

**Department of Electronic &
Telecommunication Engineering
University of Moratuwa**



**EN 2160 – Electronic Design Realization
Design Documentation**

**Verifying the accurate placement of labels on milk packets and rejecting
faulty ones**

210387D Mihiranga N.G.D.
210391J Morawakgoda M.K.I.G.

23 June 2024

Contents

Hardware Design	3
1.1. Introduction	3
1.2. Circuit design	3
1.3. Mechanical design.....	4
Software Design	5
2.1. Introduction.....	5
2.2. Background Research	5
2.3. Project Workflow and Implementation Plan	6
2.4. Dataset Preparation	6
2.4.1. Preparing the Initial Data	6
2.4.2. Data Augmentation.....	7
2.5. Data Annotation	10
2.5.1 Creating the Segmentation Masks	10
2.5.2 Creating the Labels.....	15
2.6. Training Phase	17
2.6.1 YOLOv8.....	17
2.6.2 Initial Setup	18
2.6.3 Training Script	18
2.6.4. Model Architecture	20
2.7. Evaluation Phase	21
2.8. Real Time Predictions.....	24
2.8.1. Downloading Model Weights	24
2.8.2. Implementing Real-Time Object Detection	24
2.8.3. Results of the real time detection	26
2.9. Integration with Arduino for Automated Response	28
2.9.1. Arduino Setup	28
2.9.2. Python Code Integration	28
2.9.3. Results and Observations.....	30
Appendix: Daily Progress.....	31
References	34
Reviews.....	34

Hardware Design

1.1. Introduction

Our project aims to develop an automated system capable of detecting misprints on product packaging. The primary objective of this project is to identify instances where critical information such as expiration dates, manufacturing dates, and price tags deviate from the designated white space on the packaging. Upon detection of misprints or deviations from the expected placement of information within the designated white space, the system will trigger a mechanism to reject the affected packaging from the conveyor belt.

1.2. Circuit design

The first function of the circuit is to take input from a camera and send the information to a computer for processing. After processing the data and detecting if a milk packet is faulty, a signal is sent to our pushing mechanism. The output signal from the computer is then sent to our Atmel ATmega 328P chip, which is part of the PCB. We use a USB connector(J6) to facilitate communication between the PCB and the computer.

For our project, we are using a two-layer PCB. After processing the data in the Atmel chip, the actuator movement signal is sent from the PCB to the actuator motor driver. We use JST connectors (J6) for this purpose. Finally, the output signal from the motor driver goes to the motor, activating the pushing mechanism.

The design ensures efficient detection and removal of faulty milk packets by integrating camera input, computer processing, and actuator control through the PCB and motor driver. This setup provides a reliable and automated solution for quality control in milk packet processing.

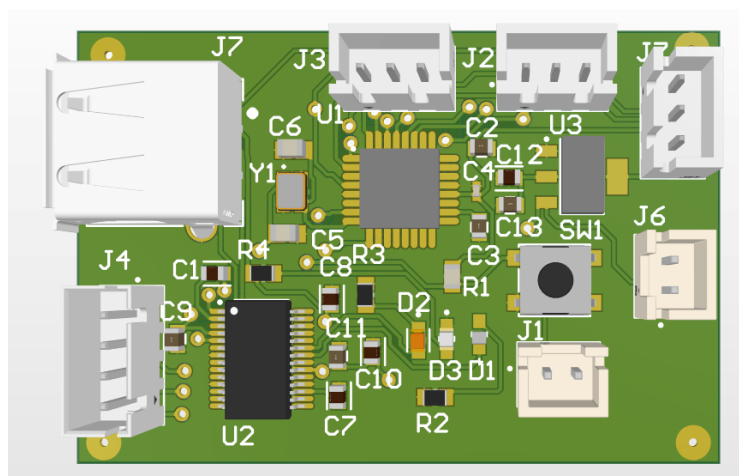
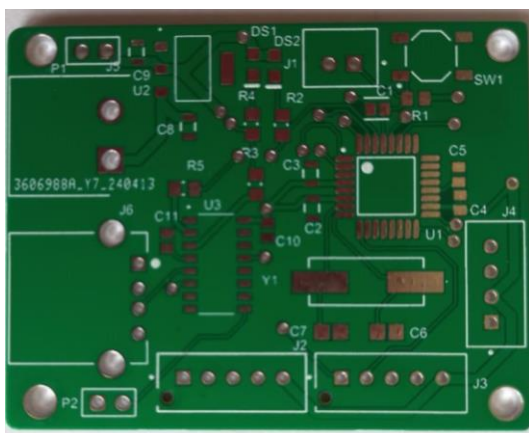


Figure 1: PCB design

1.3. Mechanical design

In the mechanical design of the components, there are three main components the main structure, the conveyor belt roller, and the pushing actuator.

The main structure of the design consists of iron box bars, providing a durable frame for the entire system. The conveyor belt roller mechanism uses 10mm pillow bearings and 10mm axles, driven by a NEMA 17 motor. This setup ensures smooth and reliable movement of the conveyor belt, allowing for precise positioning of the milk packets.

The pushing actuator is also controlled by a NEMA 17 motor, which provides the necessary force to remove faulty milk packets from the conveyor belt. The actuator mechanism is designed for quick and accurate response, ensuring that faulty packets are promptly removed from the production line.

Additionally, the integration of the mechanical components with the electronic control system, including the Atmel ATmega 328P chip and the motor driver, ensures seamless operation and coordination between the detection and removal processes. The use of USB and JST connectors facilitates reliable communication and power delivery to the various components, contributing to the overall efficiency and effectiveness of the system.

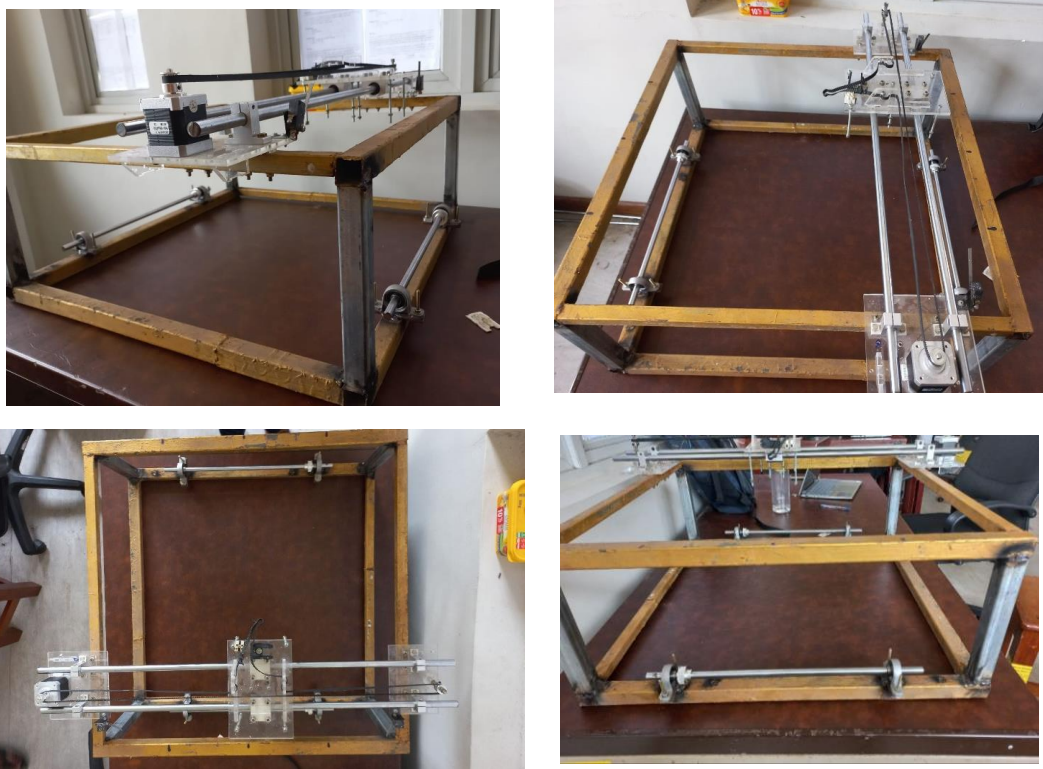


Figure 2: Mechanical design

Software Design

2.1. Introduction

In this software designing part we aim to deploy an object detection model to identify misprinted labels on milk packets, specifically focusing on labels that are not printed within the designated white space area. Ensuring the correct positioning of labels on milk packets is crucial for maintaining product quality, brand integrity, and compliance with regulatory standards. Misprinted labels can lead to misidentification of products, miscommunication of product information, and potential consumer safety issues.

