12-2021

Automated Discovery and Interpretation of ADA-Compliant Door Placards

Dale E. Parson

John J. Feilmeier

AUTOMATED DISCOVERY AND INTERPRETATION OF ADA-COMPLIANT DOOR PLACARDS

A Thesis

Presented to the Faculty of

the Department of Computer Science

Kutztown University of Pennsylvania

Kutztown, Pennsylvania

In Partial Fulfillment

of the Requirements for the Degree

Master of Science

By

John Joseph Feilmeier

December, 2021

Abstract

A familiar difficulty to any new student on campus is making one's way from classroom A to classroom B. Facilities with different wings, multiple floors, and irregular floorplans can magnify this challenge, while students with vision impairments are impacted even more by the challenge of identifying the destination.

This thesis explored different methods of discovering Americans with Disabilities Act (ADA)-compliant room identifying placards ("plaques") and identifying the text on the sign. The plaque detection was accomplished with both standard image manipulation techniques and a Histogram of Oriented Gradients (HOG) (Dalal & Triggs, 2005) object detector. The text reading utilized both standard image manipulation tools as well as an implementation of the Efficient and Accurate Scene Text detector (EAST) (Zhou et al., 2017) to isolate text, while Tesseract (Smith, 2007) was used to interpret the text. Different methods of dataset generation were utilized to train the detectors, including manual gathering, internet search scraping, and dataset generation.

Results of testing these different methods on a dataset of image frames gathered from filming the Computer Science/Information Technology (CSIT) hallway of Kutztown University's Old Main building proved the combination of HOG and EAST to be an effective method for identifying and transcribing room identification plaques. In the case of consistent visual design of rooms signs, the generated dataset proved to be nearly as effective as training the detector on real annotated images.

Acknowledgements

If not for the support of my wife Meghan, this thesis would likely have been abandoned.

Great thanks for the patience, assistance, and advice of my thesis advisor Dr. Dylan Schwesinger, who helped me define the scope of the project.

Contents

List of Figures

List of Tables

1 Project Overview

The problem of identifying and transcribing a regulated-but-variable identification agent is very similar to Automated License Plate Recognition, or ALPR. Du et al. (2012) described some of the issues facing ALPR as

> ... plates usually contain different colors, are written in different languages, and use different fonts; some plates may have a single color background and others have background images. The license plates can be partially occluded by dirt, lighting, and towing accessories on the car.

These issues echo some of the challenges involved in identifying rooms signs; while they must meet certain height and contrast requirements (U.S. DOJ, 2010, ss 703.4) they may generally vary in style and shape, with different color ways, in variable lighting, and with the possibility of wall-mounted communication elements interfering with the plaque.

2 Methodology

2.1 Attitude of approach

The principal driving thought behind this project holds that computer science and scientific thought can be used to solve real-world problems for real-world people. The open-source philosophy has yielded an incredibly rich variety of software and documentation. Exploring the idea that open-source projects and tutorials can be leveraged by the motivated computer scientist to make a real-world impact, this project explores a framework built around these freely-available code bases, updating the code where necessary for development and comprehension.

2.2 Image libraries

The main library used for this project is OpenCV (Bradski & Kaehler, 2000), which is an open-source library used for image and video manipulation. There are many tutorials and guides for the use of this library in Python, while the official documentation (at the time of development) is geared mostly to the Java and C++ implementation. The creation and use of the images is straightforward, based around arrays of pixel information. In order to "simplify" the use of this library for image creation, I created a loose wrapper around the class, removing the head space required to save, open, show, copy, etc., the images. I later expanded this abstraction for activities like drawing lines and rectangles, and simplifying thresholding, blurring, and color management. The idea behind this activity was to allow for easier and faster development, as well as an exercise in pythonic object-oriented programming. I found that after returning to this project, I had to both recall how to use my abstraction, as well as the code it was abstracting; I discarded it in favor of the normal OpenCV methods for the later development in image identification and classification.

SKImage (Van der Walt et al., 2014) is another Image processing library which is quite powerful, but here mostly used for normalizing grayscale images.

3 Data Collection and Synthesis

3.1 Problem Definition

In order to construct a performant machine learning pipeline, one (with few exceptions) must be able to feed in some quality training and testing data. In the domain of computer vision and object detection, visual data can be retrieved from a variety of sources, such as internet searches, generating the images via photography, generating the images programmatically, or some combination of the three. What follows is a journey into these different methods, necessitated by restraints of time and resources.

3.2 Literature Review

"Machine learning algorithms learn from data. It is critical that you feed them the right data for the problem you want to solve." (Brownlee, 2013). Mitsa (2019) points out that the performance of traditional machine learning algorithms will plateau after a certain amount of data is reached, but deep learning methods, which use multiple layers of processing to extract a finer-grained feature set, continue to see performance increases as the dataset grows.

Generally, the quality factors of a data set are based on its size, its level of normalization, and the presence and accuracy of labels. The needs of different machine learning methods vary, and the requirements that fall upon the data set change accordingly. In evaluating the neural net approach for plaque recognition, the ideal dataset was a large dataset (over 1,000 images in both training and testing) that was labeled ('has a plaque' or 'does not have a plaque') and normalized (all images similar colorspace and same size). It should also show the plaques from a range of different angles and scales to provide a more complete demonstration of the visual impact of the Americans with Disabilities Act (ADA) plaque. In this case, no existing collection of high-quality, labeled data existed for the specific domain of ADA compliant plaques mounted in a real environment.

There are several methods of acquiring a data set for training and evaluating a machine learning model. The most straightforward method is to find an existing data set. Sites like Google OpenImages (Kuznetsova et al., 2020) and Kaggle host various datasets for machine learning challenges and well-known, very large datasets like NOAA (“NOAA (National Oceanic and Atmospheric Administration Data”, n.d.) weather data and Famous People Faces (“Famous People Faces Dataset”, n.d.), a collection of images of human faces. The practical benefit here is that data is often already normalized and labeled, requiring minimal processing to train performant models. In some cases the dataset may not be large enough to properly train a model, which is where data augmentation can be useful. In the case of an image-based dataset, the researcher may employ several image manipulation techniques to ‘stretch’ a somewhat limited dataset. Images can be rotated, cropped (like zooming in), sheared and stretched, flipped (vertically or horizontally), color spaces changed, and noise added to generate a stronger feature map in the resulting model. These traditional “affine” transformations are “... fast, reproducible, and reliable.” (Mikołajczyk & Grochowski, 2018) This can grow a dataset to behave as a much richer dataset, as well as saving the extra labeling effort.

Where a dataset is not already available, it must be created. The simplest way to do this for a dataset of images is to use a film or still camera to collect images manually. This gives the researcher a higher level of control over the contents of the dataset, but also may lead to bias, as “... predictions are only as reliable as the human collecting and analyzing the data ...” (Mendis, 2019). This method is simple, but also time consuming and impractical depending on the subject matter. Another source for images is the popular search engine, Google Image Search (“Google Image Search”, n.d.). Google image search returns images based on specified search criteria, and can be used to generate a dataset of images. Scraping these image sets from the internet provides a raw data set that must be filtered, normalized, and labeled.

After working with both real images and googled images, I found that the imagery itself was fairly detail-sparse: essentially, looking for a high-contrast simple signage (a solid shape) next to a door (a bigger solid shape). This makes sense as the ADA signs are meant to be easily readable by those with limited vision (“U.S. Department of Justice, 2010 ADA Standards for Accessible Design”, 2010). After working with images gathered photographically and images gathered from the internet, and utilizing data augmentation techniques, I generated a dataset using some of the same techniques employed in data augmentation.

