Classification of images of producs (Fruits and Vegetables)

Our original given dataset consisted of 3625 images split into three categories: Train, Test, Validation, and further into different types of produce.

Train sets usually consist of around 100 images per produce, test and validation of around 10.

After performing exploratory analysis, comprising of examing data sizes, formats and color palletes/depths/counts, we started preprocessing.

The dataset had some same pictures in both test/validation and train data, which inflated accuracy, so those were removed from train data. Furthermore, to offset the removal of data we added some additional data of our own (both pictures taken by overselves, found from open sources and modifications of existing data as described later).

Our preprocessing consists of conversion to RGB, resizing the images to 150x150 px. We decided for a higher pixel count to maintain as much data as possible. However, it lowered performance of everything for obvious reasons.

To counter that, we used python pickle package to transform the data into byte stream for faster loading of data.

Between those two steps, we also randomly added pictures that were randomly augmented (flipped or rotated) to enhance our dataset.

Lastly we normalized the pixel values by dividing them by 255 (maximum value possible at the current point of preprocessing).

As for the models, our focus was on our custom CNN model, but we also implemented KNN, decision tree and random forest models.

In [ ]:
```
!pip install colorthief
```

```
Collecting colorthief
  Using cached colorthief-0.2.1-py2.py3-none-any.whl (6.1 kB)
Requirement already satisfied: Pillow in /usr/local/lib/python3.7/dist-packages (f
rom colorthief) (7.1.2)
Installing collected packages: colorthief
Successfully installed colorthief-0.2.1
```

Importing required packages.

In [ ]:
```
from google.colab import drive
drive.mount('/content/drive')
```

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

In [ ]:
```
import seaborn as sns
import numpy as np
import os  # Traversing file paths
```

```
import pickle  # Package for compressing image data in the form of the numpy array
import matplotlib.pyplot as plt
import plotly.express as px
import plotly.graph_objects as go
import tensorflow as tf
import pandas as pd

# 3 packages for exploratory analysis
from PIL import Image
from colorthief import ColorThief
from matplotlib.image import imread

# Packages for chosen models
from keras.preprocessing.image import ImageDataGenerator
from math import sqrt, inf
from random import random

from sklearn import svm
from sklearn.model_selection import GridSearchCV
from sklearn.metrics import accuracy_score, confusion_matrix, classification_repor
from sklearn.tree import DecisionTreeClassifier
from sklearn.ensemble import RandomForestClassifier
from sklearn.neighbors import KNeighborsClassifier
from sklearn.pipeline import Pipeline
from sklearn.preprocessing import MinMaxScaler

from tensorflow import keras
from keras import layers

from keras.models import Sequential, Model
from keras.layers import Dense, Dropout, Activation, Flatten, Conv2D, MaxPooling2D
from keras.applications.vgg16 import VGG16, preprocess_input
from keras.preprocessing import image
```

Definition of constants and categorical dictionaries.

In [ ]:

```
IMAGE_SIZE = 150
BATCH_SIZE = 128

categories = {"apple": 1, "banana": 2, "beetroot": 3,
              "bell pepper": 4, "cabbage": 5, "capsicum":6,
              "carrot": 7, "cauliflower": 8, "chilli pepper": 9,
              "corn": 10, "cucumber": 11, "eggplant": 12,
              "garlic": 13, "ginger": 14, "grapes": 15,
              "jalepeno": 16, "kiwi": 17, "lemon": 18,
              "lettuce": 19, "mango": 20, "onion": 21,
              "orange": 22, "paprika": 23, "pear": 24,
              "peas": 25, "pineapple": 26, "pomegranate": 27,
              "potato": 28, "raddish": 29, "soy beans": 30,
              "spinach": 31, "sweetcorn": 32, "sweetpotato": 33,
              "tomato": 34, "turnip": 35, "watermelon": 36}

for key, value in categories.items():
    categories[key] = value-1

rev_cat = {}
for veg, num in categories.items():
    rev_cat[num] = veg
```

Function for basic preprocessing. Resizing the images and saving tham in the compressed format with the pickle library. This was done locally in our systems. Data is uploaded in the compressed format already.

We have chosen the size of the images 150x150.

This function will not work in jupyter notebook as the raw images are not present due to their size.

After resizing every image we decided to rotate or flip images to introduce image augmentation. There is a chance that image won't be altered at all.

In [ ]:
```
def rotate_or_flip(image):
    if random() < 0.5:
        f = random()
        if f <= 0.3:
            return np.rot90(image, 1)
        elif f <= 0.6:
            return np.rot90(image, 2)
        return np.rot90(image, 3)

    if random() < 0.5:
        return np.flipud(image)
    return np.fliplr(image)


def change_size_and_pickle(subfolder="train"):
    image_size = 150
    X = []
    y = []

    for root, directories, files in os.walk(PATH + "\\" + "dataset\\" + subfolder)
        for file in files:
            file_path = os.path.join(root, file)
            with Image.open(file_path) as image:
                path = "\\" + root.split("\\")[-1] + "_" + file.split(".")[0] + ".
                image = image.resize((image_size, image_size))
                image = np.array(image.convert('RGB'))

                if random() < 0.5:
                    new_image = rotate_or_flip(image)
                    X.append(new_image)
                    y.append(categories[root.split("\\")[-1]])

                X.append(image)
                y.append(categories[root.split("\\")[-1]])

    y = np.array(y)
    X = np.array(X)

    X = X / 255.0
    X = X.reshape(-1, image_size, image_size, 3)

    with open('X_pickle_' + subfolder, 'wb') as pickle_out:
        pickle.dump(X, pickle_out)
    with open('y_pickle_' + subfolder, 'wb') as pickle_out:
        pickle.dump(y, pickle_out)
```

Function for loading the pickled data into the numpy arrays.

In [ ]:
```
def get_data():
    X = pickle.load(open('/content/drive/MyDrive/dataset//X_pickle_train', 'rb'))
    y = pickle.load(open('/content/drive/MyDrive/dataset//y_pickle_train', 'rb'))

    X_val = pickle.load(open('/content/drive/MyDrive/dataset//X.pickle_validation'
```

```
    y_val = pickle.load(open('/content/drive/MyDrive/dataset//y.pickle_validation'

    X_test = pickle.load(open('/content/drive/MyDrive/dataset//X_pickle_test', 'rb
    y_test = pickle.load(open('/content/drive/MyDrive/dataset//y_pickle_test', 'rb
    return X, y, X_val, y_val, X_test, y_test


X, y, X_val, y_val, X_test, y_test = get_data()
```

Functions for better printing of matrices without heatmaps.

In [ ]:
```
def print_row(row):
    line = ""
    for elem in row:
        line = line + "{:2d} ".format(int(elem))
    print(line)


def print_cnf(cm):
    for row in cm:
        print_row(row)
```

Creation and displaying of model improvements over time.

Heatmap visualisation of confusion matrices.

In [ ]:
```
def process_history(history, cm, y_test):
    pd.DataFrame(history.history)[['accuracy','val_accuracy']].plot()
    plt.title("Accuracy")
    plt.show()

    pd.DataFrame(history.history)[['loss','val_loss']].plot()
    plt.title("Loss")
    plt.show()


    plt.figure(figsize = (15,10))
    sns.heatmap(cm,
                annot=True,
                xticklabels = sorted(set(categories.keys())),
                yticklabels = sorted(set(categories.keys())),
                )
    plt.title('Confusion Matrix')
    plt.show()
```

Baseline

As our baseline model we implemented a simple model that computes the average color of a given picture and then compares it to the average dominant color of every produce category found in the color matrix we have created during out exploratory analysis.

The next function is the one used during the analysis, which was before we moved onto the pickled data format. As such, it will only run with full unprocessed dataset. To save space, we have provided the result gained on our local machine.

