Ryerson University

**Department of Aerospace Engineering**
Faculty of Engineering & Architectural Science

**Semester, Year:**

| | |
|---|---|
| **Semester, Year:** | ☑ Fall ☐ Winter ☐ Spring/Summer 2023 |
| **Course Code:** | AER 850 |
| **Course Title:** | Introduction to Machine Learning |
| **Section Number:** | 01 |
| **Instructor:** | Reza Faieghi |
| **Submission:** | ☐ Assignment ☐ Lab Report ☑ Project Report ☐ Thesis ☐ Other: _____ |
| **Due Date:** | December 17, 2023 |

# Project 3 Report

## December 17, 2023

| Authors/Contributors: | Student Number (XXXX12345): | Signature*: |
|---|---|---|
| Kotasthane, Gaurang | 500922614 | G.K. |
| | | |
| | | |
| | | |
| | | |

**By signing the above you attest that you have contributed to this submission and confirm that all work you contributed to this submission is your own work. Any suspicion of copying or plagiarism in this work will result in an investigation of Academic Misconduct and may result in a "0" on the work, and "F" in the course, or possibly more severe penalties, as well as a Disciplinary Notice on your academic record under the Student Code of Academic Conduct, which can be found online at [www.ryerson.ca/academicintegrity/students/ryersons-academic-integrity-policy-policy-60/](www.ryerson.ca/academicintegrity/students/ryersons-academic-integrity-policy-policy-60/)*

# Table of Contents

# Discussion

## Object Masking

### Thresholding

#### Method

Image thresholding can be done using a few different methods. These methods are readily available to be implemented using the OpenCV library. In this project the following threshold functions were implemented

1. Threshold Binary
2. Threshold Binary Inverse
3. Threshold Trunc
4. Threshold to zero
5. Threshold to zero inverse

All these functions work on a simple principle, the look at the no of pixels at a particular location in the image and then depending on the threshold that was set, it assigns a value of intensity for that location in the image. The report looks at 5 different methods to choose the optimal thresholding function for this project.
The threshold value for the functions was chosen based on a histogram of the intensity and then we can look at different peaks and choose a threshold in between those peaks to effectively separate the feature that needs to be extracted from the image. As illustrated below, a threshold of 60 was chosen for this project based on the histogram.

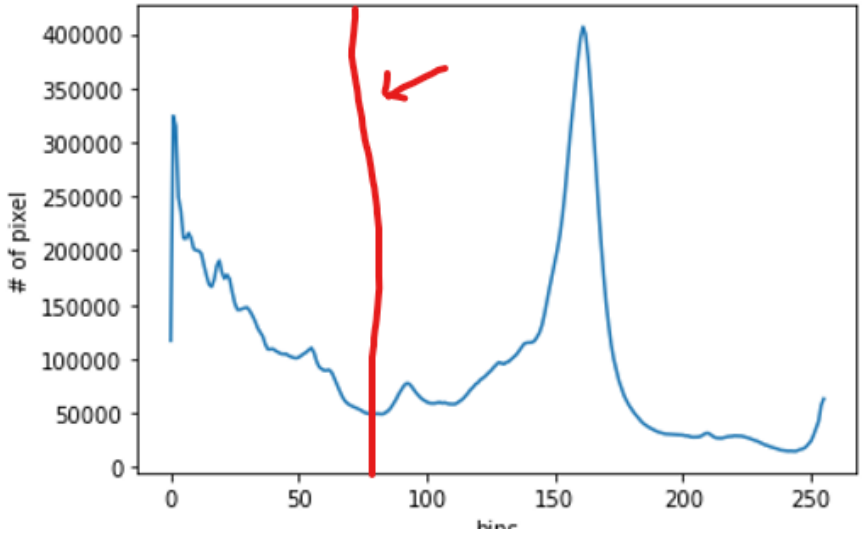


Figure 1: Histogram

Results

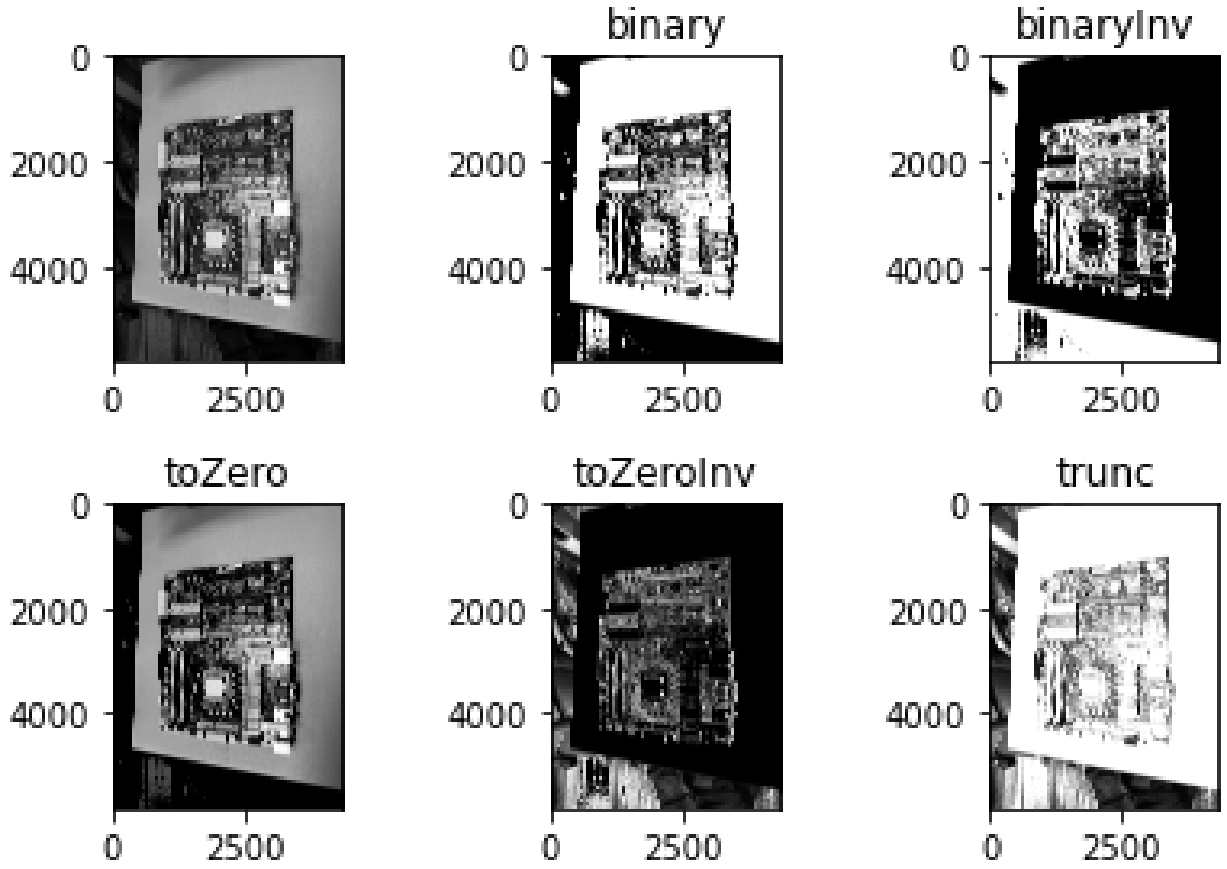


Figure 2: Plot of different thresholding functions and their results

By analyzing the plot, it is apparent that the binaryInv function works the best to separate the motherboard from its background.

