

✓ Introduction to Computer Vision: Plant Seedlings Classification

✓ Problem Statement

✓ Context

In recent times, the field of agriculture has been in urgent need of modernizing, since the amount of manual work people need to put in to check if plants are growing correctly is still highly extensive. Despite several advances in agricultural technology, people working in the agricultural industry still need to have the ability to sort and recognize different plants and weeds, which takes a lot of time and effort in the long term. The potential is ripe for this trillion-dollar industry to be greatly impacted by technological innovations that cut down on the requirement for manual labor, and this is where Artificial Intelligence can actually benefit the workers in this field, as **the time and energy required to identify plant seedlings will be greatly shortened by the use of AI and Deep Learning**. The ability to do so far more efficiently and even more effectively than experienced manual labor, could lead to better crop yields, the freeing up of human involvement for higher-order agricultural decision making, and in the long term will result in more sustainable environmental practices in agriculture as well.

✓ Objective

The aim of this project is to Build a Convolutional Neural Netowrk to classify plant seedlings into their respective categories.

✓ Data Dictionary

The Aarhus University Signal Processing group, in collaboration with the University of Southern Denmark, has recently released a dataset containing **images of unique plants belonging to 12 different species**.

- The dataset can be download from Olympus.
- The data file names are:
 - images.npy
 - Labels.csv
- Due to the large volume of data, the images were converted to the images.npy file and the labels are also put into Labels.csv, so that you can work on the data/project seamlessly without having to worry about the high data volume.
- The goal of the project is to create a classifier capable of determining a plant's species from an image.

List of Species

- Black-grass
- Charlock
- Cleavers
- Common Chickweed
- Common Wheat
- Fat Hen
- Loose Silky-bent
- Maize
- Scentless Mayweed
- Shepherds Purse
- Small-flowered Cranesbill
- Sugar beet



Note: Please use GPU runtime on Google Colab to execute the code faster.

Please read the instructions carefully before starting the project.

This is a commented Python Notebook file in which all the instructions and tasks to be performed are mentioned.

- Blanks '___' are provided in the notebook that need to be filled with an appropriate code to get the correct result
- With every '___' blank, there is a comment that briefly describes what needs to be filled in the blank space
- Identify the task to be performed correctly and only then proceed to write the required code
- Fill the code wherever asked by the commented lines like "# write your code here" or "# complete the code"
- Running incomplete code may throw an error
- Please run the codes in a sequential manner from the beginning to avoid any unnecessary errors
- Add the results/observations derived from the analysis in the presentation and submit the same in .pdf format

✓ Importing necessary libraries

```
# Installing the libraries with the specified version.
# uncomment and run the following line if Google Colab is being used
!pip install tensorflow==2.15.0 scikit-learn==1.2.2 seaborn==0.13.1 matplotlib==3.7.1 numpy==1.25.2 pandas==2.0.3 opencv-python==4.8.0.76 -i

# Installing the libraries with the specified version.
# uncomment and run the following lines if Jupyter Notebook is being used
#!pip install tensorflow==2.13.0 scikit-learn==1.2.2 seaborn==0.11.1 matplotlib==3.3.4 numpy==1.24.3 pandas==1.5.2 opencv-python==4.8.0.76
```

Note: After running the above cell, kindly restart the notebook kernel and run all cells sequentially from the start again.

```
import os
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
import math
import cv2
import seaborn as sns

# Importing numpy for Matrix Operations
# Importing pandas to read CSV files
# Importing matplotlib for Plotting and v
# Importing math module to perform mathema
# Importing openCV for image processing
# Importing seaborn to plot graphs

# Tensorflow modules
import tensorflow as tf
from tensorflow.keras.preprocessing.image import ImageDataGenerator
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense,Dropout,Flatten,Conv2D,MaxPooling2D,BatchNormalization
from tensorflow.keras.optimizers import Adam,SGD
from sklearn import preprocessing
from sklearn.model_selection import train_test_split
from sklearn.metrics import confusion_matrix
from sklearn.preprocessing import LabelBinarizer
# Display images using OpenCV
from google.colab.patches import cv2_imshow
from sklearn.model_selection import train_test_split
from tensorflow.keras import backend
from keras.callbacks import ReduceLROnPlateau
import random
# Ignore warnings
import warnings
warnings.filterwarnings('ignore')

# Importing the ImageDataGenerator for data
# Importing the sequential module to defin
# Defining all the layers to build our CNN
# Importing the optimizers which can be us
# Importing the preprocessing module to pr
# Importing train_test_split function to s
# Importing confusion_matrix to plot the c

# Importing cv2_imshow from google.patches
```

✓ Loading the dataset

```
# Uncomment and run the below code if you are using google colab
# from google.colab import drive
# drive.mount('/content/drive')
```



```
# Load the image file of dataset
images = np.load('/content/images.npy') # Complete the code to read the dataset

# Print the shape of the loaded array to verify its dimensions
print("Shape of loaded images array:", images.shape)

# Attempt to reshape the array if the number of elements match the expected shape
expected_elements = 4750 * 128 * 128 * 3
if images.size == expected_elements:
    images = images.reshape(4750, 128, 128, 3)
    print("Reshaped images array:", images.shape)
else:
    print("The number of elements in the loaded array does not match the expected shape.")
```

```
# Load the labels file of dataset
labels = pd.read_csv('/content/Labels.csv') # Complete the code to read the dataset
```

```
↳ Shape of loaded images array: (4750, 128, 128, 3)
   Reshaped images array: (4750, 128, 128, 3)
```

```
# # Load the image file of dataset
# images = np.load('/content/images.npy') # Complete the code to read the dataset

# # Print the shape of the loaded array to verify its dimensions
# print("Shape of loaded images array:", images.shape)

# # Load the labels file of dataset
# labels = pd.read_csv('/content/Labels.csv') # Complete the code to read the dataset
```

▼ Data Overview

▼ Understand the shape of the dataset

```
print(images.shape) # Complete the code to check the shape
print(labels.shape) # Complete the code to check the shape
```

