Interim Project Report

Part Identification Number Extraction from Printed Circuit Boards

Nick Gardner

October 5, 2022

1 Introduction

The goal of this study is to create a robust system for identifying and analyzing part numbers present on *printed circuit boards* (PCBs). This work is inspired by an automation need for a remanufacturing company, CoreCentric. This company receives approximately 4000 PCBs per week, which are currently sorted manually by human operators. These human operators have a very high attrition rate, as the role is demanding and laborious. As a final deliverable for this study, it is desired that an end-to-end processing system is devised to image PCBs, localize labels, and analyze part numbers to inform sorting.

The main learning outcome is attaining a fluency in modern computer vision techniques for object localization / segmentation. I have a working understanding of current techniques, from computer vision courses at *Rochester Institute of Technology* (RIT) as well as my work at *Golisano Institute for Sustainability* (GIS), but the goal would be to gain a more comprehensive understanding especially of more modern solutions. Another important learning objective is domain knowledge in the area of remanufacturing and automation. I am acutely interested in automating labor-intensive tasks, as well as sustainability efforts such as remanufacturing. This study will give me an opportunity to explore more of these disciplines in a real-world setting. A final learning objective is the project process itself - I have worked on several projects at GIS but never have I been part of the initiation of a project as well as its conclusion. To learn the steps of creating an end-to-end system in the scope of a project with an industry partner is valuable for my professional development.

2 Efforts

2.1 Summary of Work

Week 1 (8/8 - 8/12): Initial literature review and experimentation. A small initial dataset was collected by imaging the boards already present from previous investigations related to CoreCentric. After small-scale tests were reasonably successful, it was decided to pursue YOLO as the first method of object detection.
Week 2 (8/15 - 8/19): To improve the size of the dataset, supplemental datasets were located online. These datasets quadrupled the total number of training images and quintupled the total number of labels. New systems were implemented for cleaning up supplemental datasets and labeling images with bounding boxes. Began labeling supplemental dataset.
Week 3 (8/22 - 8/26): Finished labeling supplemental dataset. Cleaned up issues with dataset creation systems. CoreCentric board shipment arrived, bringing with it questions due to the absence of labels on some boards.
Week 4 (8/29 - 9/2): First modeling efforts conducted. Very reasonable success achieved naively training for a small number of epochs on the newly enlarged dataset. Met with industry partner to clarify expectations and update with progress. Based on this meeting, decided to focus efforts on boards, with labels, and to return to boards, with part numbers printed directly on the board, later. Manually assessed localization

performance on random untrained captures from CoreCentric's new boards – 73/93 labels were located with 49/50 relevant (containing part number) labels located.
∗ **Week 5 (9/5 - 9/9):** Met with faculty sponsor to update on progress. Conducted literature review into OCR options and decided to focus on Tesseract. Began experimentation with Tesseract.
∗ **Week 6 (9/12 - 9/16):** Experimentation revealed that commercial solutions (such as Google Vision API) are able to complete both the localization and OCR tasks. Extremely low costs for service mean that this solution makes more sense for industry partner (than continued iteration and reinvention for a custom solution). Met with faculty sponsor and agreed that a pivot would make the most sense for doing work that would lead up to a thesis. Met with industry partner again to discuss use of commercial solution and internal part number database.
Week 7 (9/19 - 9/23): More limited week due to career fair and associated events. Met with industry partner again for clarification and to begin efforts on their end to ready a dataset for damage assessment (rather than part number identification).
∗ **Week 8 (9/26 - 9/30):** Finished efforts looking into Tesseract to understand technical limitations. Created end-to-end implementation of part identification system using Google Cloud Vision API and Trie data structure. Began working on interim project report.
∗ **Week 9 (10/3 - 10/7):** Finished draft of interim project report and met with faculty sponsor to review.

2.2 Dataset

The first dataset was collected in early August and consisted of 33 different boards of 4 different types. These boards were supplied to GIS by CoreCentric for previous projects and explorations, and were therefore readily available for this experiment. The camera, an industrial color camera manufactured by *The Imaging Source* (ISC), was set at maximum resolution and placed 14.5 inches from the surface of the board. The goal of this placement was to capture the entire board while maximizing the board resolution. To image each board, the board was rotated slightly in each direction, then flipped 180 degrees and rotated slightly again in each direction. This resulted in six captures for each individual board, and 198 total captures for the dataset.

The second dataset utilized online sources to bolster the size of the dataset. In machine learning tasks, the general adage is that more data will allow the model to better learn and generalize. This is not always true – bad data can teach the model to identify incorrect elements – but enough good data can allow even simple models to be very effective. In the spirit of this, two relevant datasets were located through internet searches. Neither was intended for training label location, and therefore the label bounding boxes had to be assigned later, but they consisted of well-imaged printed circuit boards with labels present. The first supplemental dataset, [2], was produced in the hopes of aiding computer vision applications for printed circuit boards especially in the recycling area. This was the largest contribution to improving the overall dataset, as this supplemental dataset contained 748 images of PCBs. Some of these images contained boards without labels, and were therefore excluded, but the majority were utilized. The second supplemental dataset was considerably smaller than the first, but still contributed to the overall dataset. This dataset, [3], was created in order to analyze individual components on boards. As such, it is a much higher resolution image (close-up) of each board. After boards without labels were excluded, the total number of images in the dataset was increased from 198 to 859. Importantly as well, the number of labels increased from 401 to 2039.

The third, and final, dataset incorporated a new shipment of boards from CoreCentric. These boards were curated by the industry partner to include a wide selection of boards and manufacturers with varying label locations and types. Due to project developments, only about half of this new shipment was imaged (in the same manner that the first dataset was imaged). The model was never trained on these boards – instead these boards were held out as a testing set to assess the performance of the model on unseen data. The additional benefit of this shipment data was that the relevant labels (those with part numbers on them) were marked by CoreCentric, allowing for differentiation between 'relevant' and 'non-relevant' labels. This dataset also opened questions about the approach. Several of the boards did not have labels (stickers) on the board, and instead had relevant part information printed directly on the board. Consultation with the industry partner revealed that the majority of boards are produced by a small number of manufacturers, which do place stickers on their boards. It was decided to continue to focus on this main application and return to numbers printed on the board as a second iteration for localization.