To address this challenge, we are developing a robust object detection system that can automatically inspect milk packets for correct label placement in real-time

2.2. Background Research

At the initial stage of this project, we explored two potential solutions for detecting misprinted labels:

1) Detect the misprinted labels using Hough lines and the Canny edges without using any deep learning techniques:

This traditional computer vision approach involves using edge detection (Canny edge detector) and line detection (Hough Transform) techniques to identify the boundaries of the labels.

Challenges:

- Low Accuracy: The accuracy of this method is significantly lower because the results can vary depending on external conditions like lighting and camera angles.
- External Conditions: Variations in lighting, shadows, and camera positioning can drastically affect the edge detection and line detection results, leading to inconsistent performance.

2) Detect the misprinted labels by developing an object detection model using deep learning techniques:

This approach involves training a deep learning model on a custom dataset of milk packet images to detect misprinted labels.

Advantages:

- High Accuracy: Deep learning models can be trained to handle varying conditions by including diverse examples in the training dataset. This makes them more robust to changes in lighting, camera angles, and other external factors.
- Industry Standard: Most similar projects in the industry are using deep learning techniques due to their superior performance in accuracy and robustness.

Challenges:

- Data Collection: Creating a large dataset manually is time-consuming and requires significant effort. The dataset needs to include images of correctly printed labels and misprinted labels under various conditions.

- **Training Time:** Training deep learning models can be computationally intensive and time-consuming, requiring powerful hardware (GPUs) and considerable time for experimentation and tuning.

Considering the importance of accuracy in detecting misprinted labels and the industry's trend towards using deep learning techniques, we decided to proceed with the second solution. Despite the challenges in data collection and training, the potential for achieving high accuracy and robustness makes deep learning the preferred approach for this project.

2.3. Project Workflow and Implementation Plan

Since we opted for a deep learning approach, we selected YOLOv8 as the model and decided to implement image segmentation, an advanced form of object detection. Our choice of YOLOv8 was driven by its balance of speed and accuracy, making it suitable for real-time applications like detecting misprinted labels on milk packets.

We chose Google Colab for the training and evaluation phases due to its accessibility to free GPU resources, seamless integration with deep learning libraries and support for collaborative work. For deploying the model into real-time predictions, we planned the workflow as follows:

- 1) **Dataset Preparation:** Gathered a comprehensive dataset of milk packet images, including examples of correctly printed labels and misprinted labels outside the white space area.
- 2) **Data Annotation:** Manually annotated the dataset to indicate the location of labels and their misprints.
- 3) **Training Phase:** Utilized Google Colab to train the YOLOv8 model on the annotated dataset, leveraging its GPU capabilities for faster training.
- 4) **Evaluation Phase:** Evaluated the trained model's performance using metrics to ensure it meets the required accuracy for detecting misprinted labels.
- 5) **Real-Time Object Detection:** Deployed the trained model using Jupyter Lab in a local environment to perform real-time object detection on incoming video streams or images of milk packets.
- 6) **Integration with Arduino for Automated Response:** Integrated the detection system with a microcontroller to send signals upon detecting misprinted labels, enabling automated quality control in production lines.

2.4. Dataset Preparation

2.4.1. Preparing the Initial Data

Dataset preparation is a key challenge in this project because, to achieve sufficient accuracy with the YOLOv8 model, we need at least 1000+ images per class (for CORRECT and WRONG labels). Initially, we prepared the data with 10 images for each class:

- **Correct Labels:** We simply took 10 photos of milk packets with correctly printed labels.
- **Wrong Labels:** Finding milk packets with misprinted labels was difficult. Therefore, we took 10 images of milk packets and edited these images to simulate misprinted labels using *Photoshop* software.



Figure 3: Correct Label

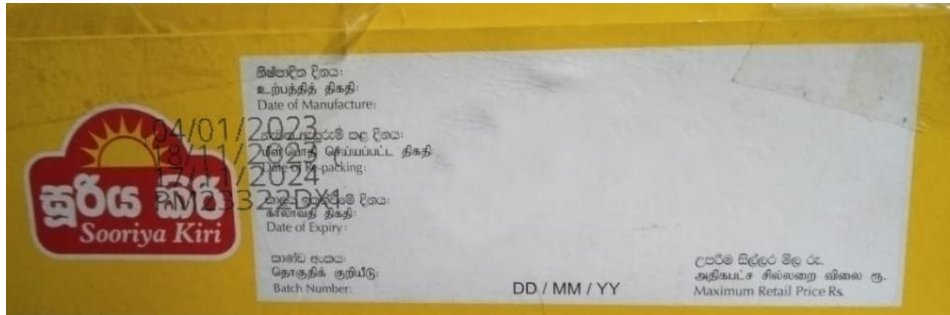


Figure 4: Wrong Label

2.4.2. Data Augmentation

To increase the number of images from our initial set to 1000+ per class, we utilized data augmentation. Data augmentation is a technique used to artificially expand the size of a dataset by creating modified versions of images in the dataset. This can include transformations such as rotations, flips, translations, and changes in brightness or contrast, among others.

Here is the process we used:

- 1) **Initial data** Folder: We created a folder containing the initial 10 images for each class.
- 2) **preview** Folder: We created a new folder called **preview** in the same directory as the **Initial data** folder.
- 3) Augmentation Process: Using the following process, we generated 2020 images (101 images from each of the 20 initial images) and saved them in the **preview** folder.

Augmentation Process

1. Installing & Importing Required Libraries

Before running the data augmentation script, it is essential to install all the necessary libraries. The following commands can be used to install the required libraries:

```
!pip install numpy tensorflow pillow
```

Following code imports necessary libraries:


```
import os
import numpy as np
from tensorflow.keras.preprocessing.image import ImageDataGenerator, load_img
from PIL import Image
```

- `os` for file and directory operations.
- `numpy` for numerical operations.
- `ImageDataGenerator` and `load_img` from `tensorflow.keras.preprocessing.image` for image augmentation and loading.
- `Image` from `PIL` for image manipulations.

2. Custom Functions for Image Conversion

```
def custom_img_to_array(img, data_format='channels_last', dtype='float32'):
    """Converts a PIL Image instance to a Numpy array."""
    x = np.asarray(img, dtype=dtype)
    if data_format == 'channels_first':
        if x.ndim == 3:
            x = x.transpose(2, 0, 1)
        elif x.ndim == 4:
            x = x.transpose(0, 3, 1, 2)
    return x
```

```
def custom_array_to_img(x, data_format='channels_last', scale=True):
    """Converts a Numpy array to a PIL Image instance."""
    if data_format == 'channels_first':
        x = x.transpose(1, 2, 0)
    if scale:
        x = (x - x.min()) / (x.max() - x.min()) * 255
    return Image.fromarray(x.astype('uint8'))
```

These functions handle the conversion between PIL images and Numpy arrays:

- `custom_img_to_array`: Converts a PIL image to a Numpy array, with an option for different data formats.
- `custom_array_to_img`: Converts a Numpy array back to a PIL image, with optional scaling.

3. Ensuring the Preview Directory Exists

```
# Ensure the preview directory exists
save_dir = 'preview'
if not os.path.exists(save_dir):
    os.makedirs(save_dir)
```

This section ensures that the directory where augmented images will be saved (`preview`) exists. If not, it creates the directory.

4. Initializing the ImageDataGenerator


```
# Initialize the ImageDataGenerator
datagen = ImageDataGenerator(
    rotation_range=20,
    width_shift_range=0.1,
    height_shift_range=0.1,
    shear_range=0.2,
    zoom_range=0.05,
    horizontal_flip=True,
    fill_mode='nearest'
)
```

The `ImageDataGenerator` is configured with various augmentation parameters:

- `rotation_range`: Randomly rotate images up to 20 degrees.
- `width_shift_range` and `height_shift_range`: Randomly shift images horizontally and vertically by up to 10% of the total width/height.
- `shear_range`: Apply random shear transformations.
- `zoom_range`: Apply random zoom within a small range ($\pm 5\%$).
- `horizontal_flip`: Randomly flip images horizontally.
- `fill_mode`: Specifies how to fill points outside the boundaries of the input when the image is transformed (here, using 'nearest').

5. Path to the Dataset Directory

```
# Path to the dataset directory
dataset_dir = 'Initial data'
```

This line sets the path to the directory containing the initial set of images.