In [ ]:
```
def dominant_color_per_picture():
    colors = []
```

```
    for root, directories, files in os.walk(PATH +  "\dataset\\train"):
        dominant = [0, 0, 0]
        count = 0
        for file in files:
            count += 1
            file_path = os.path.join(root, file)
            color = list(ColorThief(file_path).get_color(quality=1))
            dominant[0] += color[0]
            dominant[1] += color[1]
            dominant[2] += color[2]
        if count != 0:
            dominant[0] /= count
            dominant[1] /= count
            dominant[2] /= count
        colors.append(dominant)
    return colors
```

In [ ]:
```
average_color = [[160.2, 82.9, 73.0], [207.0, 180.33333333333334, 86.4444444444444
```

Baseline model itself

As stated before, we compute the average color of every test picture and compare it to the data obtained from the previous function.

Resulting produce is selected based on the distance of the average color of the testing image from the average dominant color of the each produce, choosing the shortest one as its result.

In [ ]:
```
def naive_color_baseline(average_colors, X_test, y_test, print_out=False):
    correct_guesses = 0
    number_of_guesses = 0

    for i in range(len(X_test)):
        correct = y_test[i]

        avg_color_per_row = np.average(X_test[i], axis=0)
        avg_color = np.average(avg_color_per_row, axis=0)


        min_dist = inf
        category = -1
        for j in range(len(average_colors)):
            dist = sqrt((avg_color[0] - (average_colors[j][0]/255)) ** 2 +
                        (avg_color[1] - (average_colors[j][1]/255)) ** 2 +
                        (avg_color[2] - (average_colors[j][2]/255)) ** 2)
            if dist < min_dist:
                min_dist = dist
                category = j
        if correct == category:
            correct_guesses += 1
        else:
          if print_out:
              print("Predicted: {:13s} ({:2d}) | actually: {:13s} ({:2d})".format(
                                                                                  (
                                                                                  (
                                                                                  (


        number_of_guesses += 1
    print("missed {:3d} out of {:3d}".format(number_of_guesses-correct_guesses,
```

```
                                                number_of_guesses))
    print("Accuracy on testing data: {:3.3f}%".format((correct_guesses / number_of_
```

Running the baseline, inputting the average colors of each produce. Outputting wrongly predicted ones.

In [ ]:
```
def predict_baseline(average_colors=None):
    X, y, X_val, y_val, X_test, y_test = get_data()
    if average_colors is None:
        average_colors = dominant_color_per_picture()
    print(average_colors)
    naive_color_baseline(average_colors, X, y)
predict_baseline(average_color)
```

```
[[160.2, 82.9, 73.0], [207.0, 180.33333333333334, 86.44444444444444], [123.1, 102.
7, 80.8], [164.33333333333334, 140.22222222222223, 93.55555555555556], [119.7, 14
8.4, 99.5], [165.0, 139.7, 71.1], [218.44444444444446, 126.66666666666667, 62.5555
5555555556], [172.7, 163.3, 131.2], [182.55555555555554, 146.44444444444446, 124.8
8888888888889], [185.2, 172.6, 81.4], [134.1, 144.9, 91.1], [76.6, 72.3, 61.7], [1
56.0, 142.3, 128.4], [186.9, 143.5, 90.4], [111.33333333333333, 93.77777777777777,
82.44444444444444], [86.0, 106.88888888888889, 57.666666666666664], [151.1, 155.6,
77.3], [200.5, 180.8, 86.0], [113.11111111111111, 161.44444444444446, 53.111111111
111114], [219.5, 166.4, 81.7], [191.2, 137.5, 114.8], [222.33333333333334, 164.111
11111111111, 53.22222222222222], [143.9, 73.7, 34.3], [160.6, 158.1, 74.5], [103.
9, 142.4, 56.9], [136.7, 131.4, 93.7], [147.6, 50.8, 52.3], [188.6, 147.2, 94.7],
[184.77777777777777, 186.88888888888889, 159.66666666666666], [181.6, 145.7, 97.
9], [135.1, 158.8, 102.1], [198.4, 177.1, 70.1], [127.0, 97.4, 78.2], [149.6, 103.
3, 65.6], [148.2, 143.1, 121.2], [146.5, 112.9, 86.7]]
missed 3644 out of 3918
Accuracy on testing data: 6.993%
```

Non CNN models

Function to compute three models: KNN, Random forest and Decision tree. Parameters in each classifier were computed from a GridSearch.

We also visualised confusion matrices and accuracy scores.

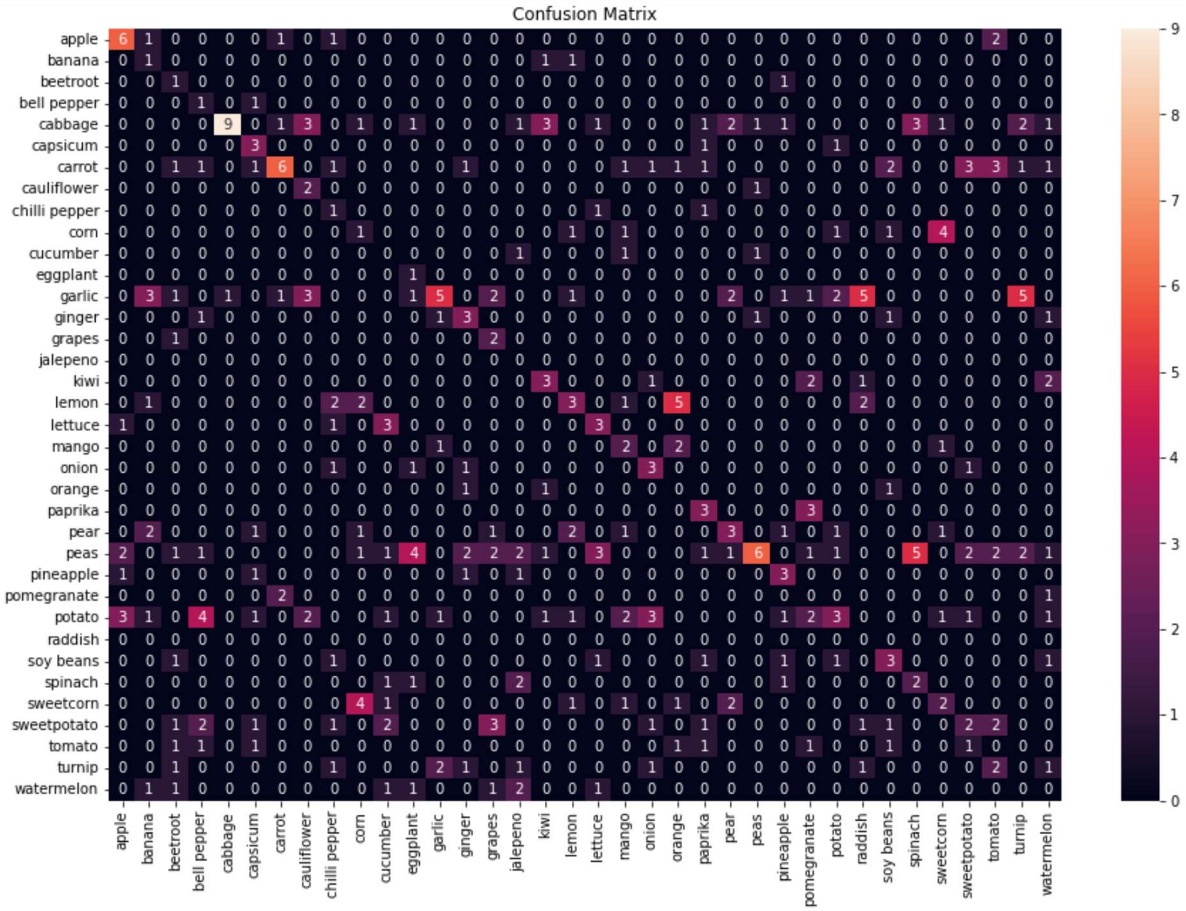
In [ ]:
```
def non_CNN_models(model="knn"):
    X, y, X_val, y_val, X_test, y_test = get_data()
    samples, nx, ny, nrgb = X.shape
    X = X.reshape((samples, nx*ny*nrgb))

    samples, nx, ny, nrgb = X_test.shape
    X_test = X_test.reshape((samples, nx*ny*nrgb))

    if model == "knn":
        classifier = KNeighborsClassifier(leaf_size=5, n_neighbors=9, weights='dis
    elif model == "rf":
        classifier = RandomForestClassifier(min_samples_leaf=3, min_samples_split=
    elif model == "dt":
        classifier = DecisionTreeClassifier(max_depth=None, min_samples_split=2, m

    classifier.fit(X, y)
    pred = classifier.predict(X_test)
    print('MODEL:', model)
    print("ACCURACY SCORE:")
    print(accuracy_score(pred, y_test))
    print("CLASSIFICATION REPORT:")
    print(classification_report(pred,y_test))
```

```
    print("CONFUSION MATRIX:")
    cm = confusion_matrix(pred, y_test)
    plt.figure(figsize = (15,10))
    sns.heatmap(cm,
                annot=True,
                xticklabels = sorted(set(categories.keys())),
                yticklabels = sorted(set(categories.keys())),
                )
    plt.title('Confusion Matrix')
    plt.show()
```