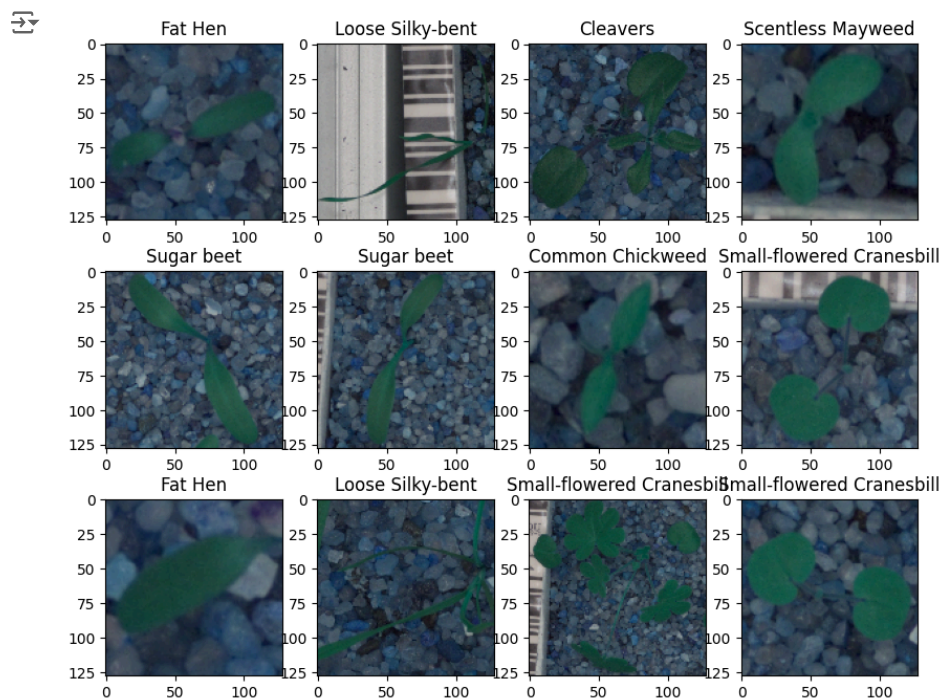
```
↳ (4750, 128, 128, 3)
   (4750, 1)
```

▼ Exploratory Data Analysis

▼ Plotting random images from each of the class

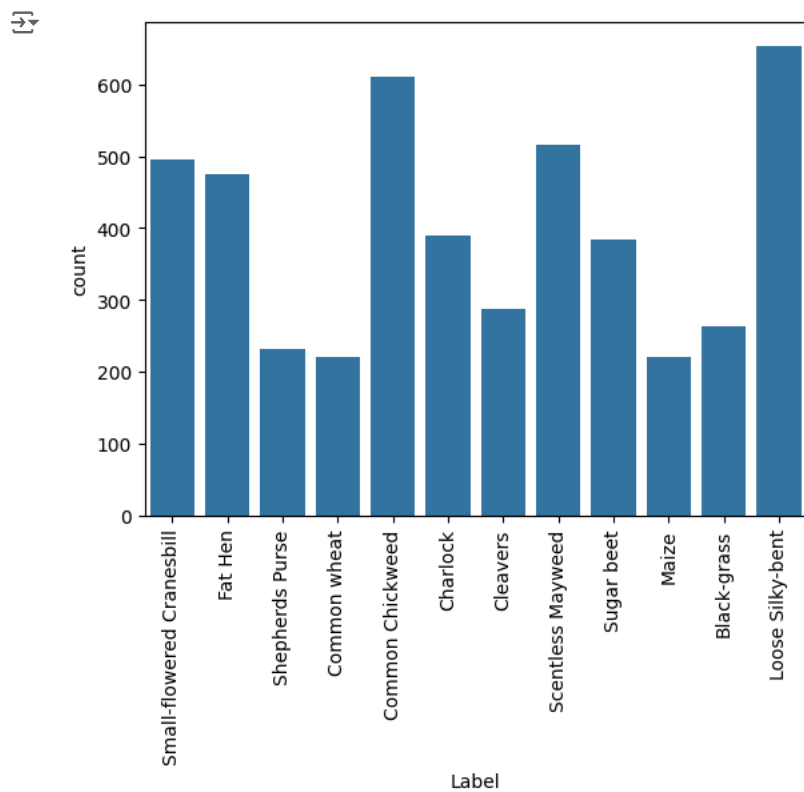
```
def plot_images(images, labels):
    num_classes=10 # Number of Classes
    categories=np.unique(labels) # Obtaining the unique classes from y_train
    keys=dict(labels['Label']) # Defining number of rows=3
    rows = 3 # Defining number of columns=4
    cols = 4 # Defining the figure size to 10x8
    fig = plt.figure(figsize=(10, 8))
    for i in range(cols):
        for j in range(rows):
            random_index = np.random.randint(0, len(labels)) # Generating random indices from the data and plotting the
            ax = fig.add_subplot(rows, cols, i * rows + j + 1) # Adding subplots with 3 rows and 4 columns
            ax.imshow(images[random_index, :]) # Plotting the image
            ax.set_title(keys[random_index])
    plt.show()

plot_images(images, labels) # Complete the code to input the images and labels to the function and plot the images with their labels
```



✓ Checking the distribution of the target variable

```
sns.countplot(x=labels['Label']); # Complete the code to check for data imbalance
plt.xticks(rotation='vertical');
```



✓ Data Pre-Processing

✓ Converting the BGR images to RGB images.

```
import cv2 # Make sure you import the cv2 module

# Converting the images from BGR to RGB using cvtColor function of OpenCV
for i in range(len(images)):
    images[i] = cv2.cvtColor(images[i], cv2.COLOR_BGR2RGB) # Use cv2.COLOR_BGR2RGB to specify the conversion
```

✓ Resizing images

As the size of the images is large, it may be computationally expensive to train on these larger images; therefore, it is preferable to reduce the image size from 128 to 64.

```
images_decreased=[]
height = 64 # Complete the code to define the height as 64
width = 64 # Complete the code to define the width as 64
dimensions = (width, height)
for i in range(len(images)):
    images_decreased.append( cv2.resize(images[i], dimensions, interpolation=cv2.INTER_LINEAR))
```