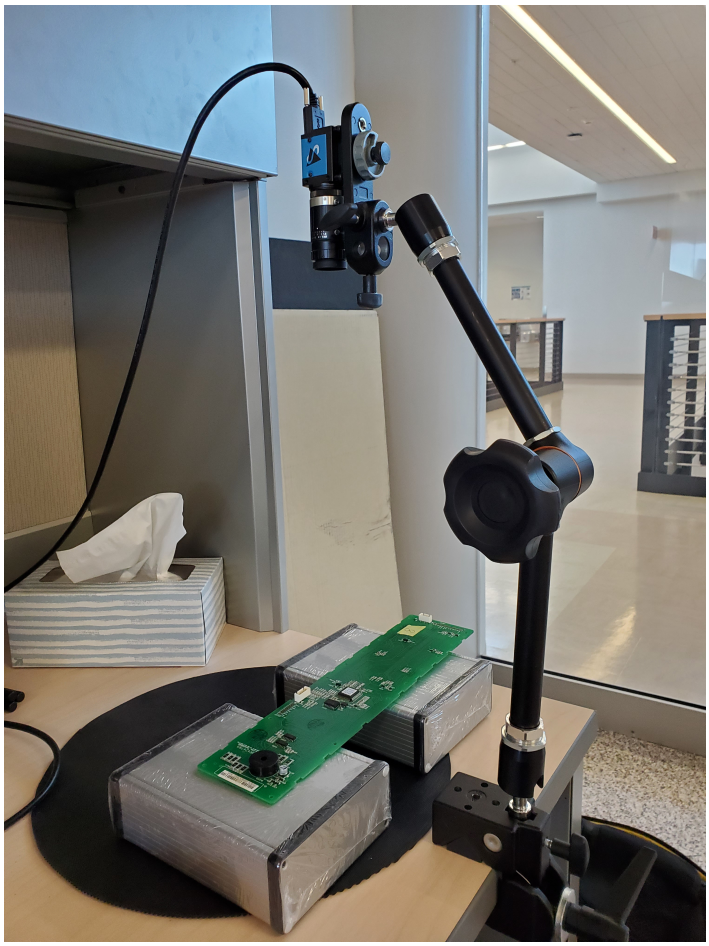

Figure 1: Station where imaging was conducted. As can be observed, lighting is not controlled as it was desired to simulate an imperfect imaging situation such as would be present on the factory floor. Background is controlled somewhat (with a black mat).

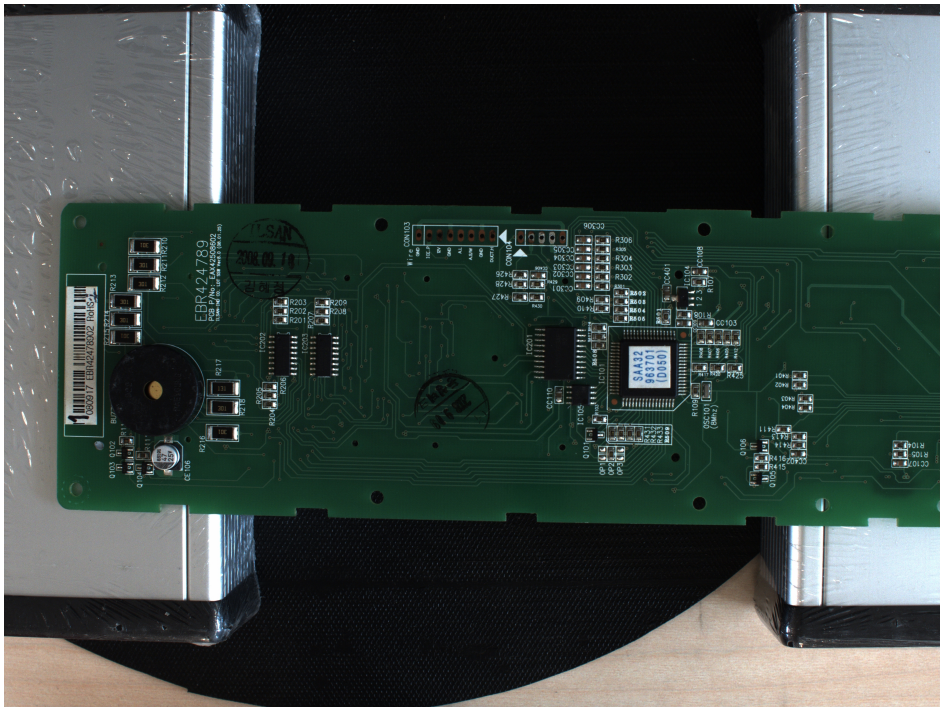

Figure 2: Board type '2' from a previous exploration into damage assessment of LEDs. The identification number that is sought is printed on the label on the left-most corner of the board. The number printed on the board contains most of the part number, but an additional identifier marking the configuration of the board is also necessary to sort this board.

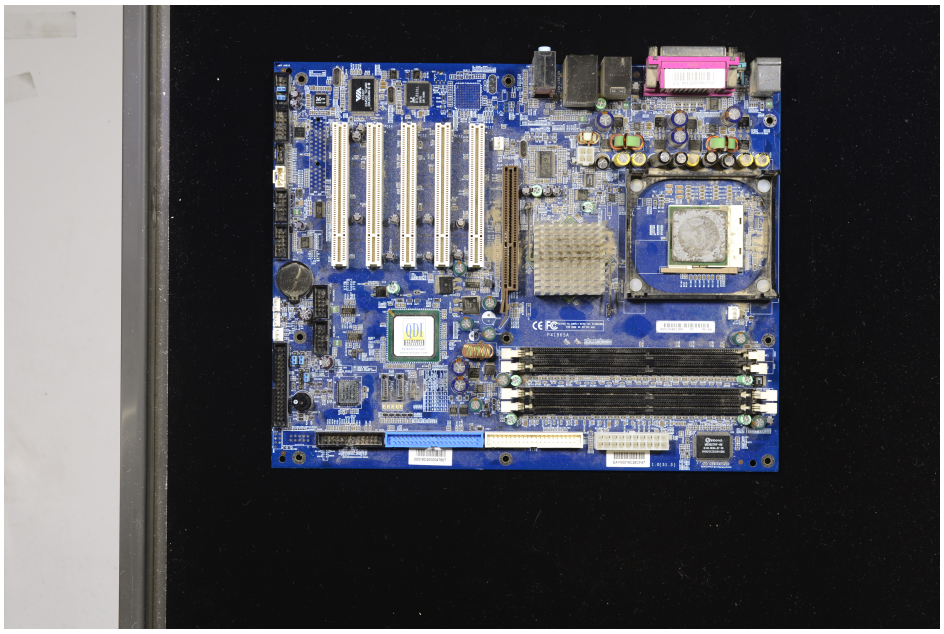

Figure 3: Example of board from supplemental dataset [2]. As this board comes from an outside source, the correct identification number is unknown. But the purpose of this supplemental dataset is to expand the size of the training dataset to learn to locate labels.