In [ ]:
```
non_CNN_models('knn')
```

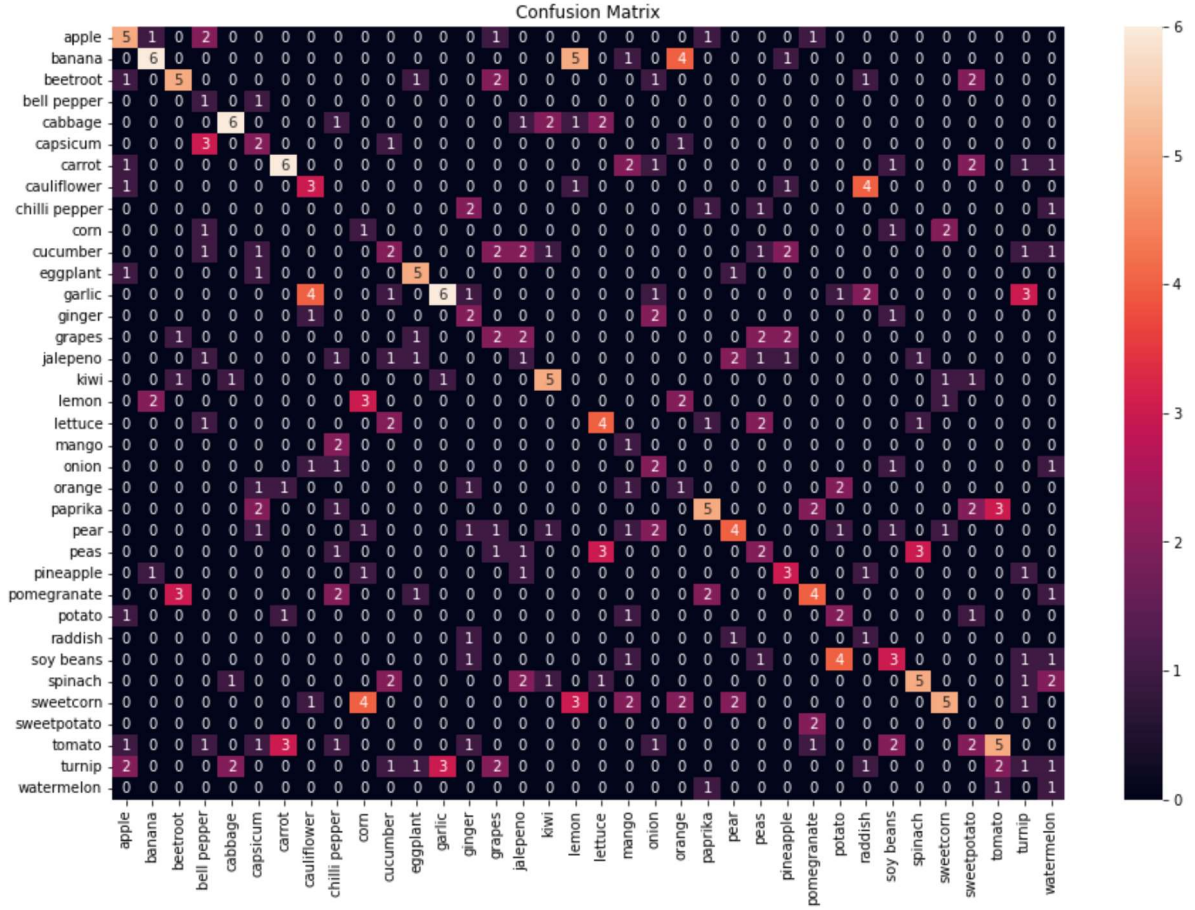
```
MODEL: knn
ACCURACY SCORE:
0.22554347826086957
CLASSIFICATION REPORT:
              precision    recall  f1-score   support

           0       0.46      0.55      0.50        11
           1       0.10      0.33      0.15         3
           2       0.10      0.50      0.17         2
           3       0.09      0.50      0.15         2
           4       0.90      0.28      0.43        32
           5       0.30      0.60      0.40         5
           6       0.55      0.24      0.33        25
           7       0.20      0.67      0.31         3
           8       0.10      0.33      0.15         3
           9       0.10      0.11      0.11         9
          10       0.00      0.00      0.00         3
          11       0.10      1.00      0.18         1
          12       0.50      0.15      0.23        34
          13       0.30      0.38      0.33         8
          14       0.18      0.67      0.29         3
          15       0.00      0.00      0.00         0
          16       0.30      0.33      0.32         9
          17       0.30      0.19      0.23        16
          18       0.30      0.38      0.33         8
          19       0.20      0.33      0.25         6
          20       0.30      0.43      0.35         7
          21       0.00      0.00      0.00         3
          22       0.27      0.50      0.35         6
          23       0.30      0.21      0.25        14
          24       0.60      0.14      0.23        42
          25       0.30      0.43      0.35         7
          26       0.00      0.00      0.00         3
          27       0.30      0.10      0.15        29
          28       0.00      0.00      0.00         0
          29       0.30      0.30      0.30        10
          30       0.20      0.29      0.24         7
          31       0.20      0.17      0.18        12
          32       0.20      0.11      0.14        18
          33       0.00      0.00      0.00         8
          34       0.00      0.00      0.00        11
          35       0.00      0.00      0.00         8

    accuracy                           0.23       368
   macro avg       0.22      0.28      0.21       368
weighted avg       0.37      0.23      0.24       368

CONFUSION MATRIX:
```

```
/usr/local/lib/python3.7/dist-packages/sklearn/metrics/_classification.py:1318: Un
definedMetricWarning: Recall and F-score are ill-defined and being set to 0.0 in l
abels with no true samples. Use `zero_division` parameter to control this behavio
r.
  _warn_prf(average, modifier, msg_start, len(result))
/usr/local/lib/python3.7/dist-packages/sklearn/metrics/_classification.py:1318: Un
definedMetricWarning: Recall and F-score are ill-defined and being set to 0.0 in l
abels with no true samples. Use `zero_division` parameter to control this behavio
r.
  _warn_prf(average, modifier, msg_start, len(result))
/usr/local/lib/python3.7/dist-packages/sklearn/metrics/_classification.py:1318: Un
definedMetricWarning: Recall and F-score are ill-defined and being set to 0.0 in l
abels with no true samples. Use `zero_division` parameter to control this behavio
r.
  _warn_prf(average, modifier, msg_start, len(result))
```