3.3 Method

3.3.1 Datasets gathered from the internet

Google Image search was used to trawl for images associated with things one could expect to find in an institutional hallway. I used search keywords such as “bulletin”, “directory”, and variations on “ADA plaque” and “classroom number”, as well as “hallways”; a sample of results is provided in Figure 1. Much of the code employed here comes from sample code provided in Adrian Rosebrock’s blog post (How to create a deep learning dataset (Rosebrook, 2017)) about image scraping and dataset retrieval from Google Images. A list of the URLs from each of these image searches was downloaded and written to a file using a bit of JavaScript in the browser’s developer console. The download_images_from_urls.py script, which uses the python requests module to hit the url and save the resulting response value, consumed these files. Separately, the program attempts to open the result with the OpenCV module. Those files that are empty, or fail to open, are deleted. I manually picked through the resulting multitude of images to remove any egregious or ridiculous inclusions, before using the clean_and_normalize_images.py script to normalize the images by adding a reflective border to “square-off” the image and resize it. The requirement for all images in the dataset to be of the same size has to do with the fully-connected convolutional neural net which was being used to classify these images. These networks have one or more hidden layers which abstracts the input and changes its size, and in the implementation used the size must be consistent.

The quality of these images, as well as the actual content, varied greatly. The bulk of the “positive” plaque labeled results were promotional materials from wholesale companies that supply custom ADA-compliant signage to institutions. These close-up images are descriptive of the variety of shapes, fonts, and colors which are generally used for such signage, but don’t do a great job of showing the context wherein these signs might be found.

The results for “negative” labeled data were incredibly diverse. Bulletins seemed to be fairly evenly split between real images of elementary and high-school decorated bulletin boards, and promotional material for cork board sales companies. The results for "directories" included many images of hallways, directory listings, and institutional building interiors, but also a collection of seemingly random images.

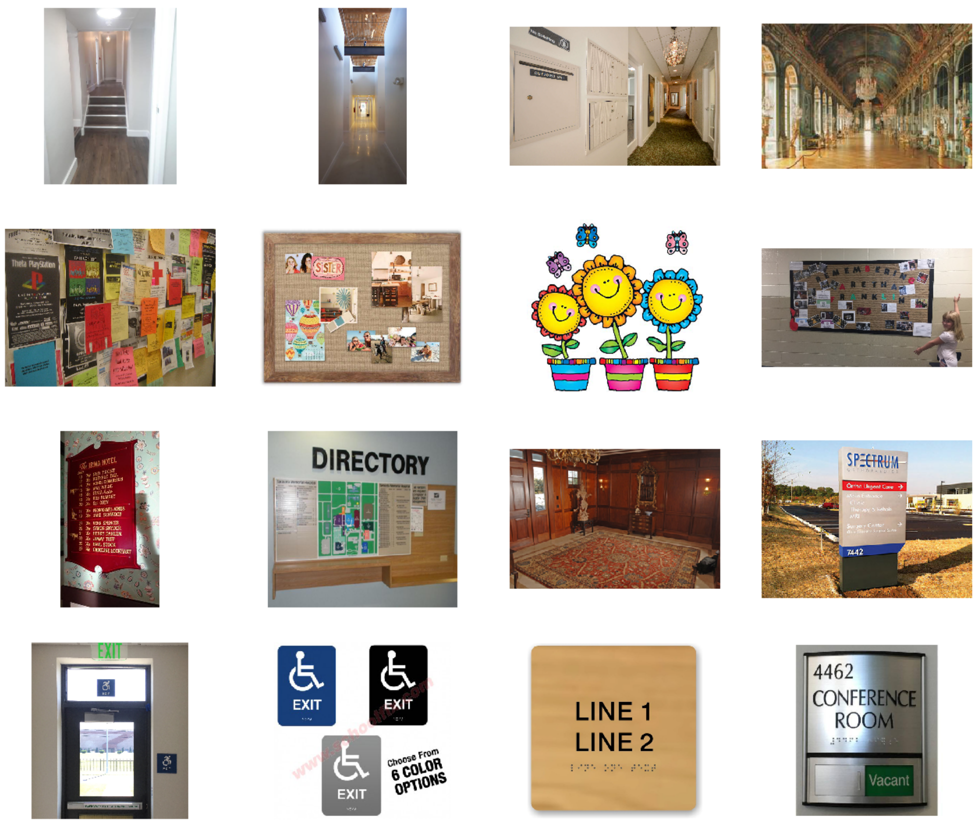

Figure 1: Sample of "hallway", "bulletin", "directory", and "plaque" images scraped from the internet

3.3.2 Dataset of images from the University Hallway

Attendance at, and full access to, the campus of a modern institution of higher learning provides an excellent opportunity to gather a dataset for this application. I had taken some initial photographs and video of Kutztown University's Old Main building before development had begun, but the quality and size of this dataset was not sufficient as the requirements became better understood. I also collected some images from buildings on the campus of U.C. Berkeley, however ADA-compliant room identification placards were not present in the buildings which I had access to. Ultimately video footage of the Computer Science hallway was collected by mounting a video camera on a cart which was pushed slowly up and down the hallway in order to prevent motion blur. I processed this footage into individual frames, which became the real-world image dataset for this project. This data was labeled and split into test and train subsets for use with the HOG detector. Plaques were cropped and labeled for use in evaluating the performance of the different text region detection methods.

3.3.3 Dataset of images with random noise

In order to satisfy the need for a dataset representative of some real-world use situations which would be encountered by the image detection, I expanded upon the tools used to manipulate a small dataset in order to *create* a dataset, reasoning that the visually simple subject matter, combined with the restricted points of view for the real-world use of the system, made this an appropriate candidate for creating a fully-synthetic dataset. The idea was that a neural network trained on synthetic data would perform equally well (or nearly so) to one trained on a real, photographed dataset. The appearance and placement of room-identifying plaques are regulated and well documented, allowing for a faithful re-creation of the small dataset collected from Kutztown University.

The file makeTrainingData.py uses the DataGenerator.ImageGenerator class to create both positive (has a plaque) and negative (does not have a plaque) images. The results are labeled and saved locally. The DataGenerator.ImageGenerator class automates the process of single image creation. It takes an IMAGECLASS argument, which specifies the type of the image being created. An instance of the ImageGenerator class is created with a seed, which is then used to set a pseudo-random number generator, which is used in turn to add noise, shapes, lines, and plaque placement in the final image. The initial seed in ImageGenerator generates a pseudo random seed used by the functions in the GeneratedImage class, generating unique objects for the dataset.

