# Smart Waste Classification System using Machine Learning

Rutva Argarde, Rani Momtaz, Richard Perocho, Angela Villadiego

School of Applied Computing, Faculty of Applied Science and Technology,
Sheridan College, Ontario, Canada

**Abstract.** This project presents a smart waste sorting system that uses computer vision to classify waste into various categories and indicate to users whether it belongs in the trash (landfill), recycling, or compost bins. The physical system integrates weight sensors, a camera, LED lights, and a Raspberry Pi 5 for a comprehensive system. Using data from public datasets *TrashNet* and *CompostNet*, as well as manually-taken waste image samples, a custom machine learning model was built to classify images of waste into various categories such as plastic, glass, trash, paper, cardboard and more. The model was created with PyTorch and runs completely on-device on the Raspberry Pi allowing for fast classification speeds.

**Keywords:** Ubiquitous computing, smart waste management, computer vision, environmental sustainability, machine learning

## 1 Introduction

### 1.1 Purpose of Proposed Research

This project was developed with the purpose of enabling more consistent and efficient waste management in consumer homes. Visual analysis of waste products would analyze their structure and sort them into the 3 main trash categories: landfill, recycling and compost. Further research into this aspect of civilian life would reduce the chance of misidentification and enhance the positive environmental impact of proper waste management.

### 1.2 Nature of Proposed Research

The datasets used to train the model are the *TrashNet* and *CompostNet* datasets, and the machine learning tool used to classify trash items is PyTorch. Data from both sets were combined with manually-taken photos to form the training set for analyzing trash items. Once trained, the model is tested with manually-taken photos of a small handful of items to measure judgement accuracy. By gaining a thorough understanding of the appearance of various landfill, recycling and compost trash items, we aimed to train the

machine learning model to recognize those same patterns in trash items scanned by consumers. The scanning part of the process is performed via a hardware system that utilizes context awareness to detect trash items and then sends the relevant data to the machine learning model for analysis.

## 2 Literature Review

### 2.1 Foundational Approaches to Automated Waste Classification: Reviewing Yang and Thung's Contributions

Prior research by Yang and Thung explored classifying single-object waste images into recycling categories such as glass, paper, metal, plastic, cardboard, and trash. In their study, a custom dataset of approximately 2,400 images was developed. Using this dataset, their work compared support vector machines (SVMs) with scale-invariant feature transform (SIFT) features and convolutional neural networks (CNNs). The authors chose SVMs as it had better initial classification algorithms and was not as complicated when compared to CNNs [1].

A takeaway from their research is the challenge of achieving robust performance with limited, specialized data. Although their use of SIFT descriptors and bag-of-features modelling proved effective for SVM classification, a lack of hyperparameter tuning for CNN resulted in its poor performance [1]. The authors also stressed the importance of more diverse and larger datasets, as well as refined training strategies to allow deep learning strategies that exceed simpler training methods.

The findings of this paper demonstrate classification performance with their dataset size, and the use of SVM. It also highlights the need for improved data quality, careful model selection, and continued experimentation with architectures and parameters. Through Yang and Thung's exploration of SVM and CNN models, their dataset provides a strong foundation for this research. Through the help of their dataset of waste images, this research project was able to develop an on-device classification model on Raspberry Pi. This paper extends to the foundation established by Yang and Thung, with a more practical implementation approach in smart waste classification systems.

### 2.2 Meal Waste Image Classification: Reviewing Frost et al.'s Contributions

The research paper "CompostNet: An Image Classifier for Meal Waste" discusses a solution for waste classification which has become a significant problem as the authors highlight the confusing need of "consumers to decipher instructional text, icons, or images in order to sort their waste accurately" [2]. CompostNet built upon Thung and Yang's TrashNet dataset, which originally consisted of 2527 images and augmented the dataset by adding 224 new images, particularly focusing on compostable waste [2]. CompostNet explored two distinct neural network architectures to base their

classification system: a transfer learning model utilizing MobileNet (Version A) and a custom convolutional neural network (CNN) (Version B). A key takeaway from their research is the significant improvement in classification accuracy by leveraging transfer learning and the paper describes as well as compared the findings from both machine learning architectures succinctly.

Version A, built on the pre-trained MobileNet, achieved a test accuracy of 77.3%, outperforming the 75% accuracy of TrashNet's CNN. Version B which was trained exclusively on the augmented dataset achieved a lower accuracy of 22.7%, because of overfitting, and the authors draw an important conclusion backed by their research – the importance of robust model architectures and the benefits of transfer learning particularly with limited and specialized datasets [2].

The authors developed a prototype iOS application for the deployment of their research. The solution developed by Frost et.al., highlights the need for a more comprehensive dataset and improved hyperparameters of machine learning models to improve accuracy of their findings [2]. The paper extends the solution described in TrashNet by Yang and Thung and refines it using a more robust architecture and aims to highlight the need for research and improved datasets in waste classification.

# 3 Methodology

The following section will outline the methodologies used in the design and development of the project. This will include a summary of events during project inception, as well as the implementation of the physical system and the creation of the ML model.

## 3.1 Project Inception

The inception phase of the project consisted of a few sessions, the first of which was a session to conceptualize the project, understand its goals, its scope, and potential user flows. During these sessions, surface-level research gathering was performed to understand the current state of research into this subject as well as understand the availability of resources such as datasets, research articles, and hardware.