I will be using Contour Detection to mask out the image of the mother board, but I have also attached my results from using Canny and Harris Edge detection methods since I explored those as well. One of the possible reasons for nor choose Canny/Harris was that this provided image has a lot of edges which makes the algorithm hard to find corners of the motherboard at a certain frequency that can be used to mask out the mother board.

Results for Contour detection is attached below.

# Edge Detection

## Canny Edge Detection

### Results – Canny Edge Detection Method

Figure 3: Canny Edge detection

## Harris Edge Detection

### Results – Harris Edge Detection Method

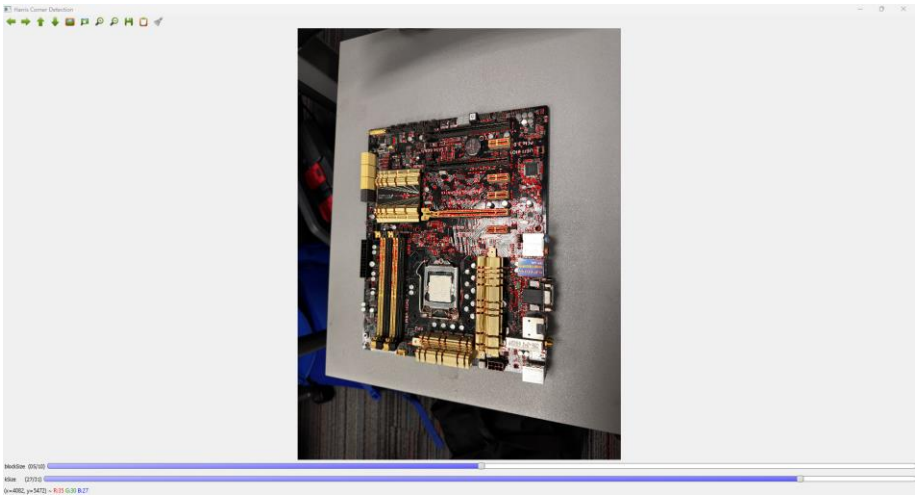

Figure 4: Harris Edge Detection

# Contour Detection

## Results – Contour Detection Method

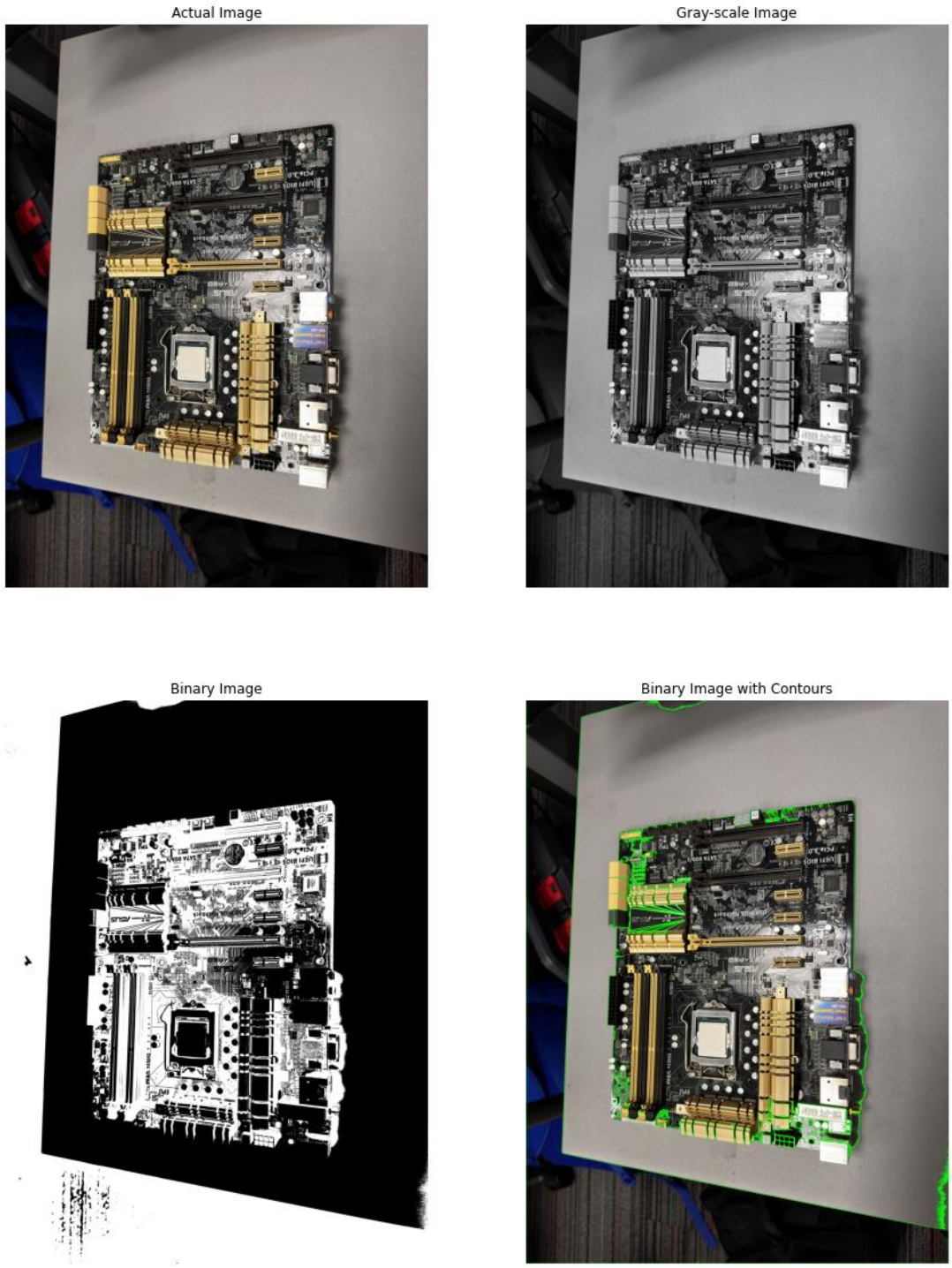


Figure 5: Contour Detection

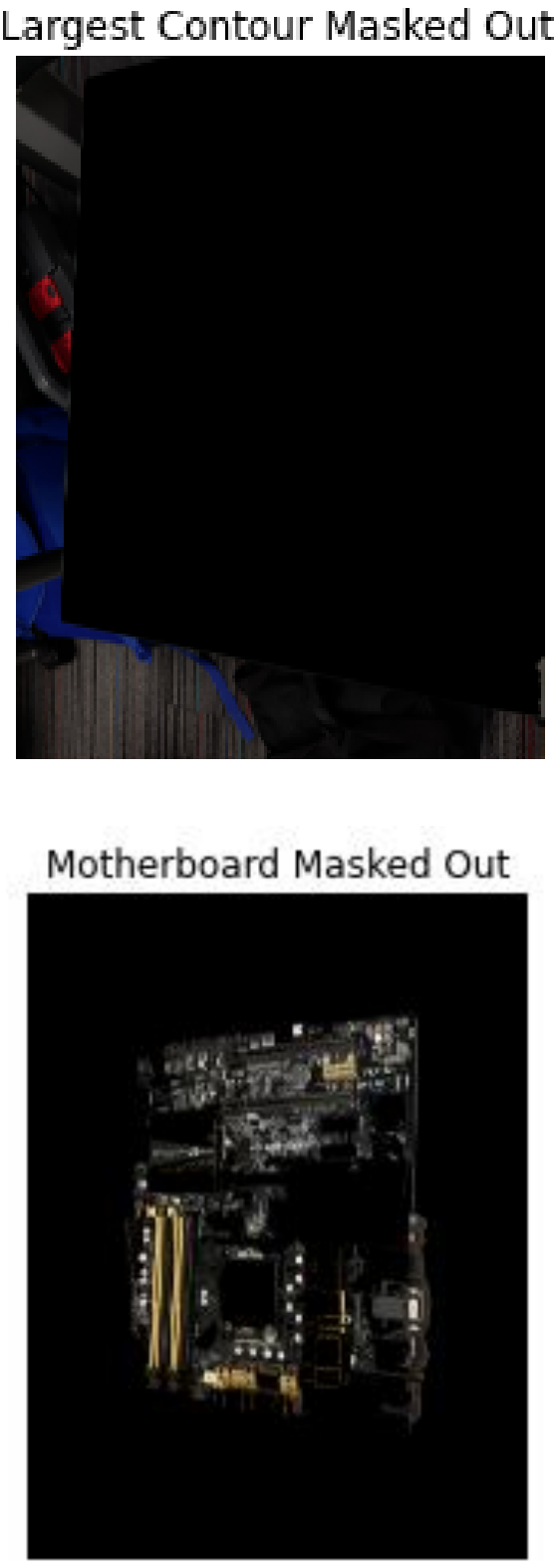


Figure 6: Contour Detection Masking Results

# Contour Detection and Masking Discussion

## Approach

The approach involved several key steps to isolate the motherboard from its background in the given image. Initially, the image was read and preprocessed by converting it to grayscale. This conversion is crucial for the subsequent thresholding process, where the image is transformed into a binary format. Binary thresholding simplifies the image to black and white pixels, making it easier to detect.

Once in binary form, the image was then processed using OpenCV's contour detection algorithm. This powerful tool in image processing identifies continuous lines or curves (contours) in the binary image, representing the boundaries of objects. The areas of these detected contours were calculated using OpenCV's `cv.contourArea` function.

The largest contour area corresponded to the table on which the motherboard was placed, rather than the motherboard itself. Hence, the second largest contour within the array of detected contours, under the assumption that this would more accurately represent the motherboard.

Utilizing this second largest contour, a mask was created and applied to the original image. This process effectively isolated the motherboard from the background, highlighting it as a distinct entity. The use of contours for this purpose showcases the effectiveness of shape analysis techniques in computer vision for object isolation and feature extraction in complex images.