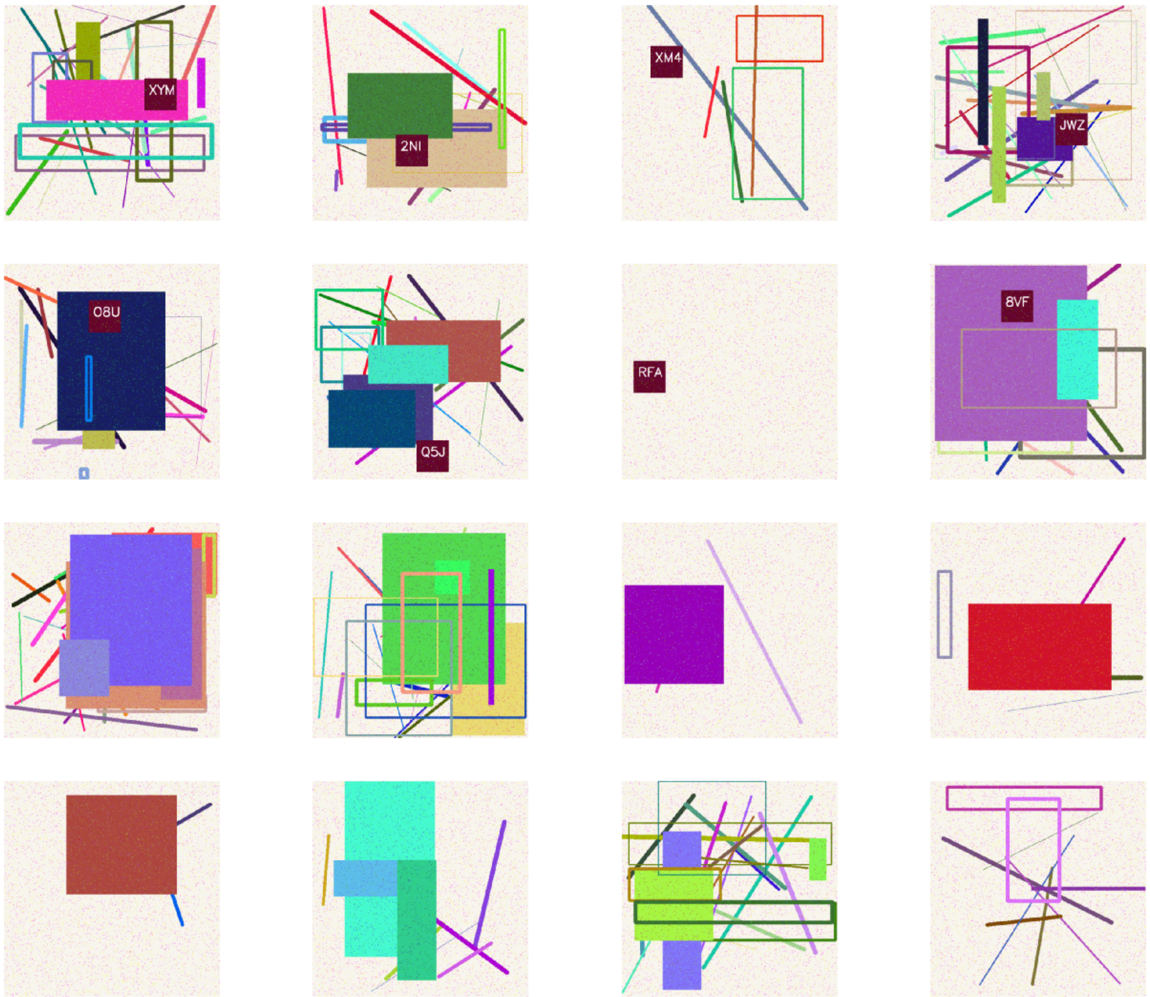

Figure 2: Images generated with random elements and noise in the background

The create_canvas returns the “blank” image for further processing. Both the make_false_image and make_true_image share much of the same functionality, except for the placement of a plaque in the image for “true” images. On top of the blank image generated by create_image, the add_stuff function populates the image with a specified number of lines and rectangles using the Generated-Image methods. This calculates two random points and draws a line of random color and thickness. This is meant to replicate (in a general way) the kinds of items found in a scholastic hallway setting, such as pipes, conduit, bulletin boards, flyers, posters, passers-by, and other kinds of wall decoration.

The results of this exercise, as shown in Figure 2, seemed promising from an aesthetic perspective, but also were quite abstract compared to what one would find in real-world images of hallways. After evaluating the performance of an object detector trained on this data, a more specifically-designed dataset was generated to give a better description of the plaques actually encountered in the testing images.

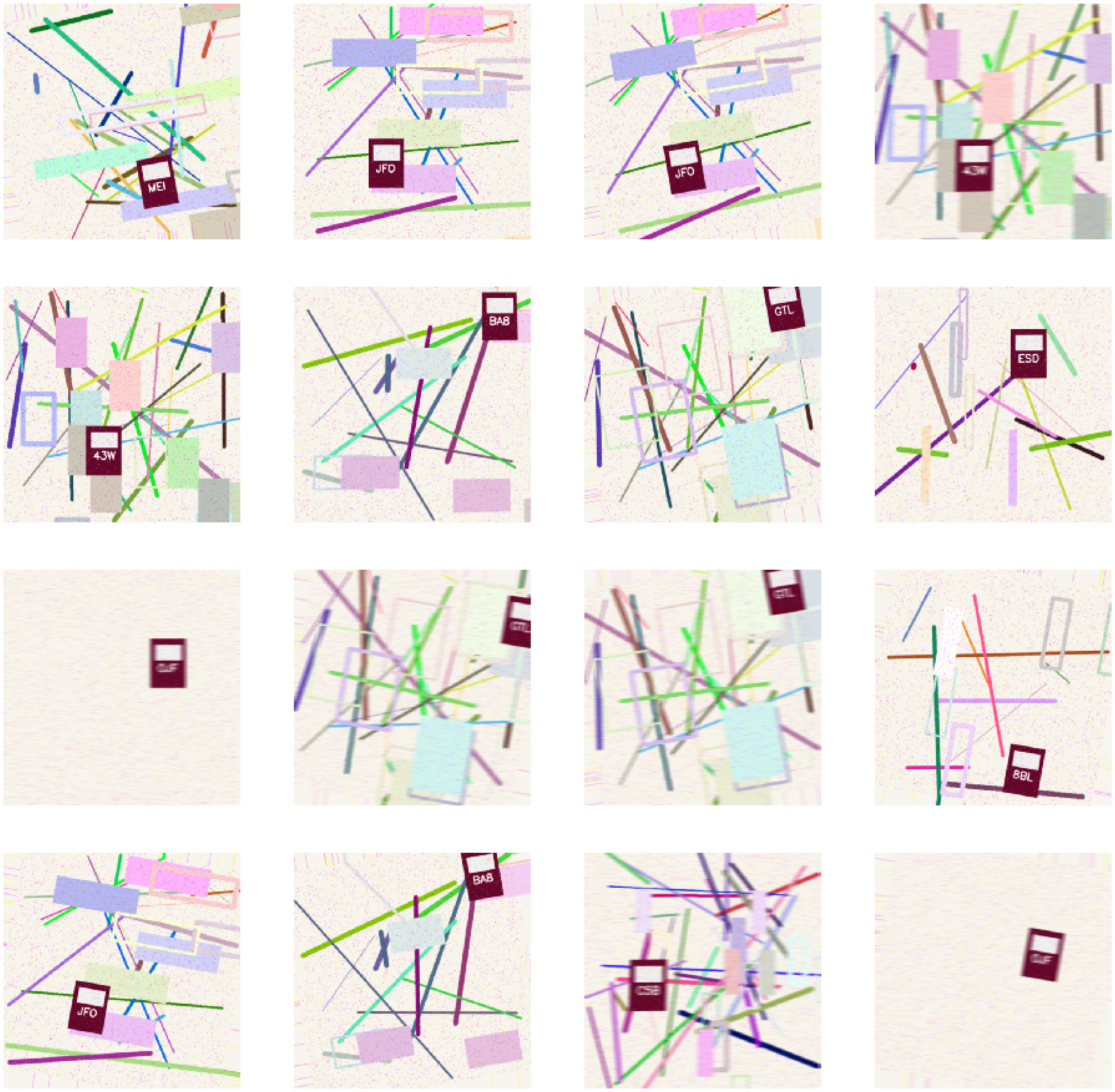

Figure 3: More purposefully-designed plaques

Here, the shape of the plaque was adjusted to reflect the rectangular nature of the room identifying plaques (comparison of these different plaques in Figure 4), with the addition of the white window used to identify the office occupants.

Figure 4: Comparison of the generic plaque, real plaque, and purpose-designed plaque

Additional shapes are added in draw_special_room_sign(), which first creates a rectangle with a similar height/width ratio to the university room plaques, adds a white rectangle, and returns pixel coordinates for drawing the room number in a more accurate position. Blur, rotation, and skew are applied to the generated images (as in Figure 3) and saved.

Dataset of constructed hallways After evaluating the appearance and performance of the randomly-generated images, I took the image generation concept a step further. Instead of only creating a room-identification plaque, why not create the hallway? The ADA compliance guidelines for plaque placement in a hallway are well-documented and the structure of a scholastic hallway is generally very simple.