Image before resizing

```
plt.imshow(images[3])
```

 <matplotlib.image.AxesImage at 0x7994282c6b60>

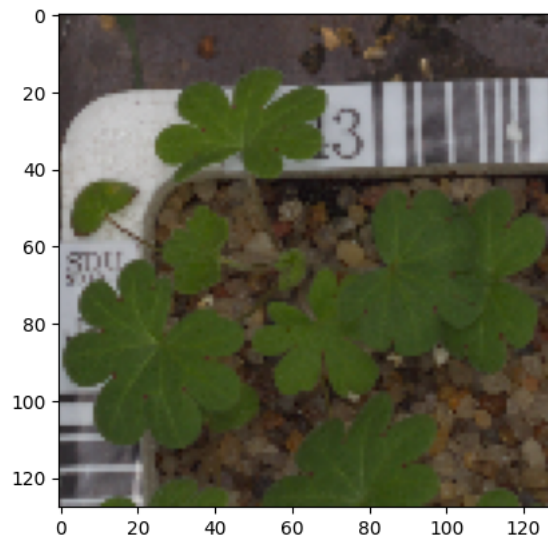

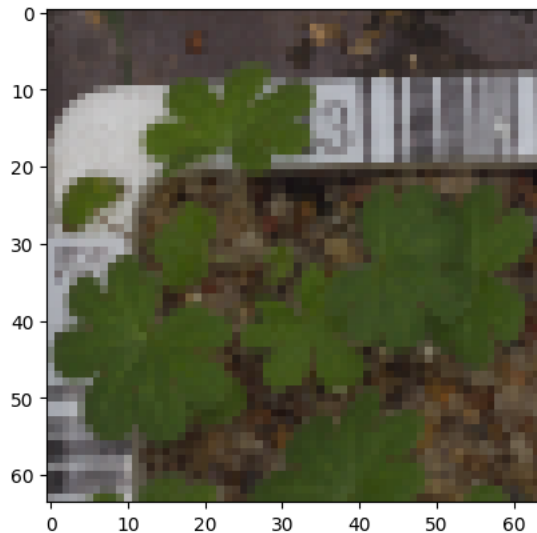


Image after resizing

```
plt.imshow(images_decreased[3])
```



 <matplotlib.image.AxesImage at 0x799428160040>




✓ Data Preparation for Modeling

- As we have less images in our dataset, we will only use 10% of our data for testing, 10% of our data for validation and 80% of our data for training.
- We are using the `train_test_split()` function from scikit-learn. Here, we split the dataset into three parts, train, test and validation.


```
X_temp, X_test, y_temp, y_test = train_test_split(np.array(images_decreased), labels, test_size=0.1, random_state=42, stratify=labels) # C
X_train, X_val, y_train, y_val = train_test_split(X_temp, y_temp, test_size=0.1, random_state=42, stratify=y_temp) # I
```

```
# Complete the code to check the shape of train, validation and test data
print(X_train.shape, y_train.shape)
print(X_val.shape, y_val.shape)
print(X_test.shape, y_test.shape)
```

 (3847, 64, 64, 3) (3847, 1)
(428, 64, 64, 3) (428, 1)
(475, 64, 64, 3) (475, 1)

✓ Encoding the target labels

```
print(y_train.head())
```

 Label
1032 Shepherd's Purse
3199 Scentless Mayweed
428 Small-flowered Cranesbill
3019 Scentless Mayweed
4081 Black-grass

```
## Convert labels from names to one hot vectors.
## We have already used encoding methods like onehotencoder and labelencoder earlier so now we will be using a new encoding method called
## Labelbinarizer works similar to onehotencoder
```

```
# enc = LabelBinarizer
# y_train_encoded = enc.fit_transform(y_train)
# y_val_encoded = enc.transform(y_val)
# y_test_encoded = enc.transform(y_test)
```

Complete the code to initialize the labelBinarizer
Complete the code to fit and transform y_train.
Complete the code to transform y_val
Complete the code to transform y_test

```

# # Convert labels from names to one hot vectors.
# # We have already used encoding methods like onehotencoder and labelencoder earlier so now we will be using a new encoding method called
# # Labelbinarizer works similar to onehotencoder
enc = LabelBinarizer() # Complete the code to initialize the labelBinarizer
y_train_labels = y_train['Label'].tolist() # Extract labels as a list
y_train_encoded = enc.fit_transform(y_train_labels) # Complete the code to fit and transform y_train.

y_val_labels = y_val['Label'].tolist()
y_val_encoded=enc.transform(y_val_labels) # Complete the code to transform y_val

y_test_labels = y_test['Label'].tolist()
y_test_encoded=enc.transform(y_test_labels) # Complete the code to transform y_test

y_train_encoded.shape,y_val.shape,y_test.shape # Complete the code to check the shape of train, validation and test data

```

→ ((3847, 12), (428, 1), (475, 1))

▼ Data Normalization

Since the **image pixel values range from 0-255**, our method of normalization here will be **scaling** - we shall **divide all the pixel values by 255 to standardize the images to have values between 0-1**.

```

# Complete the code to normalize the image pixels of train, test and validation data
X_train_normalized = X_train.astype('float32')/255.0 # Divide by 255.0 to normalize
X_val_normalized = X_val.astype('float32')/255.0 # Divide by 255.0 to normalize
X_test_normalized = X_test.astype('float32')/255.0 # Divide by 255.0 to normalize

```

▼ Model Building

```

# Clearing backend
backend.clear_session()

# Fixing the seed for random number generators
np.random.seed(42)
random.seed(42)
tf.random.set_seed(42)

```



```

# Initializing a sequential model
model1 = Sequential()

# Complete the code to add the first conv layer with 128 filters and kernel size 3x3 , padding 'same' provides the output size same as the
# Input_shape denotes input image dimension of images
model1.add(Conv2D(128, (3, 3), activation='relu', padding="same", input_shape=(64, 64, 3)))

# Complete the code to add the max pooling to reduce the size of output of first conv layer
model1.add(MaxPooling2D((2, 2), padding = 'same'))

# Complete the code to create two similar convolution and max-pooling layers activation = relu
model1.add(Conv2D(64, (3, 3), activation='relu', padding="same"))
model1.add(MaxPooling2D((2, 2), padding = 'same'))

model1.add(Conv2D(32, (3, 3), activation='relu', padding="same"))
model1.add(MaxPooling2D((2, 2), padding = 'same'))

# Complete the code to flatten the output of the conv layer after max pooling to make it ready for creating dense connections
model1.add(Flatten())

# Complete the code to add a fully connected dense layer with 16 neurons
model1.add(Dense(16, activation='relu'))
model1.add(Dropout(0.3))
# Complete the code to add the output layer with 12 neurons and activation functions as softmax since this is a multi-class classification
model1.add(Dense(12, activation='softmax'))

# Complete the code to use the Adam Optimizer
opt=Adam()
# Complete the code to Compile the model using suitable metric for loss fucntion
model1.compile(optimizer='Adam', loss='categorical_crossentropy', metrics=['accuracy'])

# Complete the code to generate the summary of the model
model1.summary()