# YOLO v8 Training & Evaluation

## Model Training Process

In this project the model was trained on a custom dataset that was provided and using YOLOv8. For the training I trained two models and I'll be sharing the thought process and the results below.

As highlighted in the project, the model can only be tuned on three hyperparameters, namely, number of epochs, batch size and image size. The higher the value of these parameters results in higher accuracy and better performing model.

To train, Google Collab was used since it was impossible to train it locally.]

Before going into the architecture and the results of the models, it is important to highlight what I tried and didn't seem to work.

1. High batch size: Since we are dependent on the virtual ram access provided by Google Collab. The default batch size for the 'batch' argument in the training of YOLO is 16. Using this value, the program was always stopped since Google Collab only gives access to 15 GB of VRAM while a batch size of 16 requires much more VRAM than that. Through lots of trial and error, the maximum batch size was determined to be 4. The VRAM used for this was about 10-14 GB which was withing the limits of Google Collab.

Please note that I have included my findings for **two** (**2)** models. This report also has results and evaluations for both the models.

## Model 1

### Architecture

```
!yolo task=detect mode=train model=yolov8m.pt
data=data.yaml epochs=100 imgsz=1216 plots=True batch=2
```

Epochs = 100
Imgsz= 1216
Batch=2

## Results – Model 1

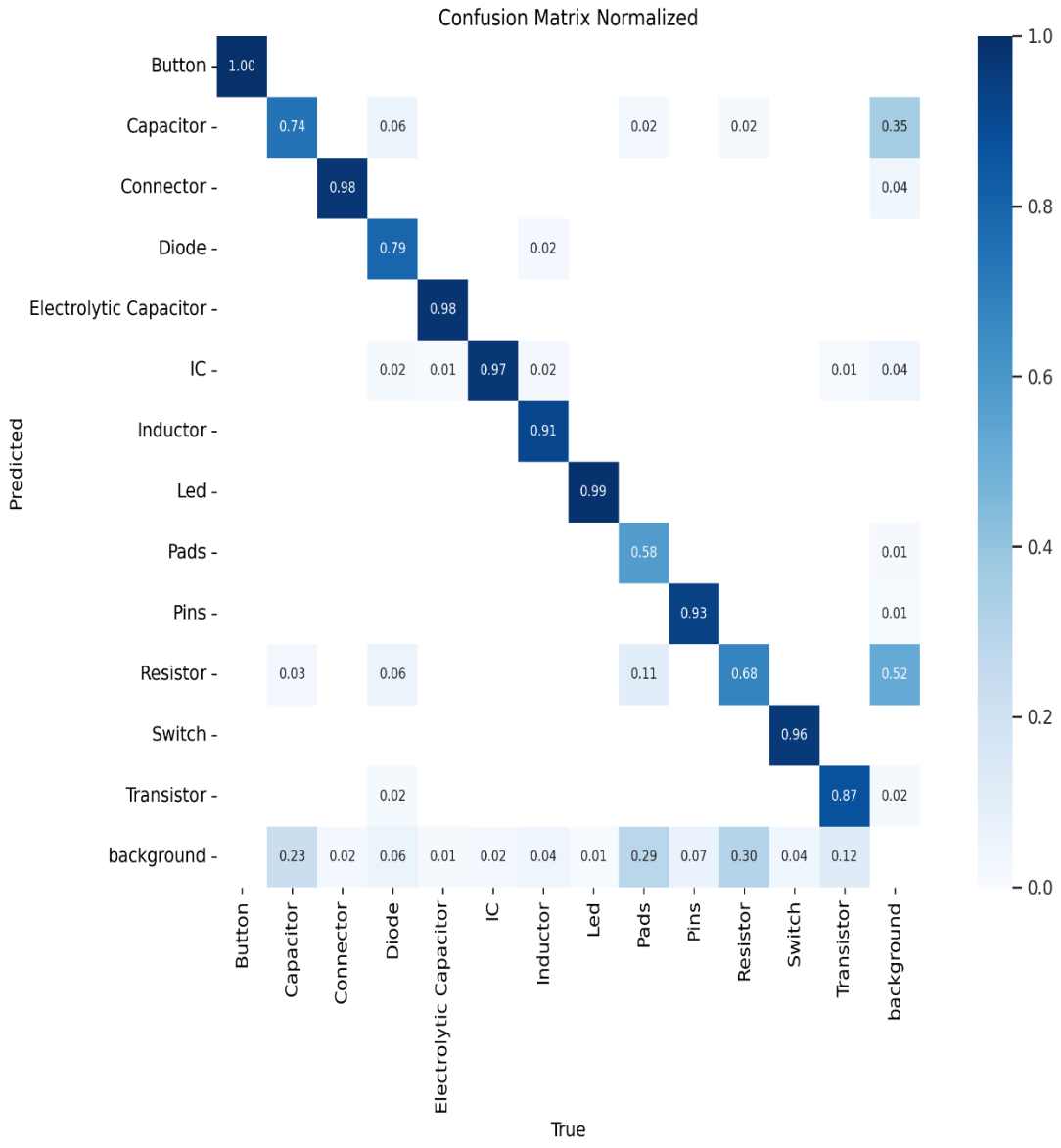


Figure 7. Normalized Confusion Matrix (Model 1)

The main diagonal represents the proportion of correct predictions for each class (True positives). It can be inferred from this there are some classes that are misclassified a lot more times than others. For example, ‘Pads’ class has only a 0.58 value on that diagonal which means its only predicted 58% of times which is not satisfactory. The model can be improved on the following classes ‘Pads’, ‘Resistor, and identifying ‘backgrounds.