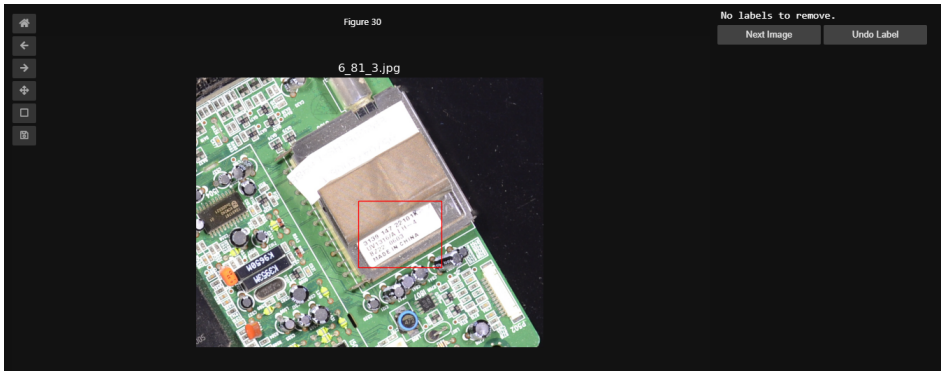

Figure 4: A demonstration of the custom tool created to label the dataset. This board image is zoomed on a particular label to better align the box. The boxes are then recorded for each image and later made into a YOLO dataset.

2.3 Object Detection / Localization

Initial literature review pointed to a few different options for object detection (localization of labels containing part numbers). The most classic approach would be to employ a multi-step convolutional neural network [1]. The first step is trained to identify potential bounding boxes, which is then followed by a feature extractor trained to find relevant features. Finally, these features are classified to known classes by a third model. More recently, models in the family of *You Only Look Once* (YOLO) objects detection and U-Net image segmentation have gained popularity. YOLO models [5] have somewhat lower accuracy in general than region-based CNNs, but they operate directly on the image in a single step and are therefore much quicker. U-Net [6] is a family of recurrent convolutional models, originally designed for biomedical image segmentation, that employ a U-shaped architecture that preserves information from previous steps to maximize spatial and feature information. As the goal of this study was, in addition to the learning objectives, to produce an end-to-end system that could be employed by CoreCentric, speed of processing was an important consideration. Additionally, similar recent projects at GIS had shown reasonable success for out-of-the-box YOLO, while U-Net experienced some drawbacks. For these reasons, the first options that was explored was YOLO.

After the first dataset was collected, bounding boxes were manually labeled using a custom tool. Using another custom tool, these bounding boxes and images were transformed into a YOLO dataset. With the data ready, model choice was the next step. YOLOv5 was chosen, as this is a stable version with easy distribution. YOLO's pretrained weights were used additionally, with the small model (rather than nano, large, or extra large) chosen. This was selected because it is a reasonable compromise between speed and training / evaluation time. The first experiment was to see if YOLOv5's out-of-the-box weights would detect labels. Images of the boards were run through the model to no effect. Sometimes YOLO would detect the board, but no labels. Next, YOLO was trained on the first dataset for a small number of epochs, starting with pre-trained weights. This was broadly successful, able to find validation labels and even some completely unseen label shapes and types. But the generalization was not nearly sufficient for the amount of anticipated variance in labels that CoreCentric sees on the factory floor.

Rather than attempting to finesse the model, which has diminishing returns and often leads to overfitting, the dataset was the next target for improvement due to the relatively small number of examples currently available. As discussed in the dataset subsection, supplemental datasets were found and integrated into a much larger and more comprehensive second dataset. Using the same experimental setup as for the previous dataset (additional training on pre-trained weights), YOLO was trained on the new dataset. This trained model showed large improvements compared to the model trained on the previous dataset. On unseen data, labels were consistently recognized. Other training methods were attempted including various forms of transfer learning (freezing just the backbone or all layers but the final prediction layer), as well as varying numbers of epochs, but the best performance came from naive additional training, with no transfer learning, for 75 epochs.

Because the new model seemed to perform quite reasonably on validation data, a more quantitative

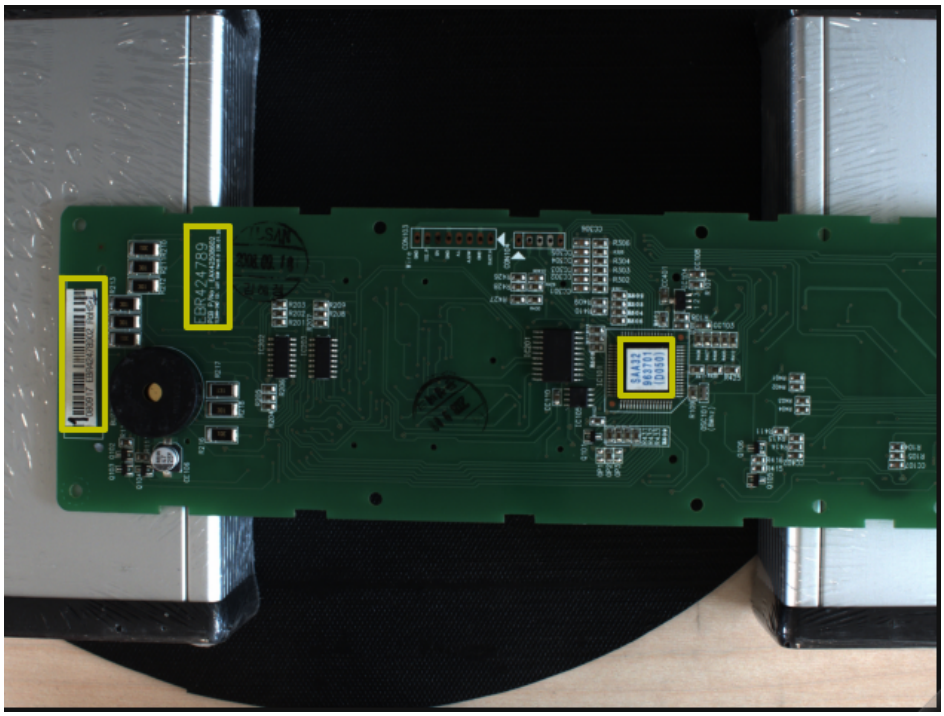

Figure 5: Validation image with bounding boxes shown. The model was trained on this board type but not on this particular board.