```

Model: "sequential"

Layer (type)	Output Shape	Param #
conv2d (Conv2D)	(None, 64, 64, 128)	3584
max_pooling2d (MaxPooling2D)	(None, 32, 32, 128)	0
conv2d_1 (Conv2D)	(None, 32, 32, 64)	73792
max_pooling2d_1 (MaxPooling2D)	(None, 16, 16, 64)	0
conv2d_2 (Conv2D)	(None, 16, 16, 32)	18464
max_pooling2d_2 (MaxPooling2D)	(None, 8, 8, 32)	0
flatten (Flatten)	(None, 2048)	0
dense (Dense)	(None, 16)	32784
dropout (Dropout)	(None, 16)	0
dense_1 (Dense)	(None, 12)	204
Total params: 128828 (503.23 KB)		
Trainable params: 128828 (503.23 KB)		
Non-trainable params: 0 (0.00 Byte)		

Fitting the model on the train data

```

# Complete the code to fit the model on train and also using the validation data for validation
history_1 = model1.fit(
    X_train_normalized, y_train_encoded, # Provide training data and labels
    epochs=30,
    validation_data=(X_val_normalized,y_val_encoded),
    batch_size=32,
    verbose=2
)

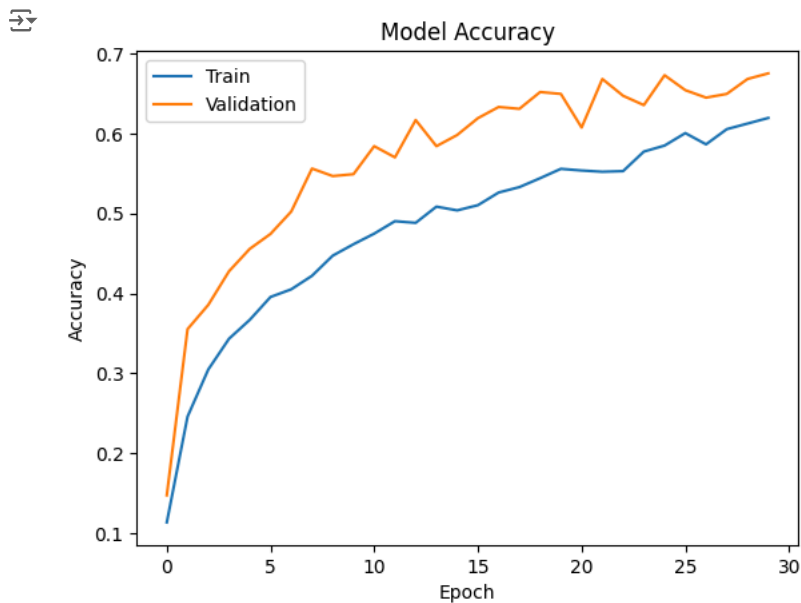
```




Epoch 1/30
121/121 - 8s - loss: 2.4510 - accuracy: 0.1133 - val_loss: 2.4269 - val_accuracy: 0.1472 - 8s/epoch - 66ms/step
Epoch 2/30
121/121 - 1s - loss: 2.2321 - accuracy: 0.2456 - val_loss: 2.0370 - val_accuracy: 0.3551 - 1s/epoch - 9ms/step
Epoch 3/30
121/121 - 1s - loss: 2.0065 - accuracy: 0.3047 - val_loss: 1.8032 - val_accuracy: 0.3855 - 1s/epoch - 9ms/step
Epoch 4/30
121/121 - 1s - loss: 1.8948 - accuracy: 0.3431 - val_loss: 1.7155 - val_accuracy: 0.4276 - 1s/epoch - 9ms/step
Epoch 5/30
121/121 - 1s - loss: 1.7861 - accuracy: 0.3668 - val_loss: 1.6026 - val_accuracy: 0.4556 - 1s/epoch - 9ms/step
Epoch 6/30
121/121 - 1s - loss: 1.7105 - accuracy: 0.3954 - val_loss: 1.5505 - val_accuracy: 0.4743 - 1s/epoch - 10ms/step
Epoch 7/30
121/121 - 1s - loss: 1.6601 - accuracy: 0.4050 - val_loss: 1.4233 - val_accuracy: 0.5023 - 1s/epoch - 11ms/step
Epoch 8/30
121/121 - 1s - loss: 1.5816 - accuracy: 0.4219 - val_loss: 1.3107 - val_accuracy: 0.5561 - 1s/epoch - 9ms/step
Epoch 9/30
121/121 - 1s - loss: 1.5099 - accuracy: 0.4471 - val_loss: 1.3148 - val_accuracy: 0.5467 - 1s/epoch - 9ms/step
Epoch 10/30
121/121 - 1s - loss: 1.4890 - accuracy: 0.4614 - val_loss: 1.3427 - val_accuracy: 0.5491 - 1s/epoch - 9ms/step
Epoch 11/30
121/121 - 1s - loss: 1.4323 - accuracy: 0.4747 - val_loss: 1.2270 - val_accuracy: 0.5841 - 1s/epoch - 9ms/step
Epoch 12/30
121/121 - 1s - loss: 1.3954 - accuracy: 0.4903 - val_loss: 1.2017 - val_accuracy: 0.5701 - 1s/epoch - 9ms/step
Epoch 13/30
121/121 - 1s - loss: 1.3668 - accuracy: 0.4882 - val_loss: 1.1784 - val_accuracy: 0.6168 - 1s/epoch - 9ms/step
Epoch 14/30
121/121 - 1s - loss: 1.3262 - accuracy: 0.5084 - val_loss: 1.2012 - val_accuracy: 0.5841 - 1s/epoch - 9ms/step
Epoch 15/30
121/121 - 1s - loss: 1.3356 - accuracy: 0.5038 - val_loss: 1.1716 - val_accuracy: 0.5981 - 1s/epoch - 9ms/step
Epoch 16/30
121/121 - 1s - loss: 1.3212 - accuracy: 0.5103 - val_loss: 1.1016 - val_accuracy: 0.6192 - 1s/epoch - 9ms/step
Epoch 17/30
121/121 - 2s - loss: 1.2556 - accuracy: 0.5261 - val_loss: 1.0717 - val_accuracy: 0.6332 - 2s/epoch - 14ms/step
Epoch 18/30
121/121 - 2s - loss: 1.2231 - accuracy: 0.5329 - val_loss: 1.0806 - val_accuracy: 0.6308 - 2s/epoch - 15ms/step
Epoch 19/30
121/121 - 1s - loss: 1.2201 - accuracy: 0.5441 - val_loss: 1.0484 - val_accuracy: 0.6519 - 1s/epoch - 9ms/step
Epoch 20/30
121/121 - 1s - loss: 1.1867 - accuracy: 0.5558 - val_loss: 1.0264 - val_accuracy: 0.6495 - 1s/epoch - 9ms/step
Epoch 21/30
121/121 - 1s - loss: 1.1689 - accuracy: 0.5537 - val_loss: 1.1322 - val_accuracy: 0.6075 - 1s/epoch - 9ms/step
Epoch 22/30
121/121 - 1s - loss: 1.1655 - accuracy: 0.5521 - val_loss: 1.0305 - val_accuracy: 0.6682 - 1s/epoch - 9ms/step
Epoch 23/30
121/121 - 1s - loss: 1.1455 - accuracy: 0.5529 - val_loss: 1.0433 - val_accuracy: 0.6472 - 1s/epoch - 9ms/step
Epoch 24/30
121/121 - 1s - loss: 1.1064 - accuracy: 0.5773 - val_loss: 1.0921 - val_accuracy: 0.6355 - 1s/epoch - 9ms/step
Epoch 25/30
121/121 - 1s - loss: 1.0871 - accuracy: 0.5849 - val_loss: 1.0257 - val_accuracy: 0.6729 - 1s/epoch - 9ms/step
Epoch 26/30
121/121 - 1s - loss: 1.0504 - accuracy: 0.6005 - val_loss: 1.0445 - val_accuracy: 0.6542 - 1s/epoch - 9ms/step
Epoch 27/30
121/121 - 1s - loss: 1.0756 - accuracy: 0.5864 - val_loss: 1.0366 - val_accuracy: 0.6449 - 1s/epoch - 10ms/step
Epoch 28/30
121/121 - 1s - loss: 1.0315 - accuracy: 0.6054 - val_loss: 1.0806 - val_accuracy: 0.6495 - 1s/epoch - 11ms/step
Epoch 29/30
121/121 - 1s - loss: 1.0391 - accuracy: 0.6124 - val_loss: 0.9923 - val_accuracy: 0.6682 - 1s/epoch - 10ms/step