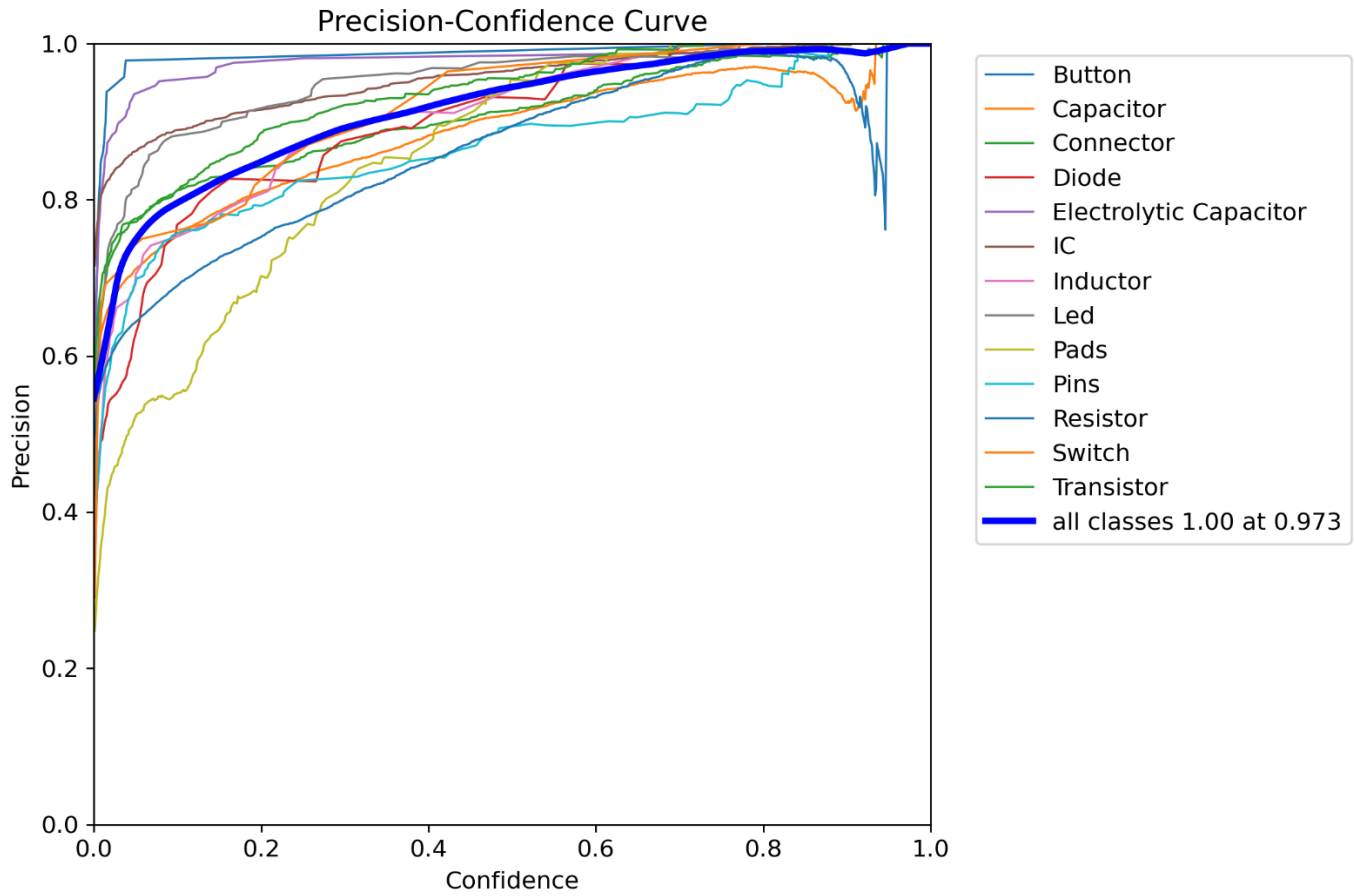


Figure 8: Precision Confidence Curve (Model 1)

The dark blue line indicates that classification of all classes reaches a maximum of 1.0 with 97.5% confidence which seems to be a pretty good result. If we look at different classes, for example classes such as ‘Pads’ have lower precision at lower confidence levels which suggest lower reliability which is something that needs to improve with Model 2.