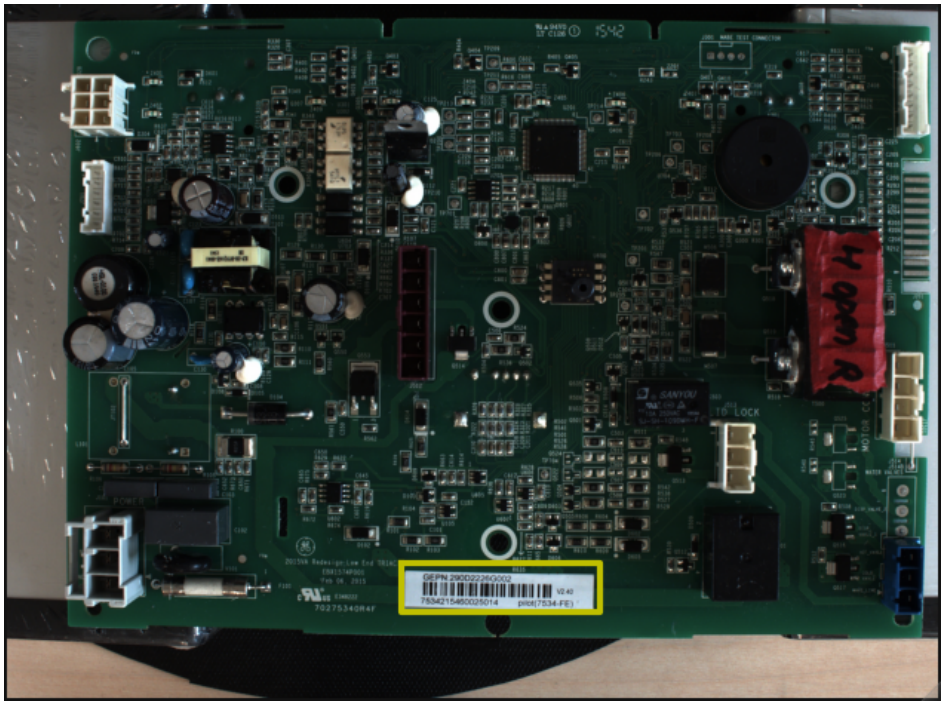

Figure 6: Validation image with bounding boxes shown. The model was trained on this board type but not on this particular board.

Figure 7: Validation image with bounding boxes shown. This example was contrived from random found items, intended to see if the model can generalize to entirely new examples. The only visible label is detected almost perfectly.

Figure 8: Validation image with bounding boxes shown. This example was contrived from random found items, intended to see if the model can generalize to entirely new examples. The two visible labels are almost entirely missed, with the model only able to find the QR codes on the label.

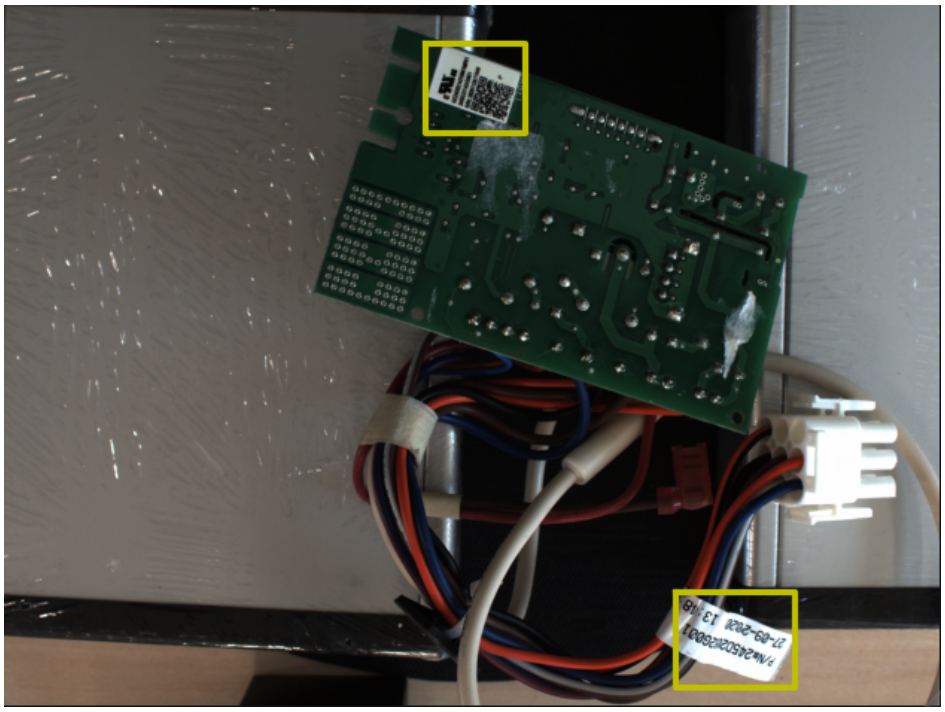

Figure 9: Unseen testing image with bounding boxes shown. Both labels are successfully located, including a skewed and wrinkled label connected to the board harness.

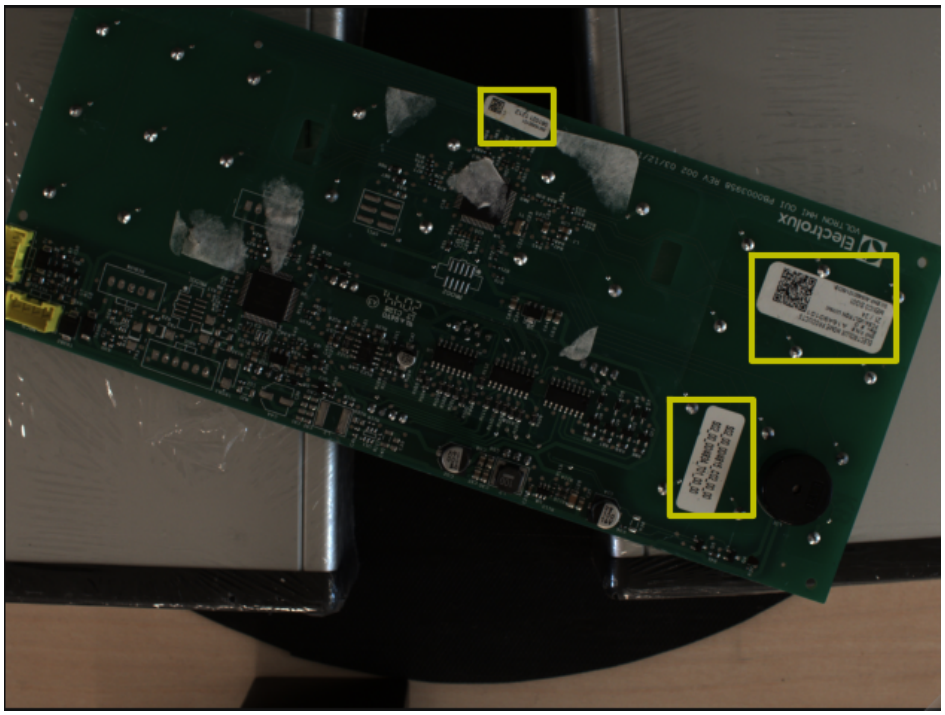

Figure 10: Unseen testing image with bounding boxes shown. All three labels are located with none missing and no false positives.