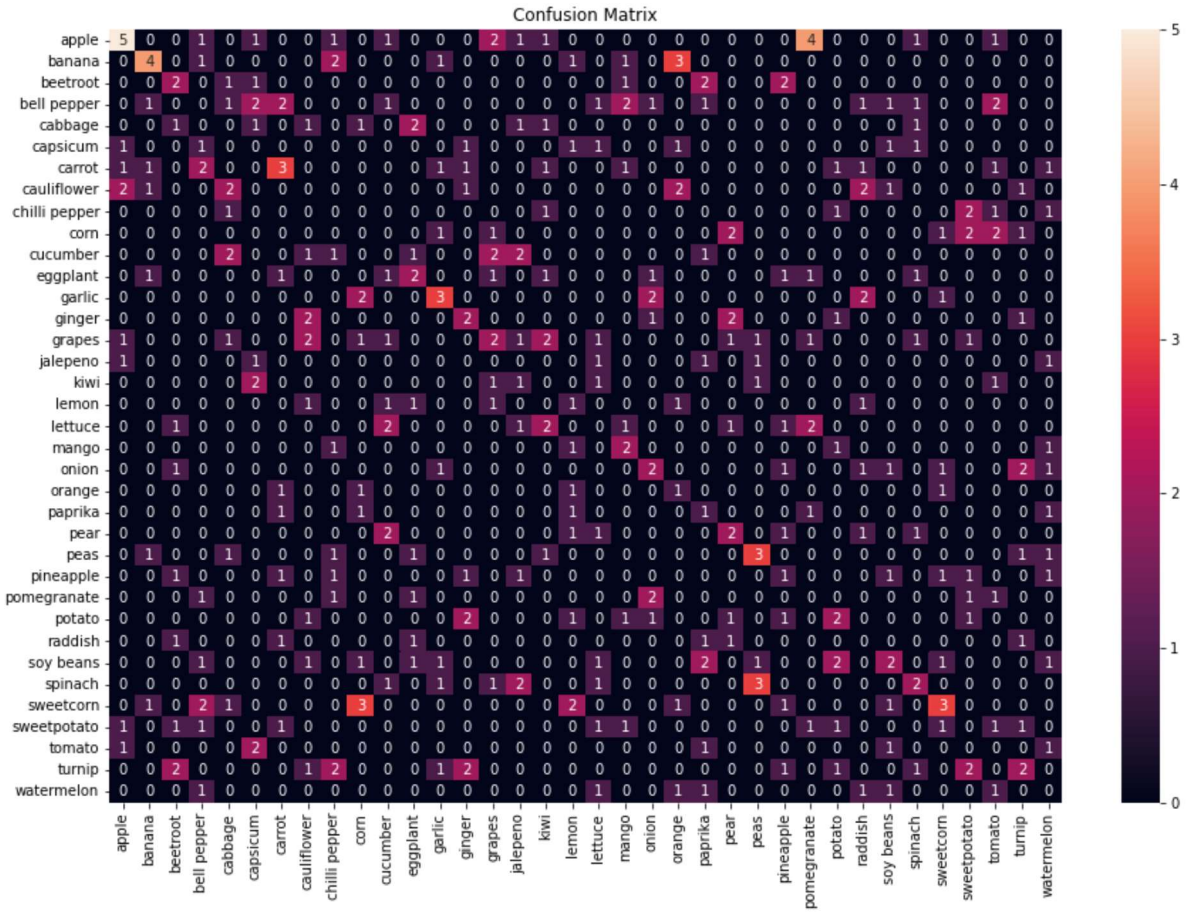
In [ ]:
```
non_CNN_models('rf')
```

```
MODEL: rf
ACCURACY SCORE:
0.2907608695652174
CLASSIFICATION REPORT:
              precision    recall  f1-score   support

           0       0.38      0.45      0.42        11
           1       0.60      0.35      0.44        17
           2       0.50      0.38      0.43        13
           3       0.09      0.50      0.15         2
           4       0.60      0.46      0.52        13
           5       0.20      0.29      0.24         7
           6       0.55      0.40      0.46        15
           7       0.30      0.30      0.30        10
           8       0.00      0.00      0.00         5
           9       0.10      0.20      0.13         5
          10       0.20      0.14      0.17        14
          11       0.50      0.62      0.56         8
          12       0.60      0.32      0.41        19
          13       0.20      0.33      0.25         6
          14       0.18      0.20      0.19        10
          15       0.10      0.10      0.10        10
          16       0.50      0.50      0.50        10
          17       0.00      0.00      0.00         8
          18       0.40      0.36      0.38        11
          19       0.10      0.33      0.15         3
          20       0.20      0.33      0.25         6
          21       0.10      0.14      0.12         7
          22       0.45      0.33      0.38        15
          23       0.40      0.27      0.32        15
          24       0.20      0.18      0.19        11
          25       0.30      0.38      0.33         8
          26       0.40      0.31      0.35        13
          27       0.20      0.33      0.25         6
          28       0.10      0.33      0.15         3
          29       0.30      0.25      0.27        12
          30       0.50      0.33      0.40        15
          31       0.50      0.25      0.33        20
          32       0.00      0.00      0.00         2
          33       0.45      0.26      0.33        19
          34       0.10      0.06      0.08        16
          35       0.10      0.33      0.15         3

    accuracy                           0.29       368
   macro avg       0.29      0.29      0.27       368
weighted avg       0.36      0.29      0.31       368

CONFUSION MATRIX:
```



In [ ]:
```
non_CNN_models('dt')
```

```
MODEL: dt
ACCURACY SCORE:
0.12771739130434784
CLASSIFICATION REPORT:
              precision    recall  f1-score   support

           0       0.38      0.26      0.31        19
           1       0.40      0.31      0.35        13
           2       0.20      0.22      0.21         9
           3       0.00      0.00      0.00        17
           4       0.00      0.00      0.00         9
           5       0.00      0.00      0.00         8
           6       0.27      0.20      0.23        15
           7       0.00      0.00      0.00        12
           8       0.00      0.00      0.00         7
           9       0.00      0.00      0.00        10
          10       0.00      0.00      0.00        10
          11       0.20      0.18      0.19        11
          12       0.30      0.30      0.30        10
          13       0.20      0.22      0.21         9
          14       0.18      0.12      0.14        17
          15       0.00      0.00      0.00         6
          16       0.00      0.00      0.00         7
          17       0.10      0.14      0.12         7
          18       0.00      0.00      0.00        11
          19       0.20      0.33      0.25         6
          20       0.20      0.18      0.19        11
          21       0.10      0.20      0.13         5
          22       0.09      0.17      0.12         6
          23       0.20      0.22      0.21         9
          24       0.30      0.30      0.30        10
          25       0.10      0.10      0.10        10
          26       0.00      0.00      0.00         7
          27       0.20      0.18      0.19        11
          28       0.00      0.00      0.00         6
          29       0.20      0.13      0.16        15
          30       0.20      0.18      0.19        11
          31       0.30      0.20      0.24        15
          32       0.00      0.00      0.00        11
          33       0.00      0.00      0.00         6
          34       0.20      0.13      0.16        15
          35       0.00      0.00      0.00         7

    accuracy                           0.13       368
   macro avg       0.13      0.12      0.12       368
weighted avg       0.15      0.13      0.13       368

CONFUSION MATRIX:
```



Functions that implement GridSearch so that we may find the best paramaters from given options that are then used in the previous non CNN models.