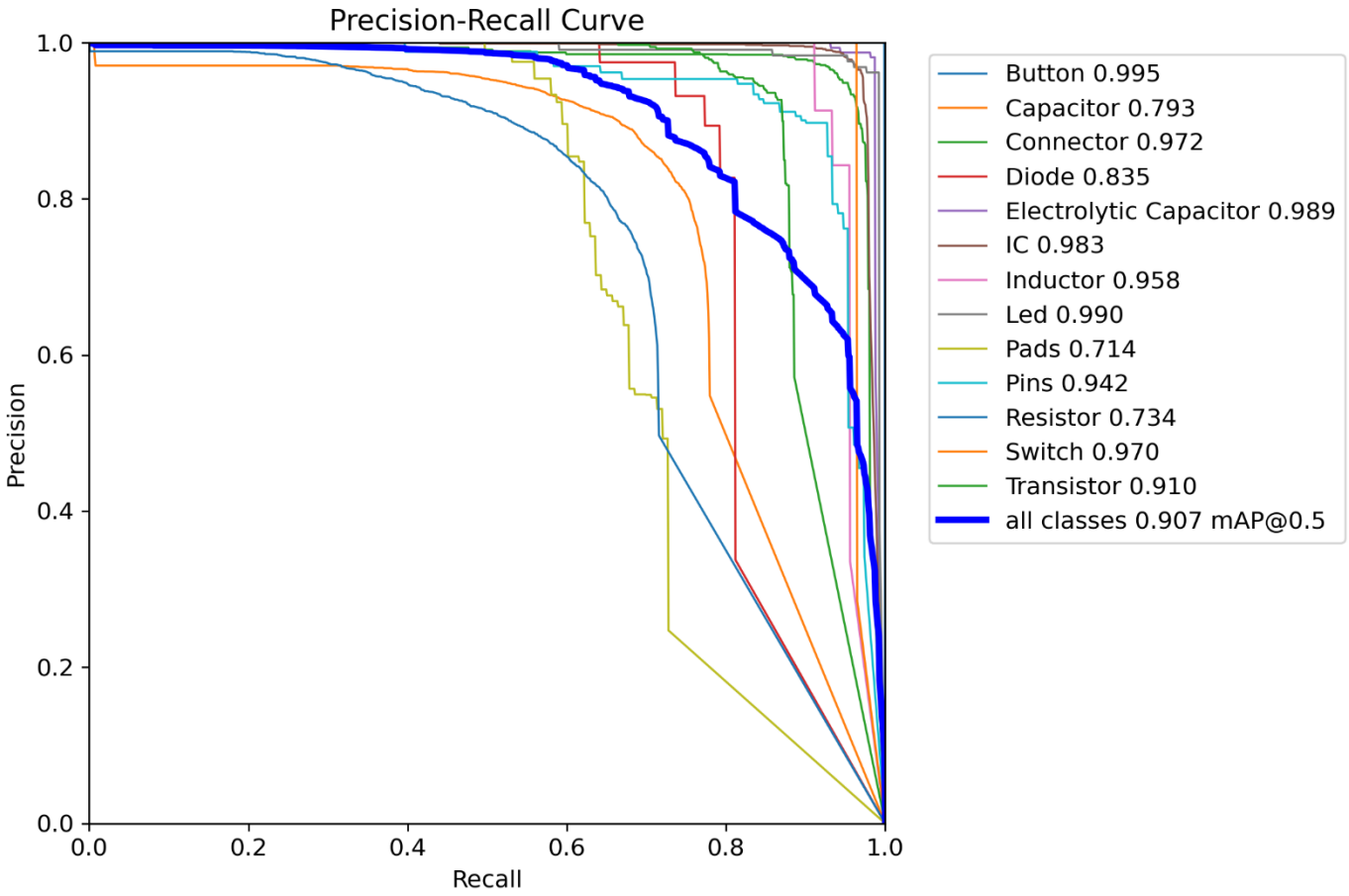


Figure 9: Precision-Recall Curve (Model 1)

Each curve represents a trade-off between precision and recall for a given class. In terms of overall performance, the dark blue line suggests the mean average of 0.907 at an IoU of 0.5 which suggests that this model is performing pretty good.

Evaluation – Model 1

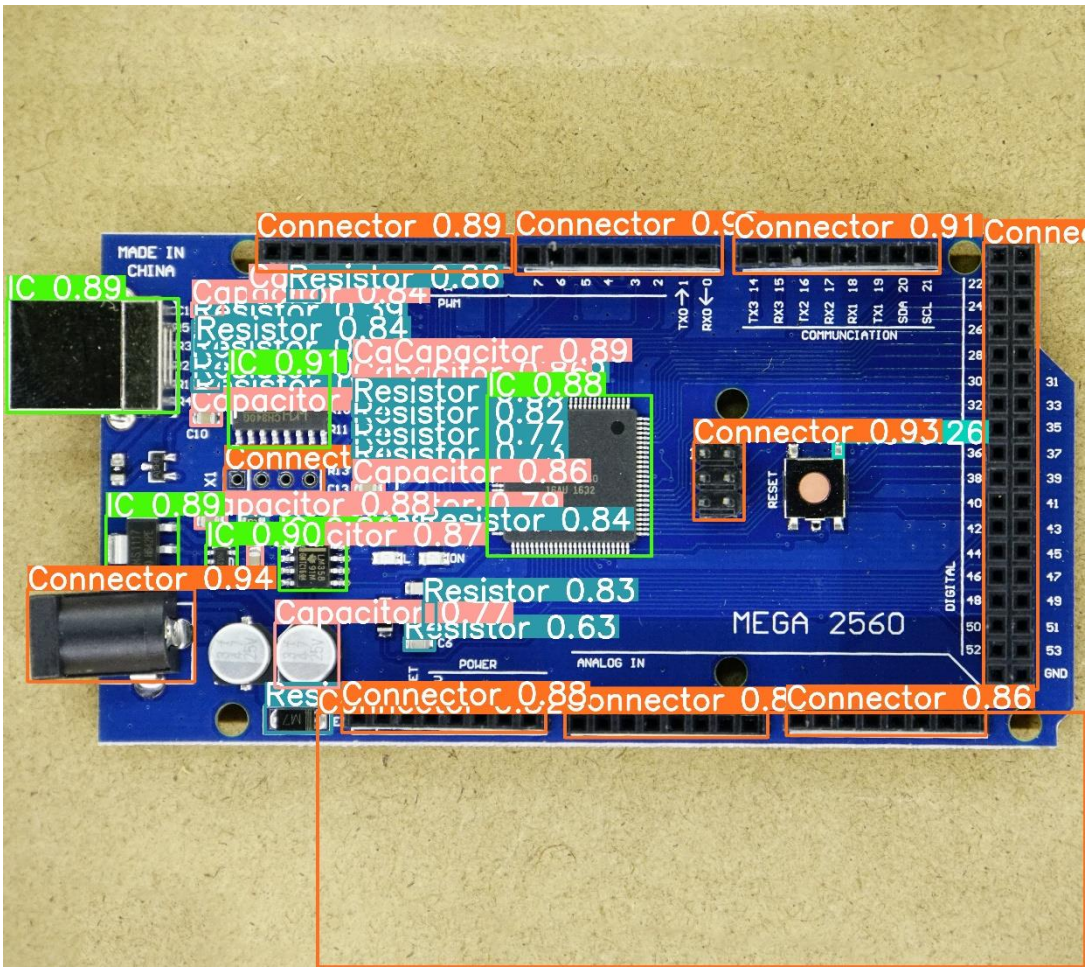


Figure 10: IMG: ardmega evaluation (Model 1)

Note: Missed a resistor and capacitor on mid-left of the board. Otherwise, the model performed pretty good.

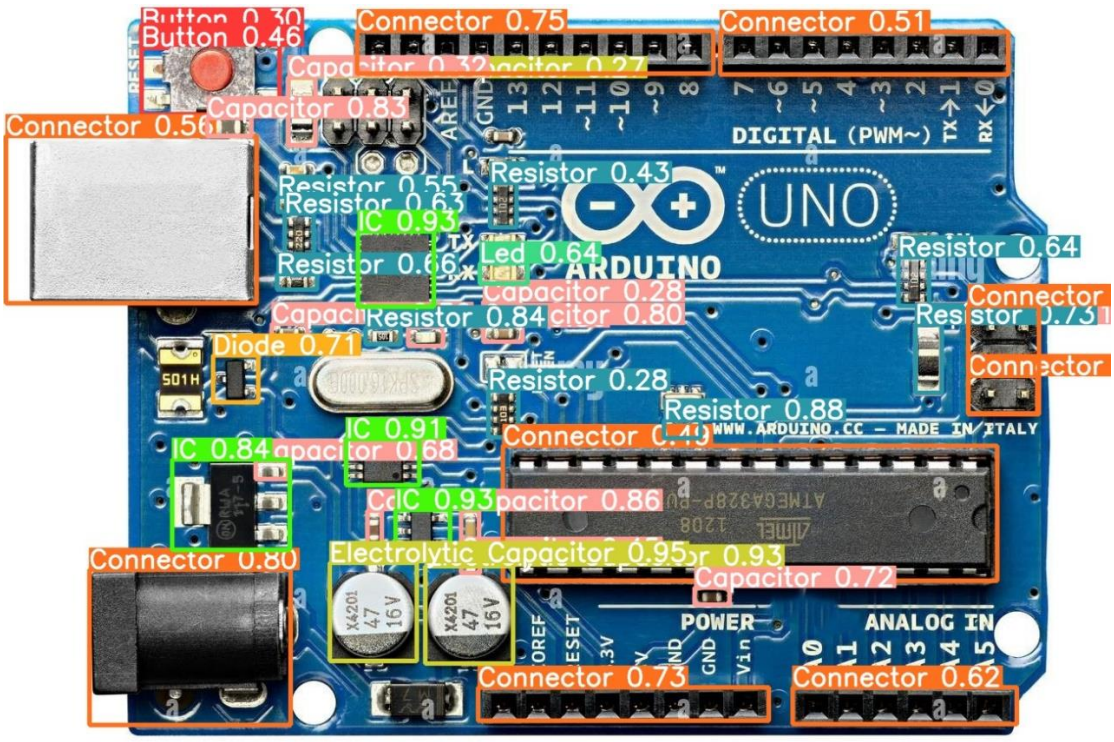


Figure 11: IMG: arduno evaluation (Model 1)