Model Evaluation

```
plt.plot(history_1.history['accuracy'])
plt.plot(history_1.history['val_accuracy'])
plt.title('Model Accuracy')
plt.ylabel('Accuracy')
plt.xlabel('Epoch')
plt.legend(['Train', 'Validation'], loc='upper left')
plt.show()
```



Evaluate the model on test data

```
accuracy = model1.evaluate(X_test_normalized, y_test_encoded, verbose=2) # Complete the code to evaluate the model on test data
```

```
15/15 - 1s - loss: 1.0051 - accuracy: 0.6821 - 625ms/epoch - 42ms/step
```

Plotting the Confusion Matrix

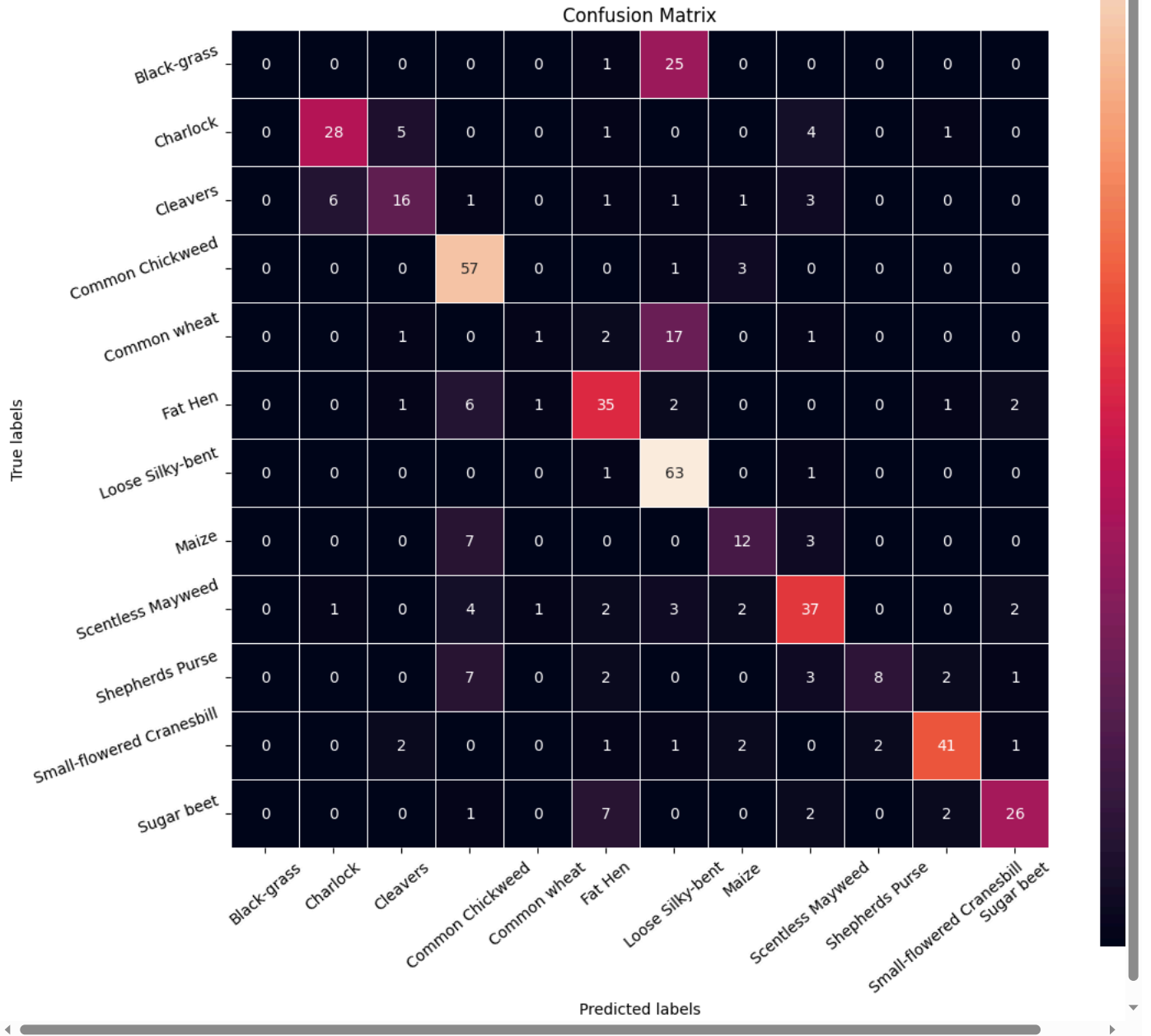
```
# Here we would get the output as probabilities for each category
y_pred=model1.predict(X_test_normalized) # Complete the code to predict the output probabilities
```

```
15/15 [=====] - 0s 6ms/step
```

```
# Obtaining the categorical values from y_test_encoded and y_pred
y_pred_arg=np.argmax(y_pred,axis=1)
y_test_arg=np.argmax(y_test_encoded,axis=1)
```

```
# Plotting the Confusion Matrix using confusion matrix() function which is also predefined in tensorflow module
confusion_matrix = tf.math.confusion_matrix(y_test_arg, y_pred_arg) # Complete the code to plot the confusion matrix
f, ax = plt.subplots(figsize=(12, 12))
sns.heatmap(
    confusion_matrix,
    annot=True,
    linewidths=.4,
    fmt="d",
    square=True,
    ax=ax
)
# Setting the labels to both the axes
ax.set_xlabel('Predicted labels');ax.set_ylabel('True labels');
ax.set_title('Confusion Matrix');
ax.xaxis.set_ticklabels(list(enc.classes_),rotation=40)
ax.yaxis.set_ticklabels(list(enc.classes_),rotation=20)
plt.show()
```