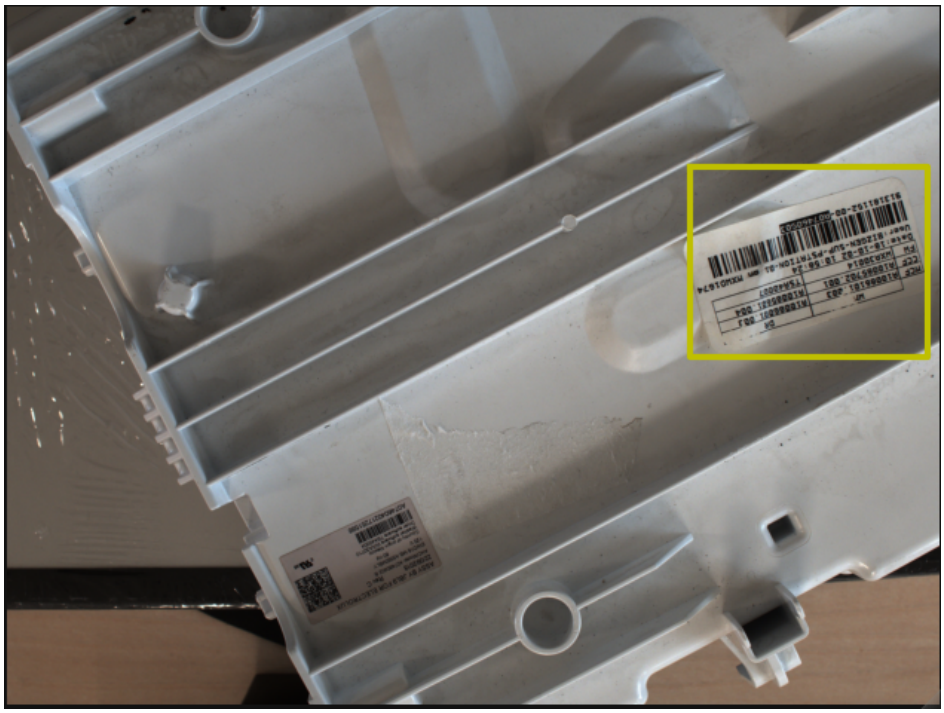

Figure 11: Unseen testing image with bounding boxes shown. Only one out of the possible two labels is located, but this is the 'relevant' label (unique identifier is present on this label).

metric was desired. A manual analysis was performed for model performance on images from the new board shipment. These board captures (and board types) were entirely unseen by the model and represent extrapolation / generalization. 50 captures from the new shipment were chosen at random, with all imaged unique boards present in the sample. The trained model was then evaluated on each image, and it was recorded how many labels the model was able to find successfully. Additionally, it was determined if the model was able to find the 'relevant' label, where relevancy relates to whether the label contains the part number that uniquely identifies the board. For a sample of 50 random board captures, the model found 73 out of 93 possible labels, and 49 out of 50 relevant labels. The total label count was reasonably successful, but the relevant label count was very impressive. While this was likely a bit of an over-performance in this category, it was still a sufficient showing to warrant moving on to the next aspect of the project. Rather than working on one part of the pipeline until it was flawless, it was decided to flesh out the basics of each step and return to improve each step as necessary.

2.4 Optical Character Recognition

After labels have been located, the text within each label needs to be analyzed to extract the appropriate part numbers. This can be created in a custom manner, but this is an extremely tough and well-studied problem that benefits greatly from massive amounts of data and training. It is also not a unique problem – while finding labels is not a common application, recognizing text is instrumental for document processing. With this in mind, the first goal was to find an open-source library that implements this. The literature review was rather brief, as the predominant option for open-source python character recognition is Tesseract. Originally produced by Hewlett-Packard in the 90's, it was re-released by Google as an open source library in 2006 [7]. Tesseract first extracts outlines with connected component analysis, then gathers those outlines into blobs. The blobs are then subdivided into text lines, and character spacing is used to break lines into words. Then, text is recognized in a two-pass operation. The first pass uses a standard classifier and returns any words it recognizes to a satisfactory level. These recognized words (and the text blobs that produced them) are then used to train an adaptive classifier. After training, the new classifier is applied to the image

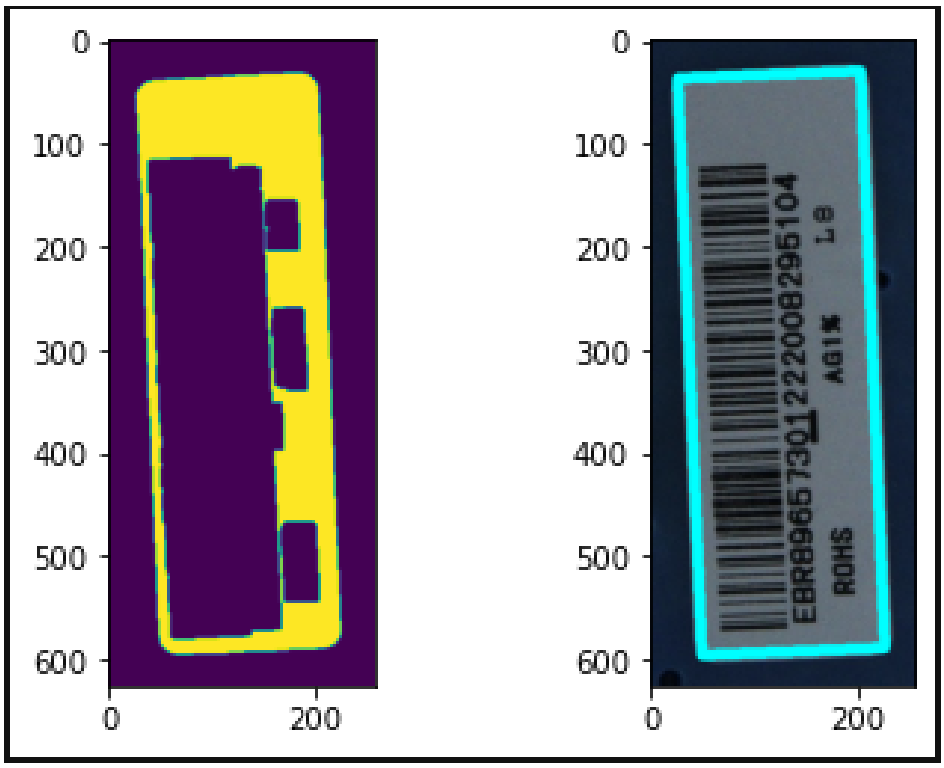

Figure 12: Demonstration of using erosion and countour detection to find a minimum area rectangle that encloses the label. This rotation information will be used to fix the label to horiztonal.

in a second pass to attempt to reveal any text not discovered in the first pass.