The hallway construction is achieved in the hall_driver.py file, which uses the ImageGenerator class' make_hallway method.

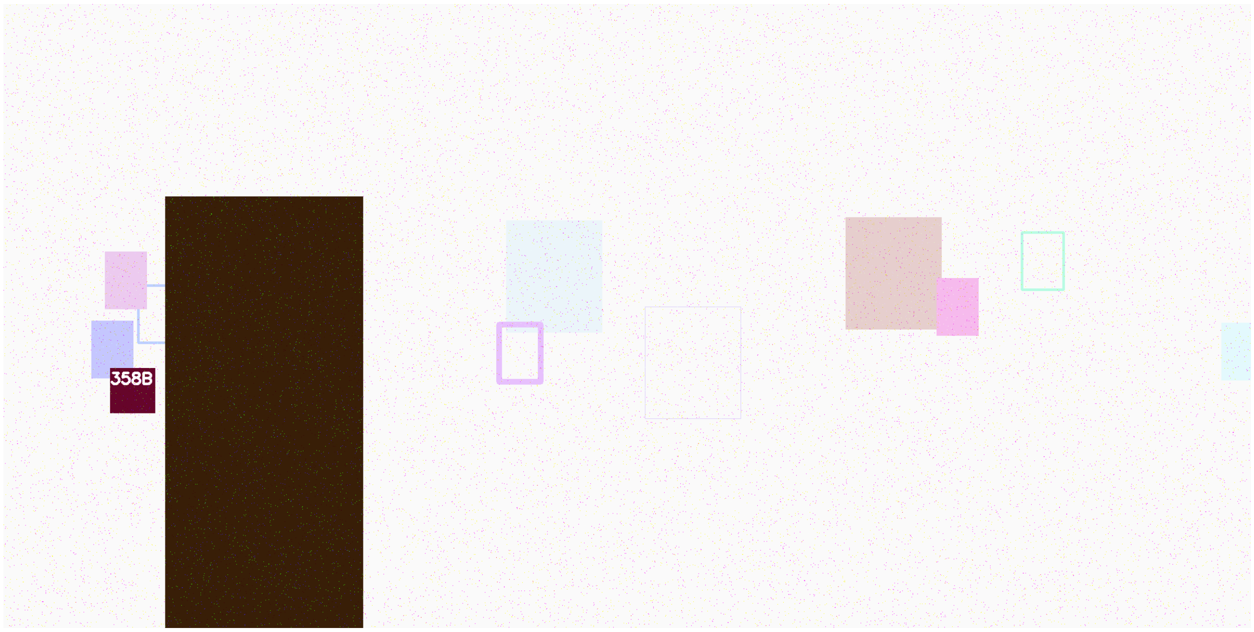

Figure 5: Image of a constructed "hallway" complete with doorway, plaque, and posters on the wall

This hallway construction uses the ADA.py module to specify the proper font size and placement for the given image size. Rectangles representing the papers, notices, and billboards commonly seen on scholastic hallways are generated as 8 * 11 for papers, and a random value for height and width between 12 and 36 inches to provide some realistic boundaries on possible background noise. These are peppered about in a "visibility zone", between what would be 80 and 36 inches from the ceiling. This achieves an average center height of what would be 57 inches, the recommended hanging height for visual artifacts (Reddigari & Vila, 2020).

A plaque is then placed at the ADA-compliant height of between 48 and 60 inches from the ground (U.S. DOJ, 2010, ss 703.4.1) at a random spot in the image. Depending on the proximity of this plaque element to the edge of the canvas, the door is placed either on the left or the right of the plaque, wherever there is enough space to contain the whole shape.

After these elements are complete, a light misting of random noise (salt and pepper) is added over everything to complete the illusion, as in Figure 5.

These files are saved with a specific notation which allows for further processing to "know" where the plaque is and apply the correct label to images. It is in the format

<top left X>_<top left Y>.<bottom right X>_<bottom right Y>_<nth image>.png

These “hallway” images are then processed in snapshot.py, where labeled snapshots are created. The location of the plaque is interpreted from the filename, and for both positive and negative results, a set of valid coordinates are calculated.

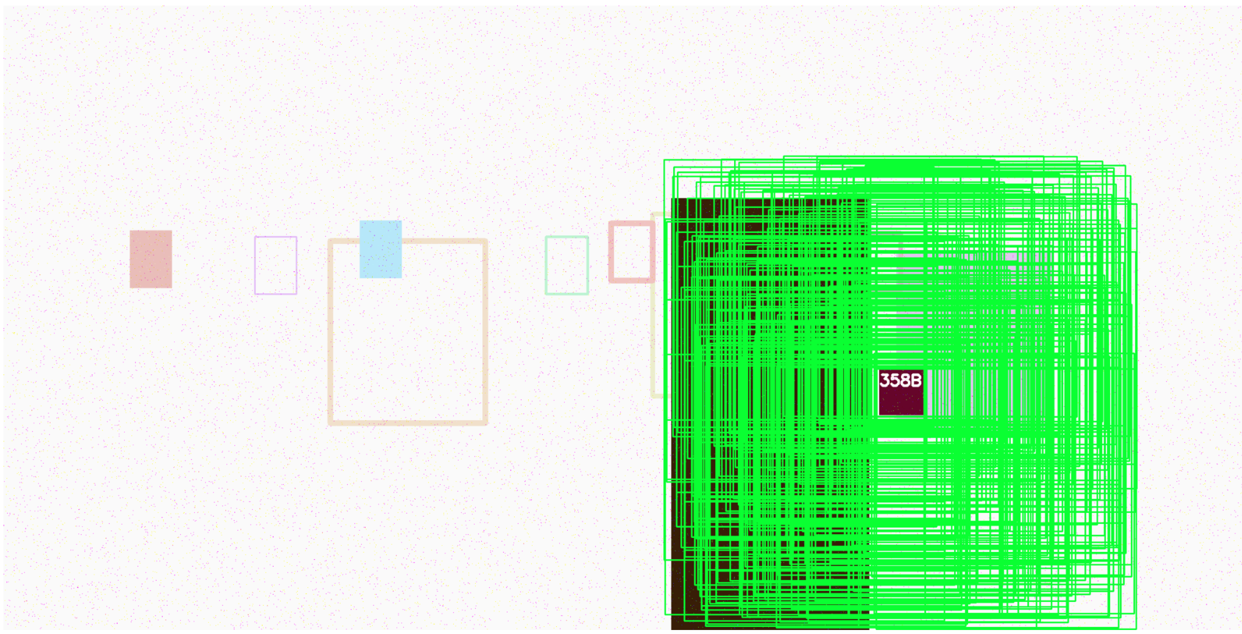

Figure 6: The realm of possibilities, for 200 images of 500 pixel dimension

For positive images (those that will contain a plaque), the smaller “x” and “y” coordinates (what will be the top left of the image) are chosen at random from a range of numbers between right boundary - window size and the left boundary, which will make sure the “x” coordinate will always include the entire plaque. The “y” is done in the same way, where the window size is subtracted from the greater number (in this case the bottom) providing the lower boundary for our valid coordinates. This is illustrated by green rectangles in Figure 6.