Plotting Classification Report

```
from sklearn import metrics
```

```
# Plotting the classification report
cr=metrics.classification_report(y_test_arg, y_pred_arg) # Complete the code to plot the classification report
print(cr)
```



	precision	recall	f1-score	support
0	0.00	0.00	0.00	26
1	0.80	0.72	0.76	39
2	0.64	0.55	0.59	29
3	0.69	0.93	0.79	61
4	0.33	0.05	0.08	22
5	0.66	0.73	0.69	48
6	0.56	0.97	0.71	65
7	0.60	0.55	0.57	22
8	0.69	0.71	0.70	52
9	0.80	0.35	0.48	23
10	0.87	0.82	0.85	50
11	0.81	0.68	0.74	38



accuracy			0.68	475
macro avg	0.62	0.59	0.58	475
weighted avg	0.65	0.68	0.65	475

✓ Model Performance Improvement

Reducing the Learning Rate:

ReduceLROnPlateau() is a function that will be used to decrease the learning rate by some factor, if the loss is not decreasing for some time. This may start decreasing the loss at a smaller learning rate. There is a possibility that the loss may still not decrease. This may lead to executing the learning rate reduction again in an attempt to achieve a lower loss.

```
# Complete the Code to monitor val_accuracy
learning_rate_reduction = ReduceLROnPlateau(monitor='val_accuracy',
                                             patience=3,
                                             verbose=1,
                                             factor=0.5,
                                             min_lr=0.00001)
```

✓ Data Augmentation

```
# Clearing backend
from tensorflow.keras import backend
backend.clear_session()

# Fixing the seed for random number generators
import random
np.random.seed(42)
random.seed(42)
tf.random.set_seed(42)

# Complete the code to set the rotation_range to 20
train_datagen = ImageDataGenerator(
    rotation_range=20,
    fill_mode='nearest'
)
```



```

# Initializing a sequential model
model2 = Sequential()

# Complete the code to add the first conv layer with 64 filters and kernel size 3x3 , padding 'same' provides the output size same as the i
# Input_shape denotes input image dimension images
model2.add(Conv2D(64, (3, 3), activation='relu', padding="same", input_shape=(64, 64, 3)))

# Complete the code to add max pooling to reduce the size of output of first conv layer
model2.add(MaxPooling2D((2, 2), padding = 'same'))

model2.add(Conv2D(32, (3, 3), activation='relu', padding="same"))
model2.add(MaxPooling2D((2, 2), padding = 'same'))
model2.add(BatchNormalization())

# flattening the output of the conv layer after max pooling to make it ready for creating dense connections
model2.add(Flatten())

# Adding a fully connected dense layer with 16 neurons
model2.add(Dense(16, activation='relu'))

# Complete the code to add dropout with dropout_rate=0.3
model2.add(Dropout(rate=0.3))
# Complete the code to add the output layer with 12 neurons and activation functions as softmax since this is a multi-class classification
model2.add(Dense(12, activation='softmax'))

# Complete the code to initialize Adam Optimizer
opt='Adam'
# Complete the code to Compile model
model2.compile(optimizer=opt, loss='categorical_crossentropy', metrics=['accuracy'])

# Generating the summary of the model
model2.summary()

```

Model: "sequential"

Layer (type)	Output Shape	Param #
conv2d (Conv2D)	(None, 64, 64, 64)	1792
max_pooling2d (MaxPooling2D)	(None, 32, 32, 64)	0
conv2d_1 (Conv2D)	(None, 32, 32, 32)	18464
max_pooling2d_1 (MaxPooling2D)	(None, 16, 16, 32)	0
batch_normalization (Batch Normalization)	(None, 16, 16, 32)	128
flatten (Flatten)	(None, 8192)	0
dense (Dense)	(None, 16)	131088
dropout (Dropout)	(None, 16)	0
dense_1 (Dense)	(None, 12)	204
Total params: 151676 (592.48 KB)		
Trainable params: 151612 (592.23 KB)		
Non-trainable params: 64 (256.00 Byte)		

Fitting the model on the train data



```
# Complete the code to fit the model on train data with batch_size=64 and epochs=30
```

```
# Epochs
```

```
epochs = 30
```

```
# Batch size
```

