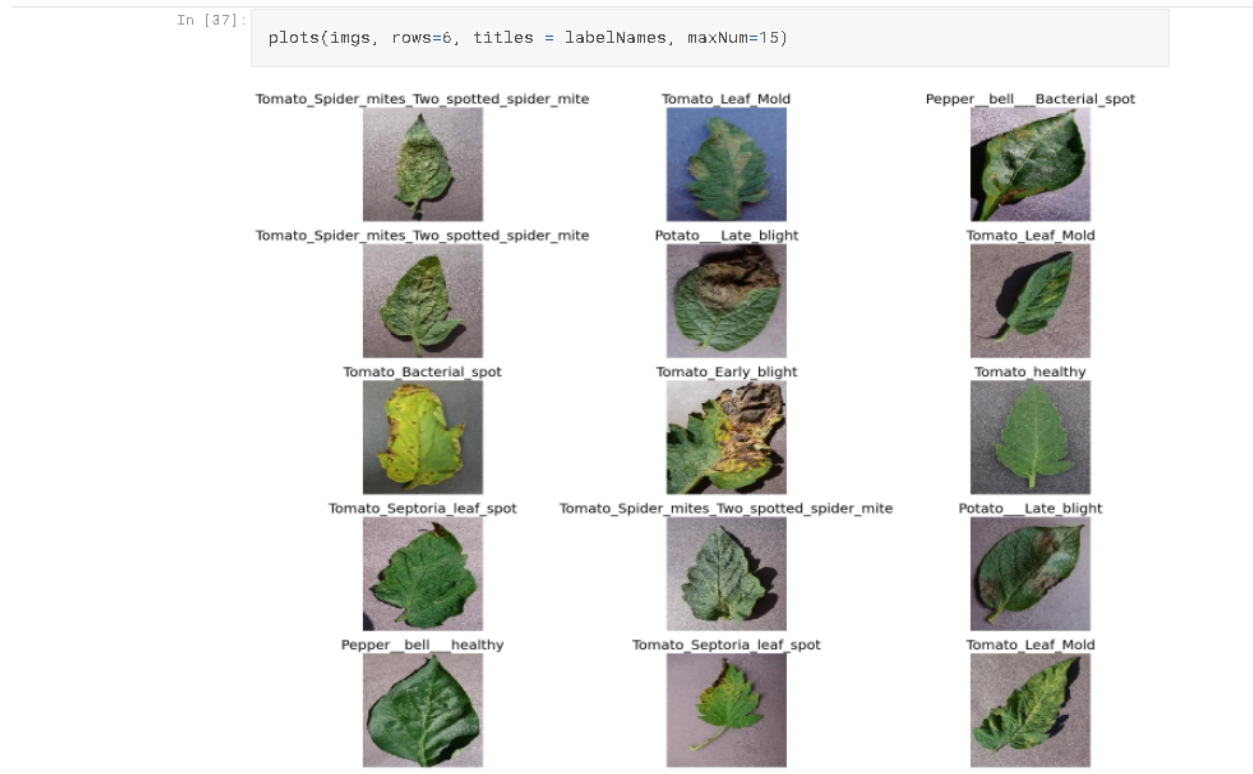


Fig 5. Visualization of Plant disease image dataset

on the million of images from the ImageNet database which consist of only single fully connected layer and can classify 1000 image categories . The top layer was removed from the