In [ ]:
```
def KNN_with_grid():
    X, y, X_val, y_val, X_test, y_test = get_data()

    samples, nx, ny, nrgb = X_val.shape
    X_val = X_val.reshape((samples, nx*ny*nrgb))

    samples, nx, ny, nrgb = X_test.shape
    X_test = X_test.reshape((samples, nx*ny*nrgb))

    knn = KNeighborsClassifier()

    params = [{'n_neighbors': [3, 5, 9, 11],
         'weights': ['uniform', 'distance'],
         'leaf_size': [5, 6, 15, 18]}]
    gs_knn = GridSearchCV(knn,
                      param_grid=params,
                      scoring='accuracy',
                      cv=5)
    gs_knn.fit(X_val, y_val)
    print(gs_knn.best_params_)
    # find best model score
    print(gs_knn.score(X_test, y_test))
    return gs_knn


def RF_with_grid():
    X, y, X_val, y_val, X_test, y_test = get_data()

    samples, nx, ny, nrgb = X_val.shape
    X_val = X_val.reshape((samples, nx*ny*nrgb))
```

```
    samples, nx, ny, nrgb = X_test.shape
    X_test = X_test.reshape((samples, nx*ny*nrgb))

    rf = RandomForestClassifier()

    params = [{'n_estimators': [100, 125],
         'min_samples_split': [2, 3, 5],
         'min_samples_leaf': [1, 3, 5]}]
    gs_rf = GridSearchCV(rf,
                         param_grid=params,
                         scoring='accuracy',
                         cv=5)
    gs_rf.fit(X_val, y_val)
    print(gs_rf.best_params_)
    # find best model score
    print(gs_rf.score(X_test, y_test))
    return gs_rf

def DT_with_grid():
    X, y, X_val, y_val, X_test, y_test = get_data()

    samples, nx, ny, nrgb = X_val.shape
    X_val = X_val.reshape((samples, nx*ny*nrgb))

    samples, nx, ny, nrgb = X_test.shape
    X_test = X_test.reshape((samples, nx*ny*nrgb))

    dt = DecisionTreeClassifier()

    params = [{'max_depth': [None, 5],
         'min_samples_split': [2, 3, 5],
         'min_samples_leaf': [1, 2, 3]}]
    gs_dt = GridSearchCV(dt,
                         param_grid=params,
                         scoring='accuracy',
                         cv=5)
    gs_dt.fit(X_val, y_val)
    print(gs_dt.best_params_)
    # find best model score
    print(gs_dt.score(X_test, y_test))
    return gs_dt
```

In [ ]:
```
print('KNN:')
KNN_with_grid()
```

```
KNN:
{'leaf_size': 5, 'n_neighbors': 3, 'weights': 'distance'}
0.9510869565217391
```

Out[ ]:
```
GridSearchCV(cv=5, estimator=KNeighborsClassifier(),
             param_grid=[{'leaf_size': [5, 6, 15, 18],
                          'n_neighbors': [3, 5, 9, 11],
                          'weights': ['uniform', 'distance']}],
             scoring='accuracy')
```

In [ ]:
```
print('Random forest:')
RF_with_grid()
```

```
Random forest:
{'min_samples_leaf': 3, 'min_samples_split': 5, 'n_estimators': 125}
0.9510869565217391
```

Out[ ]:
```
GridSearchCV(cv=5, estimator=RandomForestClassifier(),
             param_grid=[{'min_samples_leaf': [1, 3, 5],
                          'min_samples_split': [2, 3, 5],
                          'n_estimators': [100, 125]}],
             scoring='accuracy')
```

In [ ]:
```
print('Decision tree:')
DT_with_grid()
```

```
Decision tree:
{'max_depth': None, 'min_samples_leaf': 1, 'min_samples_split': 5}
0.8668478260869565
```

Out[ ]:
```
GridSearchCV(cv=5, estimator=DecisionTreeClassifier(),
             param_grid=[{'max_depth': [None, 5], 'min_samples_leaf': [1, 2, 3],
                          'min_samples_split': [2, 3, 5]}],
             scoring='accuracy')
```

CNN model

Convolutional Neuron Network model. It consists of multiple layers with repeating pattern of 2D convolutions and consequent max-pooling.

We flatten the data from 2D to 1D in order to dense it. Last dense layer is the output layer and the number of units is equal to the number of categories.

To reduce the impact of overfitting, we added dropout layers to our CNN model.

For the validation, we used given validation data instead of usually used validation split.

In [ ]:
```
def model_CNN(X, y, X_val, y_val, epochs=6):
    model = Sequential()
    model.add(Conv2D(32, (3,3), input_shape = X.shape[1:]))
    model.add(Activation('relu'))
    model.add(MaxPooling2D(pool_size=(2)))
    model.add(Dropout(0.4))

    model.add(Conv2D(32, (3,3)))
    model.add(Activation('relu'))
    model.add(MaxPooling2D(pool_size=(2)))
    model.add(Dropout(0.6))

    model.add(Flatten())
    model.add(Dense(128))
    model.add(Activation('relu'))

    model.add(Dense(36))
    model.add(Activation('softmax'))

    model.compile(loss='sparse_categorical_crossentropy', optimizer='adam', metric
    history = model.fit(X, y, batch_size=128, validation_data=(X_val, y_val), epoc
                        callbacks=[tf.keras.callbacks.EarlyStopping(monitor='val_l
                                                                    patience=2,
                                                                    restore_best_w
    return model, history
```

Testing the CNN model on the given test data taken from the pickled files.

Predicted results are compared to the expected values and the incorrect predictions are displayed.

We tried to run the model with 50 epochs but the program stopped after 13 because of RAM shortage.

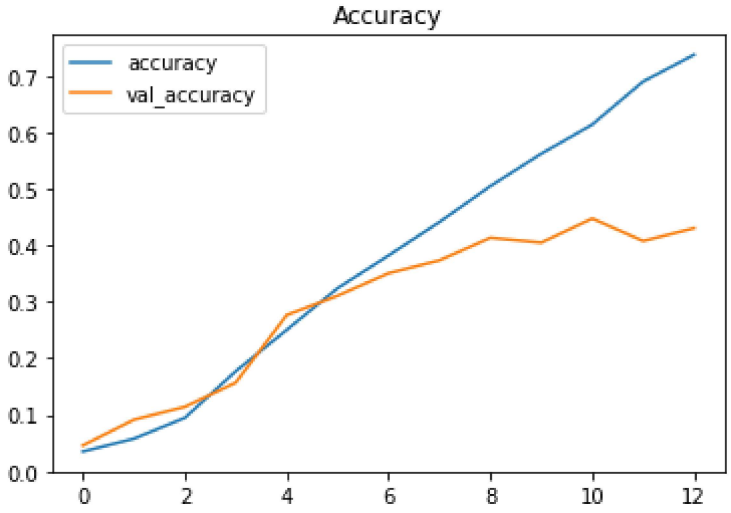
In [ ]:
```
def predict_CNN_results(epochs):
    X, y, X_val, y_val, X_test, y_test = get_data()

    m, h = model_CNN(X, y, X_val, y_val, epochs)

    pred = m.predict(X_test)
    counter = 0
    print(pred.shape)
    best_pred = []
    for i in range(len(pred)):
      best_pred.append(np.argmax(pred[i]))
    print(y_test.shape)
    cm = confusion_matrix(best_pred, y_test)
    for i in range(len(pred)):
        j = np.argmax(pred[i])
        if (abs(y_test[i] - j) > 1):
            print("Predicted: {:13s} ({:2d}) | actually: {:13s} ({:2d})".format(rev
            counter += 1
    print("missed {:3d} out of {:3d}".format(counter, len(pred)))
    print("Accuracy on testing data: {:3.3f} %".format((1-counter/len(pred))*100))
    process_history(h, cm, y_test)



predict_CNN_results(50)
```