From these initial phases, a project inception document was written. This document outlined the goals of the project, a diagram showcasing the expected appearance and functionality of the full system, and a list of hardware expected to be used during its creation.

## 3.2 System Hardware Design

After the inception of the full system, ideation regarding the minimum viable product (MVP) requirements commenced. Based on available time and resources, the decision was made to simplify the MVP to use a camera, weight sensors, and Raspberry Pi.

**Table 1.** List of hardware elements used for the system

| Hardware item | Count | Function |
|---|---|---|
| Load cell weight strain sensors (50kg) | 4 | Weight sensing |
| HX711 Analog-to-Digital (AD) module | 1 | Converting load cell output to digital signals |
| Logitech QuickCam Messenger Webcam | 1 | Image capturing |
| Raspberry Pi 5 | 1 | Central control and ML model usage. |
| LED (blue, red, green) | 3 | Indicate correct waste bin |
| Resistors | 3 | Limiting current to LEDs |

With the list of minimum required hardware finalized, the wiring and construction was ready to begin. The load cells could be used individually, wired as a pair, or by wiring all four together. To achieve more accurate sensing, as well as to have a more stable base, all four sensors were used together. To connect header pins to the HX711 module, the load cells to each other, and the load cells to the module, the pins and wires were soldered together and connections protected using electrical tape. The module was then connected directly to the Raspberry Pi 5 using female-to-female connectors (see Figure 1 for full wiring diagram).

Using a breadboard, the three LEDs were connected to the Raspberry Pi's GPIO and ground pins with resistors for correct voltage. Finally, the webcam was connected to the Raspberry Pi using its USB connector.

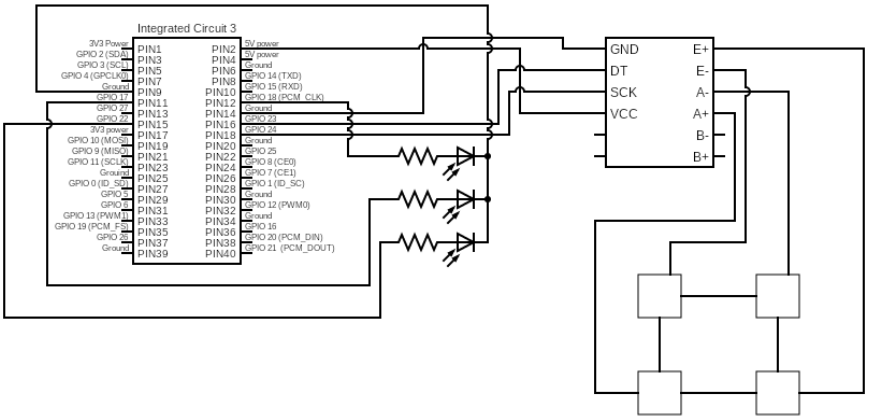


**Fig. 1.** Wiring diagram of the system

During and immediately following wiring, a multimeter was used to ensure proper connections throughout the system. Small Python tests were also written in order to ensure and test that the components were working correctly, including image capturing, LED control, and weight sensing.

### 3.3 Machine Learning Model

After the wiring was complete, it was time to create the ML model to convert images taken from the camera into meaningful classes. The most accessible and high-quality datasets found for this were *TrashNet* [3] and *CompostNet* [4]. While *TrashNet* contains over 2,500 images pre-labelled as trash, plastic, metal, paper, cardboard, and glass, it did not contain any labelled images for compost. For this, the addition of datasets A and B from *CompostNet* was necessary, which provided 175 image samples of compost. This resulted in around 2,700 images spread across 7 categories: trash, plastic, metal, paper, cardboard, glass, and compost.

While the *CompostNet* GitHub included a ready-built ML model for classification, to add the already small body of existing trash classification systems, a different method was used. Following a similar method to another trash-classification system that did not include compost created by Aadhav Vignesh on Kaggle [5], transfer learning on a pre-trained copy of ResNet50 – a 50-layer convolutional neural network (CNN) well-known for image classification – was used in the creation of the waste classification model.

The model was created and trained in a Google Colab document [6]. Making use of PyTorch, the combined dataset was analyzed, split into test, train, and validate sets, and the model created. The final layer of ResNet50 was modified to provide the correct number of outputs to match the 7 total classes in the dataset. The model was then trained in 8 epochs with a batch size of 32.

The initial training showed promising results, achieving around 95% accuracy on the testing set. With this confidence, some photos found online and taken with a cell-phone camera of compost and other trash items were fed to the model. However, the majority of those additional photos were incorrectly classified. In particular, the small number and variety of compost images caused images of other kinds of compost – such as banana peels – to be incorrectly classified as “glass”, the category containing the most sample images. This misclassification was also potentially due to differences in lighting, having background colors not being perfectly white, and the fact that the images in the *TrashNet* dataset contained mostly close-up images of trash, whereas the images planned for this waste classification system would have images of the entire item across a white background (similar to the images from *CompostNet*).

Predicting the accuracy of the model would only get worse with the low-quality images produced by the webcam of the system, around 50 images of trash, compost, glass, and plastic items were taken using the webcam in front of a white background and added to the training dataset. This greatly improved the quality of the model, achieving around 85% accuracy in practice. Exact data will be presented later in *Findings* section.