Note: Model seems to have identified everything correctly

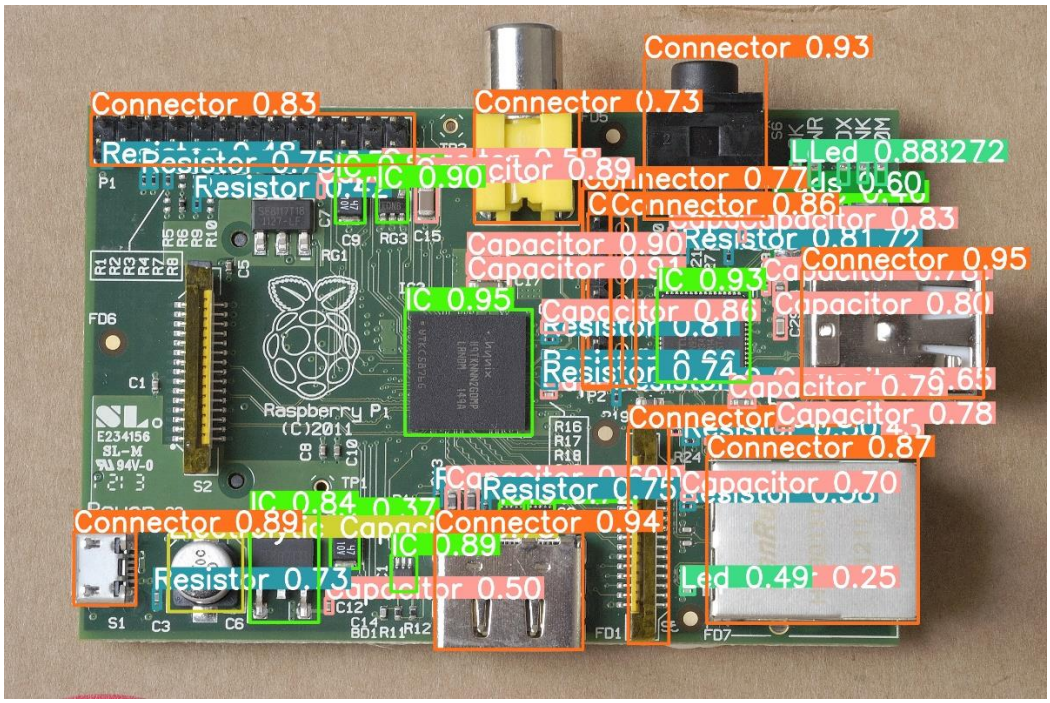


Figure 12: IMG: rasppi evaluation (Model 1)

Note: Missed a Connector, capacitor on the mid-left section. Rest of the classes were identified

## Model 2

### Architecture

```
yolo task=detect mode=train
model=runs/detect/train/weights/last.pt data=data.yaml
epochs=200 imgsz=1216 plots=True batch=4 save_period=10
resume=True
```

Epochs = 200
Imgsz= 1216
Batch=4

### Results – Model 2

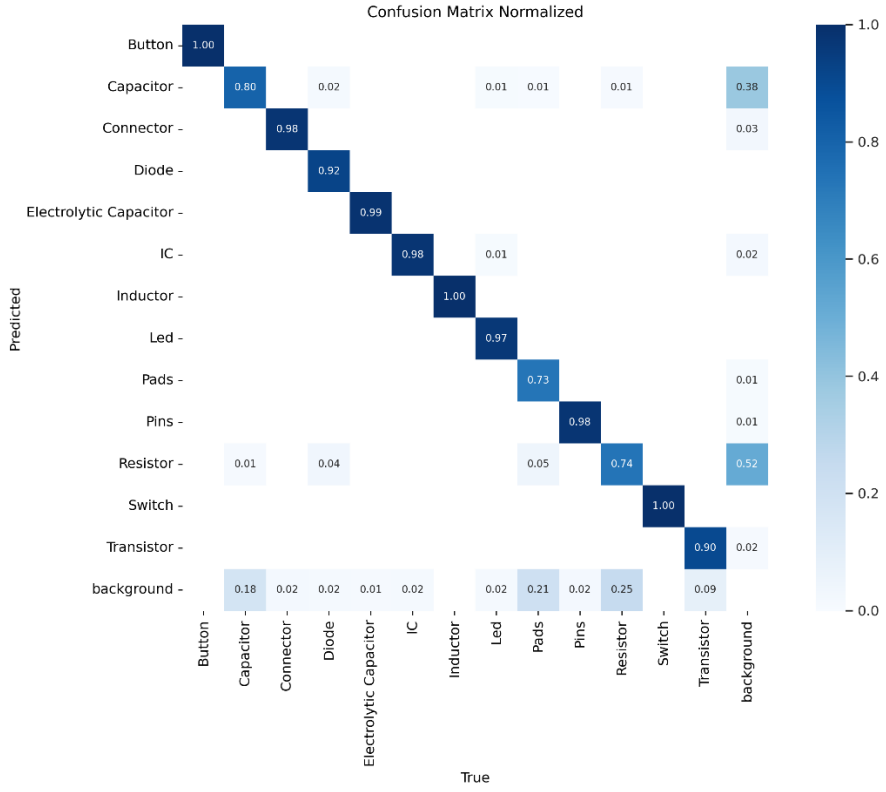


Figure 13. Normalized Confusion Matrix (Model 2)