In order to experiment with Tesseract, several examples of cropped labels (from dataset three) were gathered. A basic online tutorial was followed, involving the pre-processing necessary to prepare an image for Tesseract. Additionally, Tesseract was instructed to use any model available (although the legacy models are not available to PyTesseract) and to find all sparse text (rather than searching for a single character, or a column of text, or any other preset for Tesseract). Some success was had with basic labels, but small inaccuracies plagued the process. While one or two characters incorrect is decent accuracy, part numbers must be entirely correct for a table lookup to be successful. More complex pre-processing steps were attempted but diminshing returns were reached, revealing the systemic problems with custom text recognition.

The first issue is that Tesseract is very sensitive to orientation. It will not recognize text that is not upright and horizontal. To solve this, erosion and contour detection was used to find a minimum area rectangle that encompasses the label. This rotated rectangle gives rotation information that was used to correct the label to a horizontal orientation. Tesseract has a mode, *Orientation and Script Detection* (OSD), that can be used to correct 90 degree rotations. It does not always work, especially for labels with too few characters, but trying a 180 flip as a pre-processing step and taking the better result could easily solve this in a simple manner. What is a much more challenging orientation issue is if the text on the label itself is slanted. This cannot be solved through orienting the entire label, and would require subdivision on text blocks.

The second major issue is that line thickness has a strong effect on Tesseract's performance. If characters are too thick or are partially touching, Tesseract will attempt to identify the character that best matches this new amalgamation. If the characters are too thin, elements of each character might be missed and the same result occurs. This is also not insurmountable – perhaps being solvable with adding an erosion and a dilation step into the preset chain – but it adds more required permutations for the pre-processing steps.

The final major issue is that Tesseract works best with specific character heights (dpi). Some images work better scaled down, and some work better scaled up. No official source on this was found, as Tesseract's official documentation is scant, but an informal study [4] was found that analyzed a large amount of text

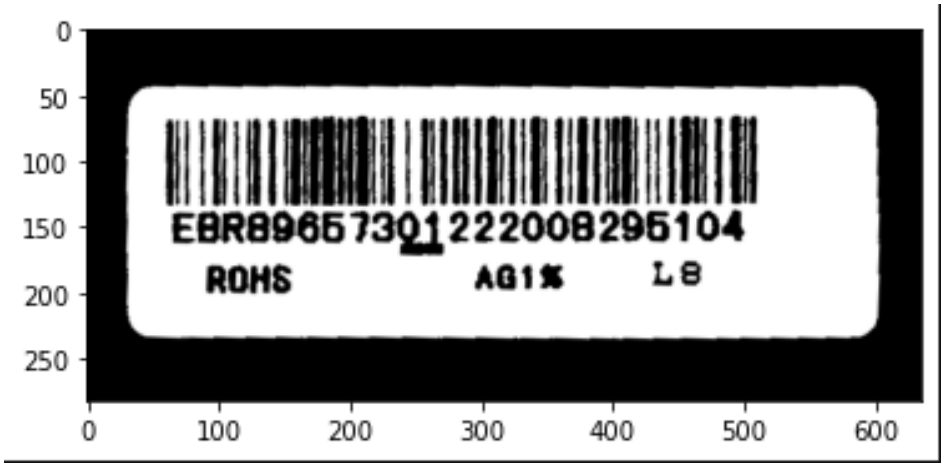

Figure 13: Label (from Figure 12) with corrected orientation. In this particular case, the label also needed to be rotated 180 degrees to reach this position. But the Tesseract OSD was able to detect that the label was upside down.

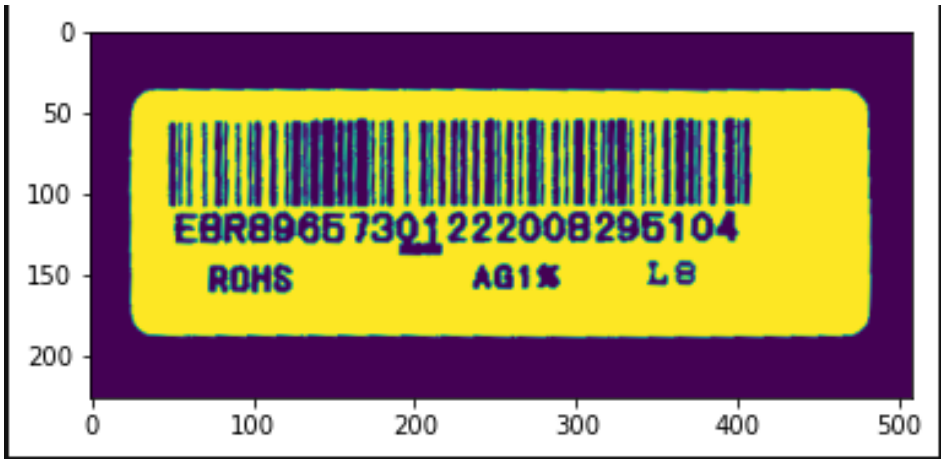

Figure 14: This label is a scaled version of the original label (importantly the original label thickness). Tesseract's output for this image was: 'LU\n\n167391 P22\n\x0c', which is not the 'EBR89657301' that we are looking for here.