At this point, the model was considered finalized for the MVP. It was saved, along with a Pickle file of the desired classes, and downloaded onto the Raspberry Pi to be loaded and used on device from this point forward.

### 3.4 System Software

With the model and the hardware completed, the on-device software was the last portion to tie the system together. As mentioned earlier, smaller portions of Python code were created separately to test and investigate the tools needed to control each of the three main hardware components: the LEDs, the camera, and the weight sensors.

The LEDs were the simplest components to integrate with Raspberry Pi. A Python library called *gpiozero* provided functions to turn an LED on and off intuitively by simply providing a GPIO pin number for each LED and then calling the *.on()* and *.off()* functions. Using this and some sleep timers, the three LEDs could be controlled separately, or turned on and off in series to create a kind of "wave" function. A Python file was created to act as an API for control over the LEDs, providing functions to turn on or off a specific light or to start or stop a "continuous behaviour" in a separate thread, such as having the lights wave to indicate the start of a different process (see Appendix A for the code implementation).

The camera provided a few more challenges; the output format of the camera was not widely supported as it was an older model. However, using the *CV2* library from *OpenCV* worked, and *CV2* provided similarly intuitive instructions for initializing a camera, taking a photo, and saving it to the device. With this completed, another Python file was created as a camera API containing the expected functions needed to initialize the camera, take a photo and save it, and release the camera (see Appendix B).

The final component that needed to be controlled were the weight sensors. This proved to be the most difficult of the three. While the HX711 module contains many Python libraries due to its popularity in the field, internally, the majority of them used GPIO libraries like *RPi.GPIO* which were not compatible with Raspberry Pi 5. The GPIO pins on the Pi 5 communicate differently and as such, these libraries would produce errors when attempting to write to and read from the system clock (SCK) connection of the HX711 module. After more digging, a Pi 5-compatible HX711 library written for C++ was found on GitHub, and the GitHub contained a link to Python bindings for that library [7].

The Python GitHub project was cloned onto the Raspberry Pi and the C++ code was built and installed. With this, the HX711 outputs were successfully retrieved by the Raspberry Pi. The library contained a file to calibrate the weight sensors and retrieve a reference unit and zero value which would be needed to ensure the accuracy of readings.

Another Python file to act as an API for the handmade scale was created. This API contains the logic to detect when an item was placed or removed by continuously checking for readings changing by more than 4 grams from one reading to the next. This could not be done by checking the readings against an apparent weight of 0 grams because the weight sensors quite quickly grew over time, which is a well-known issue with the kind of load cells and AD module used and is referred to as "drifting". Functions to initialize the weight sensors, "listen" to and receive notifications about placed or removed items, and end the listening of the sensors, were created (see Appendix C).

The last API to be created was the model API, which had functions to load the model file and re-initialized it with the same settings as the Google Colab file, as well as a function to retrieve a prediction from the model on an image (see Appendix D).

Finally, the main control file was written, which delegated tasks to the four APIs above and made use of concurrency to have processes such as LED waving and weight sensing happen concurrently with other tasks such as model loading, image taking, and image classification (see Appendix E). The waving of LEDs was built to indicate to the user both that the system was loading and that the system detected an item on the scale and an image would be taken after 3 seconds. The second functionality was added to ensure that the user would be able to remove their hand before an image was taken.

This file also handled mapping the 7 outputs of the model to the three desired ones. The trash and compost class remained the same, while paper, glass, plastic, metal, and cardboard were all mapped to recycling. Based on the results from the model classification, the appropriate LED would be turned on: red for trash, green for compost, and blue for recycling.

The entire code base was uploaded to GitHub for future reference [8].

### 3.5 Testing and Analysis

The system as a whole was tested manually for general accuracy. It performed the best in classifying glass and plastic bottles, but generally had slightly more difficulty differentiating between paper and trash. Additionally, due to the lighter weights of items like paper, it was difficult for the system to reliably determine if an item had been placed on the scale. The most apparent issue with the system was its inability to maintain accuracy when photos were taken in different environments. To further test the model, 198 images were taken from the system. 81 of these images were intentionally lit with additional external light sources.

In a new Google Colab notebook, this external testing dataset and the existing model were loaded and the results analyzed to try and figure out if the inaccuracies were caused by low-light situations.

## 4 Findings

The results of the analysis on the testing set were interesting. It was theorized that the error rate of the model rose unexpectedly when the system was moved to a new environment due to the lower lighting in that environment. However, the accuracy rate on the additionally lit images was 72.8%, while the accuracy rate on the images taken with regular room lighting (“unlit”) images was much higher at 87.2%. The confusion matrices for each of these results were produced and can be seen in figures 2 and 3.

