VIETNAMESE - GERMAN UNIVERSITY
DEPARTMENT OF COMPUTER SCIENCE

Introduction to Information Technology Project
Fruititionator: Fruit Detection and Nutrition Facts Retrieval Tool

Student 1: Đào Gia Hưng - 10423048
Student 2: Đoàn Bửu Phúc - 10423091
Student 3: Hoàng Đức Tài - 10423101
Student 4: Lê Huỳnh Văn Hóa - 10423041
Student 5: Vũ Đức Minh Hoàng - 10423042
Student 6: Nguyễn Hồ Nguyễn - 10423126

BÌNH DƯƠNG PROVINCE, 05/ 2024

Content

1 Introduction

As part of the Introduction to Information Technology module, this project aims to familiarize students with: concepts of the Internet of Things (IoT); processing Machine Learning (ML) models in Python; and publishing output results to an IoT server. The group's self-proposed objective was to define an image recognition tool that detects and classifies different types of fruits and their quality, using a camera feed input and a rehearsed model; then compiles the information with data from an online database and publishes it to an IoT server for storage and future processing. For this specific spinoff, Google Teachable Machine was selected as an image-based ML model training tool, Adafruit IO as a remote IoT server, which uses MQTT as a network transfer protocol, and the USDA FoodData Central[1][4] as database. The students have chosen a subset of 7 types of fruits from the USDA database on which to perform calculations. Extra features were implemented as needed (to be elaborated on below). The finished prototype presents potentially practical applications, such as integration with a diet or fitness program. The project's repository can be accessed at `https:/github.com/emberfox205/fruititionator`[1]. The following report details the steps and results of the project.

Below is an example of the algorithm's output to Adafruit IO. These include a class label for the detected object, a confidence score, a raster picture of the captured object, and nutrition values in JSON format.

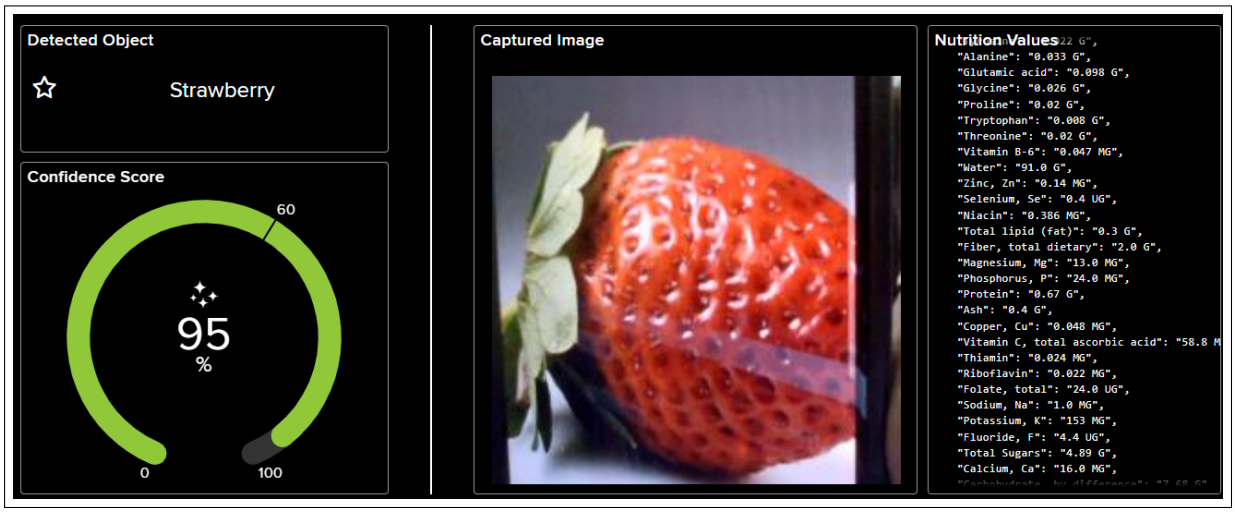

Figure 1: Output to Adafruit IO dashboard: Strawberry

[1]Foundation Foods: Data for food components including nutrients derived from analyses on individual samples of commodity/commodity-derived minimally processed foods with insights into variability. Foundation Foods highlight information on samples and acquisition details. Documentation and further details about Foundation Foods. API link: `https://api.nal.usda.gov/fdc/v1/foods/list`.

2 Teachable Machine Overview

Google Teachable Machine (TM) is an online tool that simplifies the process of training ML models. It abstracts the complexity of training algorithms for novice users and presents them with a graphical user interface (GUI). To use Google TM, a user needs only to provide classes and input imagery training data, the rest is left for Google TM to export a complete model. Various settings, such as batch size, epochs, and train sets, are available to users as toggles and sliders.

3 Dataset Collection

For Google TM to train the detection model, multiple fruit labels were assigned to corresponding classes and appropriate numbers of pictures were used to identify each class. The majority of these images were sourced from Shreya Maher's Fruits dataset[2] [2] and Mihai Oltean's Fruits-360 dataset[3] [3]. The rest of the project's image data was gathered on Google Images, for redundancy purposes.

Below are the 8 classes and their labels that have been implemented into Google TM's model. They include:

- Apple Granny Smith Raw
- Apple Red Delicious Raw
- Banana Raw
- Green Grape Raw
- Orange Raw
- Peach Raw
- Strawberry
- Nothing (exists as the background class)

3.1 Class 1: Apple Granny Smith Raw

Images are collected from different Internet sources and uploaded as samples for the Apple Granny Smith Raw class.

[2]Fruits Dataset (Images): Exploring 9 Popular Fruits through a Comprehensive Image Dataset.
[3]Fruits-360 dataset: A dataset of images containing fruits, vegetables and nuts. Version: 2020.05.18.0.

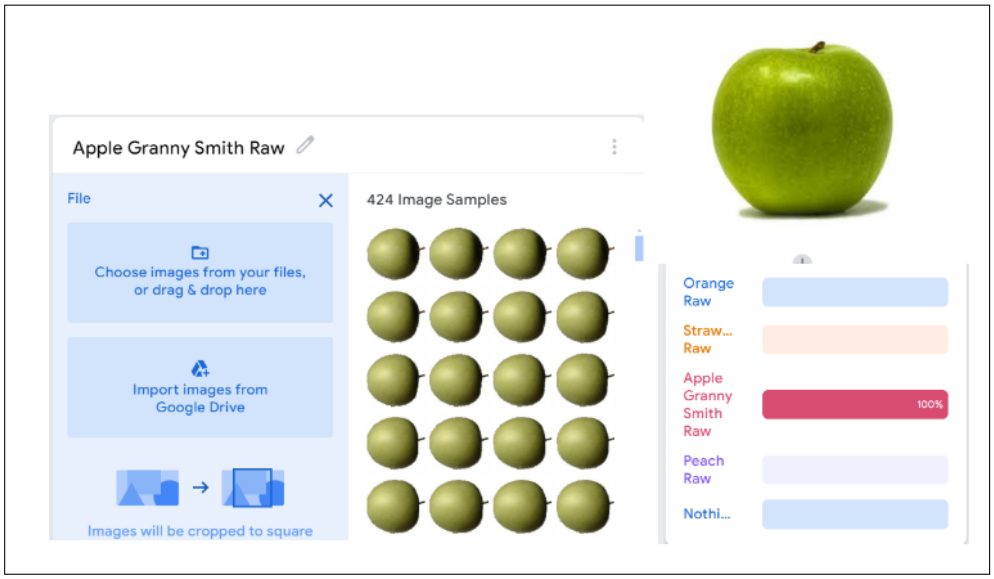

Figure 2: Training data for the Apple Granny Smith Raw class

The Granny Smith apple is characterized by its round shape and bright green skin, which may occasionally exhibit a red or pink blush. The flesh of the apple is white, sometimes with a slight green tint.