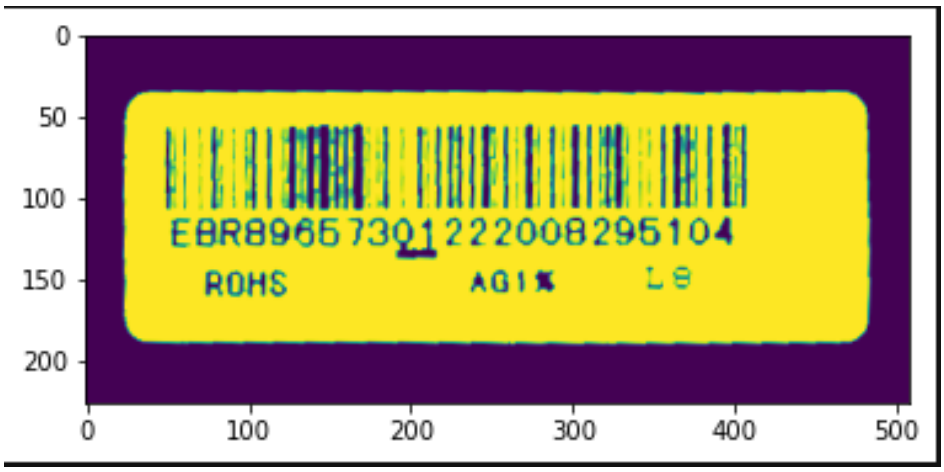

Figure 15: This label is a scaled and dilated version of the original label (reduced the thickness of the label characters). Tesseract's output for this image was: 'EBRB96573Q222008295104\n\nUUUTA\n\x0c', whose beginning is much closer to the 'EBR89657301' that we are looking for here.

and found an optimal character height of around 30 pixels. In my own exploration, I found a similar result. Once again, this is not an impossible issue on its own. But it would require more character-specific analysis or adding more pre-processing permutations.

If this work were to continue, there are a couple of reasonable paths. The simplest, but maybe least efficient, method would be to perform a grid search on different orientations, line thicknesses, and image scales. Any part numbers found for any preset would be presented as potential candidates. Another potential method would be to use a second level of YOLO to identify the position of each character, then run Tesseract in character recognition mode on each individual character. This would likely have some of the same issues that the current approach has, but recognizing a singular character is an easier and less sensitive task.

2.5 Commercial Solutions

After revealing these limitations with open source character recognition, other solutions were examined. As part of this exploration, it was discovered that commercial solutions exist that can perform successful character recognition on the board without need of localization of any kind. It is unfortunate that such options were not explored sooner, as it would have saved some time and effort attempting to reinvent the wheel. Additionally, it should not be surprising that this technology already exists. Document scanning and processing is part of almost every large enterprise, creating a massive demand for effective solutions. Massive companies like Google and Amazon have services that offer nearly flawless recognition for less than a dollar per 1000 scans. In the face of this reality, and knowing the limitations of a custom solution, it does not make sense to attempt to reinvent this technology, especially considering that the goal of this study is to begin work on what will eventually be a master's thesis. The learning objectives of this study have been accomplished – a comprehensive study and analysis of object detection and character recognition was performed and an end-to-end system was implemented. But in the face of new knowledge, a new system was implemented utilizing a third-party API.

Using the Google Vision API [8], essentially all of the text on any given board can be recognized. Orientation, lighting, wrinkle, skew, and any other issue does not appear to affect it. This is the advantage of a commercial solution – it was developed with the essentially infinite resources of a massive technology company and can be relied on to work immediately with no long cycles of development and iterative improvement (as the custom solution would). The API does take around 5 seconds to process the image, but this is a small price to pay for such an effective solution. Once the text is recognized, it must be parsed, sorted, and reduced to candidate solutions. This is done in two steps: first, known part naming rules – such as EBR often preceeds part numbers or P/N indicates a part number is likely to follow – are utilized to build a list of initial candidate solutions. If none of these pan out, likely strings are then compared directly against the

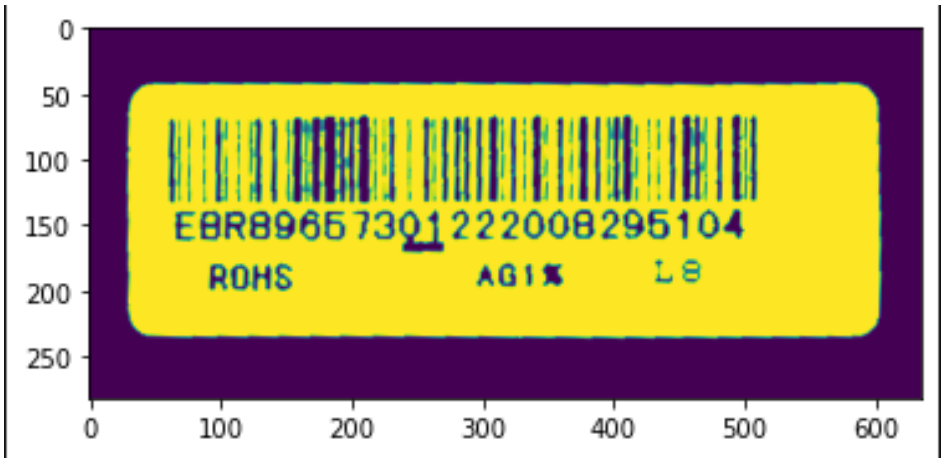

Figure 16: This label is the original scale of the image, (with some dilation to solve the other issue). Tesseract is not successful at reading this image, outputting 'I\n\nI\n\nNS\n\n8R896573Q4222008295104\n\nAGI\n\nL\n\nRO\n\x0c'. This failure is not as dramatic as some other examples, but the difference is clear when compared with Figure 17.

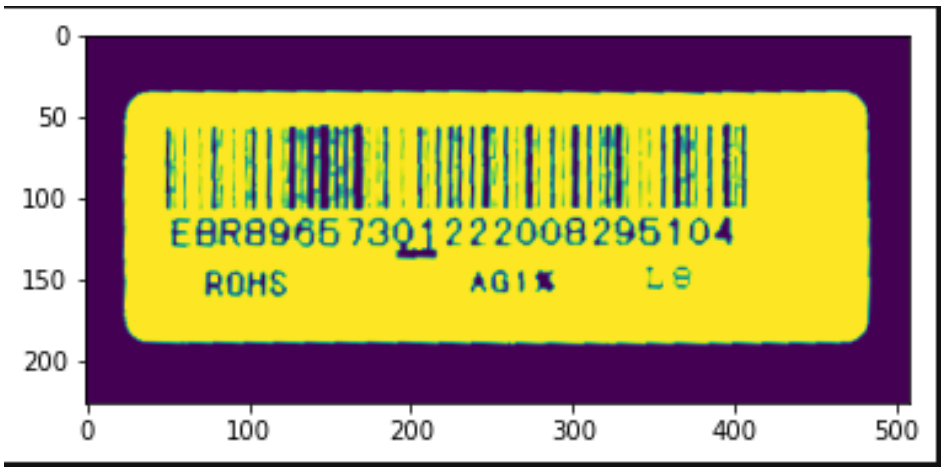

Figure 17: This label is scaled down by 0.8. This small change is enough to cause Tesseract to yield 'EBRB96573Q222008295104\n\nUUUTA\n\x0c', which is much closer to the desired value.