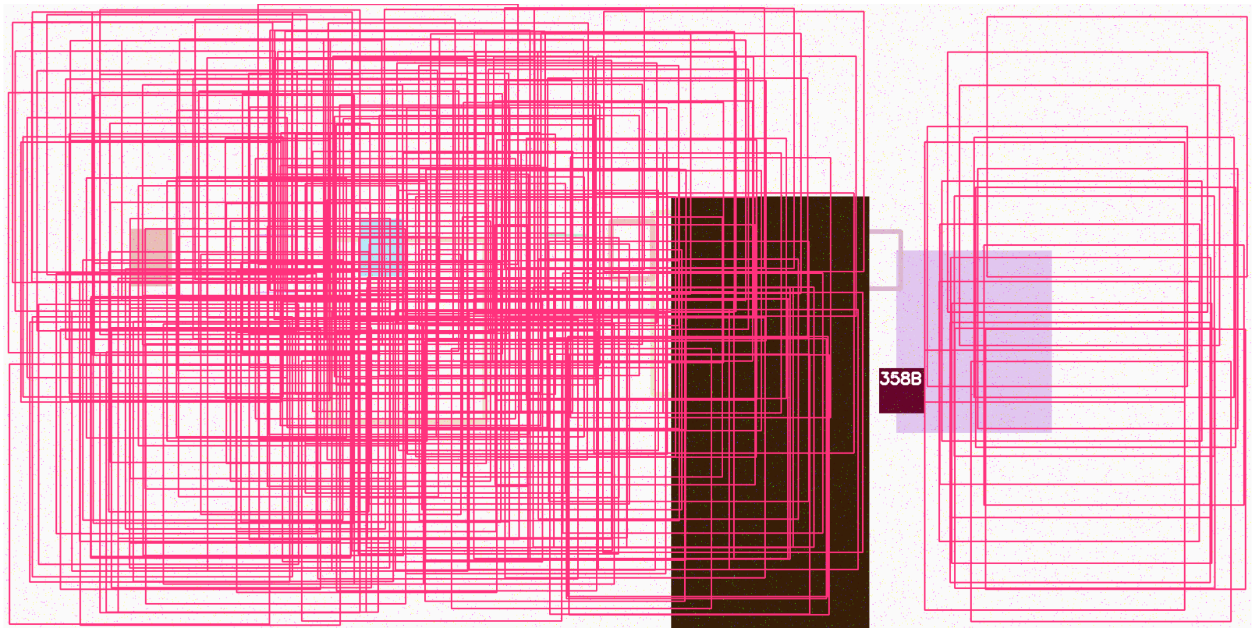

Figure 7: Possible crops from this image which will not include a plaque

Negative images (those that should not contain a plaque) take a different approach. In order to make sure the plaque is not included, a "danger zone" is specified between the x coordinates of the plaque. The left_of_plaque and right_of_plaque are the valid ranges of starting x coordinates for non-plaque snapshots. These also take into consideration the window size, preventing tiny slivers of image from being used in model training or evaluation. These sets are combined, creating the empty "danger zone" visualized in Figure 7. While there is an unused realm above and below the plaque, these areas are not much different from the rest of the generated hallway, and this method simplifies the implementation.

Plaques in close proximity to the edges of the canvas must also be considered, as in Figures 8 and 9. The possibilities are limited but may still provide additional volume to the dataset.

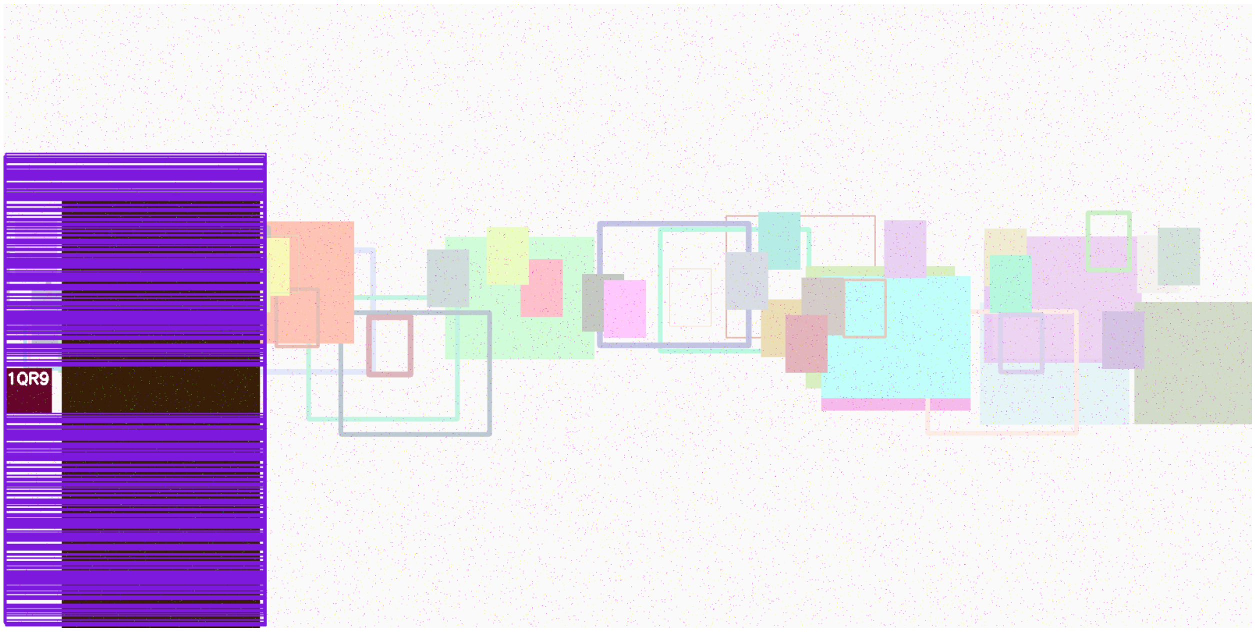

Figure 8: 200 limited possibilities for a plaque so near the left edge of the image