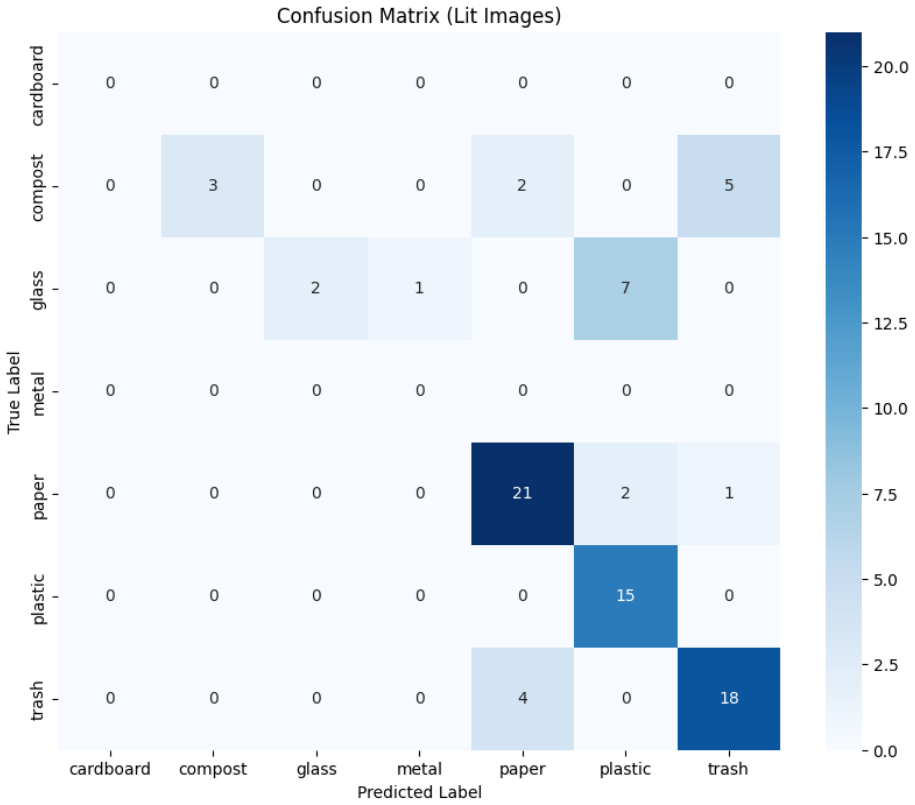


**Fig. 2.** Confusion matrix for prediction results on externally lit images

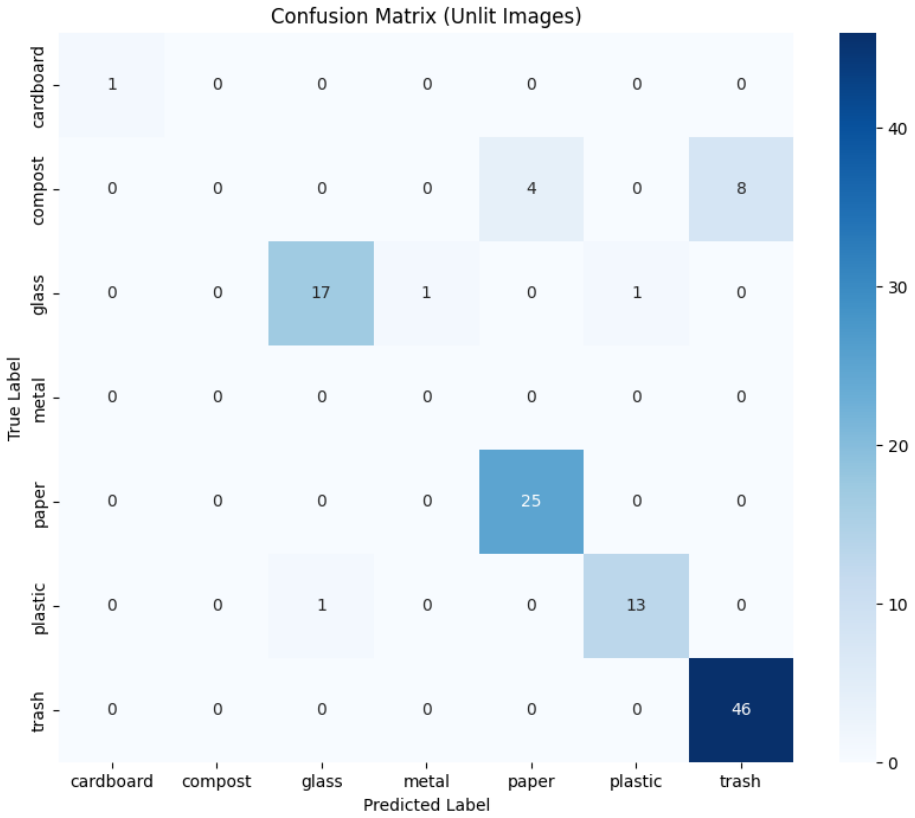


**Fig. 3.** Confusion matrix for prediction results on unlit images

These results were initially confusing, as it did not make sense for the model to perform *better* in lower-light situations. However, the confusion matrix for lit images showed a high accuracy within the paper, plastic, and trash categories. Investigating the test data showed that images in those three classes were taken in the same environment the system was originally tested in, the same one where the original photos that were added to the training dataset were taken. All of the unlit photos were also taken in the original room.

Because of this, the only results that truly show the difference between more and less lit situations is among those three classes. The “unlit” dataset provided an accuracy of 98.8% between these three classes, while the lit dataset had an accuracy of 88.5%.

Additionally, it is interesting to note that images of glass had a much lower accuracy in lit situations, and that compost images were incorrectly classified the majority of times in both lit and unlit situations. However, the lit images of compost provided a slightly *higher* accuracy than the unlit compost images. This is likely due to the fact

that the unlit compost images were *considerably* darker than other images in the set due to the camera needing to be placed differently (pointing down rather than forward toward the object).