6. Iterating Through and Augmenting Images

```
# Iterate through all images in the dataset directory
for filename in os.listdir(dataset_dir):
    if filename.lower().endswith(('.png', '.jpg', '.jpeg')):
        img_path = os.path.join(dataset_dir, filename)
        img = load_img(img_path) # This is a PIL image
        x = custom_img_to_array(img)
        x = x.reshape((1,) + x.shape)

        # Generate and save batches of randomly transformed images manually
        i = 0
        for batch in datagen.flow(x, batch_size=1):
            transformed_img = custom_array_to_img(batch[0])
            save_name = f"{os.path.splitext(filename)[0]}_{i}.jpg"
            fname = os.path.join(save_dir, save_name)
            transformed_img.save(fname)
            i += 1
        if i > 100:
            break # Otherwise, the generator would loop indefinitely
```

This section processes and augments each image in the dataset directory:

- **File Iteration:** Iterates through all image files in the dataset directory.
- **Image Loading and Conversion:** Loads each image and converts it to a Numpy array.
- **Data Augmentation Loop:** Uses `datagen.flow` to generate batches of augmented images from each original image. For each batch:
 - Converts the augmented Numpy array back to a PIL image.
 - Constructs a filename for the augmented image.
 - Saves the augmented image in the `preview` directory.
 - Limits the number of augmented images per original image to 100 to prevent indefinite looping.

2.5. Data Annotation

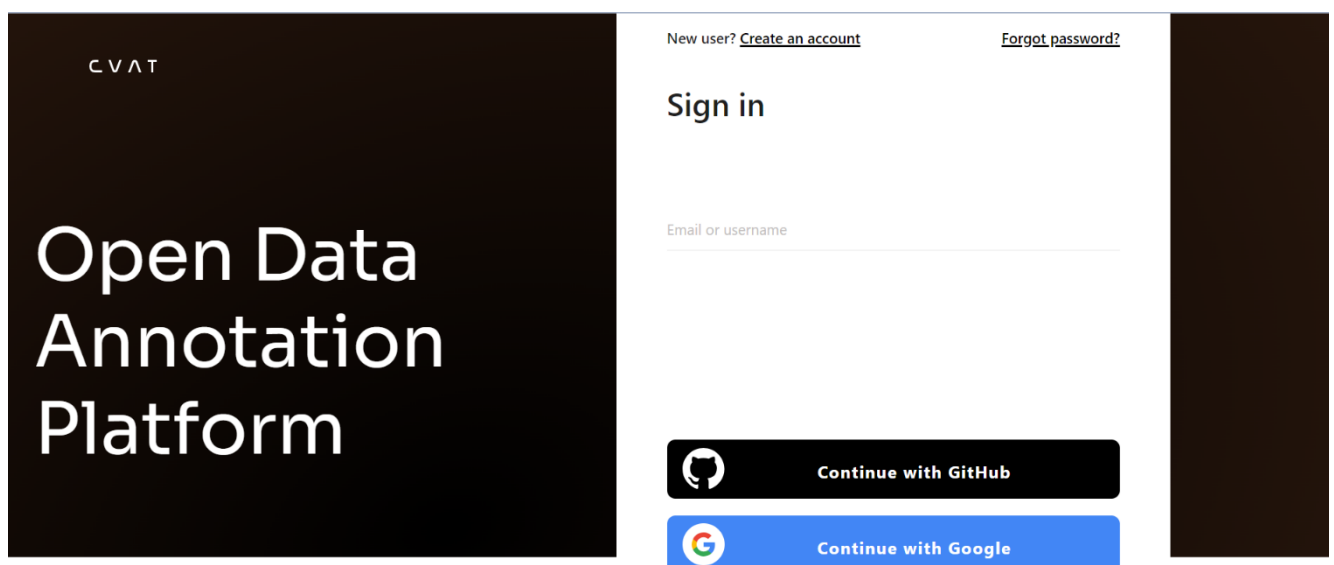
After the data augmentation process, a new folder called images was created, and all images from both the preview and initial data folders were copied into it. This resulted in a dataset of 2040 images, with 1020 images per class. The next step is to create segmentation masks for the dataset and convert them into labels.

2.5.1 Creating the Segmentation Masks

For creating segmentation masks for our custom dataset, we used an open data annotation platform called CVAT (<https://www.cvat.ai/>).

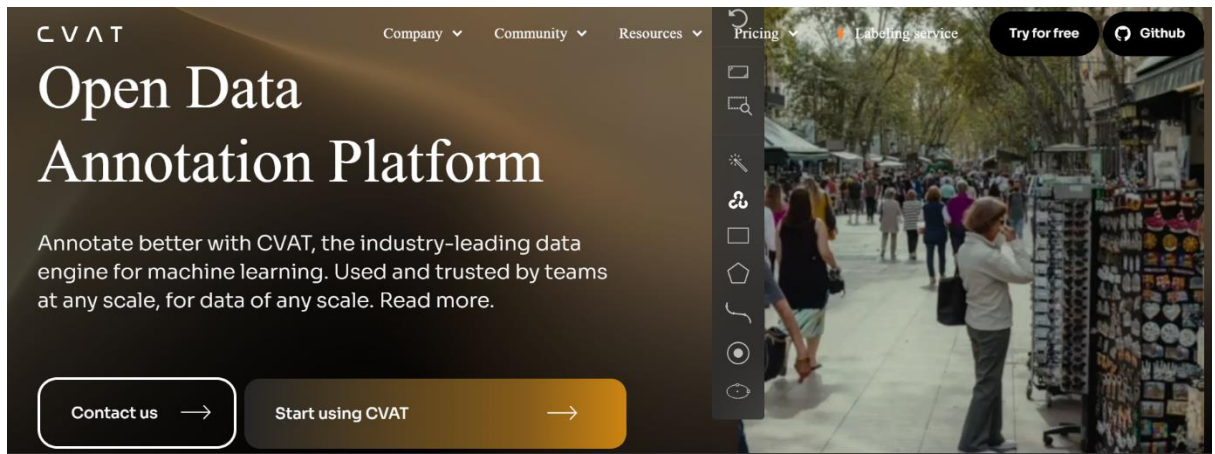
Process Followed:

1. **Create an Account:**
 - Sign up using Google or GitHub.



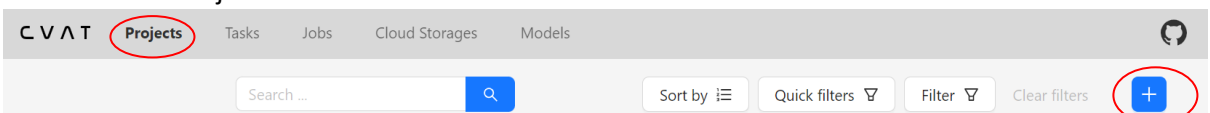
2. Start Using CVAT:

- Click "Start using CVAT" after logging into your account.

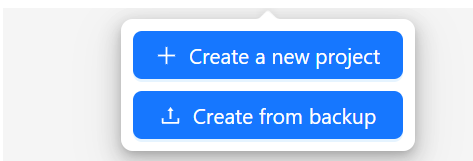


3. Create a New Project:

- Go to the Projects section and click the "+" button.



- Click "Create a new project".

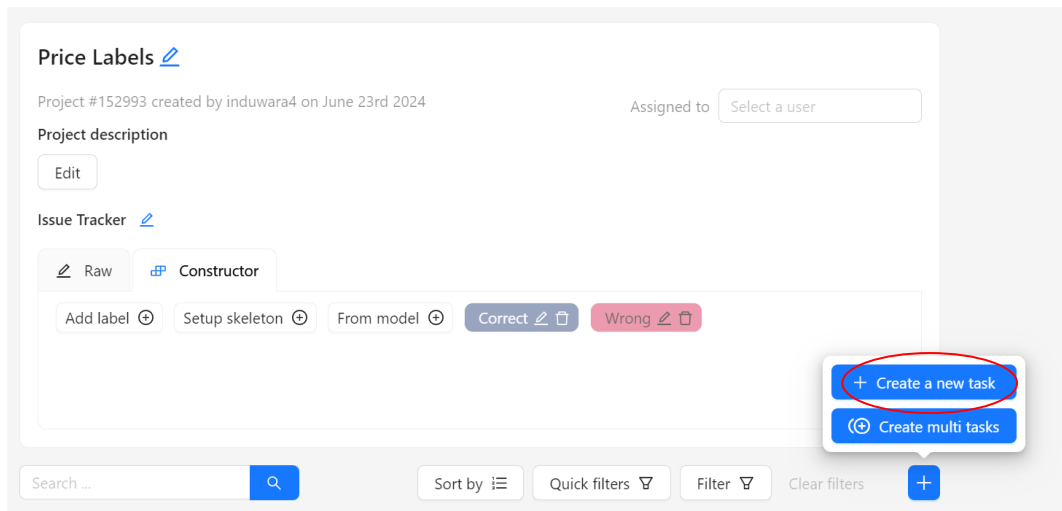


- Name the project (e.g., "Price Labels").
- Add labels:
 - Click "Add label" in the Labels field, enter "Correct" as the label name, and click "Continue".
 - Enter "Wrong" as the label name and click "Continue".
 - Click "Cancel" and then "Submit and open".