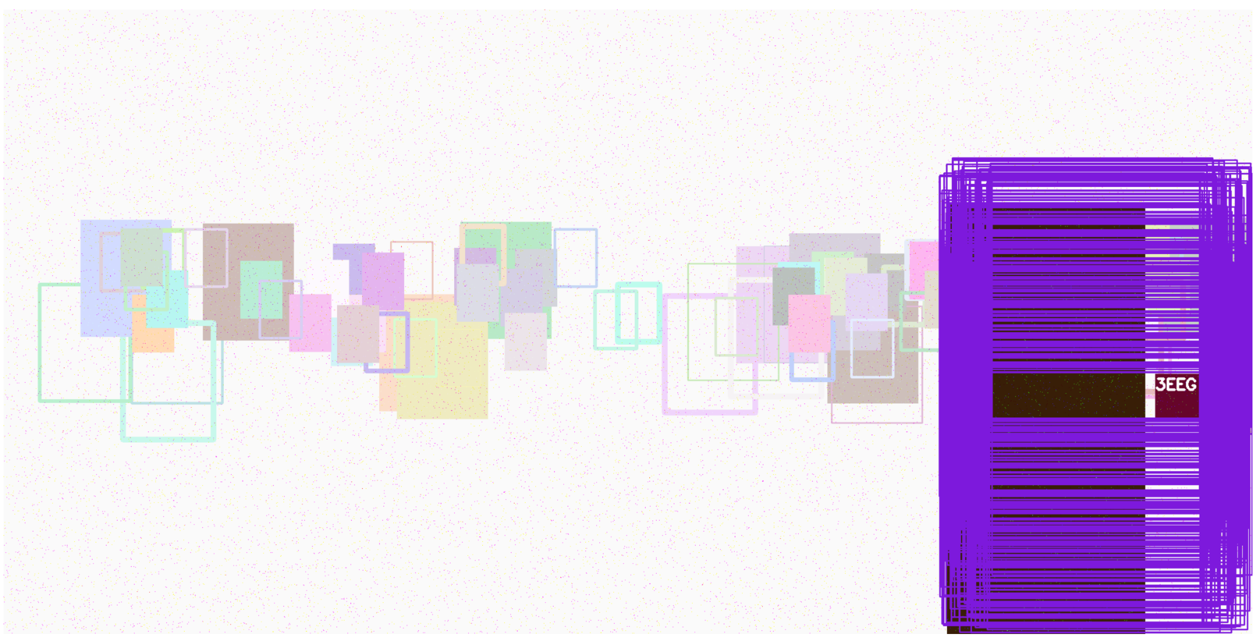

Figure 9: 200 of the possible snapshots from this plaque placement

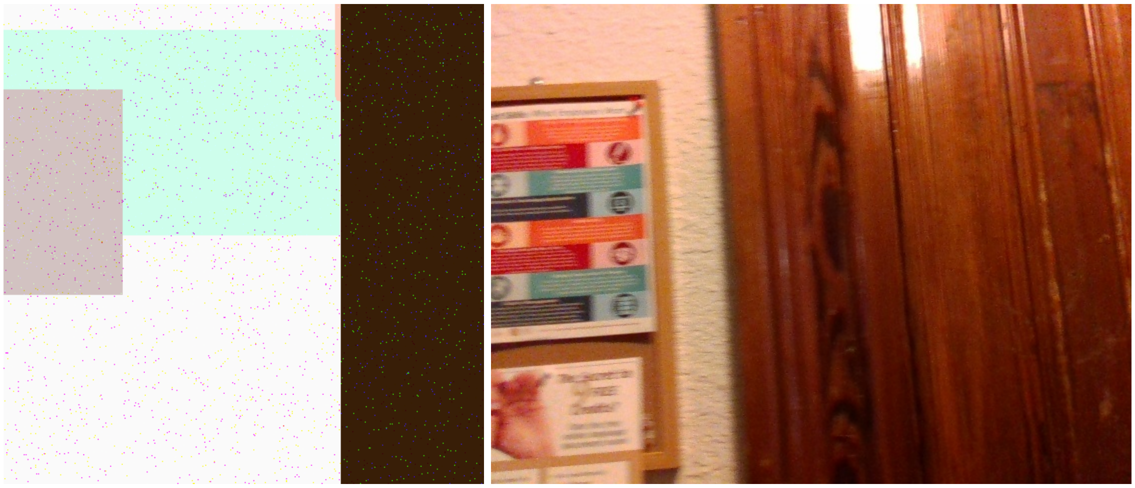

Figure 10: Left: snapshot of hallway with no plaque in it, Right: actual no-plaque image

Figure 11: Left: snapshot of hallway with plaque in it, Right: actual plaque-having image

With the above methods, a novel and fully-labeled dataset can be constructed programmatically, producing, in this case, results which bear a significant similarity to images gathered through photography as seen in Figures 10 and 11.

3.4 Results

For a discussion of the performance of these different datasets in training models, please refer to the relevant method in the Feature Extraction and Text Extraction sections below.

3.5 Data Collection Conclusion

For the specific purpose of identifying plaques in the CSIT hallway of Kutztown University's Old Main facility, capturing images via pictures or frames of video was the simplest. Generating data, given the simplicity of the subject (shape on wall) proved to be effective, and opens the door for generating a dataset specific to the needs of the building. Internet searches for this particular subject were inconsistent and did not provide much value.

4 Feature Extraction

4.1 Problem Definition

Discovering whether an image or video frame contains a plaque is valuable, but less valuable in this use case than discovering *where,* precisely, that plaque is situated. The more accurate the information on where the plaque is in the image, the better it can be associated with a specific pose information for use in the actual mapping. A variety of methods were experimented with in order to reliably discover the location of a plaque in an image. The first is a manual calibration, done before a dataset of images is run through the detection pipeline. The second uses a detector based on the Histogram of Oriented Gradients, trained on annotated images.

4.2 Literature Review

HOG, or Histogram of Oriented Gradients, is a popular and state-of-the art method for accomplishing object detection efficiently. From the paper that introduced this method for detecting humans with a sliding window framework Dalal and Triggs (2005), “The basic idea is that local object appearance and shape can often be characterized ... by the distribution of local intensity gradients or edge directions”. This is accomplished by splitting an image up into a grid. Each section of the grid is normalized to reduce the impact of variable luminosity in the image, and the gradient vector of this area is calculated. This gradient describes the direction of maximal slope (where the values change the most, like a division between light and dark areas) as well as the magnitude of that change. This results in a set of descriptors based on the content of the training images, visualized in Figure 12.

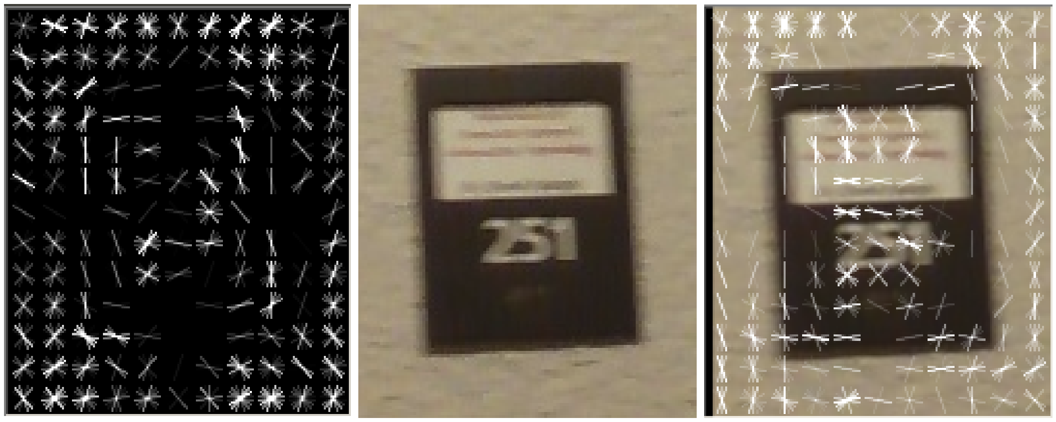

Figure 12: Visualization of the HOG descriptor trained on images of KU Old Main hallway with an example and the overlay. The visualization shows how the edges of the plaque and textual elements are represented.

The implementation used in this project is provided in the dlib machine learning toolbox for python. This implementation, described by King (2015), uses "max-Margin Object Detection" to find the parameters of the object detector and make full use of the data in the image, such as windows which overlap with a target window. It is also very easy to use in the simple_object_detector class.

4.3 Method

The first method relies on active human intervention to pick a representative plaque image and then choose the plaque from the various options presented. It utilizes the cv2.findContours() function to find contours in the image, and draw polygons around these possible shapes, presenting a menu to choose the correct polygon. OpenCV Contours can be thought of as "a curve joining all the continuous points (along the boundary), having [the] same color or intensity" (Bradski & Kaehler, 2000). Generally the image is converted to grayscale, and a threshold applied to this grayscale image to accentuate the edges of the different shapes in the image as in Figure 13.

After the contours are found, a bounding rectangle is drawn around the contour's area, labeled, and presented along with a radio button, shown in Figure 14.