Based off these results, it can be concluded that while lighting conditions do affect the accuracy of the system, the color and direction of the lighting appears to be at least as important as the amount of light in the model's accuracy. It seems that the model is not very generalizable to any changes in lighting situations. Additionally, the model is fairly inconsistent with classifying compost than those that exist in the training set, especially those that don't explicitly exist within the training set.

## 5 Discussion

While there is some existing research and developed solutions in garbage classification, finding datasets proved to be difficult. *TrashNet* was the only accessible pre-labelled dataset for garbage classification that was found, with the addition of *CompostNet* being the only labelled compost image dataset. When combined with the additional manually-taken photos, this created a dataset of 2,864 images for model creation. While this seems like a decently large size, there's a common consensus in the research community that, especially in more complex applications such as this one, around 1,000 images or more for each class is recommended for better results [9, 10, 11]. Within the 7 total classes, these 2,864 images did not reach that level.

**Table 2.** Images per class in the original dataset

| Class | Count |
|---|---|
| Cardboard | 403 |
| Compost | 214 |
| Glass | 500 |
| Metal | 410 |
| Paper | 662 |
| Plastic | 491 |
| Trash | 184 |

The compost and trash classes had the lowest amount of image samples, and these were also consistently the lowest-accuracy classes during testing, supporting the fact that it is unlikely that the amount of data used is sufficiently generalizable for identifying these classes of waste.

Additionally, while on the surface it seems that these seven classes of waste are sufficient for accurate sorting of trash, the problem of waste classification is more convoluted. Items that may appear recyclable on the surface, such as plain paper, may not be if they are soiled with food or grease. This would render them trash. Also, not all plastics, metals, cardboard, and paper are recyclable; some cannot be recycled at all, and

others are actually compostable. The images in the dataset do not take this into account, and the model as it is has no ability to differentiate between different kinds of plastic or soiled versus unsoiled materials.

Additionally, with such a complex classification issue, there is potential for multiple specialized models to perform classification subtasks and then be combined to produce a final trash/compost/recycling class. For example, while the model created here classifies into 7 separate classes, a different model could be used on the plastics class to identify different kinds of plastics, and another could be used to classify items as being soiled or unsoiled.

The findings discussed above regarding different lighting situations also showed that different lighting has a very big impact on the system's accuracy. As it stands now, the system does not perform any preprocessing steps to the images taken from the camera. This causes the wild fluctuations seen in testing. Image preprocessing steps such as sharpening, brightening, white balance, and more could and should be added to help mitigate this issue. One such way to do this would be to have the system take an image at initialization of the white background. It can then calibrate the image so that the lighting appears more consistent with the training data. This, however, may introduce other issues with over-exposed, noisy, or low-contrast images. The addition of a newer, higher-quality camera would also likely contribute to better classification results in practice and could mitigate some of the above-mentioned issues with image brightening.

Finally, the addition of weight sensors to the project proved helpful for the detection of items, but imperfect. This could be improved with more accurate weight sensors or perhaps additional proximity sensors to aid in lower-weight situations. The use of a weight sensor in the system also spurred the idea that the combination of image and weight data together could potentially aid in waste classification. No datasets were found for the weight of garbage items at the time this report was written.

# 6 Conclusion

## 6.1 Summary of Main Findings

In conclusion, the developed smart waste classification system demonstrated a promising performance for on-device waste identification. With the integration of hardware components (Raspberry Pi 5, camera, load cells, and LEDs), and software (PyTorch-based ML model trained with *TrashNet* and *CompostNet* datasets), the system achieved around 95% accuracy in test conditions, reliably categorizing trash and recycling. However, despite these promising results, the limitations in dataset diversity for compost samples became apparent when the system struggled to reliably identify organic materials. This led to misclassifications for this category and highlights the importance of providing sufficient data for the model, as well as the need for more robust evaluation under real-world lighting conditions, angles, and background context.

### 6.2 Limitations

To address the limitations experienced in this research, several enhancements are recommended. Future work should expand the compost dataset to bring accuracy levels in this category to the same level as the more successful categories. Employing active learning or synthetic data generation may also improve class balance and reduce bias. Another potential enhancement for the future is more robust preprocessing steps. Real-world lighting conditions and angles heavily affected the accuracy of the system, therefore employing image enhancing techniques to improve visual clarity would be beneficial. Techniques such as noise reduction, contrast adjustment, de-skewing, and brightness adjustment may help in overall performance improvement.

### 6.3 Recommendations

Beyond technical enhancements, gathering feedback from waste management professionals may better align the system's usability in real-world applications. These recommendations may guide further development into a more effective waste management and sorting solution.

The lack of data, particularly data that can help identify different types of plastic, glass, and differentiates between soiled and non-soiled material proves a gap in the current state of research into ML waste classification solutions. Additionally, image data is the only readily-available data that exists for this purpose. To add to the current body of research, the effects of additional image data and other kinds of data such as weight data could be investigated, as well as potential solutions for the complexity of classification required for accurate trash classification, such as multi-model systems.