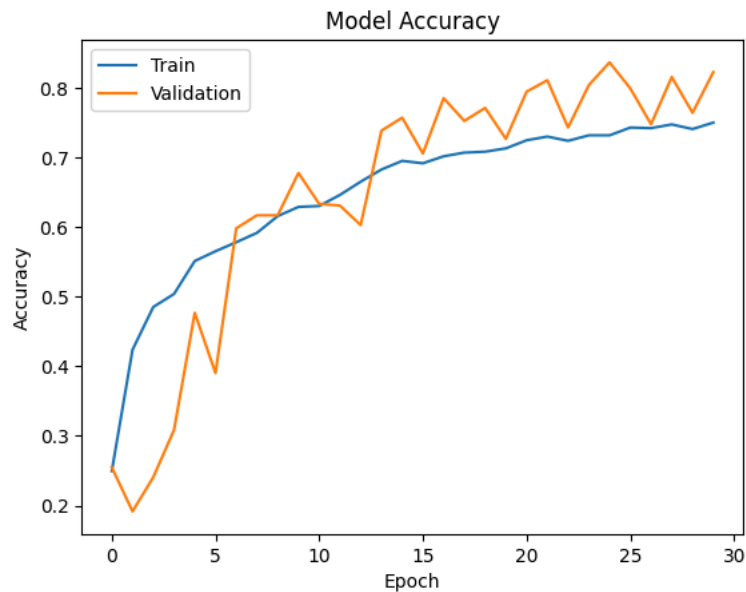
```
batch_size = 64
```

```
history = model2.fit(train_datagen.flow(X_train_normalized, y_train_encoded,
                                         batch_size=batch_size,
                                         shuffle=False),
                    epochs=epochs,
                    steps_per_epoch=X_train_normalized.shape[0] // batch_size,
                    validation_data=(X_val_normalized, y_val_encoded),
                    verbose=1, callbacks=[learning_rate_reduction])
```

```
Epoch 1/30
60/60 [=====] - 8s 103ms/step - loss: 2.1320 - accuracy: 0.2495 - val_loss: 2.4059 - val_accuracy: 0.2547 - 1
Epoch 2/30
60/60 [=====] - 4s 69ms/step - loss: 1.6607 - accuracy: 0.4237 - val_loss: 2.2870 - val_accuracy: 0.1916 - 1
Epoch 3/30
60/60 [=====] - 5s 92ms/step - loss: 1.4740 - accuracy: 0.4851 - val_loss: 2.2436 - val_accuracy: 0.2407 - 1
Epoch 4/30
60/60 [=====] - 4s 70ms/step - loss: 1.3974 - accuracy: 0.5038 - val_loss: 2.1233 - val_accuracy: 0.3084 - 1
Epoch 5/30
60/60 [=====] - 5s 88ms/step - loss: 1.2698 - accuracy: 0.5511 - val_loss: 1.8757 - val_accuracy: 0.4766 - 1
Epoch 6/30
60/60 [=====] - 5s 91ms/step - loss: 1.2072 - accuracy: 0.5652 - val_loss: 1.9397 - val_accuracy: 0.3902 - 1
Epoch 7/30
60/60 [=====] - 4s 70ms/step - loss: 1.2013 - accuracy: 0.5781 - val_loss: 1.4052 - val_accuracy: 0.5981 - 1
Epoch 8/30
60/60 [=====] - 5s 92ms/step - loss: 1.1283 - accuracy: 0.5916 - val_loss: 1.4317 - val_accuracy: 0.6168 - 1
Epoch 9/30
60/60 [=====] - 5s 75ms/step - loss: 1.0979 - accuracy: 0.6156 - val_loss: 1.1780 - val_accuracy: 0.6168 - 1
Epoch 10/30
60/60 [=====] - 5s 75ms/step - loss: 1.0387 - accuracy: 0.6289 - val_loss: 0.9957 - val_accuracy: 0.6776 - 1
Epoch 11/30
60/60 [=====] - 4s 70ms/step - loss: 1.0195 - accuracy: 0.6302 - val_loss: 1.0887 - val_accuracy: 0.6332 - 1
Epoch 12/30
60/60 [=====] - 5s 91ms/step - loss: 0.9912 - accuracy: 0.6460 - val_loss: 1.1412 - val_accuracy: 0.6308 - 1
Epoch 13/30
60/60 [=====] - ETA: 0s - loss: 0.9468 - accuracy: 0.6651
Epoch 13: ReduceLROnPlateau reducing learning rate to 0.0005000000237487257.
60/60 [=====] - 5s 81ms/step - loss: 0.9468 - accuracy: 0.6651 - val_loss: 1.2719 - val_accuracy: 0.6028 - 1
Epoch 14/30
60/60 [=====] - 5s 79ms/step - loss: 0.8715 - accuracy: 0.6825 - val_loss: 0.8356 - val_accuracy: 0.7383 - 1
Epoch 15/30
60/60 [=====] - 4s 69ms/step - loss: 0.8472 - accuracy: 0.6950 - val_loss: 0.7708 - val_accuracy: 0.7570 - 1
Epoch 16/30
60/60 [=====] - 6s 92ms/step - loss: 0.8320 - accuracy: 0.6915 - val_loss: 0.9171 - val_accuracy: 0.7056 - 1
Epoch 17/30
60/60 [=====] - 4s 70ms/step - loss: 0.8092 - accuracy: 0.7016 - val_loss: 0.7527 - val_accuracy: 0.7850 - 1
Epoch 18/30
60/60 [=====] - 5s 90ms/step - loss: 0.7938 - accuracy: 0.7068 - val_loss: 0.8060 - val_accuracy: 0.7523 - 1
Epoch 19/30
60/60 [=====] - 5s 75ms/step - loss: 0.7827 - accuracy: 0.7082 - val_loss: 0.8079 - val_accuracy: 0.7710 - 1
Epoch 20/30
60/60 [=====] - ETA: 0s - loss: 0.7738 - accuracy: 0.7129
Epoch 20: ReduceLROnPlateau reducing learning rate to 0.0002500000118743628.
60/60 [=====] - 5s 83ms/step - loss: 0.7738 - accuracy: 0.7129 - val_loss: 0.8907 - val_accuracy: 0.7266 - 1
Epoch 21/30
60/60 [=====] - 4s 70ms/step - loss: 0.7385 - accuracy: 0.7246 - val_loss: 0.6824 - val_accuracy: 0.7944 - 1
Epoch 22/30
60/60 [=====] - 5s 89ms/step - loss: 0.7280 - accuracy: 0.7298 - val_loss: 0.6485 - val_accuracy: 0.8107 - 1
Epoch 23/30
60/60 [=====] - 4s 73ms/step - loss: 0.7371 - accuracy: 0.7238 - val_loss: 0.8799 - val_accuracy: 0.7430 - 1
Epoch 24/30
60/60 [=====] - 5s 76ms/step - loss: 0.7182 - accuracy: 0.7317 - val_loss: 0.6777 - val_accuracy: 0.8037 - 1
Epoch 25/30
60/60 [=====] - 4s 70ms/step - loss: 0.7240 - accuracy: 0.7317 - val_loss: 0.6321 - val_accuracy: 0.8364 - 1
Epoch 26/30
60/60 [=====] - 5s 92ms/step - loss: 0.6896 - accuracy: 0.7428 - val_loss: 0.6672 - val_accuracy: 0.7991 - 1
Epoch 27/30
60/60 [=====] - 4s 71ms/step - loss: 0.7160 - accuracy: 0.7420 - val_loss: 0.7065 - val_accuracy: 0.7477 - 1
```