The actual plaque finding method uses the catchWeirdShape() function to disregard any poly-

Figure 13: Stages of finding contours

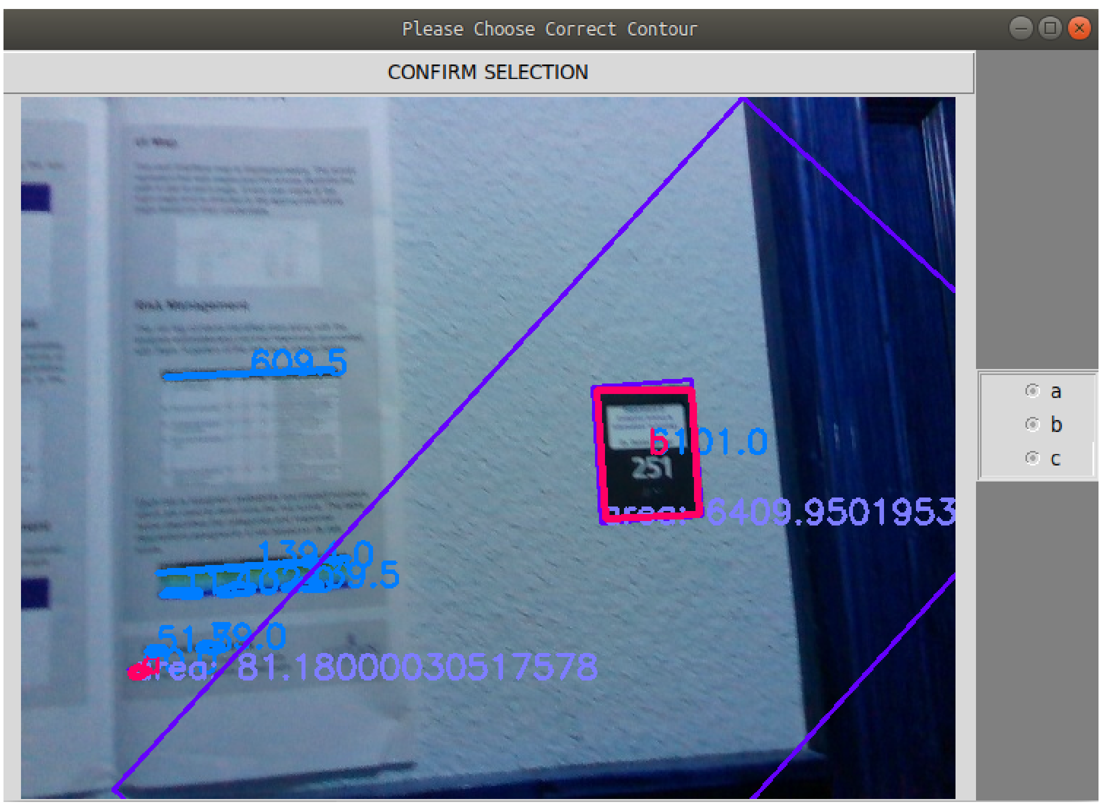

Figure 14: Only polygons which fit a reasonable shape and size are choosable

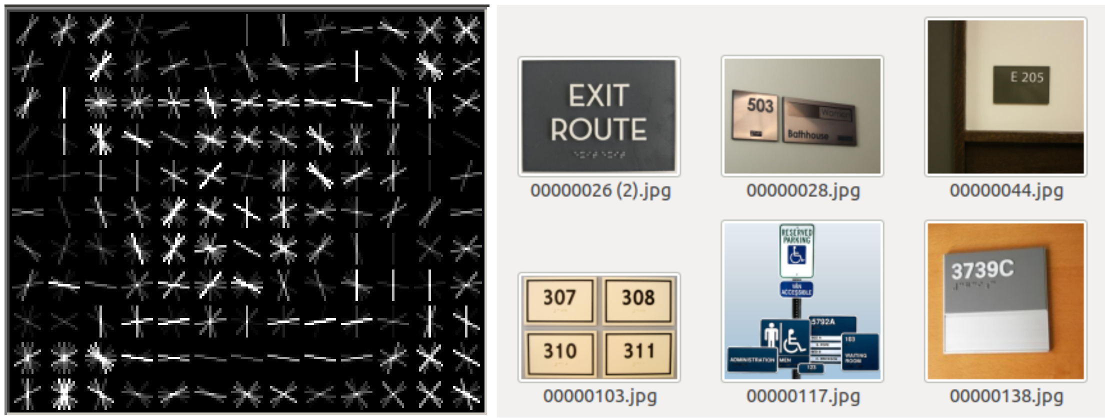

Figure 15: Results of creating HOG from images scraped from the internet. Note the variety of shapes and orientations of the images.

gons which have a width to height ratio not between ½ and 2. These numbers were found through trial-and-error, boundaries which would restrict much of the noisy contours found in an image while still letting the plaque contours through.

The output of this function returns an area and ratio to look for when trying to locate plaques in the rest of the images. If all the images are taken from an identical distance from the wall, and all from the same building, this method can work effectively, as all the plaques will be a similar shape, and a similar photographed size. This expectation is not very realistic, however, so the plaque detection pipeline relying on area and height/width ratio takes an additional parameter of cutoff_ratio, which allows a variance between the area discovered with the polygon menu and the area of a potential plaque.

The source code for the annotation of images, training and testing of the detector come from an excellent tutorial by Talari (2017) for identifying clocks. It utilizes the dlib simple_object_detector class described above to generate the object detector.

Training images are first annotated through an interactive script which allows the user to draw a rectangle around the target object, and saves this metadata to be used in the training of the detector. Separate detectors were trained on internet scraped images, generated images with generic plaques, generated images with more specific plaques, and images collected from life. The detectors were then tested against the real-world collected images to measure their effectiveness.

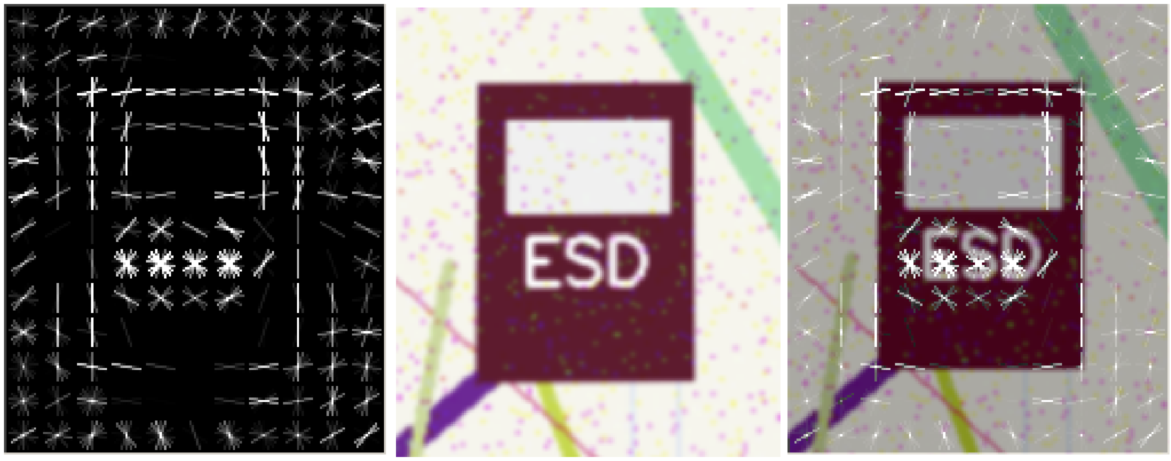

Figure 16: HOG created with a specifically-generated dataset

Different detectors and a sample of their training data are shown in Figures 15 and 16.

4.4 Results

The manual "area heuristic" detector achieved a Precision of 57.3%, Recall of 57.2%, and F-Score of 57.2%, a little better than random guessing.

	Area Found plaque	Area Found nothing
No plaque	FP = 180	TN = 648
Whole plaque	TP = 226	FN = 141
Partial plaque	TP = 16	FN = 40

Table 1: Results for area-based heuristic

The HOG -based object detector performed very well when trained on a subset of real-world images. There are 1251 total images, of which 828 have no plaque, 367 have a whole plaque, and 56 have some fragment of a plaque.

This detector achieved a Precision of 100%, Recall of 94.3%, F-Score of 97.1%.

A detector was also trained on 25 of the randomly generated plaque images.

The generated images were not an exact match for the real-world hallway data set, but still managed to capture a Precision of 84.3%, Recall of 48.8%, and F-Score of 61.8%. This detector

	HOG Found plaque	HOG found extra	HOG Found nothing
No plaque	FP = 5	FP = 0	TN = 824
Whole plaque	TP = 367	FP = 0	FN = 0
Partial plaque	TP = 33	FP = 1	FN = 22

Table 2: Results of the hog descriptor run on the real-world dataset

	HOG Found plaque	HOG found extra	HOG Found nothing
No plaque	FP = 32	FP = 3	TN = 796
Whole plaque	TP = 201	FP = 2	FN = 164
Partial plaque	TP = 4	FP = 1	FN = 51

Table 3: Results for HOG detector trained on fabricated data not imitating actual plaque

seems to be better at identifying the numerical part of the plaques, owing to the training set. It also had some issue with lighting artifacts being identified, as well as a paper towel dispenser.