## 7 References

1. Yang, M., & Thung, G.: Classification of trash for recyclability status. Unpublished manuscript, Stanford University (2016). https://cs229.stanford.edu/proj2016/report/ThungYang-ClassificationOfTrashForRecyclabilityStatus-report.pdf, last accessed 2024/12/12.
2. Frost, S., Tor, B., Agrawal, R., Forbes, A.: CompostNet: An Image Classifier for Meal Waste (2019.). https://github.com/sarahmfrost/compostnet/blob/master/2019_Frost_CompostNet_GHTC.pdf, last accessed 2024/12/12.
3. TrashNet GitHub, https://github.com/garythung/trashnet, last accessed 2024/12/12.
4. CompostNet Github, https://github.com/sarahmfrost/compostnet, last accessed 2024/12/12.
5. [PyTorch] Garbage classification (~95% accuracy), https://www.kaggle.com/code/aadhavvignesh/pytorch-garbage-classification-95-accuracy/notebook, last accessed 2024/12/12.
6. Waste Classification, https://colab.research.google.com/drive/1Bp0brf4YMm3AuXwYFHnyCpd1I9-3rEVi?usp=sharing, last accessed 2024/12/12.
7. hx711-rpi-py GitHub, https://github.com/endail/hx711-rpi-py, last accessed 2024/12/12.

8. pi-waste-classifier GitHub, https://github.com/AngelaVilladiego/pi-waste-classifier, last accessed 2024/12/12
9. How many images do you need to train a neural network?, https://pete-warden.com/2017/12/14/how-many-images-do-you-need-to-train-a-neural-network/, last accessed 2024/12/12.
10. How Many Images per Class Are Sufficient for Training a CNN?, https://www.geeksforgeeks.org/how-many-images-per-class-are-sufficient-for-training-a-cnn/, last accessed 2024/12/12.
11. Cireşan, D. C., Meier, U., Schmidhuber, J.: Transfer Learning for Latin and Chinese Characters with Deep Neural Networks (2012). https://people.idsia.ch/~ciresan/data/ijcnn2012_v9.pdf, last accessed 2024/12/12.

# 8 Appendices

## 8.1 Appendix A LED API

```
from gpiozero import LED
from time import sleep
import threading

# Constants/globals
_red_led = LED(17)
_green_led = LED(18)
_blue_led = LED(22)

stop_behaviour_flag = threading.Event()

def on(color="all"):
    match color:
        case "red":
            _red_led.on()
        case "green":
            _green_led.on()
        case "blue":
            _blue_led.on()
        case _:
            _red_led.on()
            _green_led.on()
            _blue_led.on()

def off(color="all"):
    match color:
        case "red":
```

```
            _red_led.off()
        case "green":
            _green_led.off()
        case "blue":
            _blue_led.off()
        case _:
            _red_led.off()
            _green_led.off()
            _blue_led.off()

def flash(count=1, interval=0.5):
    for i in range(count):
        on("all")
        sleep(interval)
        off("all")
        sleep(interval)

def wave(count=1, interval=0.2):
    for i in range(count):
        on("red")
        sleep(interval)
        on("green")
        sleep(interval)
        on("blue")
        sleep(interval)
        off("red")
        sleep(interval)
        off("green")
        sleep(interval)
        off("blue")
        sleep(interval)

BEHAVIOUR_MAP = {
    'flash': flash,
    'wave': wave
}

def start_continuous_behaviour(behaviour="flash"):
    stop_behaviour_flag.clear()

    behaviour_function = BEHAVIOUR_MAP.get(behaviour, flash)
```

```
    while not stop_behaviour_flag.is_set():
        behaviour_function()

def stop_continuous_behaviour():
    stop_behaviour_flag.set()
    off("all")
```

## 8.2 Appendix B Camera API

```
import cv2
import os

# Globals and constants
IMAGE_DIR = '/home/pi/Garbage_Classifier/images'
next_image_number = None
_cam = None

if not os.path.exists(IMAGE_DIR):
    os.makedirs(IMAGE_DIR)

def set_next_image_number():
    global next_image_number

    print('Initializing image file numbering...')

    existing_files = os.listdir(IMAGE_DIR)

    if not existing_files:
        next_image_number = 1
        return

    most_recent_file = max(existing_files, key=lambda f: os.path.getmtime(os.path.join(IMAGE_DIR, f)))

    number_str = most_recent_file[9:12] # Getting the number from pi_image_001.jpg format
    next_image_number = int(number_str) + 1

    print(f'File numbering initialized. Next number is {next_image_number}')

def initialize_camera():
    global _cam

    if not next_image_number:
```

```
        set_next_image_number()

    _cam = cv2.VideoCapture('/dev/video0')

    if not _cam.isOpened():
        print('Error: couldn\'t load camera.')
        exit()

    print('Camera initialized')

def take_picture():
    global _cam, next_image_number

    if not _cam.isOpened():
        print('Error: couldn\'t load camera.')
        exit()

    print('Taking photo...')

    ret,image = _cam.read()
    image_path = (os.path.join(IMAGE_DIR, f'pi_image_{next_image_num-
ber:03}.jpg'))
    cv2.imwrite(image_path, image)

    print(f'Image saved at {image_path}.')

    next_image_number = next_image_number + 1

    return image_path

def release_camera():
    global _cam

    print('Releasing camera.')
    _cam.release()
```