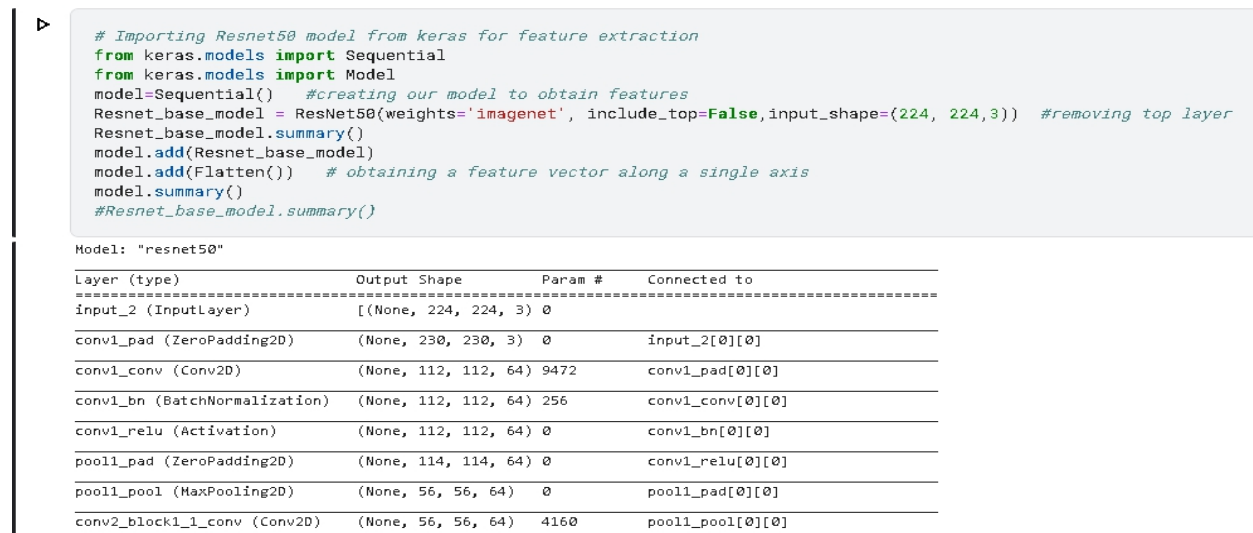


Fig .6 Declaring and analyzing ResNet-50 architecture

conv5_block3_2_relu (Activation) (None, 7, 7, 512)	0	conv5_block3_2_relu[0][0]
conv5_block3_3_conv (Conv2D) (None, 7, 7, 2048)	1050624	conv5_block3_2_relu[0][0]
conv5_block3_3_bn (BatchNormali (None, 7, 7, 2048)	8192	conv5_block3_3_conv[0][0]
conv5_block3_add (Add) (None, 7, 7, 2048)	0	conv5_block2_out[0][0] conv5_block3_3_bn[0][0]
conv5_block3_out (Activation) (None, 7, 7, 2048)	0	conv5_block3_add[0][0]
=====		
Total params: 23,587,712		
Trainable params: 23,534,592		
Non-trainable params: 53,120		
Model: "sequential_2"		
Layer (type)	Output Shape	Param #
=====		
resnet50 (Functional)	(None, 7, 7, 2048)	23587712
flatten (Flatten)	(None, 100352)	0
=====		
Total params: 23,587,712		
Trainable params: 23,534,592		
Non-trainable params: 53,120		

Fig.7 ResNet-50 architecture parameters and output feature vector

model and the convolutional layers were left behind which extracts both low and high level features from the images. The model architectures and parameters are shown in the Fig .6 and Fig .7. The sequential model was defined and a flatten layer was added which maps the multichannel data to a single 1D array which can be used as an input to another layer. It provides better insight to analyze the different arrays. Hence, the output of ResNet-50 when converted to 1-D array is (2048x7x7=100352) 100352 dimensional feature vector. All the feature vectors

```
[42]: feature_matrix=[] # declaring an empty array for feature storage
      labels = [] # declaring labels array

      # loop over the image paths
      for imagePath in imagePaths:
          # extract the class label from the filename
          label = imagePath.split(os.path.sep)[-2] # obtain a label from the class folder
          # load the image, swap color channels, and resize it to be a fixed
          # 224x224 pixels while ignoring aspect ratio
          image = cv2.imread(imagePath)
          image = cv2.cvtColor(image, cv2.COLOR_BGR2RGB)
          image = cv2.resize(image, (224, 224))
          image = image.reshape(1,224,224,3) # update the data and labels lists, respectively
          image = preprocess_input(image)
          features = model.predict(image)
          feature_matrix.append(features)
          labels.append(label)


      # converting the data and labels to NumPy arrays while scaling the pixel
      # intensities to the range [0, 1] i-e normalization
      feature_matrix=np.array(feature_matrix)/255
      labels=np.array(labels)
```

Fig .8 Feature extraction using ResNet-50 model

were stored in a single feature matrix as shown in the Fig .8. The input images were resized to 224x224x3 because its the standard input of this model.