The image shows the 'Create a new project' form. The title 'Create a new project' is at the top. Below it is a form with a 'Name' field containing 'Price Labels' and a green checkmark. Under 'Labels:', there are 'Raw' and 'Constructor' tabs. The 'Constructor' tab is active, showing a 'Correct' label with a green checkmark, a dropdown menu set to 'Any', and an 'Add an attribute' button. Below the labels are 'Continue' and 'Cancel' buttons. At the bottom, there is an 'Advanced configuration' section with a right-pointing arrow. At the very bottom, there are two buttons: 'Submit & Open' (circled in red) and 'Submit & Continue'.

4. Create a New Task:

- Click the "+" button and select "Create new task".



- Name the task ("Label_Checking").
- In the Subset field, select "Train".
- Open the **images** folder, select all 2040 images, and drag and drop them into the "Select files" field.
- Click "Submit and open".

Create a new task

Basic configuration

* Name
Label_Checking

Project
Price Labels

Subset
Train

Labels
Project labels will be used

* Select files
My computer Connected file share Remote sources Cloud Storage

Click or drag files to this area
You can upload an archive with images, a video, or multiple images

> Advanced configuration

Submit & Open Submit & Continue

5. Annotate Images:

- Click the job number in the Jobs field to open the task.

Back to project

Label_Checking

Task #753569 Created by induwara4 on June 23rd 2024

Assigned to Select a user

Issue Tracker

Subset: Train

Jobs

Copy

Sort by Quick filters Filter Clear filters

Job #977366	Assignee:	Stage:	State:	Frame range:
Created on June 23rd 2024 10:57 Last updated June 23rd 2024 10:57	Select a u...	annotation	New	0-5

- Click the polygon icon.
- Select the appropriate label (Correct or Wrong), set the number of points to 4, and click "Shape".

Draw new polygon

Label: Wrong

Number of points: 4

Shape Track

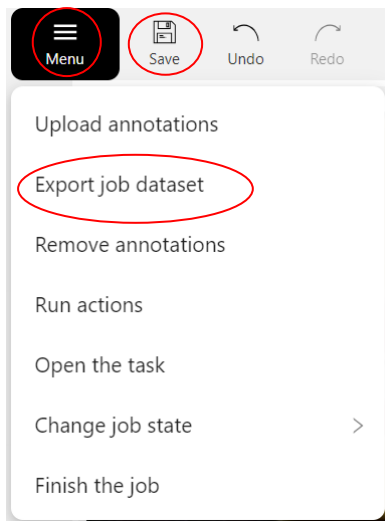
- Draw a polygon around the white space and press "F" to go to the next image.
- Annotate all images accordingly.

WRONG 1 (MANUAL)

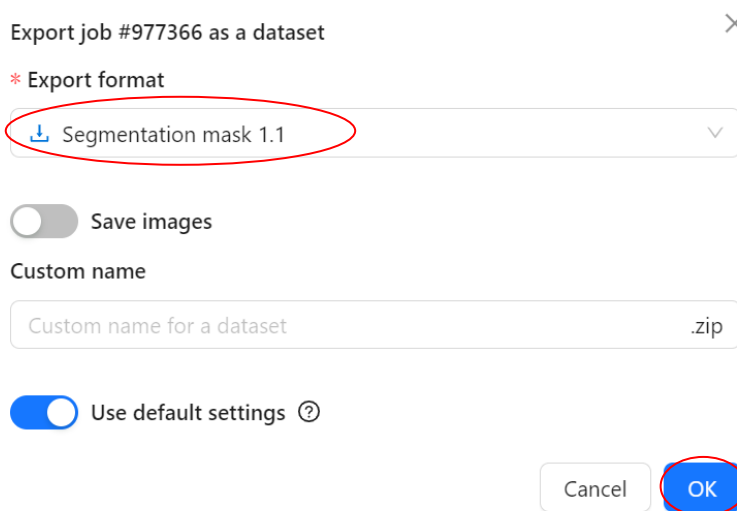
Label: Wrong

6. Save and Export Annotations:

- After annotating all images, click "Save".
- Under the menu, click "Export job dataset".

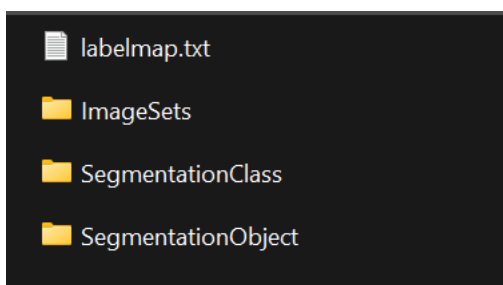


- In the Export format section, select "Segmentation mask 1.1" and click "OK" to download the segmentation masks for the dataset.



7. Extract and Review Masks:

- Extract the downloaded zip file and open it.



- The **labelmap.txt** file contains the colors used for each class in RGB format.

```
# label:color_rgb:parts:actions
Correct:85,104,148::
Wrong:224,88,121::
background:0,0,0::
```

- The **SegmentationClass** folder, contains the segmentation masks for each image.

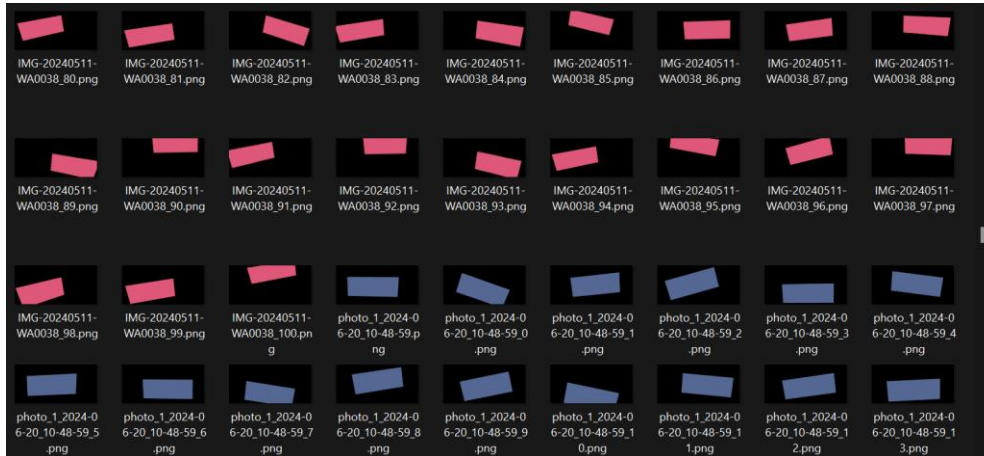


Figure 6: Created Masks in **SegmentationClass** folder

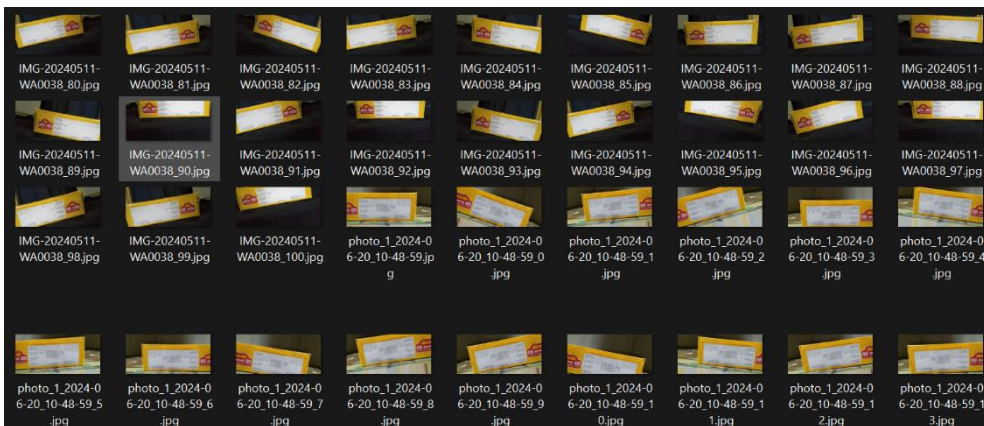


Figure 5: Actual images in **images** folder

2.5.2 Creating the Labels

After creating segmentation masks, the next step is to convert these masks into labels suitable for training our model. Here is how this process was done:

1. A new folder called **masks** was created, and all segmentation masks from the **SegmentationClass** folder were copied into it.
2. Another folder called **labels** was created to store the converted label files.
3. The following code was used to convert the segmentation masks into label files and save them in the **labels** folder:

Code to Convert Masks into Labels :

```
import os
import cv2
import numpy as np

input_dir = './tmp/masks'
output_dir = './tmp/labels'

# Define class RGB colors
class_colors = {
    'Correct': (148, 104, 85),
    'Wrong': (121, 88, 224),
}

# Map class names to numeric labels
class_labels = {
    'Correct': 0,
    'Wrong': 1,
}

def mask_to_polygons(mask, class_label):
    # Find contours
    contours, _ = cv2.findContours(mask, cv2.RETR_EXTERNAL, cv2.CHAIN_APPROX_SIMPLE)

    # Convert contours to polygons
    H, W = mask.shape
    polygons = []
    for cnt in contours:
        if cv2.contourArea(cnt) > 200: # Filter out small contours
            polygon = []
            for point in cnt:
                x, y = point[0]
                polygon.append(x / W)
                polygon.append(y / H)
            polygons.append((class_label, polygon))
    return polygons

def convert_mask_to_labels(input_dir, output_dir, class_colors, class_labels):
    if not os.path.exists(output_dir):
        os.makedirs(output_dir)

    for filename in os.listdir(input_dir):
        image_path = os.path.join(input_dir, filename)
        mask = cv2.imread(image_path)

        all_polygons = []
        for class_name, color in class_colors.items():
            class_label = class_labels[class_name]

            # Create binary mask for each class
            binary_mask = cv2.inRange(mask, color, color)

            # Get polygons for this class
            polygons = mask_to_polygons(binary_mask, class_label)
            all_polygons.extend(polygons)

        # Write to label file
        label_filename = os.path.splitext(filename)[0] + '.txt'
        output_path = os.path.join(output_dir, label_filename)
        with open(output_path, 'w') as f:
            for class_label, polygon in all_polygons:
                f.write(f"{class_label} ")
                for p_, p in enumerate(polygon):
                    if p_ == len(polygon) - 1:
                        f.write(f"{p}\n")
                    else:
                        f.write(f"{p} ")

    # Run the conversion
    convert_mask_to_labels(input_dir, output_dir, class_colors, class_labels)
```

Explanation of the Code:

1. **Library Imports:**
 - `os`: For file and directory operations.
 - `cv2`: For image processing (OpenCV).
 - `numpy`: For numerical operations.
2. **Class Colors and Labels:**
 - `class_colors`: A dictionary mapping class names to their corresponding colors.(In here we use BGR format)
 - `class_labels`: A dictionary mapping class names to numeric labels.
3. **Functions:**
 - `mask_to_polygons`: Finds contours in the binary mask and converts them to polygons normalized to the image dimensions.
 - `convert_mask_to_labels`: Iterates through each mask in the `input_dir`, extracts polygons for each class, and writes them to a label file in the `output_dir`.
4. **Processing:**
 - For each mask, the function creates binary masks for each class based on the RGB colors.
 - It then finds and converts contours to polygons, which are saved to corresponding label files.

2.6. Training Phase

2.6.1 YOLOv8

What is YOLOv8?

YOLOv8 (You Only Look Once version 8) is the latest iteration in the YOLO family of object detection models. YOLO models are renowned for their speed and accuracy, making them ideal for real-time object detection tasks. YOLOv8 continues this legacy by incorporating state-of-the-art techniques to improve detection accuracy and processing speed.

How Does YOLOv8 Work?

YOLOv8, like its predecessors, divides the input image into a grid and predicts bounding boxes and class probabilities for each grid cell. The main innovations in YOLOv8 include:

- **Improved Backbone:** The backbone network in YOLOv8 is optimized for better feature extraction.
- **Efficient Neck:** The neck component of the model refines the features further to improve detection accuracy.
- **Advanced Head:** The detection head predicts bounding boxes, class probabilities, and objectness scores with higher precision.
- **Enhanced Loss Function:** The loss function in YOLOv8 has been fine-tuned to minimize errors more effectively.

Why Use YOLOv8 for This Project?

- **Accuracy:** YOLOv8 provides high detection accuracy, crucial for identifying misprinted labels on milk packets.
- **Speed:** The model's ability to process images quickly makes it suitable for real-time applications.
- **Flexibility:** YOLOv8 can be fine-tuned to detect custom objects, making it ideal for this specific task.

2.6.2 Initial Setup

To set up the training environment, the following steps were taken:

1. **Folder Structure:**
 - A new folder called **Label_Checking** was created in Google Drive.
 - Under this folder, three subfolders/files were created:
 1. **tmp:** This folder contains the **images** and **labels** folders created previously.
 2. **dataset.yaml:** This file contains the dataset configuration for YOLOv8.
 3. **training.ipynb:** This Jupyter Notebook file is used for training the model in Google Colab.
2. **Dataset Configuration (dataset.yaml):** The **dataset.yaml** file was created using a text editor and contains the following configuration:

```
1 # dataset.yaml
2 path: '/content/gdrive/My Drive/Label_Checking/tmp' # dataset root dir
3 train: images
4 val: images
5
6 nc: 2 # Number of classes
7
8 names: ['Correct', 'Wrong']
```

- **path:** The root directory of the dataset.
- **train** and **val:** Subdirectories containing the training and validation images, respectively.
- **nc:** Number of classes (2 in this case: 'Correct' and 'Wrong').
- **names:** List of class names.

3. **Google Colab Setup:**
 - A new Jupyter Notebook file named **training.ipynb** was created in Google Colab.
 - The notebook was used to train the YOLOv8 model using the dataset stored in Google Drive.

2.6.3 Training Script

The provided script sets up and trains a YOLOv8 model in Google Colab using a custom dataset stored in Google Drive. Below is a step-by-step explanation of the script:

1. Mount Google Drive:

```
from google.colab import drive
drive.mount('/content/gdrive')
```

- The first two lines mount Google Drive to the Colab environment. This allows access to the dataset and saving results directly to Google Drive. The `drive.mount` function prompts for authorization and mounts the Google Drive at `/content/gdrive`.

2. Install YOLOv8 Library:

```
!pip install ultralytics
```

- This line installs the `ultralytics` package, which includes the YOLOv8 model and other utilities. The `!pip install` command is used to install Python packages in the Colab environment.

3. Import Libraries:

```
import os
from ultralytics import YOLO
```

- The `os` module is imported for handling file and directory operations.
- The `YOLO` class from the `ultralytics` package is imported to load and train the YOLOv8 model.

4. Load Pretrained YOLOv8 Model:

```
model = YOLO('yolov8n-seg.pt') # load a pretrained model (recommended for training)
```

- A pretrained YOLOv8 segmentation model is loaded using the `YOLO` class. The model file `'yolov8n-seg.pt'` is a lightweight version of YOLOv8 optimized for segmentation tasks.

5. Train the Model:

```
model.train(data='/content/gdrive/My Drive/Label_Checking/dataset.yaml', epochs=10, imgsz=640)
```

- The `train` method is called on the YOLO model object to start the training process.
- The `data` parameter specifies the path to the dataset configuration file (`dataset.yaml`), which contains information about the dataset structure.
- The `epochs` parameter is set to 10, indicating that the training will run for 10 epochs.
- The `imgsz` parameter is set to 640, specifying the image size to be used for training.

2.6.4. Model Architecture

The summary provided below gives a detailed layer-by-layer breakdown of the YOLOv8 segmentation model, which is used for training to identify misprinted labels on milk packets.