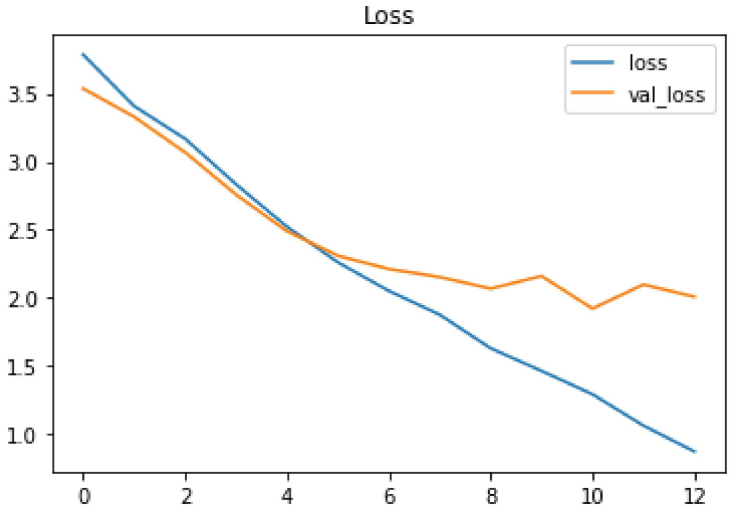
```
Epoch 1/50
31/31 [==============================] - 109s 3s/step - loss: 3.7859 - accuracy:
0.0347 - val_loss: 3.5370 - val_accuracy: 0.0456
Epoch 2/50
31/31 [==============================] - 107s 3s/step - loss: 3.4083 - accuracy:
0.0574 - val_loss: 3.3299 - val_accuracy: 0.0912
Epoch 3/50
31/31 [==============================] - 106s 3s/step - loss: 3.1690 - accuracy:
0.0944 - val_loss: 3.0691 - val_accuracy: 0.1140
Epoch 4/50
31/31 [==============================] - 106s 3s/step - loss: 2.8341 - accuracy:
0.1769 - val_loss: 2.7582 - val_accuracy: 0.1567
Epoch 5/50
31/31 [==============================] - 108s 3s/step - loss: 2.5224 - accuracy:
0.2501 - val_loss: 2.4892 - val_accuracy: 0.2764
Epoch 6/50
31/31 [==============================] - 107s 3s/step - loss: 2.2610 - accuracy:
0.3239 - val_loss: 2.3087 - val_accuracy: 0.3105
Epoch 7/50
31/31 [==============================] - 107s 3s/step - loss: 2.0507 - accuracy:
0.3818 - val_loss: 2.2116 - val_accuracy: 0.3504
Epoch 8/50
31/31 [==============================] - 114s 4s/step - loss: 1.8756 - accuracy:
0.4410 - val_loss: 2.1521 - val_accuracy: 0.3732
Epoch 9/50
31/31 [==============================] - 115s 4s/step - loss: 1.6290 - accuracy:
0.5046 - val_loss: 2.0674 - val_accuracy: 0.4131
Epoch 10/50
31/31 [==============================] - 111s 4s/step - loss: 1.4622 - accuracy:
0.5620 - val_loss: 2.1597 - val_accuracy: 0.4046
Epoch 11/50
31/31 [==============================] - 107s 3s/step - loss: 1.2908 - accuracy:
0.6136 - val_loss: 1.9202 - val_accuracy: 0.4473
Epoch 12/50
31/31 [==============================] - 107s 3s/step - loss: 1.0606 - accuracy:
0.6899 - val_loss: 2.0969 - val_accuracy: 0.4074
Epoch 13/50
31/31 [==============================] - 111s 4s/step - loss: 0.8697 - accuracy:
0.7371 - val_loss: 2.0074 - val_accuracy: 0.4302
(368, 36)
(368,)
Predicted: potato        (27) | actually: apple         ( 0)
Predicted: eggplant      (11) | actually: apple         ( 0)
Predicted: eggplant      (11) | actually: apple         ( 0)
Predicted: pomegranate   (26) | actually: apple         ( 0)
Predicted: lemon         (17) | actually: apple         ( 0)
Predicted: capsicum      ( 5) | actually: apple         ( 0)
Predicted: watermelon    (35) | actually: apple         ( 0)
Predicted: bell pepper   ( 3) | actually: apple         ( 0)
Predicted: grapes        (14) | actually: banana        ( 1)
Predicted: ginger        (13) | actually: banana        ( 1)
Predicted: ginger        (13) | actually: banana        ( 1)
Predicted: corn          ( 9) | actually: banana        ( 1)
Predicted: ginger        (13) | actually: banana        ( 1)
Predicted: sweetcorn     (31) | actually: banana        ( 1)
Predicted: lemon         (17) | actually: banana        ( 1)
Predicted: lemon         (17) | actually: banana        ( 1)
Predicted: pineapple     (25) | actually: banana        ( 1)
Predicted: pomegranate   (26) | actually: beetroot      ( 2)
Predicted: eggplant      (11) | actually: beetroot      ( 2)
Predicted: pomegranate   (26) | actually: beetroot      ( 2)
Predicted: chilli pepper ( 8) | actually: beetroot      ( 2)
Predicted: turnip        (34) | actually: beetroot      ( 2)
Predicted: pomegranate   (26) | actually: beetroot      ( 2)
```