3.2 Class 2: Apple Red Delicious Raw

Images are collected from different Internet sources and uploaded as samples for the Apple Red Delicious Raw class.

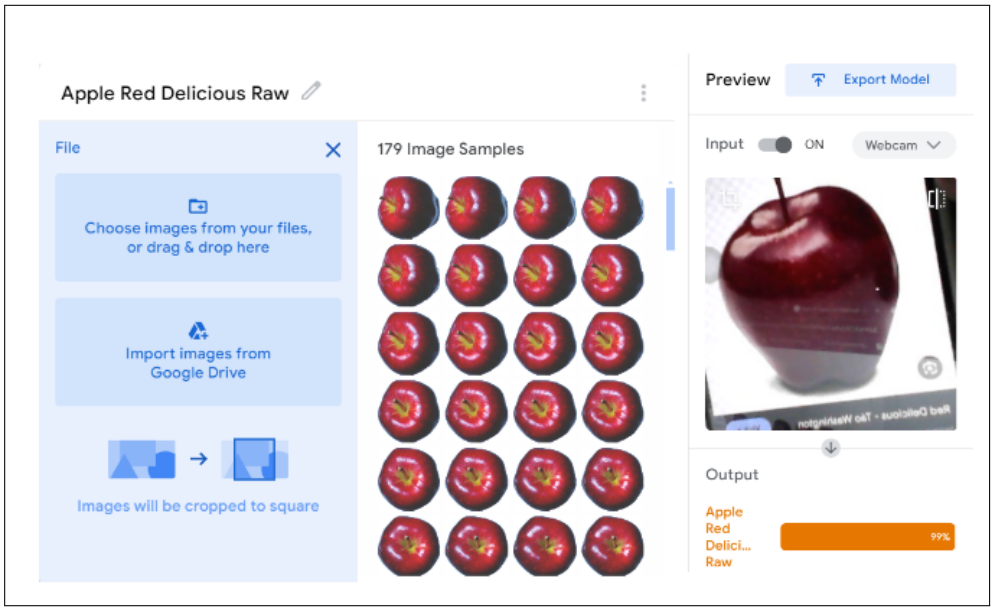

Figure 3: Training data for the Apple Red Delicious Raw class

The Red Delicious apple has deep red skin that is often shiny and smooth. The colour may

vary, ranging from a bright crimson to a darker hue. The shape of the fruit is tall and conical. Also, the flesh is of a bland white colour.

3.3 Class 3: Banana Raw

Images are collected from different Internet sources and uploaded as samples for class Banana Raw.

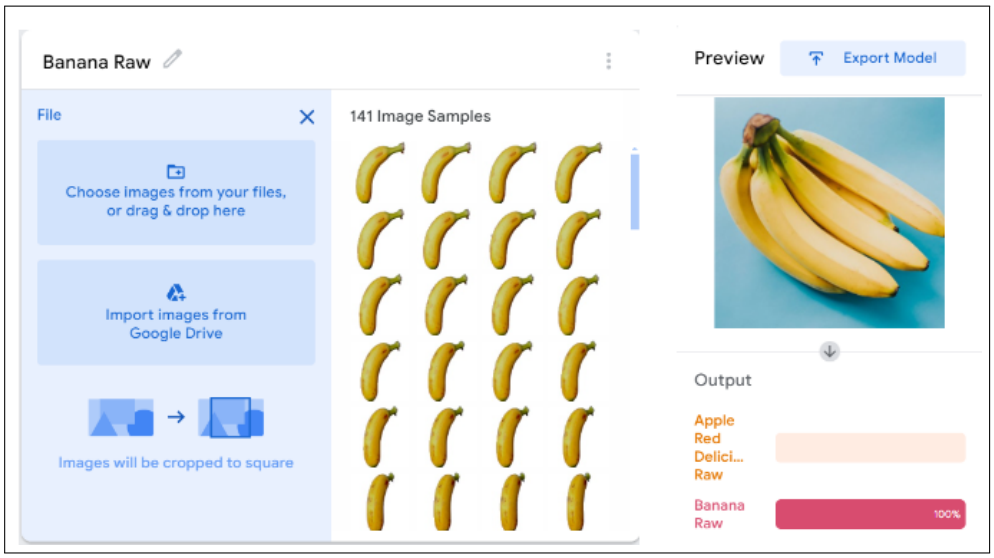

Figure 4: Training data for the Banana Raw class

A banana (which may be of various species) usually has an elongated and curved shape. The skin is smooth and glossy, whose colour transitions to yellow when ripe. Some bananas may develop small brown spots or patches. The inner flesh is usually a pale, yellowish-white colour with a soft texture.

3.4 Class 4: Green Grape Raw

Images are collected from different Internet sources and uploaded as samples for class Green Grape Raw.

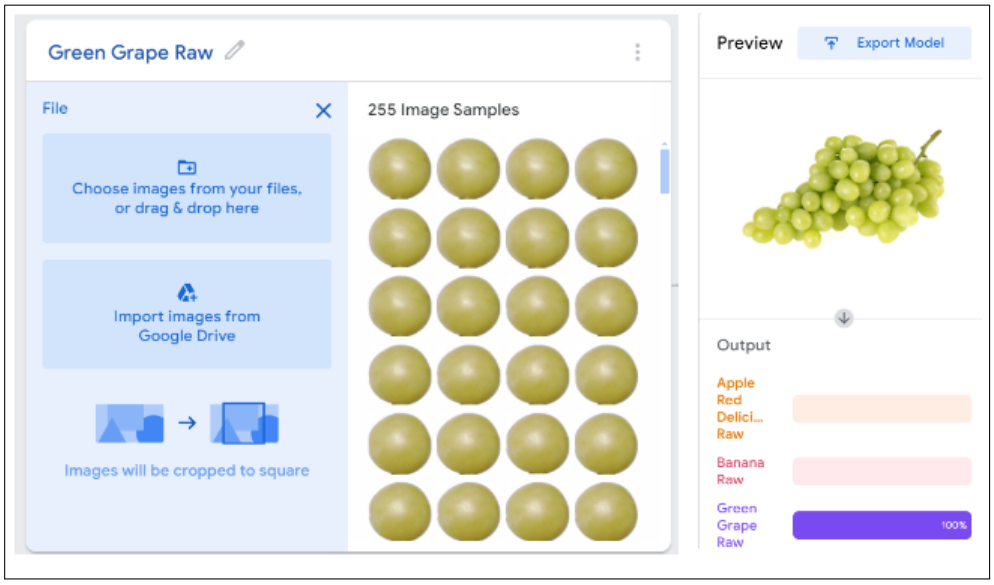

Figure 5: Training data for the Green Grape Raw class

Green grapes are small and spherical. Their skin is smooth and glossy, with a colour that can vary from a pale to a more vibrant shade of green. The interior flesh is translucent.

3.5 Class 5: Orange Raw

Images are collected from different Internet sources and uploaded as samples for class Orange Raw.