### 8.3 Appendix C Scale API

```
import time
import threading
from HX711 import *

REFERENCE_UNIT = -24
ZERO_VALUE = 122930
```

```
DATA_PIN = 23
SCK_PIN = 24

hx = None
event_dispatcher = None
last_mass = None
new_mass = None

stop_listening_flag = threading.Event()

def initialize_scale():
   global hx, event_dispatcher, last_mass

   hx = SimpleHX711(DATA_PIN, SCK_PIN, REFERENCE_UNIT,
ZERO_VALUE)
   hx.setUnit(Mass.Unit.G)
   hx.zero()

   last_mass = hx.weight(20)
   print('Scale initialized')

listen_lock = threading.Lock()
def listen(scale_event_dispatcher):
   with listen_lock:

      global event_dispatcher, last_mass, new_mass, hx

      stop_listening_flag.clear()

      event_dispatcher = scale_event_dispatcher

      item_placed_time = None
      item_placed = False
      hx.zero()

      try:
         while not stop_listening_flag.is_set():
            new_mass = hx.weight(20)
            current_time = time.time()

            mass_difference = new_mass.getValue() - last_mass.getValue()
```

```
            if stop_listening_flag.is_set():
                break

            if (not item_placed and mass_difference > 4):
                item_placed = True
                item_placed_time = current_time
                event_dispatcher.dispatch('item_placed')

            elif (item_placed and mass_difference < -4):
                item_placed = False
                event_dispatcher.dispatch('item_removed')
                hx.zero()

            # Item is placed and settled
            elif item_placed and current_time - item_placed_time >= 1:
                event_dispatcher.dispatch('item_settled')

            last_mass = new_mass

        except Exception as e:
            print(f"Error while listening to scale: {e}")

        finally:
            print("Exiting scale listening loop.")
            hx.zero()  # Zero the scale on exit

def stop_listening():
    global hx
    stop_listening_flag.set()
    hx.zero()
```

### 8.4 Appendix D Model API

```
import torch
import torchvision.transforms as transforms
from PIL import Image
from pathlib import Path
import torch.nn as nn
import torchvision.models as models
import pickle
import warnings

# Define the ResNet model class
class ResNet(nn.Module):
```

```
    def __init__(self, num_classes):
        super().__init__()
        self.network = models.resnet50(pretrained=True)
        num_ftrs = self.network.fc.in_features
        self.network.fc = nn.Linear(num_ftrs, num_classes)

    def forward(self, xb):
        return torch.sigmoid(self.network(xb))

# Global variables to hold the model, device, classes, and transformations
_model = None
_device = None
_classes = None
_transformations = None

def load_model(filepath="garbage_classification_model.pt", classes_file="clas-
ses.pkl", show_warnings=False):
    """
    Load the model and classes if not already loaded.

    Parameters:
    - filepath: Path to the model file.
    - classes_file: Path to the pickle file containing class labels.

    Returns:
    - The loaded model.
    """
    global _model, _device, _classes, _transformations

    # Configure warnings (off by default as they can be distracting)
    if show_warnings:
        warnings.filterwarnings("default", category=UserWarning, module="torch")
    else:
        warnings.filterwarnings("ignore", category=UserWarning, module="torch")
        warnings.filterwarnings("ignore", category=FutureWarning)

    if _model is None:
        print("Loading model and resources...")

        # Set the device (GPU if available, else CPU)
        _device = torch.device('cuda' if torch.cuda.is_available() else 'cpu')
```

```
        # Load class labels
        with open(classes_file, 'rb') as f:
            _classes = pickle.load(f)

        # Initialize the model
        _model = ResNet(num_classes=len(_classes))
        _model.load_state_dict(torch.load(filepath, map_location=_device))
        _model.eval()  # Set the model to evaluation mode

        # Define transformations (same as during training)
        _transformations = transforms.Compose([
            transforms.Resize((256, 256)),
            transforms.ToTensor()
        ])

        print("Model and resources loaded successfully.")
    else:
        print("Model is already loaded.")

    return _model

def predict_image(image_path):
    """
    Predict the class of an image.

    Parameters:
    - image_path: Path to the input image.

    Returns:
    - Predicted class name.
    - Probabilities for each class.
    """
    global _model, _device, _classes, _transformations

    if _model is None:
        raise RuntimeError("Model not loaded. Call load_model() first.")

    # Load and preprocess the image
    image = Image.open(image_path)
    input_tensor = _transformations(image)
    input_batch = input_tensor.unsqueeze(0)  # Create a mini-batch
```

```
    # Move the input batch to the same device as the model
    input_batch = input_batch.to(_device)

    # Perform prediction
    with torch.no_grad():
        output = _model(input_batch)
        probabilities = torch.nn.functional.softmax(output[0], dim=0)

    # Retrieve the predicted class name and probabilities
    predicted_class_index = torch.argmax(probabilities).item()
    predicted_class_name = _classes[predicted_class_index]

    probability_details = [
        {"label": _classes[i], "probability": prob.item()}
        for i, prob in enumerate(probabilities)
    ]

    return predicted_class_name, probability_details

# Optional helper to get the class list
def get_classes():
    """
    Get the list of class names.
    """
    global _classes
    if _classes is None:
        raise RuntimeError("Classes not loaded. Call load_model() first.")
    return _classes
```