## 1.6 Dataset division

The data were divided randomly between train (80%) and test set (20%) using sklearn library as shown in the Fig .9. Although the conventional practice involves validation set as well which



```
#Obtaining training and testing data using holdout

(trainX, testX, trainY, testY) = train_test_split(feature_matrix, labels,
        test_size=0.20, stratify=labels, random_state=0)
print('Train size: {train}, Test size: {test}'.format(train=trainX.shape[0], test=testX.shape[0]))
```

Train size: 2662, Test size: 666

+ Code + Markdown

```
trainX=trainX[:,0,] #get the desired shape (number of samples, size of feature vector) (i-e 2662,100352)
print(trainX.shape)
testX=testX[:,0,] #get the desired shape (number of samples, size of feature vector) (i-e 666,100352)
print(testX.shape)
```

(2662, 100352)  
(666, 100352)

Fig .9 Data partition using sklearn library

helps in determining the optimum hyperparameters and prevents overfitting, but the extreme learning machines (ELM) doesn't have much hyperparameter to be tuned and is less prone to overfitting (as far as the number of neuron are not too many) as compared to the other deep learning frameworks which uses multiple layers and repeatedly iterate to determine weights between layers. As a result they can easily overfit if the parameters are not tuned appropriately. Hence, we did not use validation set for the underlying model.

## 1.7 Extreme learning machine:

Extreme learning machine is a feed forward learning algorithm that commonly uses single hidden layer and output layer. In contrast to the other deep learning frame works, the bias and weights between input and hidden layer is initialized randomly and the output weights are calculated by using least square method between hidden layer and output layer. The whole process can be summarized by analyzing the Fig .9, 10, 11 and Fig .12 with its displayed output.



```
In [149]: INPUT_LENGTH = 100352      #FEATURE VECTOR DIMENSION
          HIDDEN_UNITS = 750        #NUMBER OF NEURONS

          Win = np.random.normal(size=[INPUT_LENGTH, HIDDEN_UNITS])      #RANDOM INITIALIZATION OF WEIGHTS
          print('Input Weight shape: {shape}'.format(shape=Win.shape))

          Input Weight shape: (100352, 750)
```

Fig .10 Determining input to hidden layer, number of neurons and random weights

```
[48]: def input_to_hidden(x):      # defining function for the dot product b/w input and its weights
          a = np.dot(x, Win)
          a = np.maximum(a, 0, a) # Implementation of ReLU activation function
          return a

          Input Weight shape: (100352, 750)
```

Fig 11. Dot product between inputs and its corresponding weights

```
def predict(x):      #defining function for prediction
    x = input_to_hidden(x)
    y = np.dot(x, Wout)
    return y
```

Fig .12 Dot product between output of hidden layer and its corresponding weights

```
In [150]: start = time.time()
          X = input_to_hidden(trainX)
          Xt = np.transpose(X)
          Wout = np.dot(np.linalg.inv(np.dot(Xt, X)), np.dot(Xt, trainY))
          print('Output weights shape: {shape}'.format(shape=Wout.shape))
          print('Total Training Time is: %s seconds ' % (time.time()-start))

          Output weights shape: (750, 15)
          Total Training Time is: 8.844124794006348 seconds
```

Fig .13 Time taken for training the model and the output weights shape

## 2 Experimental setup

The experimentation process was initiated using 2662 images from all the 15 categories for training the model and 666 images were used for testing. The only tuning parameter used in this experiment was number of neurons which was initially set to 50 and number of hidden layers was fixed to 1. Since our data is normalized between 0 and 1, the Rectified Linear Units (ReLU) function was used as an activation function for the implemented architecture. Its output is zero when its receive an input less than or equal to zero and its returns that value back for positive input values.

**2.1 Performance metrics:** The system performance was evaluated using the following defined classification metrics:

**2.1.1 Accuracy:** It is the number of correct predictions made divided by the total number of predictions made. It is expressed as:

$$\text{Accuracy} = \frac{TP + TN}{TP + TN + FP + FN} \quad (1)$$

where TP , is the True Positive values,

FP is the False positive values,

TN, is True negative

and FN is False negative

**2.1.2 Precision:** It indicates the proportion of positive predictions that was actually correct. Precision is given by :

$$\text{Precision} = \frac{TP}{TP + FP} \quad (2)$$

The model achieves 100 percent precision if FP is zero

**2.1.3 Recall:** Recall identifies the proportions of actual positives that were correct. It is given by:

$$\text{Recall} = \frac{TP}{TP + FN} \quad (3)$$

**2.1.4 F1 score:** It is the harmonic mean of precision and recall and penalizes the extreme values. It is computed as follow:

$$F1 = 2 \times \frac{\text{Recall} \times \text{Precision}}{\text{Recall} + \text{Precision}} \quad (4)$$