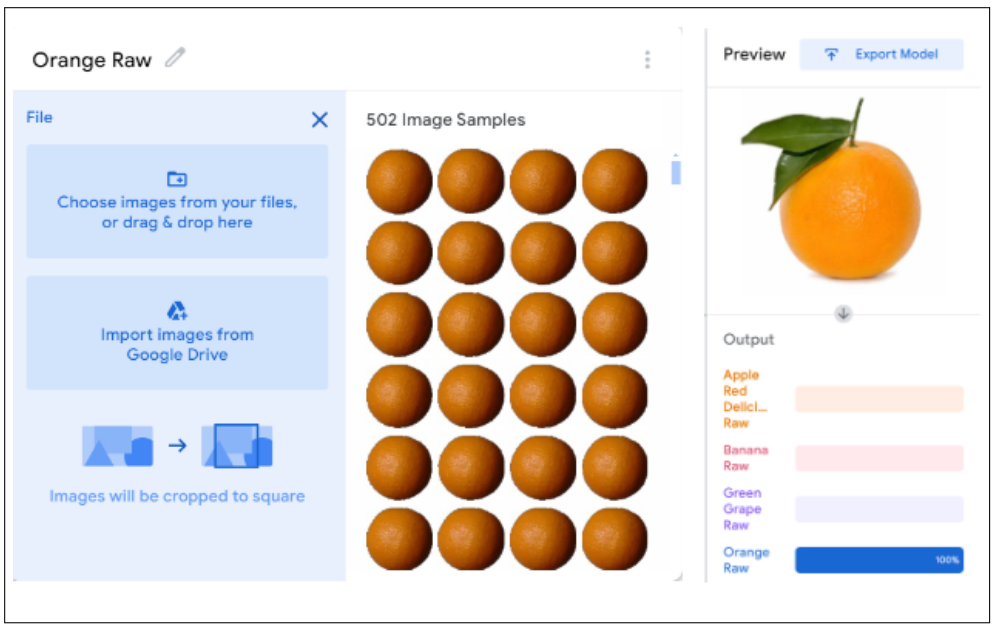

Figure 6: Training data for the Orange Raw class

Oranges are round and vibrant in colour. The skin of an orange is thick and has a pebbly

texture due to the presence of oil glands.

3.6 Class 6: Peach Raw

Images are collected from different Internet sources and uploaded as samples for class Peach Raw.

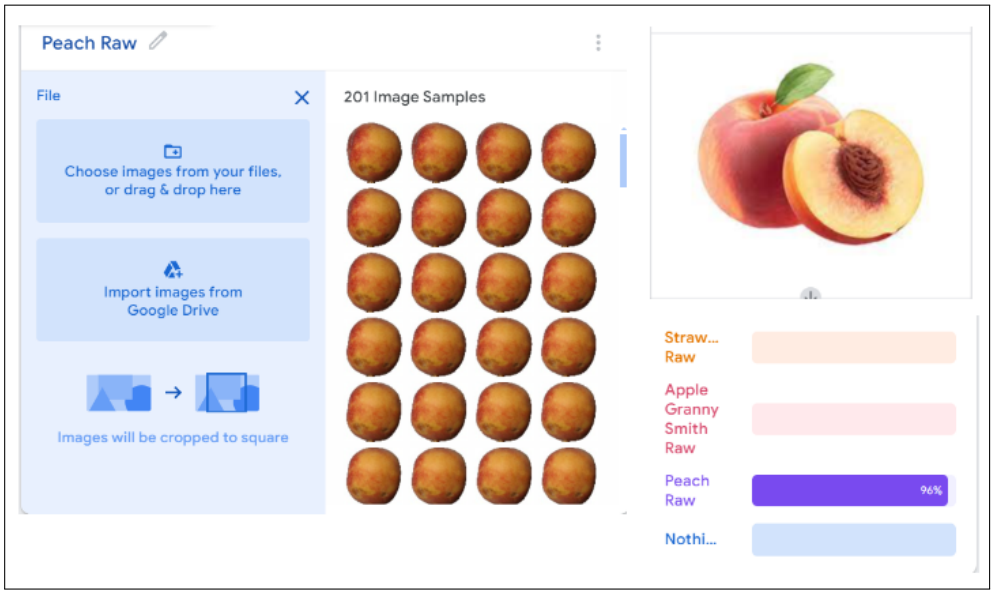

Figure 7: Training data for the Peach Raw class

A peach is round and has a distinctive central groove. The skin is velvety and can range in colour from pale yellow to deep orange. Because of the round shape and patches of red skin, which are also the characteristics of the Red Delicious apple, peach was chosen to test the model.

3.7 Class 7: Strawberry Raw

Images are collected from different Internet sources and uploaded as samples for class Strawberry.

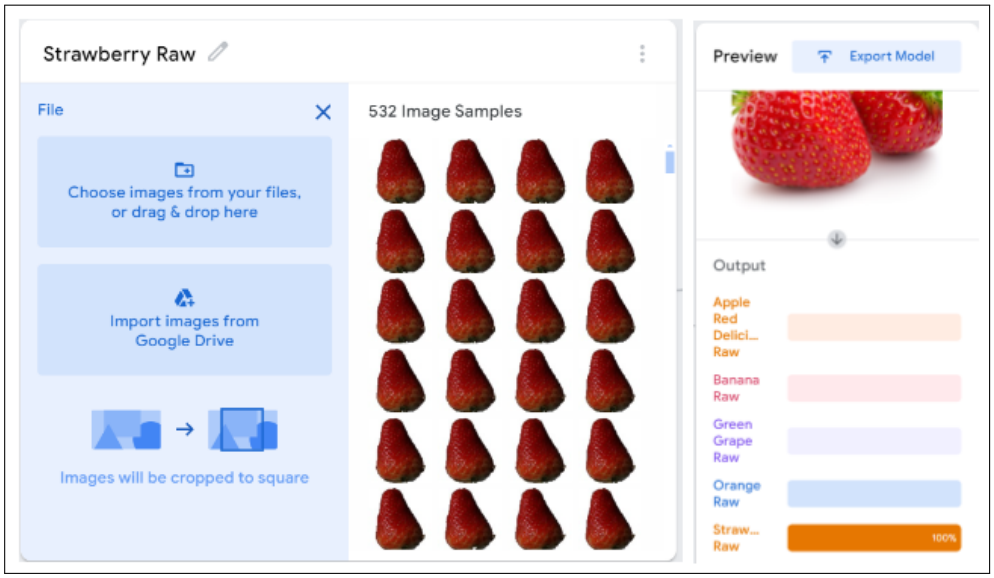

Figure 8: Training data for the Strawberry Raw class

Strawberries are commonly heart-shaped or conical. They are vibrantly red when ripe, while the surface is dotted with small, yellow seeds.

3.8 Class 8: Nothing

This part consists of images for background, which is normally required where the is nothing in front of the camera. The dataset mainly includes images of people sitting with their faces as the centrepiece. Nevertheless, it works for most scenarios where no fruit is present.

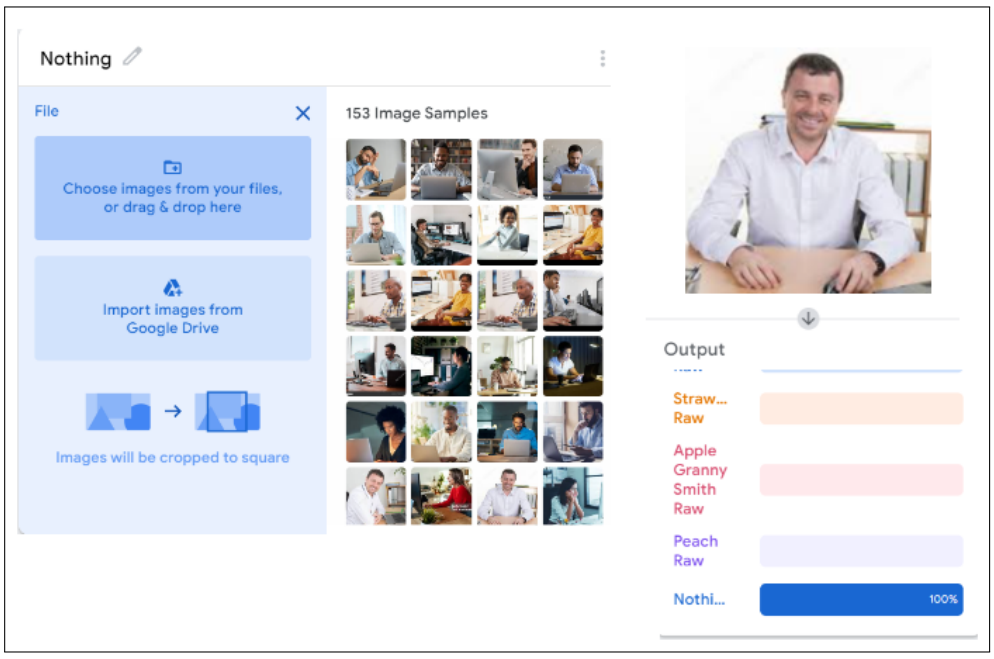

Figure 9: Training data for the Nothing class

4 Result

Input is taken from the camera feed of the device itself. Due to the lack of fruits available, samples were substituted with images displayed on smartphones. This was done instead of uploading files from the device to simulate imperfect lighting conditions and avoid having to deal with file paths.