	from	n	params	module	arguments
0	-1	1	464	ultralytics.nn.modules.conv.Conv	[3, 16, 3, 2]
1	-1	1	4672	ultralytics.nn.modules.conv.Conv	[16, 32, 3, 2]
2	-1	1	7360	ultralytics.nn.modules.block.C2f	[32, 32, 1, True]
3	-1	1	18560	ultralytics.nn.modules.conv.Conv	[32, 64, 3, 2]
4	-1	2	49664	ultralytics.nn.modules.block.C2f	[64, 64, 2, True]
5	-1	1	73984	ultralytics.nn.modules.conv.Conv	[64, 128, 3, 2]
6	-1	2	197632	ultralytics.nn.modules.block.C2f	[128, 128, 2, True]
7	-1	1	295424	ultralytics.nn.modules.conv.Conv	[128, 256, 3, 2]
8	-1	1	460288	ultralytics.nn.modules.block.C2f	[256, 256, 1, True]
9	-1	1	164608	ultralytics.nn.modules.block.SPPF	[256, 256, 5]
10	-1	1	0	torch.nn.modules.upsampling.Upsample	[None, 2, 'nearest']
11	[-1, 6]	1	0	ultralytics.nn.modules.conv.Concat	[1]
12	-1	1	148224	ultralytics.nn.modules.block.C2f	[384, 128, 1]
13	-1	1	0	torch.nn.modules.upsampling.Upsample	[None, 2, 'nearest']
14	[-1, 4]	1	0	ultralytics.nn.modules.conv.Concat	[1]
15	-1	1	37248	ultralytics.nn.modules.block.C2f	[192, 64, 1]
16	-1	1	36992	ultralytics.nn.modules.conv.Conv	[64, 64, 3, 2]
17	[-1, 12]	1	0	ultralytics.nn.modules.conv.Concat	[1]
18	-1	1	123648	ultralytics.nn.modules.block.C2f	[192, 128, 1]
19	-1	1	147712	ultralytics.nn.modules.conv.Conv	[128, 128, 3, 2]
20	[-1, 9]	1	0	ultralytics.nn.modules.conv.Concat	[1]
21	-1	1	493056	ultralytics.nn.modules.block.C2f	[384, 256, 1]
22	[15, 18, 21]	1	1004470	ultralytics.nn.modules.head.Segment	[2, 32, 64, [64, 128, 256]]

YOLOv8n-seg summary: 261 layers, 3264006 parameters, 3263990 gradients, 12.1 GFLOPs

Breakdown of YOLOv8 Model Components

1. Convolutional Layers (Conv):

- **Purpose:** Extract features from input images by applying filters.
- **Example:**
 - `ultralytics.nn.modules.conv.Conv [3, 16, 3, 2]`: This layer takes an input with 3 channels, applies 16 filters of size 3x3, and downsamples the image by a factor of 2.

2. C2f Blocks:

- **Purpose:** Implement a more complex feature extraction by stacking multiple convolutional layers and adding shortcut connections.
- **Example:**
 - `ultralytics.nn.modules.block.C2f [32, 32, 1, True]`: This block uses 32 filters and processes the feature maps in a more refined manner.

3. Spatial Pyramid Pooling Fast (SPPF):

- **Purpose:** Combines features from different scales to improve the model's ability to detect objects of varying sizes.
- **Example:**
 - `ultralytics.nn.modules.block.SPPF [256, 256, 5]`: Applies pooling operations to feature maps to aggregate contextual information.

4. Upsampling Layers:

- **Purpose:** Increase the spatial resolution of feature maps to allow finer object localization and segmentation.
- **Example:**
 - `torch.nn.modules.upsampling.Upsample [None, 2, 'nearest']`: Doubles the size of the feature maps using nearest-neighbor interpolation.

5. Concatenation Layers (Concat):

- **Purpose:** Merge feature maps from different layers to combine information from various stages of the network.
- **Example:**

- `ultralalytics.nn.modules.conv.Concat [1]`: Concatenates feature maps from two or more layers.

6. Segmentation Head:

- **Purpose:** Generates segmentation masks that delineate the precise boundaries of detected objects.
- **Example:**
 - `ultralalytics.nn.modules.head.Segment [2, 32, 64, [64, 128, 256]]`: Outputs segmentation masks for the two classes (Correct and Wrong) using multiple channels and feature scales.

Model Statistics

- **Total Layers:** 261
- **Total Parameters:** 3,264,006
- **Trainable Parameters:** 3,263,990
- **GFLOPs (Giga Floating Point Operations):** 12.1

2.7. Evaluation Phase

After the training is complete, the trained model weights and the metrics of the training process are saved in the `content/runs` directory by default. To evaluate the results, the following code can be used to visualize the training outputs.

Code for Evaluating and Plotting Results

```
import matplotlib.pyplot as plt
import glob

# Function to plot the images
def plot_results(file_path):
    img = plt.imread(file_path)
    plt.figure(figsize=(10, 10))
    plt.imshow(img)
    plt.axis('off')
    plt.show()

# Define source path where results are saved
source_path = '/content/runs'

# Check if source path exists
if os.path.exists(source_path):
    print(f"Source path exists: {source_path}")
    result_files = glob.glob(os.path.join(source_path, '**', '*.png'), recursive=True)

    # Plot each result image
    for file_path in result_files:
        print(f"Plotting: {file_path}")
        plot_results(file_path)
else:
    print(f"Source path does not exist: {source_path}")
```

Explanation of the code:

1. Import Libraries:

- `matplotlib.pyplot`: Used for plotting and visualizing images.
- `glob`: Used to retrieve files/pathnames matching a specified pattern.
- `os`: Used for interacting with the operating system (e.g., checking if a directory exists).

2. Function Definition - `plot_results`:

- **Input**: Takes the file path of an image.
- **Process**: Reads and displays the image using `matplotlib`.
- **Output**: Displays the image without axes for a cleaner look.

3. Set the Source Path:

- Defines the directory where the training results are stored (`/content/runs`).

4. Check Directory Existence:

- Uses `os.path.exists` to check if the specified directory exists.
- If the directory exists, it retrieves all PNG files in the directory (and its subdirectories) using `glob`.

5. Plotting the Results:

- Iterates through each result file found in the directory.
- Calls the `plot_results` function to display each image.

Results Validation:

After running the above code the following results were obtained :

1. Loss Curves :

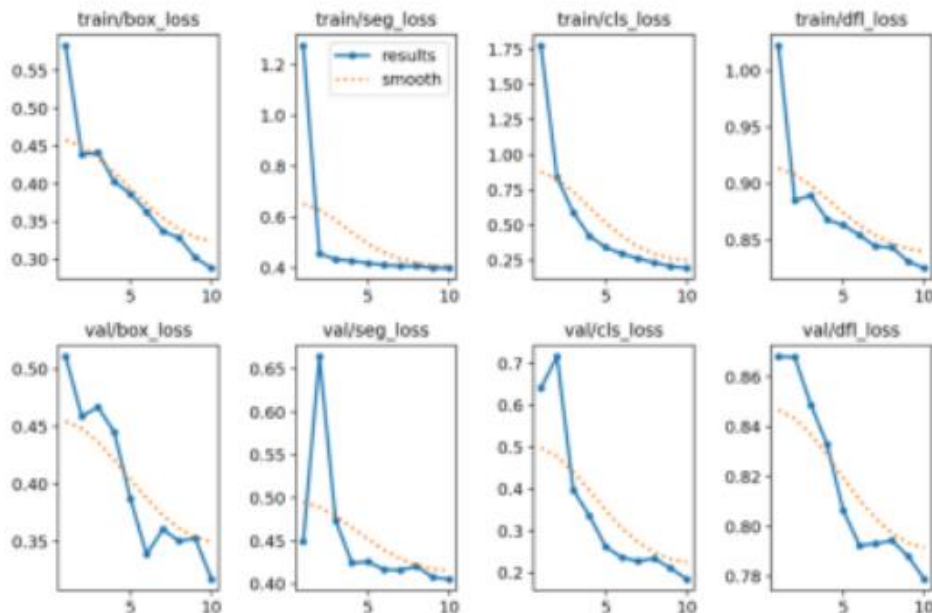


Figure 7: Loss curves

- Loss curves are crucial for understanding how well the model is learning during training.

- **Box Loss:** Measures the localization error (i.e., how well the model predicts the bounding boxes).
 - **Segmentation Loss:** Pertains to the accuracy of the segmentation masks, which are relevant if instance segmentation is used.
 - **Class Loss:** Reflects how well the model classifies objects within bounding boxes.
 - **DFL Loss:** If used, it focuses on additional features like domain-specific features or adaptations.
- The loss curves for both training and validation sets (Box Loss, Segmentation Loss, Class Loss, DFL Loss) consistently decreased across epochs, indicating that the model was learning effectively.