## 2.2 Results and Analysis

The classification results are shown in the Fig .14 which shows the highest values of system performance achieved in-terms of all the used evaluation parameters. The number of neurons

Accuracy: 0.791291

```
In [161]: print(classification_report(gr_labs, pred_labs, target_names=Classes_names))
```

	precision	recall	f1-score	support
Pepper_Bacterial	0.93	0.90	0.92	42
Potato_healthy	0.93	0.95	0.94	41
Tomato_Leaf_Mold	0.96	0.94	0.95	53
Tomato_YellowLeaf_Virus	0.75	0.77	0.76	53
Tomato_Bacterial_spot	0.80	0.93	0.86	30
Tomato_Septoria_spot	0.82	0.92	0.87	66
Tomato_healthy	0.50	0.40	0.44	40
Tomato_Two_spotted_spider_mite	0.69	0.69	0.69	39
Tomato_Early_blight	0.82	0.73	0.77	37
Tomato_Target_Spot	0.70	0.53	0.60	36
Pepper_healthy	0.67	0.78	0.72	45
Potato_Late_blight	0.55	0.57	0.56	51
Tomato_Late_blight	0.97	0.80	0.88	35
Potato_Early_blight	0.89	0.98	0.93	57
Tomato_mosaic_virus	0.85	0.80	0.83	41
accuracy			0.79	666
macro avg	0.79	0.78	0.78	666
weighted avg	0.79	0.79	0.79	666

Fig .14 Classification results for the overall system performance

used for that particular experiment was set to 1000. Initially, the neurons were set to 50 and its results are shown in the Fig .14. It can be seen that the system performance is very low.

Accuracy: 0.460961

```
In [29]: #print(test)
print(predicted)
```

12

```
In [39]: print(classification_report(gr_labs, pred_labs, target_names=Classes_names))
```

	precision	recall	f1-score	support
Pepperbell_Bacterial_spot	0.61	0.33	0.43	4
Potato_healthy	0.51	0.61	0.56	4
Tomato_Leaf_Mold	0.44	0.75	0.56	5
Tomato_Tomato_YellowLeaf_Curl_Virus	0.42	0.60	0.49	5
Tomato_Bacterial_spot	0.60	0.60	0.60	3
Tomato_Septoria_leaf_spot	0.40	0.58	0.48	6

Fig .15 Classification results for the number of neurons set to 50

Hence, the performance was improved by increasing the number of neurons with constant step size. Its results are tabulated in the Table .1. It can be seen that system performance increases with constantly increasing the number of neurons. The number of neurons were kept on increasing until a point is reached where there is no further improvement seen or the performance starts reducing. The system performs better in terms of training time for 200 neurons whereas highest accuracy was achieved at 750 neurons. It can be seen in the Fig. 13 classification report that overall, the system performed better at 1000 neurons for which highest values of precision, recall and F1 score was achieved. Its confusion matrix is also shown in the Fig .15. We can observe that not even a single class is under represented. It occurs when some times the classes are highly imbalanced and in that case the precision and recall values are very low. Hence, F1 score was used to balance class distribution if there is any imbalanced distribution.

Table .1 Results of Extreme learning classifier

Number of Neurons	Training time (s)	Test Accuracy	Precision (macro avg)	Recall (macro avg)	F1 score (macro avg)
50	-	0.46	0.49	0.44	0.42
100	-	0.56	0.58	0.54	0.52
150	-	0.65	0.66	0.64	0.63
200	246.976	0.70	0.69	0.68	0.67
250	<b>3.397</b>	0.72	0.73	0.70	0.70
300	3.833	0.74	0.74	0.73	0.73
350	4.823	0.76	0.76	0.74	0.74
500	5.189	0.77	0.76	0.75	0.75
750	8.844	<b>0.80</b>	0.76	0.75	0.75
1000	11.630	0.79	<b>0.79</b>	<b>0.78</b>	<b>0.78</b>
1500	16.453	0.76	0.76	0.75	0.75
2000	20.873	0.66	0.68	0.66	0.66