The model detected correctly for an overwhelming majority of test cases. The confidence scores of all classes have been observed to range from 60% to 100%. Lower scores are attributed to complex backgrounds, unfavourable lighting angles and over-exposure of the inner flesh, the last of which is insufficiently included in the training set.

The model sometimes struggles with the Peach Raw class, just as speculated. The Nothing class, despite the inconsistent training images, performed extremely well.

Upon further inspection, this model can be improved by feeding more training images, especially cross-sectional views of different fruits, to Google TM and increasing the number of epochs per training session. Individual testing has shown that these improvisations refine output results to certain extents. This finding may prove to be useful in future projects.

4.1 Class-specific results

4.1.1 Class 1: Apple Granny Smith Raw

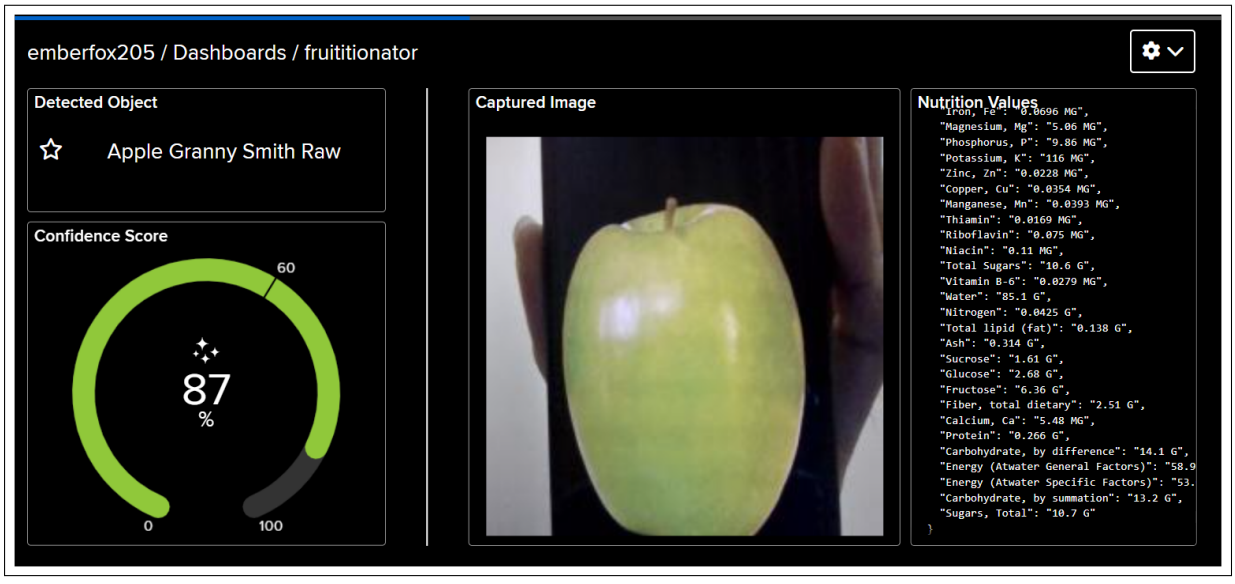

Figure 10: Result for the Apple Granny Smith Raw class

4.1.2 Class 2: Apple Red Delicious Raw

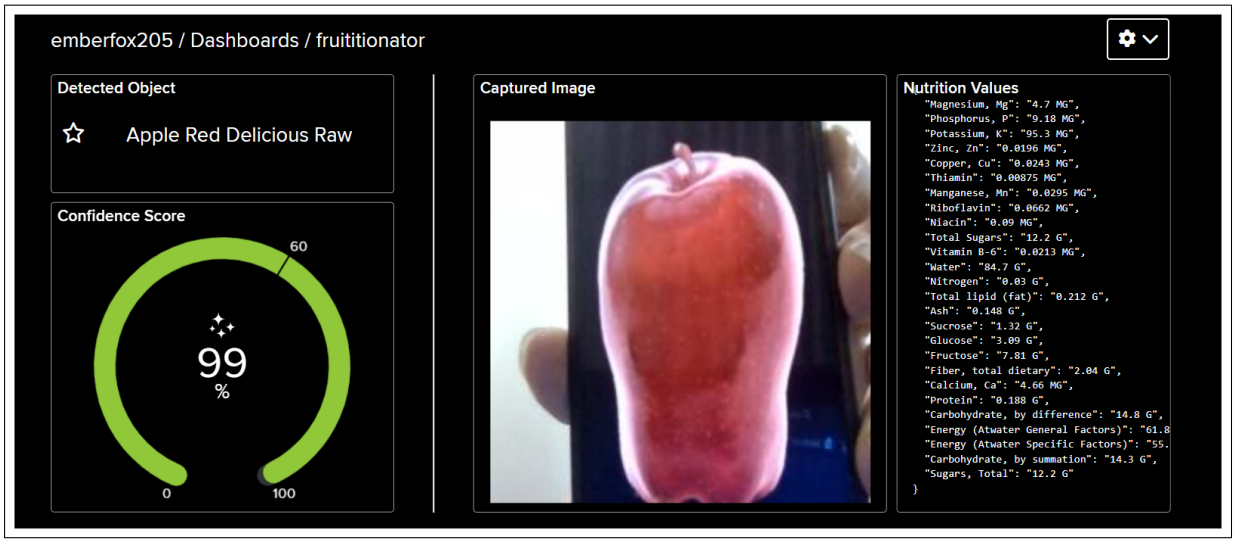

Figure 11: Result for the Apple Red Delicious Raw class

4.1.3 Class 3: Banana Raw

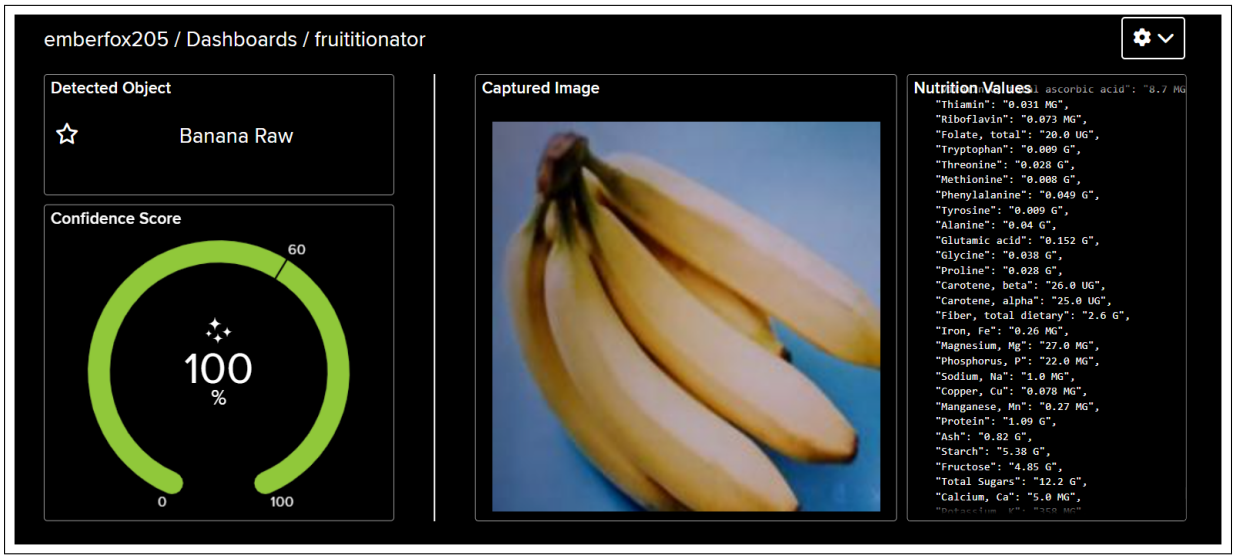

Figure 12: Result for the Banana Raw class

4.1.4 Class 4: Green Grape Raw

Figure 13: Result for the Green Grape Raw class

4.1.5 Class 5: Orange Raw

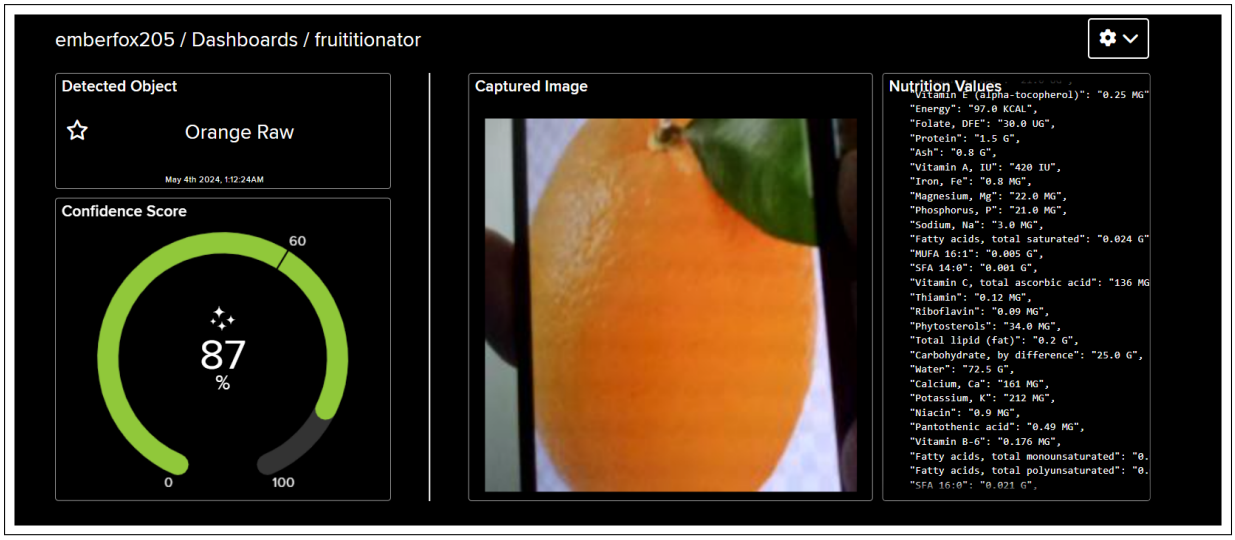

Figure 14: Result for the Orange Raw class

4.1.6 Class 6: Peach Raw

Figure 15: Result for the Peach Raw class

4.1.7 Class 7: Strawberry Raw

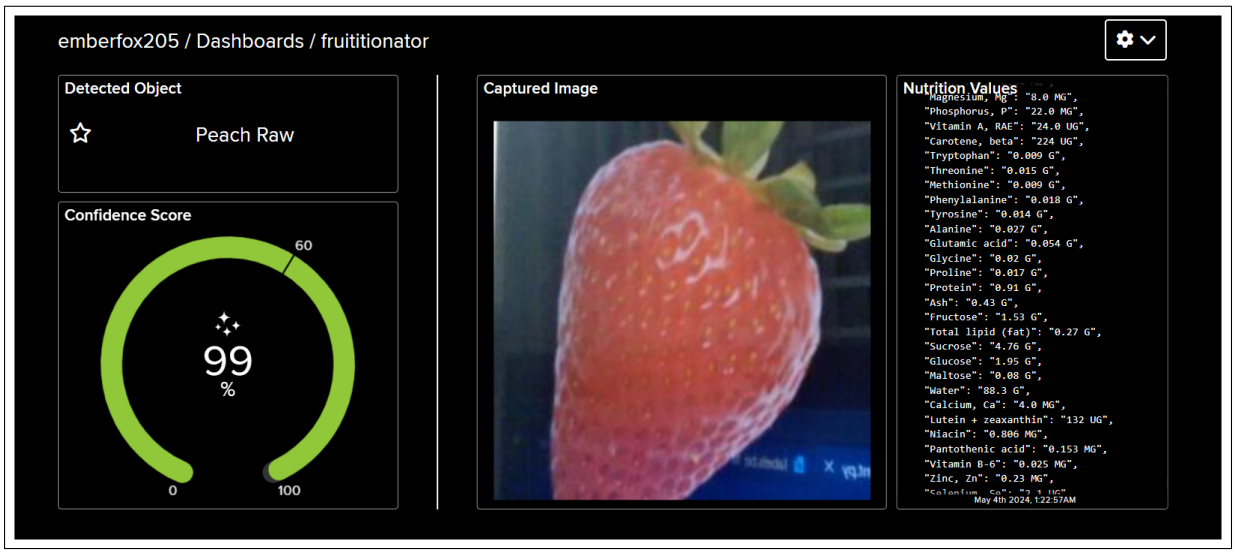

Figure 16: Result for the Strawberry Raw class

4.1.8 Class 8: Nothing

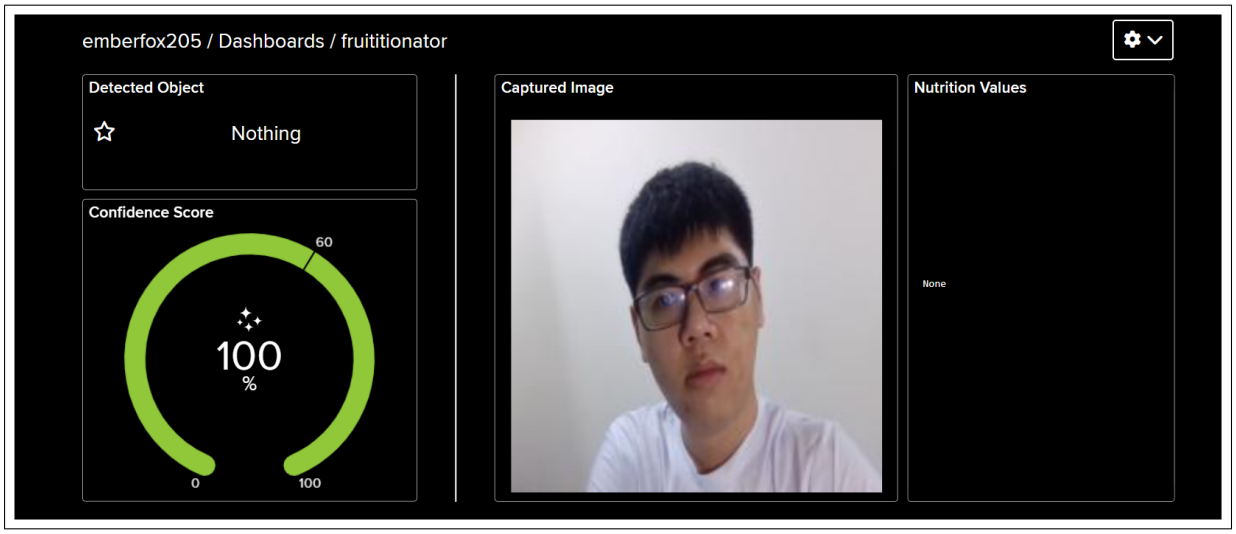

Figure 17: Result for the Nothing class

4.2 Adafruit IO Feeds

Examples of results with feed history enabled (below):