2. Confusion Matrix

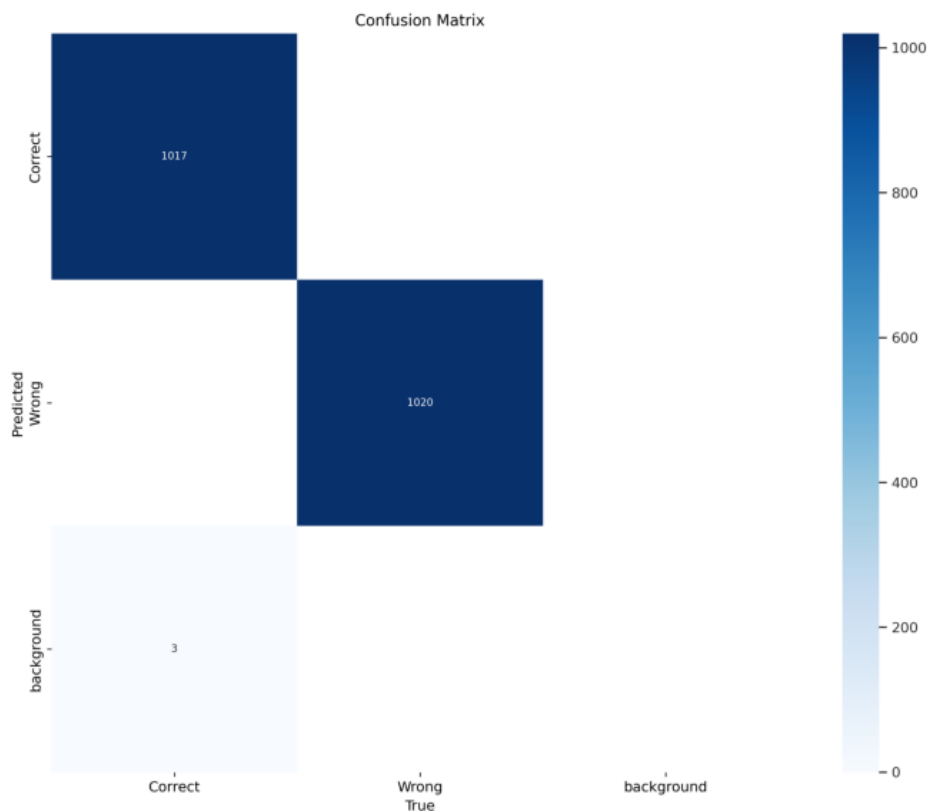


Figure 8: Confusion Matrix

- The confusion matrix provided insights into the model's classification performance:
 - **True Positives (TP):** High values indicated accurate identification of misprinted labels.
 - **True Negatives (TN):** Demonstrated correct rejection of non-misprinted labels.
 - **False Positives (FP) and False Negatives (FN):** Were minimized, indicating fewer instances of misclassification.

2.8. Real Time Predictions

2.8.1. Downloading Model Weights

To facilitate real-time predictions in a local environment, the trained model weights (`best.pt` and `last.pt`) need to be downloaded from Google Colab using the following code:

```
from google.colab import files

# Define paths to the model weights
best_model_path = '/content/runs/segment/train/weights/best.pt' # Update this path if necessary
last_model_path = '/content/runs/segment/train/weights/last.pt' # Update this path if necessary

# Check if the files exist and then download them
if os.path.exists(best_model_path):
    files.download(best_model_path)
else:
    print(f"File not found: {best_model_path}")

if os.path.exists(last_model_path):
    files.download(last_model_path)
else:
    print(f"File not found: {last_model_path}")
```

2.8.2. Implementing Real-Time Object Detection

To perform real-time object detection using the downloaded YOLOv8 model, we implemented the following code in the local environment with jupyter notebook:(prediction.ipnb)

Code Explanation:

1. Importing Libraries:

```
import os
from ultralytics import YOLO
import cv2
```

- `os`: Provides functionalities to interact with the operating system, used here to manage file paths.
- `ultralytics.YOLO`: Imports the YOLO object detection model from the Ultralytics library for use in the script.
- `cv2`: OpenCV library for handling real-time video capture and image processing.

2. Setting up Video Capture:

```
# Set up the video capture for the webcam
cap = cv2.VideoCapture(0) # 0 is the default camera
```

- Initializes video capture from the default camera (0 denotes the default camera device).

3. Checking Video Capture:

```
# Check if the webcam is opened correctly
if not cap.isOpened():
    print("Error: Could not open webcam.")
    exit()
```

- Checks if the camera capture is successful (`cap.isOpened()`). If not, prints an error message and exits the script.

4. Defining Model Path and Loading Model:

```
# Path to the trained model
model_path = os.path.join('.', 'runs', 'segment', 'train', 'weights', 'last.pt')

# Load the custom model
model = YOLO(model_path)
```

- Constructs the path to the trained YOLO model (`last.pt` file).
- Loads the YOLO model using the specified path.

5. Setting Detection Threshold:

```
threshold = 0.5
```

- Defines the confidence threshold above which detected objects will be displayed.

6. Real-Time Detection Loop:

```
while True:
    ret, frame = cap.read()
    if not ret:
        print("Error: Could not read frame from webcam.")
        break
```

- Enters a loop to continuously read frames (`cap.read()`) from the webcam.
- If unable to read a frame, prints an error message and breaks out of the loop.

7. Performing Object Detection:

```
results = model(frame)[0]
```

- Passes the current frame (`frame`) to the YOLO model (`model`) for object detection.
- `model(frame)` returns detection results, and `[0]` selects the first (and typically only) batch of results.

8. Drawing Boxes and Labels:

```
for result in results.bboxes.data.tolist():
    x1, y1, x2, y2, score, class_id = result

    if score > threshold:
        label = results.names[int(class_id)].upper()
        color = (0, 255, 0) if label != "WRONG" else (0, 0, 255) # Green for correct, red for wrong

        cv2.rectangle(frame, (int(x1), int(y1)), (int(x2), int(y2)), color, 4)
        cv2.putText(frame, label, (int(x1), int(y1 - 10)),
                    cv2.FONT_HERSHEY_SIMPLEX, 1.3, color, 3, cv2.LINE_AA)
```

- Iterates through each detection result (`results.bboxes.data.tolist()`).
- Checks if the confidence score (`score`) exceeds the threshold.
- Draws a bounding box (`cv2.rectangle`) around the detected object with a label (`cv2.putText`).

9. Displaying Real-Time Detection:

```
# Display the frame with detection boxes
cv2.imshow('YOLO Real-Time Detection', frame)
```

- Displays the current frame (`frame`) with overlaid detection boxes and labels using `cv2.imshow`.

10. Exiting Real-Time Detection:

```
# Press 'q' to exit the real-time detection
if cv2.waitKey(1) & 0xFF == ord('q'):
    break

# Release resources
cap.release()
cv2.destroyAllWindows()
```

- Releases the video capture (`cap`) and closes all OpenCV windows (`cv2.destroyAllWindows()`).

2.8.3. Results of the real time detection

Running above code enabled us to successfully identify the correct and wrong labels, as shown in the below

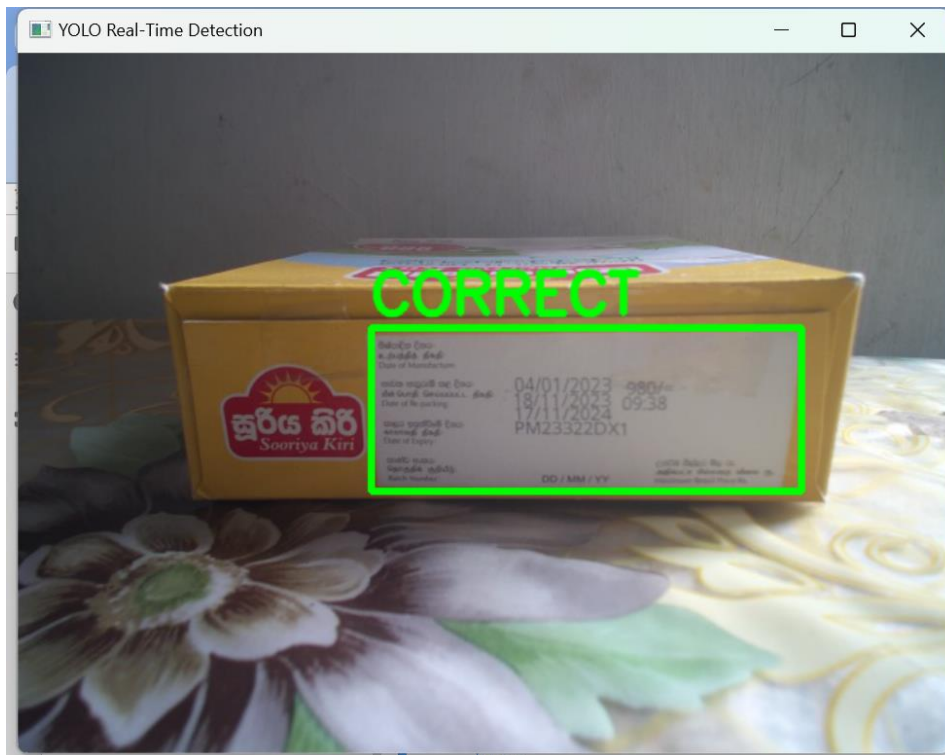


Figure 9: Real time detection of a correct label



Figure 10: Real time detection of a wrong label

2.9. Integration with Arduino for Automated Response

In this section, we describe the integration of YOLO real-time object detection with an Arduino microcontroller to enable automated responses based on detected labels.