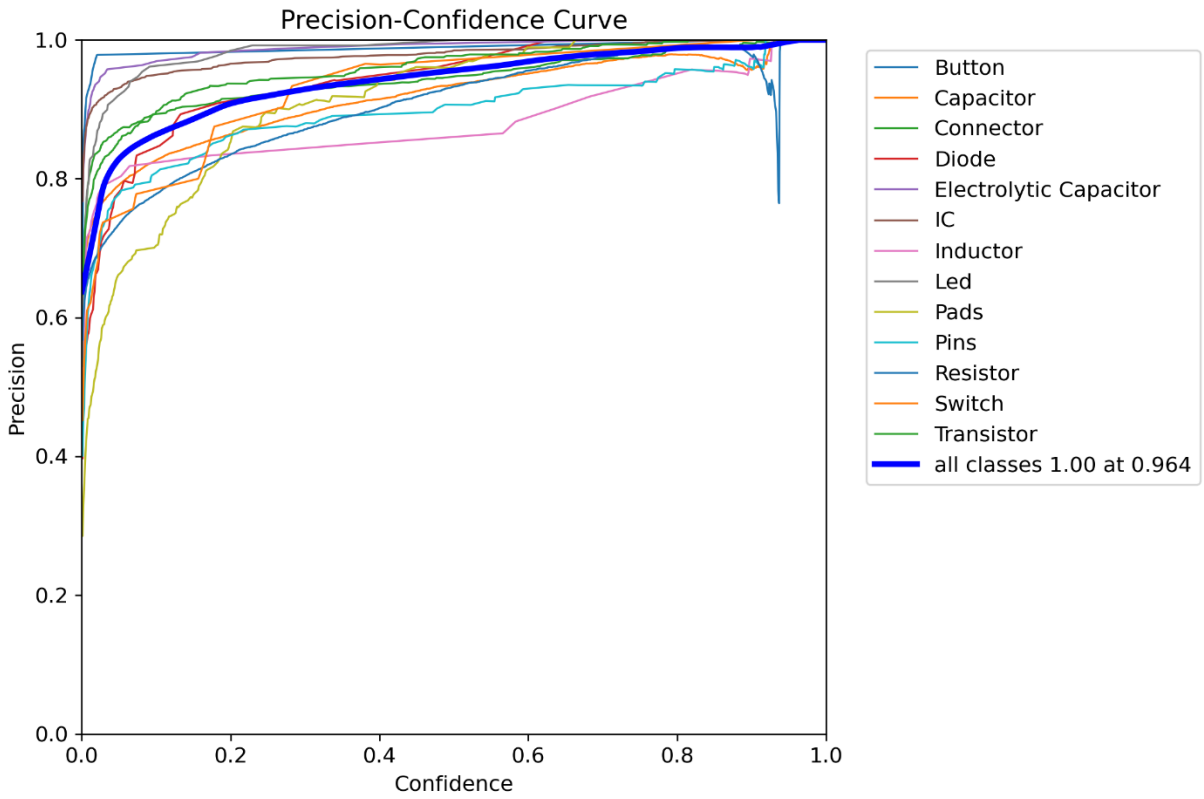


Figure 14: Precision Confidence Curve (Model 2)

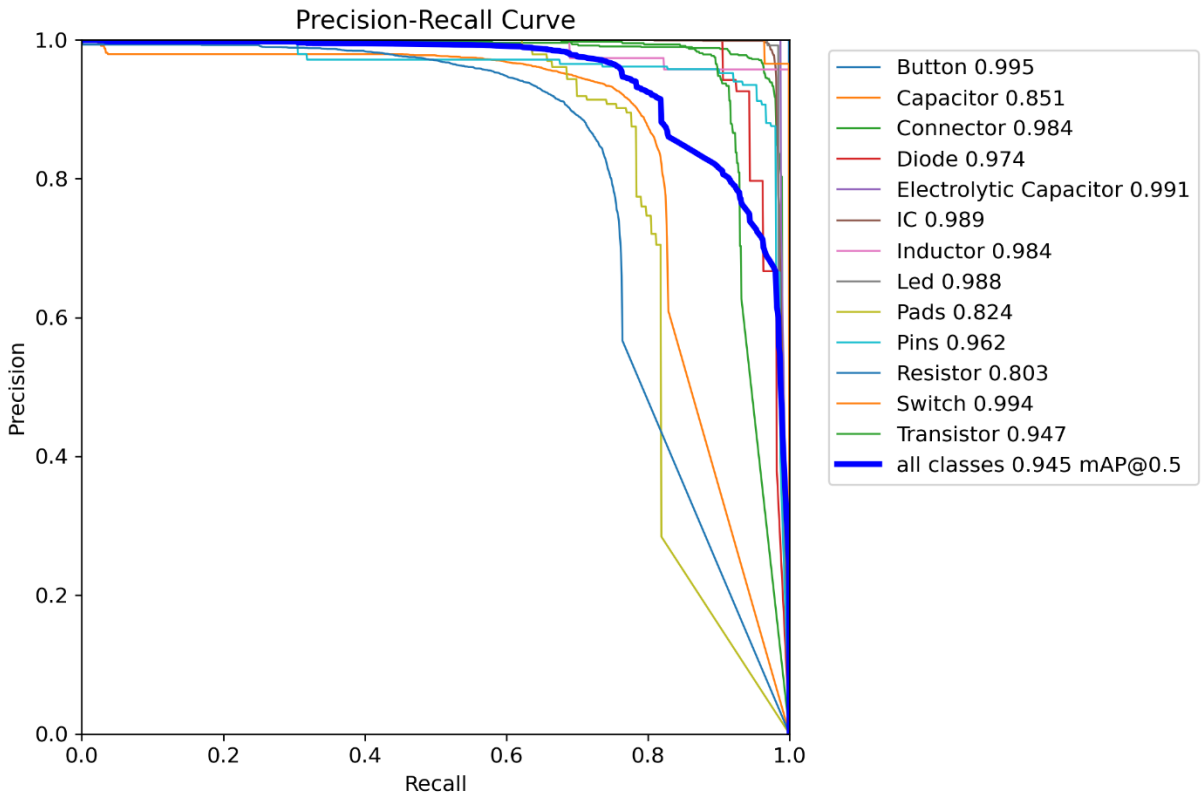


Figure 15: Precision-Recall Curve (Model 2)

Evaluation – Model 2

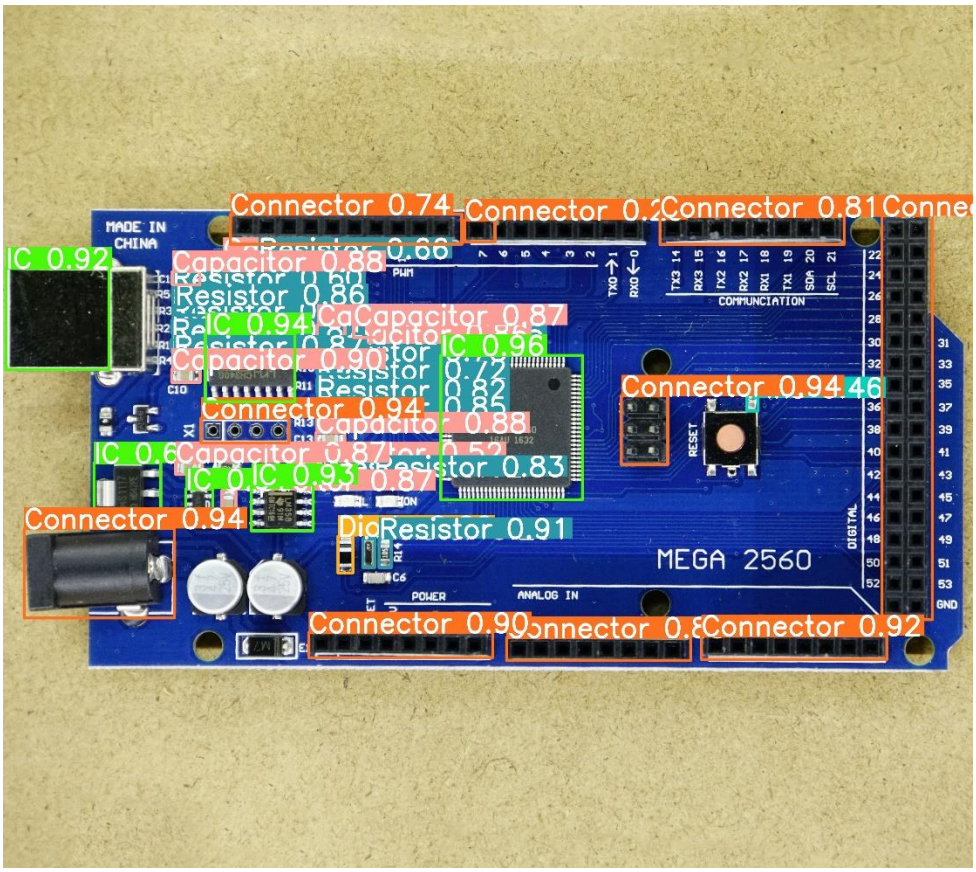


Figure 16: IMG: ardmega evaluation (Model 2)

Note: Some of the predicted inaccuracies include the same resistor and capacitor in the mid-left section, the rest of the items have been classified correctly and with a higher confidence level