Created at	Value
2024/05/03 06:49:39PM	Banana Raw
2024/05/03 06:49:37PM	Nothing
2024/05/03 06:49:35PM	Banana Raw
2024/05/03 06:49:33PM	Nothing
2024/05/03 06:49:32PM	Peach Raw
2024/05/03 06:49:31PM	Banana Raw
2024/05/03 06:49:28PM	Green Grape Raw
2024/05/03 06:49:23PM	Nothing
2024/05/03 06:49:21PM	Banana Raw
2024/05/03 06:49:17PM	Nothing
2024/05/03 06:49:08PM	Nothing

Figure 18: Detected Object Feed

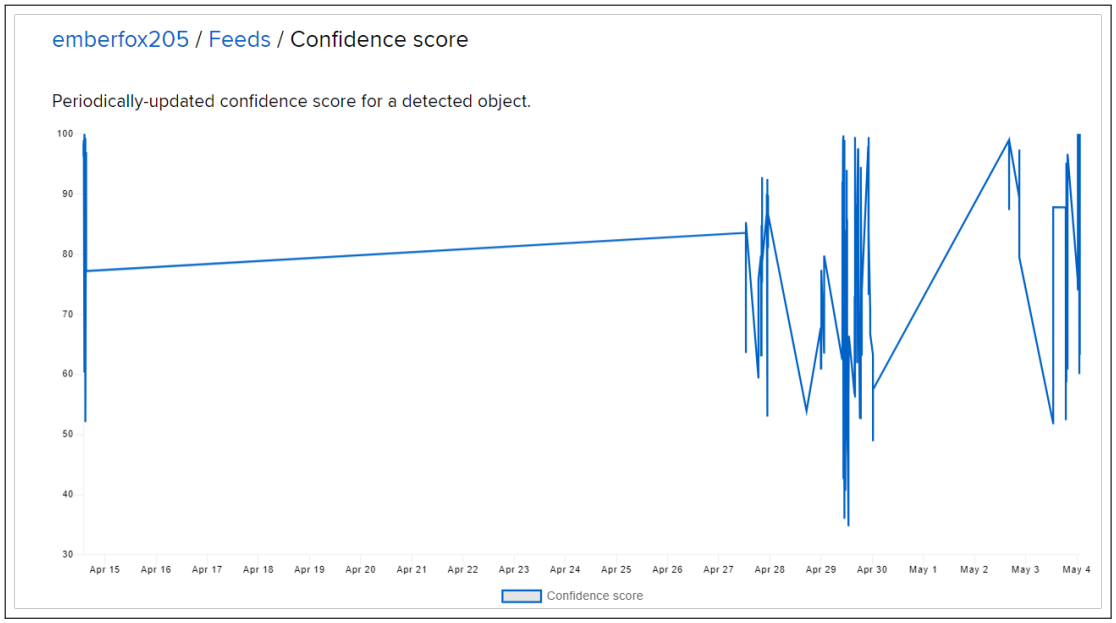

Figure 19: Confidence Score Feed

4.3 Visual Studio Code Terminal

Data printed on the terminal is more detailed with updates on all states of the program.

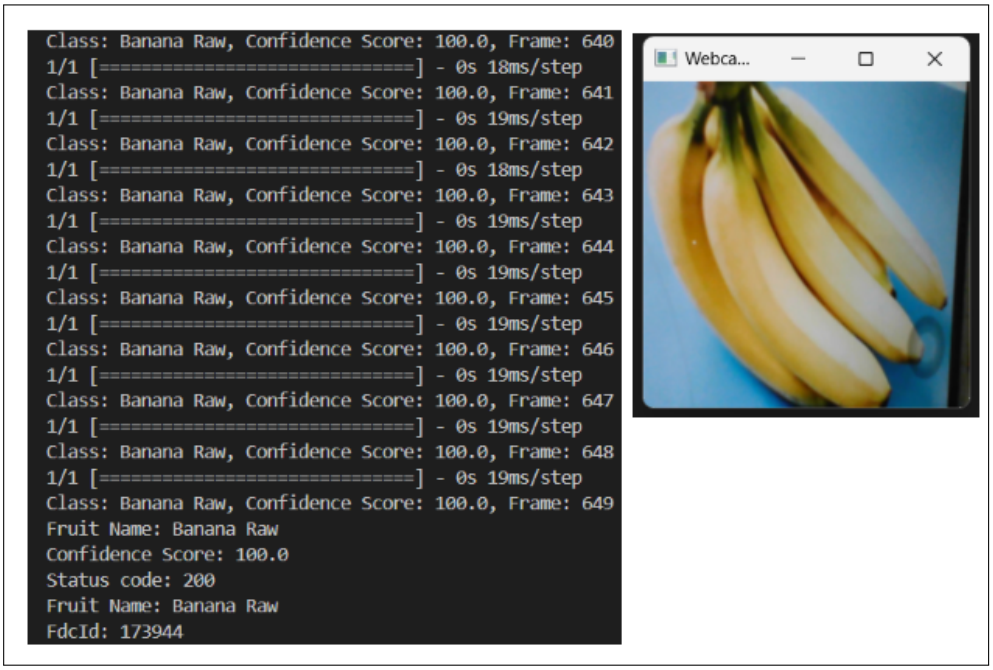

Figure 20: Terminal Outputs and Camera Feed

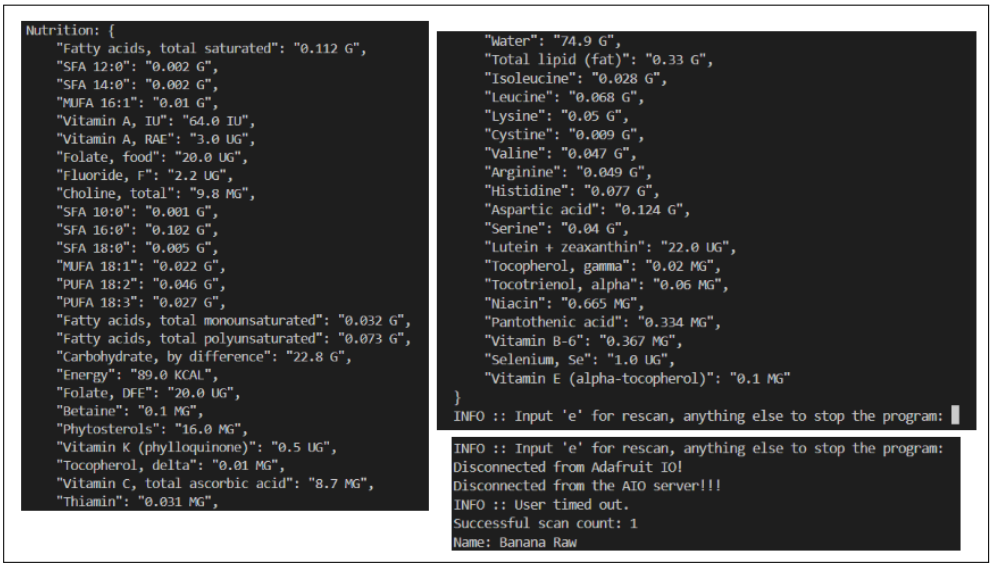

Figure 21: Terminal Outputs

5 Image Recognition Model

5.1 Python Source Code

The following is the Python code for the modified Fruit Recognition Model:

Listing 1: Fruit Recognition Model

```
import cv2
import numpy as np
from collections import Counter
from keras.models import load_model
from custom_classes import Detected_Object

def detect_fruit(client, DETECTED_OBJ_FEED_ID, CONFIDENCE_FEED_ID) -> ↩
    Detected_Object:
    # Load the model
    model = load_model("keras_model.h5", compile=False)

    # Load the labels
    with open("labels.txt", "r") as f:
        class_names = [line.strip() for line in f.readlines()]

    # Initialize camera
    camera = cv2.VideoCapture(0)

    detected_results = []
    frame = 0
    last_class = None

```

```
22      while True:
23          # Capture image from camera
24          image = camera.read()[1]
25
26          if image is None:
27              print("Error: Unable to capture image from camera.")
28              break
29
30          # Resize image to match model input size
31          image = cv2.resize(image, (224, 224), interpolation=cv2.↩
                INTER_AREA)
32          cv2.imshow("Webcam Image", image)
33
34          # Preprocess image for model input
35          image = np.asarray(image, dtype=np.float32).reshape(1, 224, ↩
                224, 3)
36          image = (image / 127.5) - 1
37
38          # Make prediction using the model
39          prediction = model.predict(image)
40          index = np.argmax(prediction)
41          class_name = class_names[index]
42          confidence_score = prediction[0][index]
43
44          print(f"Class: {class_name[2:]}, Confidence Score: {np.round(↩
                confidence_score * 100)}, Frame: {frame}")
45
46          # Limit publishing rate while keeping the camera feed smooth (↩
                about 20 FPS).
47          if frame % 30 == 0 and last_class == class_name[2:]:
48              client.publish(CONFIDENCE_FEED_ID, str(np.round(↩
                    confidence_score * 100)))
49          if frame % 15 == 0 and last_class != class_name[2:]:
50              client.publish(CONFIDENCE_FEED_ID, str(np.round(↩
                    confidence_score * 100)))
51              client.publish(DETECTED_OBJ_FEED_ID, class_name[2:])
52              last_class = class_name[2:]
53          frame += 1
54
55          # Store detected results
56          detected_results.append({
57              "fruit_name": class_name[2:],
58              "confidence_score": float(np.round(confidence_score * 100)↩
                    ),
59              "image": cv2.cvtColor(((image[0] + 1) * 127.5), cv2.↩
```

```
            COLOR_BGR2RGB).astype(np.uint8)  # Save the image for display
        })

        # Reduce the size of detected_results to its last 10 elements
        frame_threshold = 10
        if len(detected_results) > frame_threshold:
            detected_results = detected_results[-frame_threshold:]

        keyboard_input = cv2.waitKey(1)

        # Once the user quits, choose the most common object name, and from that choose the result with the highest confidence
        # 27 is the ASCII for the ESC key on your keyboard.
        if keyboard_input == 27:
            detected_results = detected_results[-frame_threshold:]
            freq_counter = Counter([result["fruit_name"] for result in detected_results])
            most_freq_results = [result for result in detected_results if result["fruit_name"] == (freq_counter.most_common(1)[0][0])]
            best_result = max(most_freq_results, key=lambda result: result["confidence_score"])
            detected_obj = Detected_Object(best_result["fruit_name"],
                                           best_result["confidence_score"],
                                           best_result["image"])
            client.publish(CONFIDENCE_FEED_ID, best_result["confidence_score"])
            client.publish(DETECTED_OBJ_FEED_ID, best_result["fruit_name"])

            print(detected_obj)

            # Release camera and close windows
            camera.release()
            cv2.destroyAllWindows()
            return detected_obj
```

5.2 Improvements

While most of the source code was readily provided by Google TM, additions of custom functions were made to make the model more compatible with the group's proposed features.

5.2.1 Functions and Objects

The fruit detection model is refactored to be inside the `detect_fruit()` function that returns a custom class. This makes it possible to pass the fruit's name and image to other modules. The `detect_fruit()` function also takes in an MQTT client instance and 2 feed IDs (`DETECTED_OBJ_FEED_ID` and `CONFIDENCE_FEED_ID`) as arguments to intermittently publish the name of the detected object and its corresponding confidence score.

5.2.2 Rate Limiter

As seen from lines 47 to 54, the program only publishes at most once every 30 frames if the same fruit item is yielded, and once every 15 frames if the item changes. From trials, it was concluded that this rate was the most appropriate, even with the standard publish rate of 30 publish attempts per minute. This logic does not delay the model with `time.sleep()` at all, thus keeping the camera viewfinder rolling at an acceptable frame rate.

5.2.3 Detection Result Selection

From lines 56 to 81, a code snippet is implemented so that a list of 10 latest detection results is constantly refreshed until the user stops the executing process. The best result is then chosen, based on its frequency and confidence score, and is published.

6 Extra Features

6.1 Image processing and publishing

To make use of the image object output by the AI, the project employs the Python Imaging Library (`PIL`) module for image manipulation, `io` module for binary data, `base64` module for encoding, and `MQTT` module for publishing to Adafruit IO feeds.

Listing 2: Image Processing and Publishing

```
1 import io
2 import base64
3 from PIL import Image
4
5 def resize_image(image, target_size=100, format="JPEG"):
6     # Calculate the target size in bytes
7     target_size_bytes = target_size * 1024
8
9     # Convert the image array to a PIL Image object
```

```
10      image_pil = Image.fromarray(image)
11
12      # Initialize the resizing factor
13      factor = 0.9
14
15      # Variable to store the encoded image
16      encoded_image = None
17
18      # Resize the image until it fits within the target size
19      while True:
20          with io.BytesIO() as output:
21              # Save the image to the output buffer
22              image_pil.save(output, format=format)
23              encoded_image = output.getvalue()
24
25          # Check if the encoded image size is within the target size
26          if len(encoded_image) <= target_size_bytes:
27              break
28
29          # Reduce the size of the image by the factor
30          width, height = image_pil.size
31          image_pil = image_pil.resize((int(width * factor), int(height
                * factor)))
32          factor *= 0.9
33
34      return image_pil
35
36  def encode_image(image_pil, format="JPEG"):
37      with io.BytesIO() as output:
38          # Save the image to the output buffer
39          image_pil.save(output, format=format)
40          encoded_image = output.getvalue()
41
42      # Encode the image as base64 and convert it to a string
43      base64_encoded_image = base64.b64encode(encoded_image).decode("utf
            -8")
44      return base64_encoded_image
45
46  def image_publisher(client, IMAGE_FEED_ID, ndarray_image, format="JPEG
        "):
47      resized_image = resize_image(ndarray_image, format=format)
48      img_str = encode_image(resized_image, format=format)
49
50      # Publish the encoded image to the client
51      client.publish(IMAGE_FEED_ID, img_str)
```

`resize_image(image, target_size=100, format="JPEG")`

This function call resizes an input image (as a NumPy array) to the target size (100KB) by iteratively reducing its dimensions by a constant factor. Once done, the function returns a PIL image object.

`def encode_image(image_pil, format="JPEG")`

This function encodes an input PIL image object into a base64 encoded string.

`def image_publisher(client, IMAGE_FEED_ID, ndarray_image, format="JPEG")`

This function preprocesses the image returned from the model by calling the aforementioned resizing and encoding functions before publishing it to an image feed.