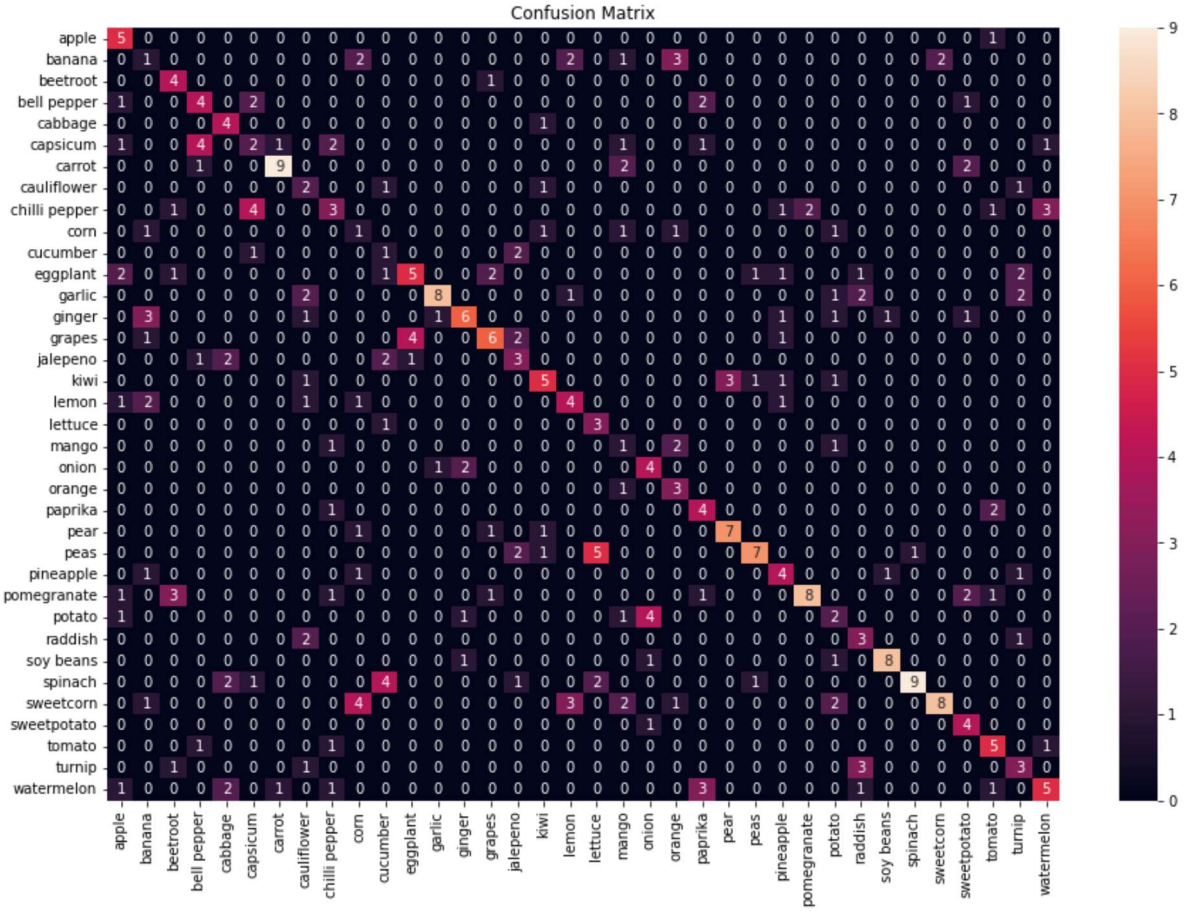
```
Predicted: capsicum      ( 5) | actually: bell pepper   ( 3)
Predicted: capsicum      ( 5) | actually: bell pepper   ( 3)
Predicted: tomato        (33) | actually: bell pepper   ( 3)
Predicted: capsicum      ( 5) | actually: bell pepper   ( 3)
Predicted: carrot        ( 6) | actually: bell pepper   ( 3)
Predicted: capsicum      ( 5) | actually: bell pepper   ( 3)
Predicted: jalepeno      (15) | actually: bell pepper   ( 3)
Predicted: jalepeno      (15) | actually: cabbage       ( 4)
Predicted: jalepeno      (15) | actually: cabbage       ( 4)
Predicted: watermelon    (35) | actually: cabbage       ( 4)
Predicted: watermelon    (35) | actually: cabbage       ( 4)
Predicted: spinach       (30) | actually: cabbage       ( 4)
Predicted: spinach       (30) | actually: cabbage       ( 4)
Predicted: chilli pepper ( 8) | actually: capsicum      ( 5)
Predicted: chilli pepper ( 8) | actually: capsicum      ( 5)
Predicted: spinach       (30) | actually: capsicum      ( 5)
Predicted: bell pepper   ( 3) | actually: capsicum      ( 5)
Predicted: bell pepper   ( 3) | actually: capsicum      ( 5)
Predicted: cucumber      (10) | actually: capsicum      ( 5)
Predicted: chilli pepper ( 8) | actually: capsicum      ( 5)
Predicted: chilli pepper ( 8) | actually: capsicum      ( 5)
Predicted: watermelon    (35) | actually: carrot        ( 6)
Predicted: ginger        (13) | actually: cauliflower   ( 7)
Predicted: kiwi          (16) | actually: cauliflower   ( 7)
Predicted: raddish       (28) | actually: cauliflower   ( 7)
Predicted: raddish       (28) | actually: cauliflower   ( 7)
Predicted: garlic        (12) | actually: cauliflower   ( 7)
Predicted: garlic        (12) | actually: cauliflower   ( 7)
Predicted: turnip        (34) | actually: cauliflower   ( 7)
Predicted: lemon         (17) | actually: cauliflower   ( 7)
Predicted: capsicum      ( 5) | actually: chilli pepper ( 8)
Predicted: capsicum      ( 5) | actually: chilli pepper ( 8)
Predicted: paprika       (22) | actually: chilli pepper ( 8)
Predicted: pomegranate   (26) | actually: chilli pepper ( 8)
Predicted: tomato        (33) | actually: chilli pepper ( 8)
Predicted: mango         (19) | actually: chilli pepper ( 8)
Predicted: watermelon    (35) | actually: chilli pepper ( 8)
Predicted: sweetcorn     (31) | actually: corn          ( 9)
Predicted: pineapple     (25) | actually: corn          ( 9)
Predicted: sweetcorn     (31) | actually: corn          ( 9)
Predicted: sweetcorn     (31) | actually: corn          ( 9)
Predicted: pear          (23) | actually: corn          ( 9)
Predicted: banana        ( 1) | actually: corn          ( 9)
Predicted: sweetcorn     (31) | actually: corn          ( 9)
Predicted: banana        ( 1) | actually: corn          ( 9)
Predicted: lemon         (17) | actually: corn          ( 9)
Predicted: spinach       (30) | actually: cucumber      (10)
Predicted: spinach       (30) | actually: cucumber      (10)
Predicted: lettuce       (18) | actually: cucumber      (10)
Predicted: spinach       (30) | actually: cucumber      (10)
Predicted: jalepeno      (15) | actually: cucumber      (10)
Predicted: spinach       (30) | actually: cucumber      (10)
Predicted: jalepeno      (15) | actually: cucumber      (10)
Predicted: cauliflower   ( 7) | actually: cucumber      (10)
Predicted: grapes        (14) | actually: eggplant      (11)
Predicted: grapes        (14) | actually: eggplant      (11)
Predicted: grapes        (14) | actually: eggplant      (11)
Predicted: grapes        (14) | actually: eggplant      (11)
Predicted: jalepeno      (15) | actually: eggplant      (11)
Predicted: onion         (20) | actually: garlic        (12)
Predicted: onion         (20) | actually: ginger        (13)
Predicted: onion         (20) | actually: ginger        (13)
Predicted: soy beans     (29) | actually: ginger        (13)
Predicted: potato        (27) | actually: ginger        (13)
```

```
Predicted: eggplant      (11) | actually: grapes        (14)
Predicted: pear          (23) | actually: grapes        (14)
Predicted: pomegranate   (26) | actually: grapes        (14)
Predicted: eggplant      (11) | actually: grapes        (14)
Predicted: beetroot      ( 2) | actually: grapes        (14)
Predicted: spinach       (30) | actually: jalepeno      (15)
Predicted: cucumber      (10) | actually: jalepeno      (15)
Predicted: peas          (24) | actually: jalepeno      (15)
Predicted: cucumber      (10) | actually: jalepeno      (15)
Predicted: peas          (24) | actually: jalepeno      (15)
Predicted: pear          (23) | actually: kiwi          (16)
Predicted: peas          (24) | actually: kiwi          (16)
Predicted: cabbage       ( 4) | actually: kiwi          (16)
Predicted: corn          ( 9) | actually: kiwi          (16)
Predicted: cauliflower   ( 7) | actually: kiwi          (16)
Predicted: banana        ( 1) | actually: lemon         (17)
Predicted: banana        ( 1) | actually: lemon         (17)
Predicted: garlic        (12) | actually: lemon         (17)
Predicted: sweetcorn     (31) | actually: lemon         (17)
Predicted: sweetcorn     (31) | actually: lemon         (17)
Predicted: sweetcorn     (31) | actually: lemon         (17)
Predicted: peas          (24) | actually: lettuce       (18)
Predicted: spinach       (30) | actually: lettuce       (18)
Predicted: peas          (24) | actually: lettuce       (18)
Predicted: peas          (24) | actually: lettuce       (18)
Predicted: spinach       (30) | actually: lettuce       (18)
Predicted: peas          (24) | actually: lettuce       (18)
Predicted: peas          (24) | actually: lettuce       (18)
Predicted: corn          ( 9) | actually: mango         (19)
Predicted: carrot        ( 6) | actually: mango         (19)
Predicted: potato        (27) | actually: mango         (19)
Predicted: sweetcorn     (31) | actually: mango         (19)
Predicted: sweetcorn     (31) | actually: mango         (19)
Predicted: capsicum      ( 5) | actually: mango         (19)
Predicted: orange        (21) | actually: mango         (19)
Predicted: carrot        ( 6) | actually: mango         (19)
Predicted: banana        ( 1) | actually: mango         (19)
Predicted: potato        (27) | actually: onion         (20)
Predicted: potato        (27) | actually: onion         (20)
Predicted: potato        (27) | actually: onion         (20)
Predicted: potato        (27) | actually: onion         (20)
Predicted: sweetpotato   (32) | actually: onion         (20)
Predicted: soy beans     (29) | actually: onion         (20)
Predicted: banana        ( 1) | actually: orange        (21)
Predicted: banana        ( 1) | actually: orange        (21)
Predicted: mango         (19) | actually: orange        (21)
Predicted: sweetcorn     (31) | actually: orange        (21)
Predicted: corn          ( 9) | actually: orange        (21)
Predicted: mango         (19) | actually: orange        (21)
Predicted: banana        ( 1) | actually: orange        (21)
Predicted: bell pepper   ( 3) | actually: paprika        (22)
Predicted: pomegranate   (26) | actually: paprika        (22)
Predicted: watermelon    (35) | actually: paprika        (22)
Predicted: watermelon    (35) | actually: paprika        (22)
Predicted: bell pepper   ( 3) | actually: paprika        (22)
Predicted: capsicum      ( 5) | actually: paprika        (22)
Predicted: watermelon    (35) | actually: paprika        (22)
Predicted: kiwi          (16) | actually: pear          (23)
Predicted: kiwi          (16) | actually: pear          (23)
Predicted: kiwi          (16) | actually: pear          (23)
Predicted: kiwi          (16) | actually: peas          (24)
Predicted: eggplant      (11) | actually: peas          (24)
Predicted: spinach       (30) | actually: peas          (24)
Predicted: grapes        (14) | actually: pineapple     (25)
```

```
Predicted: eggplant      (11) | actually: pineapple     (25)
Predicted: chilli pepper ( 8) | actually: pineapple     (25)
Predicted: ginger        (13) | actually: pineapple     (25)
Predicted: lemon         (17) | actually: pineapple     (25)
Predicted: kiwi          (16) | actually: pineapple     (25)
Predicted: chilli pepper ( 8) | actually: pomegranate   (26)
Predicted: chilli pepper ( 8) | actually: pomegranate   (26)
Predicted: sweetcorn     (31) | actually: potato        (27)
Predicted: ginger        (13) | actually: potato        (27)
Predicted: garlic        (12) | actually: potato        (27)
Predicted: corn          ( 9) | actually: potato        (27)
Predicted: sweetcorn     (31) | actually: potato        (27)
Predicted: soy beans     (29) | actually: potato        (27)
Predicted: mango         (19) | actually: potato        (27)
Predicted: kiwi          (16) | actually: potato        (27)
Predicted: turnip        (34) | actually: raddish       (28)
Predicted: turnip        (34) | actually: raddish       (28)
Predicted: garlic        (12) | actually: raddish       (28)
Predicted: watermelon    (35) | actually: raddish       (28)
Predicted: eggplant      (11) | actually: raddish       (28)
Predicted: turnip        (34) | actually: raddish       (28)
Predicted: garlic        (12) | actually: raddish       (28)
Predicted: pineapple     (25) | actually: soy beans     (29)
Predicted: ginger        (13) | actually: soy beans     (29)
Predicted: peas          (24) | actually: spinach       (30)
Predicted: banana        ( 1) | actually: sweetcorn     (31)
Predicted: banana        ( 1) | actually: sweetcorn     (31)
Predicted: carrot        ( 6) | actually: sweetpotato   (32)
Predicted: pomegranate   (26) | actually: sweetpotato   (32)
Predicted: pomegranate   (26) | actually: sweetpotato   (32)
Predicted: bell pepper   ( 3) | actually: sweetpotato   (32)
Predicted: ginger        (13) | actually: sweetpotato   (32)
Predicted: carrot        ( 6) | actually: sweetpotato   (32)
Predicted: watermelon    (35) | actually: tomato        (33)
Predicted: paprika       (22) | actually: tomato        (33)
Predicted: paprika       (22) | actually: tomato        (33)
Predicted: apple         ( 0) | actually: tomato        (33)
Predicted: chilli pepper ( 8) | actually: tomato        (33)
Predicted: pomegranate   (26) | actually: tomato        (33)
Predicted: garlic        (12) | actually: turnip        (34)
Predicted: garlic        (12) | actually: turnip        (34)
Predicted: eggplant      (11) | actually: turnip        (34)
Predicted: pineapple     (25) | actually: turnip        (34)
Predicted: eggplant      (11) | actually: turnip        (34)
Predicted: raddish       (28) | actually: turnip        (34)
Predicted: cauliflower   ( 7) | actually: turnip        (34)
Predicted: capsicum      ( 5) | actually: watermelon    (35)
Predicted: chilli pepper ( 8) | actually: watermelon    (35)
Predicted: tomato        (33) | actually: watermelon    (35)
Predicted: chilli pepper ( 8) | actually: watermelon    (35)
Predicted: chilli pepper ( 8) | actually: watermelon    (35)
missed 202 out of 368
Accuracy on testing data: 45.109 %
```