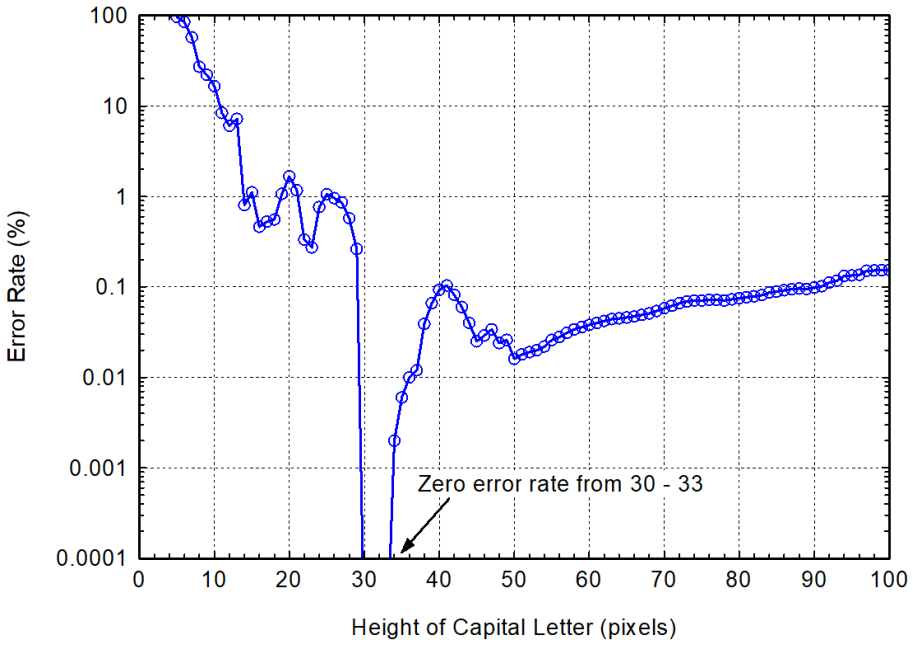

Figure 18: This chart is from the informal study [4] that was discovered when researching Tesseract character height problems. As Tesseract's official documentation is relatively non-existant, this represents a small examination of Tesseract's preference for character height. This graph's findings of around 30 pixels being optimal was reflected in the exploration of this study.

database until a solution is found or the decision is made that the board is not in the database. For a more robust text processing solution, a Trie data structure can be created that stores the known part numbers in a character tree. This structure can be queried for each potential string in a character-by-character fashion, massively reducing the number of required comparisons.

3 Next Steps

With part identification in hand, the next step will be to move on to more challenging tasks. These tasks are various kinds of damage detection and assessment for printed circuit boards, including: dim/dead LEDs, burn marks on boards, missing harnesses, cracks, and other damage modes that are deemed feasible and important to pursue. There is certainly not a commercial solution for this application, and could serve as a more interesting and novel direction to explore for potential master's thesis work. Additionally, some work has already been done for LEDs, and the first step in this new direction will likely be to explore YOLO's application for localizing these diodes. A keypoint homography system is currently utilized, but determining if YOLO can be used to generalize to new boards will be a worthy endeavour that further enhances the learning objectives of this study.

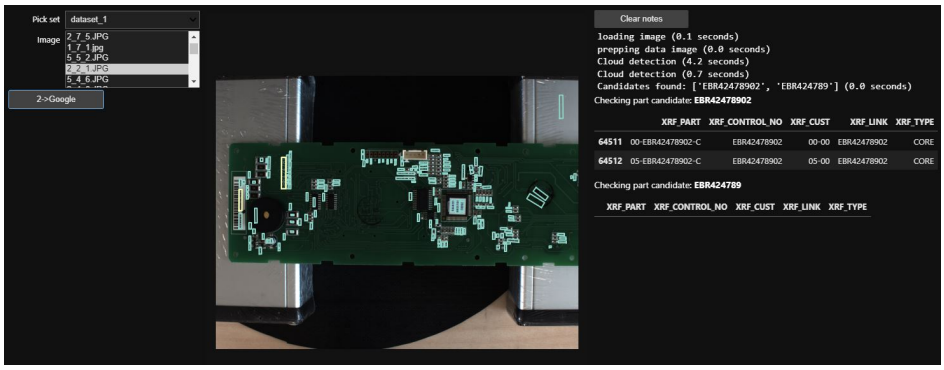

Figure 19: This image shows the custom tool designed to interface with the Google Cloud Vision API and query the industry partner's database. As can be observed from the image, the API finds essentially every piece of text on the board, and the only remaining task is to identify probably candidates and query the database. The number starting with 'EBR' is correctly identified and the query returns the board's entry in the database.

Figure 20: Another example of the custom tool. This board has considerably more text present on the board, but much of it can be excluded quickly as it is short and intended to label components on the board (for instance: 'R25' for 'Resistor 25'). In this example, the system is again successful at finding the appropriate identification number and finds the entry in the database.

References

[1] Ross Girshick et al. *Rich feature hierarchies for accurate object detection and semantic segmentation.* 2013. DOI: 10.48550/ARXIV.1311.2524. URL: https://arxiv.org/abs/1311.2524.

[2] Martin Kampel and Christopher Pramerdorfer. *PCB DSLR DATASET [Data set].* Tech. rep. 14th IAPR International Conference on Machine Vision Applications (MVA), Tokyo. Zenodo, 2020. DOI: https://doi.org/10.5281/zenodo.3886553.

[3] Chia-Wen Kuo et al. "Data-Efficient Graph Embedding Learning for PCB Component Detection". In: *2019 IEEE Winter Conference on Applications of Computer Vision (WACV)*. IEEE. 2019.

[4] *Optimal image resolution (DPI/PPI) for Tesseract 4.0.0 and eng.traineddata?* URL: https://groups.google.com/g/tesseract-ocr/c/Wdh_JJwnw94/m/xk2ErJnFBQAJ.

[5] Joseph Redmon et al. *You Only Look Once: Unified, Real-Time Object Detection.* 2015. DOI: 10.48550/ARXIV.1506.02640. URL: https://arxiv.org/abs/1506.02640.

[6] Olaf Ronneberger, Philipp Fischer, and Thomas Brox. *U-Net: Convolutional Networks for Biomedical Image Segmentation.* 2015. DOI: 10.48550/ARXIV.1505.04597. URL: https://arxiv.org/abs/1505.04597.

[7] R. Smith. "An Overview of the Tesseract OCR Engine". In: *Ninth International Conference on Document Analysis and Recognition (ICDAR 2007)*. Vol. 2. 2007, pp. 629–633. DOI: 10.1109/ICDAR.2007.4376991.

[8] *Vision AI.* URL: https://cloud.google.com/vision.