6.2 Fetching nutrition data with API

This project can also retrieve, format and publish the nutrition data of the fruit item detected by the AI model. Queries are made to the USDA FoodData Central (FDC). The primary Python packages used are `requests`, which helps send POST requests and `json`, which formats the API data into an Adafruit-IO-compatible data type. In addition, `dotenv` and `os` packages are utilised to hide the API key.

Listing 3: API Call and Formatting

```
import json
import os
import requests
from dotenv import load_dotenv
from typing import Union
from custom_classes import Fruit_Nutrition

load_dotenv()
API_KEY = os.getenv('API_KEY')


# Request and receive nutrition data
def get_api(client, NUTRITION_FEED_ID, keyword: str) -> Union[Fruit_Nutrition, None]:
    url = "https://api.nal.usda.gov/fdc/v1/foods/list"
    data = {
        "generalSearchInput": keyword,
```

```
17          "dataType": ["Foundation", "SR Legacy"],
18          "requireAllWords": True,
19          "foodCategory": "Fruits and Fruit Juices",
20          "sortBy": "score",
21          "sortOrder": "desc"
22      }
23
24      response = requests.post(url, json=data, params={"api_key": ↩
            API_KEY})
25
26      # Print the status code of the response
27      print(f"Status code: {response.status_code}")
28
29      if response.status_code == 200:
30          res_json: list = list(response.json())
31          if res_json:
32              # Format and return the received data
33              return format_data(client, NUTRITION_FEED_ID, res_json[0],↩
                    keyword)
34          else:
35              # Publish "None" to the nutrition feed if no data is found
36              client.publish(NUTRITION_FEED_ID, "None")
37              return None
38      else:
39          return None
40
41  # Format the received data for the CLI and Adafruit IO dashboard
42  def format_data(client, NUTRITION_FEED_ID, response: list, fruit_name:↩
        str) -> Fruit_Nutrition:
43      nutrition = {}
44      for nutrient in response["foodNutrients"]:
45          name = nutrient["name"]
46          unit = nutrient["unitName"]
47          amount = nutrient["amount"]
48          if amount > 0 and unit != "kJ":
49              if name not in nutrition.keys():
50                  nutrition[name] = f"{amount} {unit}"
51
52      # Create a Fruit_Nutrition object with the formatted data
53      fruit_nutrition = Fruit_Nutrition(fruit_name, response["fdcId"], ↩
            nutrition)
54
55      # Convert the nutrition data to JSON format
56      nutrition_json = json.dumps(nutrition, indent=2)
57
```

```
58      # Publish the nutrition data to the NUTRITION_FEED_ID
59      client.publish(NUTRITION_FEED_ID, nutrition_json)
60
61      return fruit_nutrition
```

`def get_api(client, NUTRITION_FEED_ID, keyword: str)`

This function first sends a POST request to the database and manages the procedures following a response. The use of POST instead of GET is to enable complex queries with filters and conditions.
The response is in JSON format but can be converted into Python code, in which case it becomes a dictionary wrapped inside a list.

`def format_data(client, NUTRITION_FEED_ID, response: list, fruit_name: str)`

If the request is successful (status code 200), this function extracts the ID and the nutrition values of the fruits, which are then formatted and compiled into a class called Fruit_Nutrition. Attributes within this class are to be printed to the terminal and published onto a feed called Nutrition Values.

6.3 CLI and MQTT Client

This main file serves as the central hub for managing all the imported modules, credentials as well as the AI model. Additionally, it contains codes for a Command Line Interface (CLI) and connections with Adafruit IO.

Listing 4: CLI and MQTT Client

```
1  import os
2  import sys
3  import time
4  from Adafruit_IO import MQTTClient
5  from dotenv import load_dotenv
6  from api_call import get_api
7  from fruit_detector import detect_fruit
8  from image_processor import image_publisher
9
10 load_dotenv()
11
12 # Load AIO_USERNAME and AIO_KEY from environment variables
13 AIO_USERNAME = os.getenv('AIO_USERNAME')
```

```
14 AIO_KEY = os.getenv('AIO_KEY')
15
16 # Define feed IDs
17 CONFIDENCE_FEED_ID = "confidence-score"
18 DETECTED_OBJ_FEED_ID = "detected-obj"
19 NUTRITION_FEED_ID = "nutrition-values"
20 IMAGE_FEED_ID = "captured-image"
21
22 def connected(client):
23     # Callback function when connected to the AIO server
24     print("Connected to the AIO server!!!!")
25
26 def disconnected(client):
27     # Callback function when disconnected from the AIO server
28     print("Disconnected from the AIO server!!!")
29     sys.exit(1)
30
31 def message(client, feed_id, payload):
32     # Callback function when a message is received
33     print("Received: " + payload)
34
35 client = MQTTClient(AIO_USERNAME, AIO_KEY)
36 client.on_connect = connected
37 client.on_disconnect = disconnected
38 client.on_message = message
39
40 def main():
41     scan_count = 0
42     fruits_detected = []
43     while True:
44         # Call the program's primary functions
45         detected_fruit = detect_fruit(client, DETECTED_OBJ_FEED_ID, ↩
               CONFIDENCE_FEED_ID)
46         time.sleep(0.5)
47         image_publisher(client, IMAGE_FEED_ID, detected_fruit.image)
48         fruit_nutrition = get_api(client, NUTRITION_FEED_ID, ↩
               detected_fruit.name)
49
50         if fruit_nutrition is None:
51             print("WARNING :: Unable to get data.")
52         else:
53             scan_count += 1
54             fruits_detected.append(fruit_nutrition.name)
55         print(fruit_nutrition)
56
```

```
57          # Handle retries and exit
58          prompt = input("INFO :: Input 'e' for rescan, anything else to stop the program: ")
59          if prompt == 'e':
60              print("__RESTARTING__")
61              time.sleep(0.5)
62          else:
63              client.disconnect()
64              time.sleep(0.01)
65              print("INFO :: User timed out.")
66              print(f"Successful scan count: {scan_count}")
67              for fruit in fruits_detected:
68                  print(f"Name: {fruit}")
69              break
70
71  if __name__ == '__main__':
72      client.connect()
73      client.loop_background()
74      main()
```

This Command Line Interface (CLI) upon execution will continuously scan for fruits, detect them and capture images. It employs `Adafruit_IO` for MQTT communications, along with the `api_call`, `fruit_detector`, and `image_processor` modules for fruit detection, API calls and image processing. Users can choose to initiate rescans or terminate the program.

7 Conclusion

With this project completed, the team has achieved the goal of experimenting with Google Teachable Machine and Adafruit IO's MQTT protocol in aiding its members' learning in the field of Internet of Things. Students were able to utilize the opportunity to tackle tasks that produce tangible and applicable results. Ultimately, knowledge in various technologies relating to Artificial Intelligence and the Internet of Things serves as an enablement for future endeavours. Also, experience in team coordination, version control and debugging has been beneficial for participants of this project.

References

[1] emberfox205, "Fruititionator: A python project to detect fruits using computer vision and return their nutrition values.," GitHub, https://github.com/emberfox205/fruititionator (accessed May 5, 2024).

[2] S. Maher, "Fruits dataset (images)," Kaggle, https://www.kaggle.com/datasets/shreyapmaher/fruits-dataset-images (accessed May 4, 2024).

[3] M. Oltean, "Fruits-360 dataset," Kaggle, https://www.kaggle.com/datasets/moltean/fruits (accessed May 4, 2024).

[4] U.S. Department of Agriculture. "Fooddata Central Search Results," FoodData Central, https://fdc.nal.usda.gov/fdc-app.html#/ (accessed May 4, 2024).