**Note:** The bold value indicates the highest performance achieved in terms of its specified parameter

## 2.3 Discussion and Recommendations

ELM is considered faster int-terms of computational time taken for training as compared to other Deeplearning networks such as Convolutional neural networks and recurrent neural networks. Initially it was developed with a single hidden layer and as the research was carried out more variant were introuduced which used more than one hidden layer. We have seen that it took very less time i-e few seconds to train the model whereas the deep networks requires hours and even days for very large data to train the model. One of the possible reasons that ELM took less time

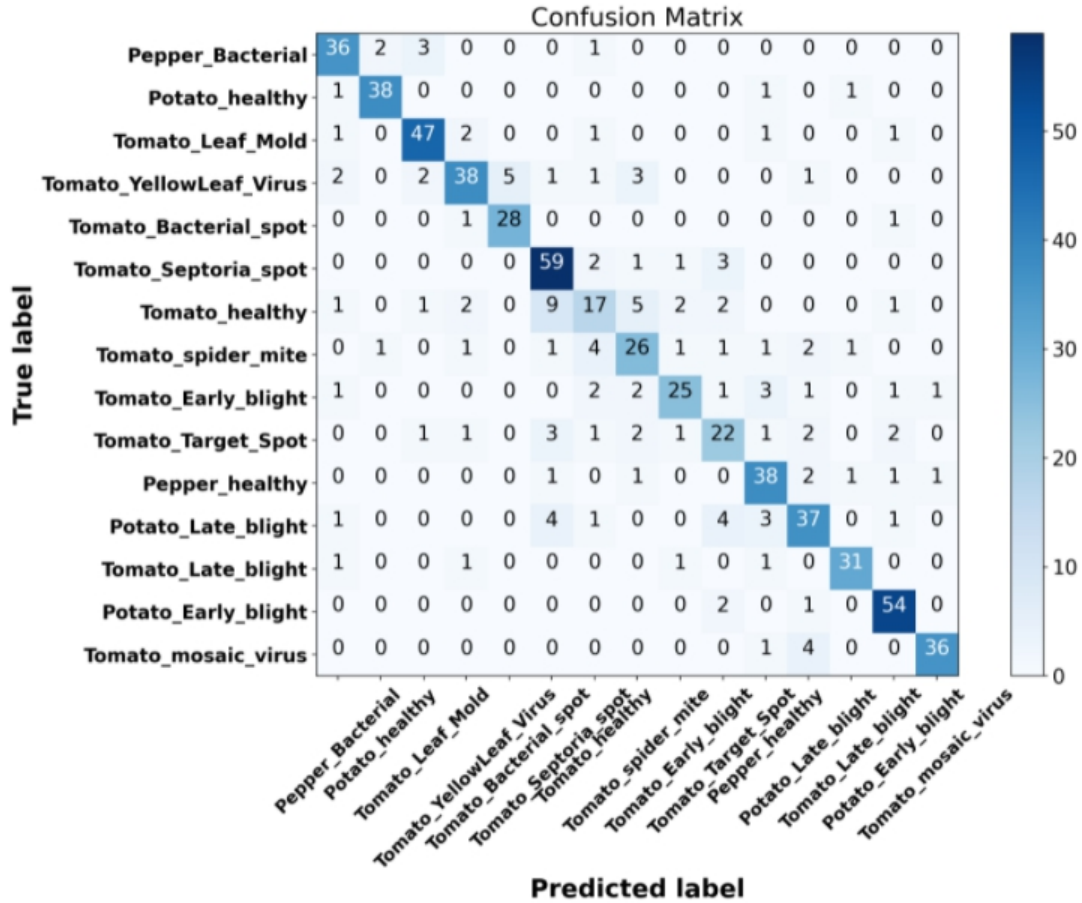


Fig .16 Confusion matrix of the system for its highest performance

for training was due to its single hidden layer architecture as well as its random initialization of weights. Hence it can be utilized in the time critical applications. Furthermore the ELM model accuracy improves at the expense of increased training time, so it can be also be utilized in the specific application areas where time is not of paramount importance. We have also seen that by increasing the number of neurons from a certain threshold the model overfits and its generalization capability is reduced. Similarly using of too few neurons resulted in an underfitting. However, based on the above results we can conclude that ELM has shown some promising results for the classification of plant diseases and in the future it can be used with other pre-trained models with multiple hidden layers to analyze its performance. Further it can also be used as a stand alone frame work (i-e without utilizing any pre-trained model for feature extraction) with multiple hidden layers and then evaluating its performance on the same dataset.