Accuracy
accuracy
val_accuracy
0.0
0.1
0.2
0.3
0.4
0.5
0.6
0.7
0
2
4
6
8
10
12


Loss
loss
val_loss
1.0
1.5
2.0
2.5
3.0
3.5
0
2
4
6
8
10
12


Confusion Matrix
apple
banana
beetroot
bell pepper
cabbage
capsicum
carrot
cauliflower
chilli pepper
corn
cucumber
eggplant
garlic
ginger
grapes
jalepeno
kiwi
lemon
lettuce
mango
onion
orange
paprika
pear
peas
pineapple
pomegranate
potato
raddish
soy beans
spinach
sweetcorn
sweetpotato
tomato
turnip
watermelon
0
1
2
3
4
5
6
7
8
9

SVM model - OBSOLETE AND ABANDONED

Model based on the support vector machine. We have included grid search (looking back, don't do this kids). This model takes approximately 8 hours to run on our 16GB ram machine with 8 cores CPU.

This model was abandoned after our initial tests with it, but we decided to keep it here for old time's sake and as a warning for those that shall oppose the one true ruling model, CNN.

BEWARE: run at your own risk.

In [ ]:
```
def model_lin_svm(X, y, X_val, y_val):
    X_new = []
    X_val_new = []

    print("Data loaded")
    for i in range(len(X)):
        X_new.append(X[i].flatten())
    for i in range(len(X_val)):
        X_val_new.append(X_val[i].flatten())

    X_new = np.array(X_new)
    X = X_new
    X_val_new = np.array(X_val_new)
    X_val = X_val_new
    print(X_val.shape, y_val.shape)


    param_grid = {'kernel': ['rbf', 'poly']}
    print("param_grid created")
    svc = svm.SVC(probability=True)
    model = GridSearchCV(svc, param_grid)
    print("model created")
    model.fit(X, y)
    return model
```

Testing the SVM model on the test data. It is required to reshape the images from 4D (numberOfImages, height, width, color) to 2D (numberOfImages, flattenedImage).

Predicted results are compared to the expected values and the incorrect predictions are displayed.

In [ ]:
```
def predict_SVM_results():
    X, y, X_val, y_val, X_test, y_test = get_data()

    X_test_new = []
    for i in range(len(X_test)):
        X_test_new.append(X_test[i].flatten())
    X_test_new = np.array(X_test_new)
    X_test = X_test_new

    m = model_lin_svm(X, y, X_val, y_val)

    pred = m.predict(X_test)
    counter = 0
```

```
    for i in range(len(pred)):
        j = pred[i]
        if y_test[i] != j:
            print("Predicted: {:13s} ({:2d}) | actually: {:13s} ({:2d})".format(rev
            counter += 1
    print("missed {:3d} out of {:3d}".format(counter, len(pred)))
    print("Accuracy on testing data: {:3.3f}%".format((1-counter/len(pred))*100))

#predict_SVM_results()
```