Model Evaluation

```
plt.plot(history.history['accuracy'])
plt.plot(history.history['val_accuracy'])
plt.title('Model Accuracy')
plt.ylabel('Accuracy')
plt.xlabel('Epoch')
plt.legend(['Train', 'Validation'], loc='upper left')
plt.show()
```



Evaluate the model on test data

```
accuracy = model2.evaluate(X_test_normalized, y_test_encoded, verbose=2) # Complete the code to evaluate the model on test data
```



```
15/15 - 0s - loss: 0.7480 - accuracy: 0.7558 - 228ms/epoch - 15ms/step
```

Plotting the Confusion Matrix

```
# Complete the code to obtain the output probabilities
```

```
y_pred=model2.predict(X_test_normalized)
```



```
15/15 [=====] - 0s 2ms/step
```

```
# Obtaining the categorical values from y_test_encoded and y_pred
```

```
y_pred_arg=np.argmax(y_pred,axis=1)
```

```
y_test_arg=np.argmax(y_test_encoded,axis=1)
```

```
# Plotting the Confusion Matrix using confusion matrix() function which is also predefined in tensorflow module
```

```
confusion_matrix = tf.math.confusion_matrix(y_test_arg,y_pred_arg) # Complete the code to obtain the confusion matrix
f, ax = plt.subplots(figsize=(12, 12))
```

```
sns.heatmap(
    confusion_matrix,
    annot=True,
    linewidths=.4,
    fmt="d",
    square=True,
    ax=ax
)
```

```
# Setting the labels to both the axes
```

```
ax.set_xlabel('Predicted labels');ax.set_ylabel('True labels');
```

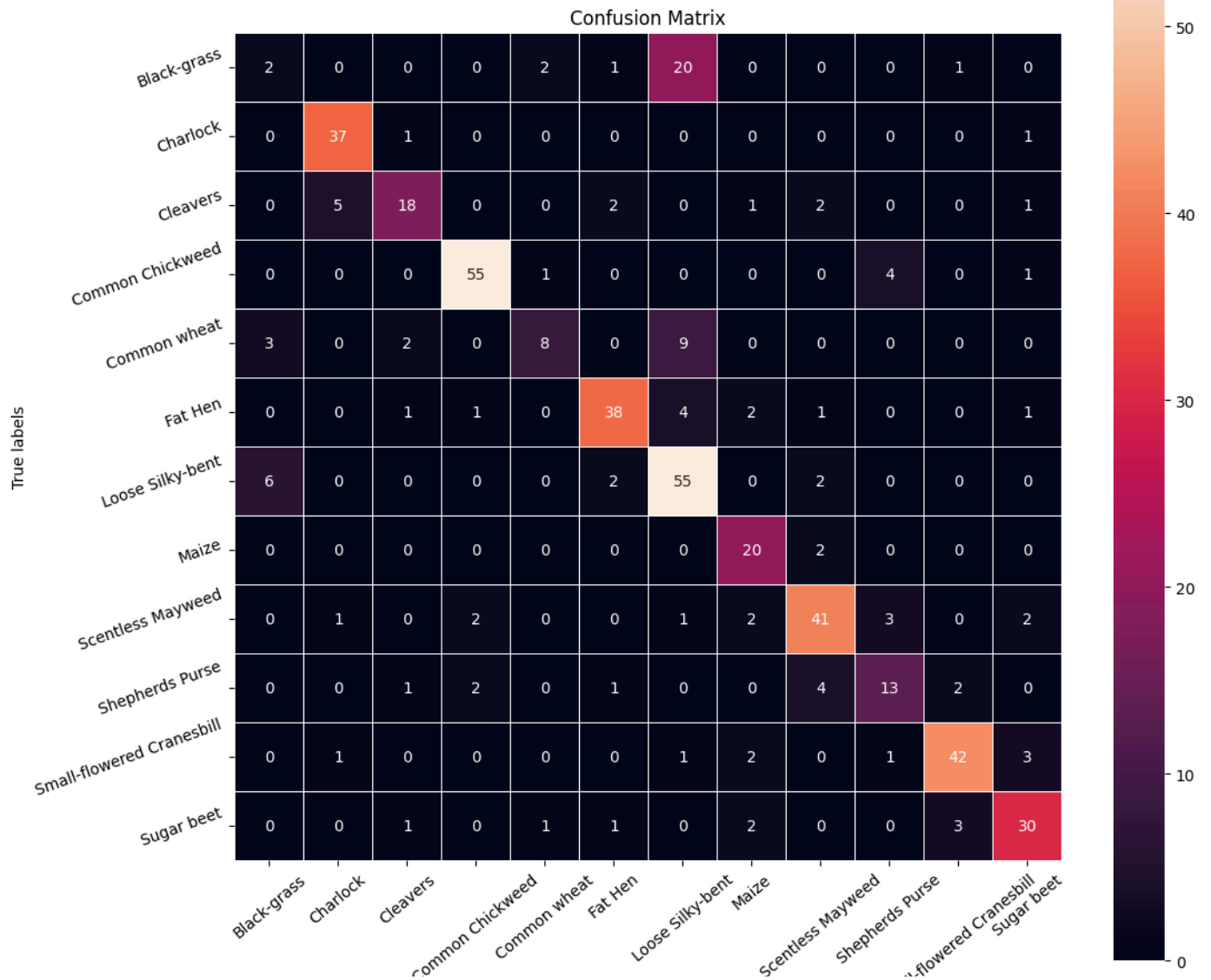
```
ax.set_title('Confusion Matrix');
```

```
ax.xaxis.set_ticklabels(list(enc.classes_),rotation=40)
```

```
ax.yaxis.set_ticklabels(list(enc.classes_),rotation=20)
```

```
plt.show()
```





Plotting Classification Report

```
# Plotting the classification report
cr=metrics.classification_report(y_test_arg, y_pred_arg)
print(cr)
```

Complete the code to plot the classification report