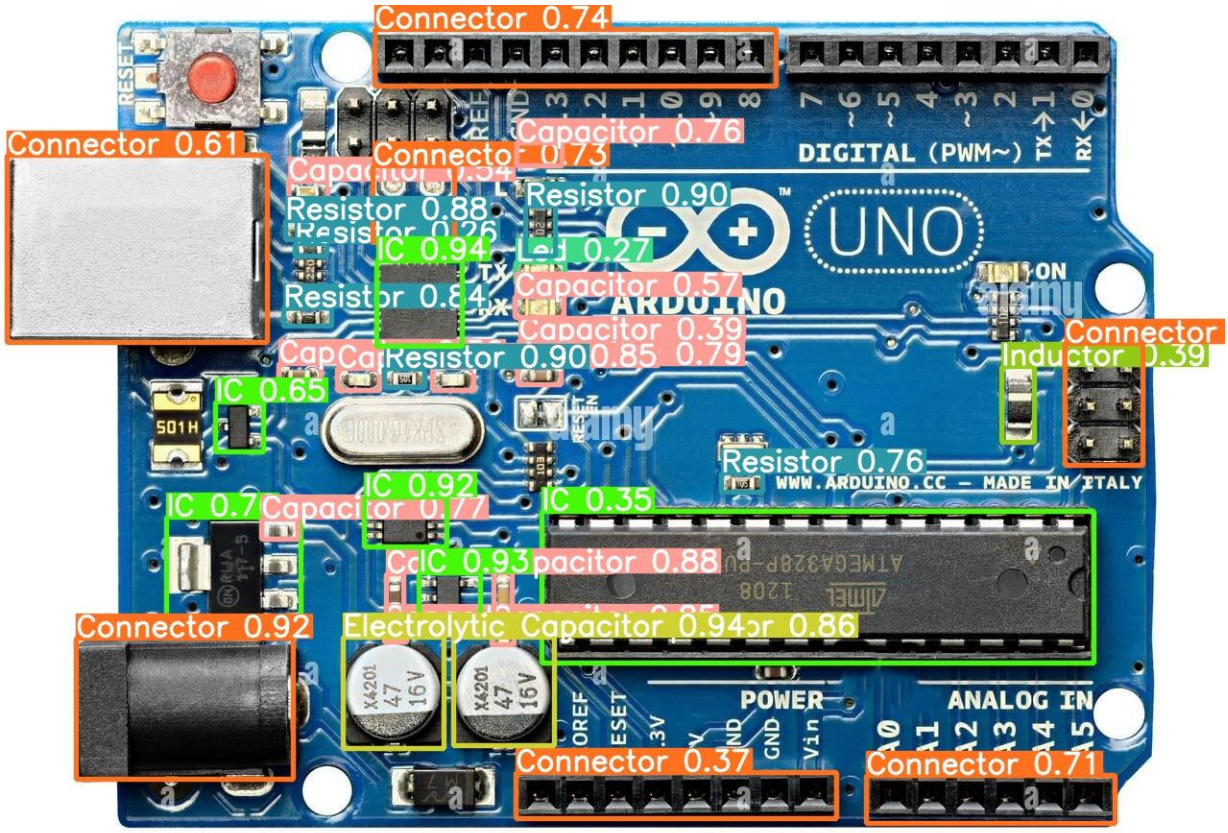


Figure 17: IMG: rasppi evaluation (Model 2)

Note: Model seemed to have missed the connector on the top right, bottom-mid of the board and the buttons on the top left. For this board the model seems to have performed worse than model 1. Although in the results provided on D2L, the button on the top left is misclassified as a capacitor so small errors are to be expected.

Figure #: IMG: arduno evaluation (Model 2)

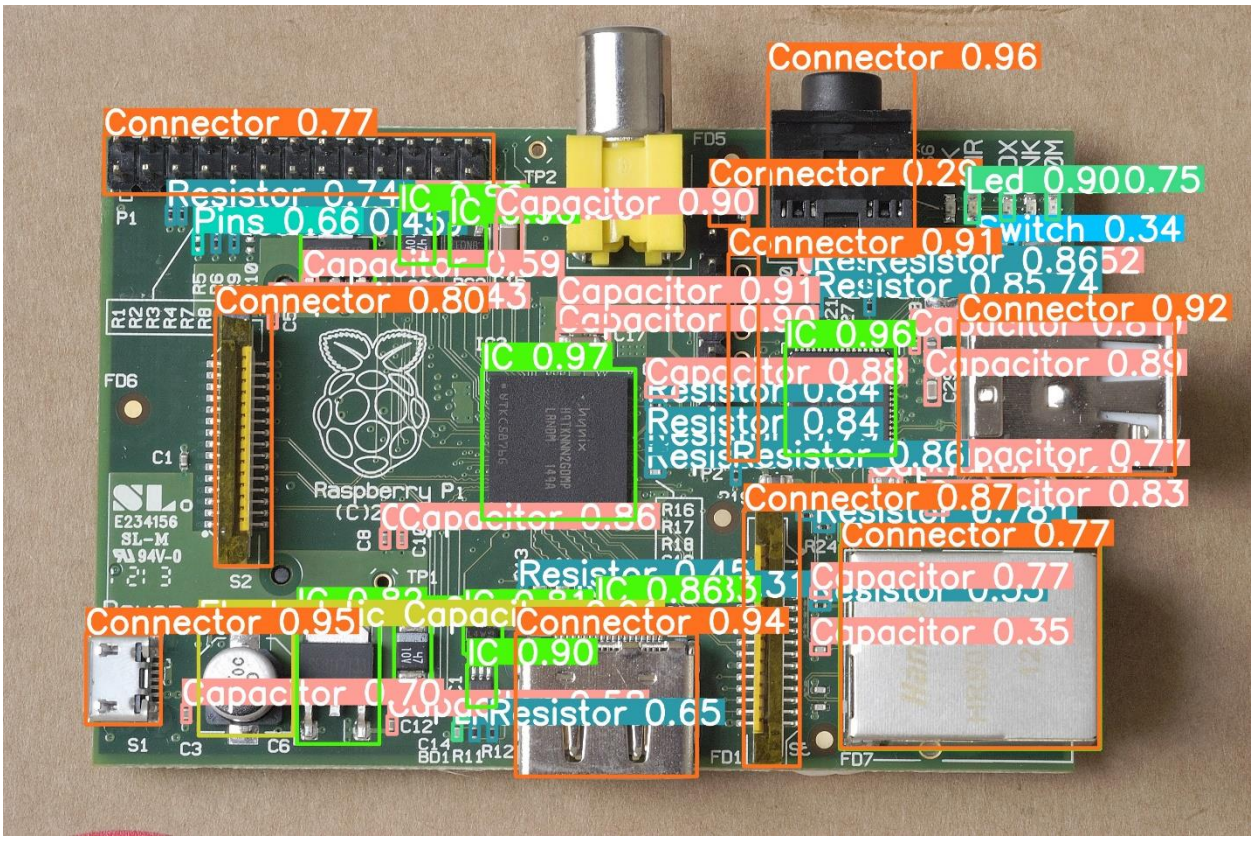


Figure 18: IMG: rasppi evaluation (Model 2)

Note: Model seems to have missed the connector on the top-mid section of the board, rest of the items have been identified correctly

## Conclusion – Yolov8 Performance

In terms of the model performance both models seemed to have performed exceptionally well, as apparent from the confusion matrix as well the P-C and P-R curves.
The model can be improved by increasing the batch size and the number of epochs and perhaps object masking since one of the biggest misclassifications was the ‘background’ class.

All the results can be found in the GitHub repository that was shared for authenticity.