### 8.5 Appendix E Main control program

```
import model_api as model
import camera_api as camera
import lights_api as lights
import scale_api as scale
import time
from lights_api import start_continuous_behaviour
import threading
import signal
import sys

# Constants and globals
WASTE_CATEGORY_MAP = {
    'cardboard': 'recycling',
```

```
    'compost': 'compost',
    'glass': 'recycling',
    'metal': 'recycling',
    'paper': 'compost',
    'plastic': 'recycling',
    'trash': 'trash'
}

LED_COLOR_MAP = {
    'recycling': 'blue',
    'trash': 'red',
    'compost': 'green'
}
lights_thread = None
scale_thread = None
event_dispatcher = None
processing_complete_event = threading.Event()
shutdown_flag = threading.Event()

def start_lights(behaviour):
    global lights_thread
    lights_thread = threading.Thread(target=start_continuous_behaviour, args=(behav-
iour,))
    lights_thread.daemon = True # Daemon thread will exit when the main program exits
    lights_thread.start()

def stop_lights():
    lights.stop_continuous_behaviour()
    if lights_thread is not None and threading.current_thread() is not lights_thread:
        lights_thread.join()

def start_listening_to_scale():
    global scale_thread, event_dispatcher

    if shutdown_flag.is_set():
        return

    if scale_thread is not None and scale_thread.is_alive():
        return

    def scale_listen_wrapper():
        try:
```

```
            if not shutdown_flag.is_set():
                scale.listen(event_dispatcher)  # Adjust as necessary
        except Exception as e:
            if not shutdown_flag.is_set():
                print(f"Error in scale listener: {e}")
        finally:
            print("Scale listener stopped.")

    scale_thread = threading.Thread(target=scale_listen_wrapper)
    scale_thread.daemon = True # Daemon thread will exit when the main program exits
    scale_thread.start()
    print('Listening to scale...')

def stop_listening_to_scale():
    print('Turning off scale listening...')
    scale.stop_listening()
    if scale_thread is not None and threading.current_thread() is not scale_thread:
        scale_thread.join()

def on_item_placed():
    if shutdown_flag.is_set():
        return
    print('Item placed on scale. Turning on lights.')
    lights.on()

def on_item_settled():
    if shutdown_flag.is_set():
        return

    print('Processing...')

    stop_listening_to_scale()
    lights.off()

    # Indicate that picture will be taken
    start_lights("wave")
    time.sleep(3)
    stop_lights()
    lights.on()

    # Take picture
    camera.initialize_camera()
```

```
    image_path = camera.take_picture()
    camera.release_camera()
    lights.off()

    # Predict image
    predicted_class, probabilities = model.predict_image(image_path)

    print("Probabilities:")
    for prob in probabilities:
        print(f'\t{prob}')

    print(f"Predicted class: {predicted_class}")

    waste_category = WASTE_CATEGORY_MAP.get(predicted_class)

    print(f'Waste belongs in {waste_category}.')

    lights.on(color=LED_COLOR_MAP.get(waste_category))

    print()

    processing_complete_event.set()

def on_item_removed():
    if shutdown_flag.is_set():
        return
    print('Item removed from scale. Turning off lights.')
    stop_lights()

class ScaleEventDispatcher:
    def __init__(self):
        self.handlers = {
            'item_placed': on_item_placed,
            'item_settled': on_item_settled,
            'item_removed': on_item_removed
        }

    def dispatch(self, event):
        if shutdown_flag.is_set():
            return
        if event in self.handlers:
            self.handlers[event]()
```

```
        else:
            print(f'Unknown event: {event}')

def init():
    global event_dispatcher

    model.load_model(filepath="garbage_classification_model.pt",   classes_file="clas-
ses.pkl")
    # camera.initialize_camera()
    scale.initialize_scale()
    event_dispatcher = ScaleEventDispatcher()

def shutdown_program():
    print('Shutting down...')

    # Stop threads
    stop_listening_to_scale()
    stop_lights()

    # Release resources
    camera.release_camera()
    lights.off()

    print('Shutdown complete.')

def signal_handler(sig, frame):
    print('\nInterrupt signal received. Shutting down...')
    stop_listening_to_scale()
    shutdown_flag.set() # Notify threads to stop


    # Wait briefly to allow threads to finish
    time.sleep(0.5)

    print('Exiting program.')
    sys.exit(0)

def main(args):

    try:
        # Indicate initializing state with flashing lights
        start_lights("wave")
```

```
        init()
        stop_lights()

        # Indicate ready state with solid lights
        lights.on()
        print('\nReady to start.')

        while not shutdown_flag.is_set():
            input('Please remove any items from the scale and press Enter to start a new
waste classification.')
            lights.off()

            # Start event loop listening
            print('Starting waste classification.')
            start_listening_to_scale()

            processing_complete_event.wait()

            print('Waste classification complete.')
            processing_complete_event.clear()

    except KeyboardInterrupt:
        print('Shutdown requested.')

    finally:
        shutdown_program()

if __name__ == '__main__':
    signal.signal(signal.SIGINT, signal_handler) # Handle CTRL+C
    signal.signal(signal.SIGTERM, signal_handler) # Handle termination
    import sys
    sys.exit(main(sys.argv))
```