2.9.1. Arduino Setup

The Arduino is configured to receive signals from the computer via serial communication. Upon receiving specific signals, it activates an LED to indicate a detected event. Below is the Arduino code used for this setup:

```
void setup() {
    Serial.begin(9600); // Start serial communication at 9600 baud
    pinMode(LED_BUILTIN, OUTPUT); // Use the built-in LED for demonstration
}

void loop() {
    if (Serial.available() > 0) {
        char signal = Serial.read();
        if (signal == 'W') { // Assuming 'W' stands for "wrong"
            digitalWrite(LED_BUILTIN, HIGH); // Turn on LED
            delay(100); // Keep it on for 0.1 second
            digitalWrite(LED_BUILTIN, LOW); // Turn off LED
        }
    }
}
```

2.9.2. Python Code Integration

The Python application interfaces with the webcam, performs real-time object detection using YOLO, and sends signals to the Arduino upon detecting specific labels. Here is the Python code snippet illustrating this integration:

```
import os
from ultralytics import YOLO
import cv2
import serial
import serial.tools.list_ports
import time

arduino_port = 'COM3'
baud_rate = 9600
ser = serial.Serial(arduino_port, baud_rate)
time.sleep(2) # Wait for the serial connection to initialize

# Set up the video capture for the webcam
cap = cv2.VideoCapture(0) # 0 is the default camera
```

```

# Check if the webcam is opened correctly
if not cap.isOpened():
    print("Error: Could not open webcam.")
    exit()

# Path to the trained model
model_path = os.path.join('.', 'runs', 'segment', 'train', 'weights', 'last.pt')

# Load the custom model
model = YOLO(model_path)

threshold = 0.5

while True:
    ret, frame = cap.read()
    if not ret:
        print("Error: Could not read frame from webcam.")
        break

    results = model(frame)[0]
    wrong_detected = False

    for result in results.boxes.data.tolist():
        x1, y1, x2, y2, score, class_id = result

        if score > threshold:
            label = results.names[int(class_id)].upper()
            color = (0, 255, 0) if label != "WRONG" else (0, 0, 255)
            if label == "WRONG":
                wrong_detected = True

            cv2.rectangle(frame, (int(x1), int(y1)), (int(x2), int(y2)), color, 4)
            cv2.putText(frame, label, (int(x1), int(y1 - 10)),
                        cv2.FONT_HERSHEY_SIMPLEX, 1.3, color, 3, cv2.LINE_AA)

    if wrong_detected:
        ser.write(b'W') # Send signal to Arduino

    # Display the frame with detection boxes
    cv2.imshow('YOLO Real-Time Detection', frame)

    # Press 'q' to exit the real-time detection
    if cv2.waitKey(1) & 0xFF == ord('q'):
        break

# Release resources
cap.release()
ser.close()
cv2.destroyAllWindows()

```


2.9.3. Results and Observations

Running the integrated system successfully detects and responds to labels in real-time. The Arduino's LED lights up briefly upon detecting a "WRONG" label, demonstrating effective communication and automated response capabilities.

Appendix: Daily Progress

January 29-February 4

Initial Submission

- **January 29:** Project proposal drafted and submitted for initial review.
- **January 30:** Feedback received; revisions made to proposal.
- **January 31:** Finalized project proposal and submitted.
- **February 1-4:** Awaiting approval and preparing initial research materials.

February 5-February 11

Finding Information - Market and Other Analysis

- **February 5:** Began market analysis; identified key competitors and market needs.
- **February 6-7:** Conducted detailed analysis of existing solutions and their limitations.
- **February 8:** Compiled data on market trends and potential demand.
- **February 9:** Identified key features needed for our system to stand out.
- **February 10-11:** Summarized findings and prepared a report on market analysis.

February 11-February 18

Finding Information - Stakeholders

- **February 11:** Identified potential stakeholders and their interests.
- **February 12:** Reached out to industry experts for insights.
- **February 13-15:** Conducted interviews and surveys with potential users.
- **February 16:** Analyzed feedback from stakeholders.
- **February 17-18:** Compiled a comprehensive report detailing stakeholder requirements and expectations.

February 19-February 25

Initial PCB Design and Software Setup

- **February 19:** Started conceptual design of PCB and set up software development environment.
- **February 20-21:** Determined essential components for PCB and selected Python and OpenCV for image processing.
- **February 22:** Developed initial script to capture and preprocess images from the camera.
- **February 23-24:** Created initial schematic diagrams and implemented basic image segmentation techniques.
- **February 25:** Reviewed and refined PCB design; tested initial image processing pipeline with sample data.

February 26-March 3

Research and Software Development

- **February 26:** Researched available sensors and their integration into the system.
- **February 27-28:** Evaluated different microcontrollers for system compatibility.
- **March 1:** Identified suitable image processing libraries for misprint detection; integrated YOLO object detection model.
- **March 2-3:** Investigated communication protocols between PCB and software; trained YOLO model on sample misprint data.

March 4-March 10

Conceptual Design Basic Structure and Software Optimization

- **March 4:** Drafted the basic structural layout of the system.
- **March 5-6:** Defined the workflow for detecting misprints and triggering the rejection mechanism; optimized detection accuracy and speed.
- **March 7-8:** Designed the initial mechanical setup for the conveyor belt system.
- **March 9-10:** Integrated feedback to refine the conceptual design; developed interface for real-time detection and feedback.

March 11-March 17

Research and Software-Hardware Integration

- **March 11:** Explored advanced image processing techniques for accurate detection.
- **March 12:** Studied machine learning models for improving detection accuracy; integrated software with PCB for signal transmission.
- **March 13-15:** Tested different algorithms for real-time processing.
- **March 16-17:** Compiled research findings and prepared a summary report; refined software based on hardware integration tests.

March 18-March 24

PCB Initial and Prototype Development

- **March 18:** Developed a detailed PCB layout.
- **March 19-21:** Simulated the PCB design for validation.
- **March 22:** Made necessary adjustments based on simulation results.
- **March 23-24:** Prepared the final design for prototype production.

March 25-March 31

Prototype Assembly and Testing

- **March 25-26:** Assembled initial hardware prototype; implemented additional software features for misprint classification.

- **March 27:** Integrated PCB with sensors and microcontroller; enhanced real-time processing capabilities.
- **March 28-30:** Conducted initial tests on the prototype; refined real-time processing software.
- **March 31:** Documented prototype assembly and testing results.

April 1-April 7

Prototype Refinement and Integration Testing

- **April 1-3:** Continued refining the prototype based on test results.
- **April 4-5:** Implemented software integration for real-time processing; conducted comprehensive software-hardware integration tests.
- **April 6-7:** Conducted comprehensive testing of the prototype; finalized and documented software for deployment.

April 8-April 14

Ordering Components

- **April 8:** Finalized the list of required components.
- **April 9:** Sourced suppliers and obtained quotes.
- **April 10-11:** Placed orders for all necessary components.
- **April 12-14:** Followed up on order confirmations and delivery schedules.

April 15-April 21

Ordering PCB and Final Initial Mechanical Parts Design

- **April 15:** Finalized PCB design based on prototype testing feedback.
- **April 16-17:** Placed order for PCBs.
- **April 18:** Designed final initial mechanical parts for the conveyor belt system.
- **April 19-21:** Ordered mechanical parts and confirmed delivery dates.

April 29-May 5




Design Document Review and PCB Soldering

- **April 29:** Completed the design document and prepared it for review.
- **April 30:** Submitted the design document for review by another group.
- **May 1:** Received feedback and signed declaration from the review group.
- **May 2:** Incorporated feedback and finalized the design document.
- **May 3:** PCB and parts received.
- **May 4-5:** Soldered components onto the PCB and performed initial tests.

References

- Ultralytics, "YOLO Object Detection Documentation," Ultralytics, 2024. [Online]. Available: <https://docs.ultralytics.com>.
- G. Jocher, A. Chaurasia, J. Qiu, and A. Stoken, "YOLOv5," 2020. [Online]. Available: <https://github.com/ultralytics/yolov5>.
- Arduino, "Arduino Language Reference," Arduino, 2024. [Online]. Available: <https://www.arduino.cc/reference/en/>.
- OpenCV, "OpenCV Documentation," OpenCV, 2024. [Online]. Available: <https://docs.opencv.org/>.

Reviews

Reviewer	Index No	Signature
LAKSHAN W.D.T.	210335T	
GUNAWARDANA W.M.T.V.	210198A	
WICKRAMASINGHE M.M.M.	210707L	
RAJAPAKSHA N.N.	210504L	