The results of the HOG trained on a dataset of more specific generated plaque shapes performed even better: Precision of 100%, Recall of 81.8%, F-Score of 90%. It caught all of the well-lit normal room-identifying plaques (Figure 17), but missed all but one of the Restroom signs, as well as a few of the more dimly-lit plaques (Figure 18).

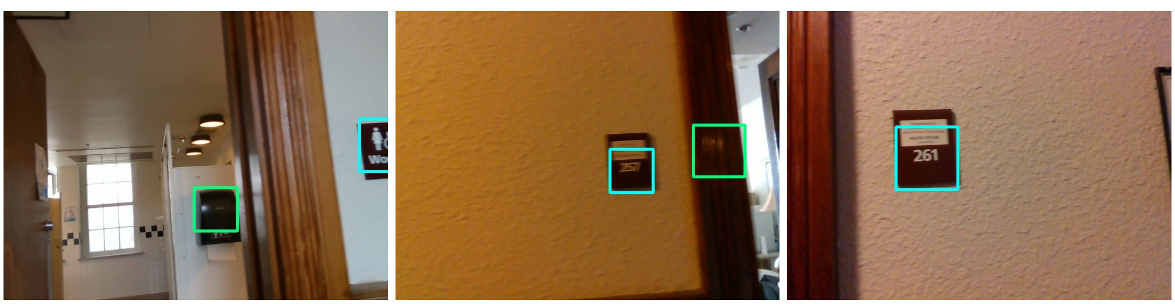

Figure 17: Misidentified towel dispenser, and a preference for the squarish bottom half of plaques

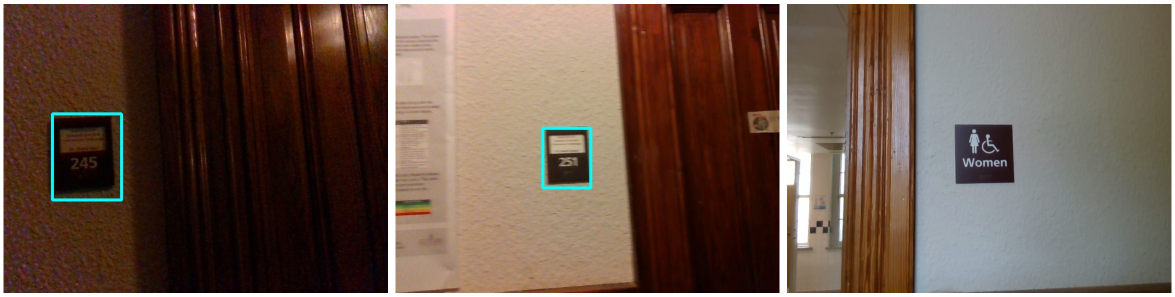

Figure 18: Captured plaques, and missed bathroom sign

	HOG Found plaque	HOG found extra	HOG Found nothing
No plaque	FP = 0	FP = 0	TN = 828
Whole plaque	TP = 307	FP = 0	FN = 60
Partial plaque	TP = 39	FP = 0	FN = 17

Table 4: Results of HOG trained on purpose-built dataset

4.5 Feature Extraction Conclusion

4.5.1 Manual Heuristic

The exploration of an area heuristic was mainly an effort to show why other tools and methods exist for object detection, even with something so simple as a rectangle with words on it. The test results on the KU dataset were slightly better than random guessing, but not by much. In a real world application, where this system is running on video footage, it still may be good enough to discover each plaque at least once, but that relies heavily on the camera operator capturing at least one frame of each plaque at the correct distance to get the plaque at the specified area.

The limitations of the manual "area heuristic" method are straightforward. Even in a best-case scenario, where all images are taken from an exact distance from the wall, all in the same building with consistently shaped and sized plaques, it would still generate many false positives, as in Figure 19.

Figure 19: False positives are hard to handle when a detector only considers area

Any other contours found in the image, which fall within the allowable cutoff_ratio, will be labeled as plaques. This includes posters, documents, or even doorknobs.

The limitations of this method makes sense, as the range of eligible areas required to catch the bulk of the plaques also will capture more “wall noise”, such as the billboards, papers, and artifacts created by the thresholding process used to find the contours shown in Figure 20.

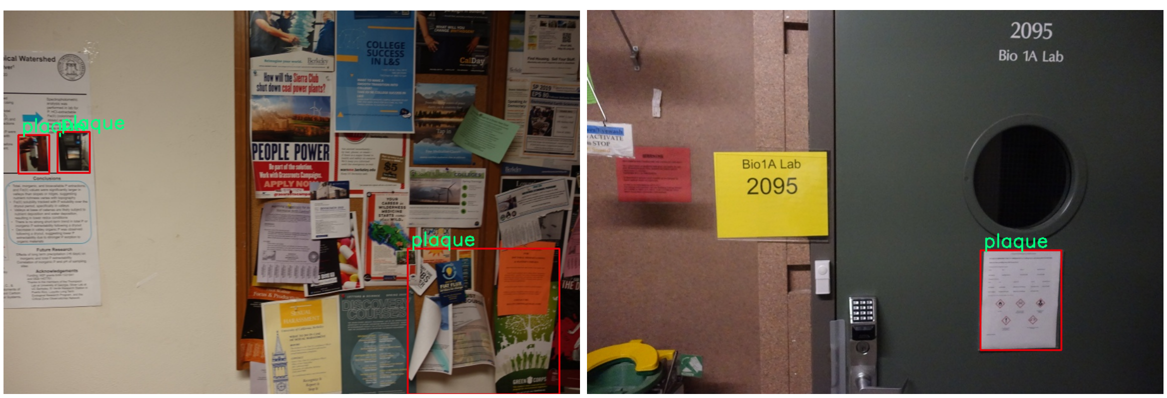

Figure 20: Images from a poster, as well as a collection of documents on a billboard are marked as plaques, and the improvised plaque is not found, while a printout is incorrectly labeled as a plaque

4.5.2 Histogram of Oriented Gradients

The plaque finding, using the HOG method, was fairly accurate for the dataset gathered from Kutztown University. This may be due to the fact that all signs follow a standard visual identity, making it simpler to train a detector on a small and/or fabricated dataset. If there were multiple types of signs, with different shapes and layouts, perhaps a different detector for each plaque type could be trained and all detectors run on the dataset, which would have given better results for the differently laid-out "restroom" signs. Future work could be done on a larger, more varied dataset (perhaps gathered by computer science students across the Pennsylvania Higher Education network) with variations on detectors trained on each type of room-identifying plaque, and other detectors trained on a mix of all images. In a real-world scenario, this tool (when properly trained) could do a thorough job of detecting plaques and room signs.

Since we would rather find a false positive than miss a plaque, this detector would work well for buildings with rectangular plaques with a window above the room number.

There were some instances where it missed a partial plaque, or identified a false positive (Figure 21), but it performed perfectly on images with a complete plaque, and also in low-light conditions as in Figure 22.

Figure 21: Hallucinated plaque, misidentified paper towel dispenser, missed partial plaque

Figure 22: Successful identification of partial plaque, and success in a darkened hallway

For the purpose-built dataset, examining an overlay of the HOG visualization and some samples, one can see that the descriptors are very specifically fitted to the shape and text conventions of the training data, as the overlays in Figure 23 demonstrate. As the Women's restroom sign is of a fairly different layout, the HOG trained on the synthetic data did not recognize this as an interesting object.

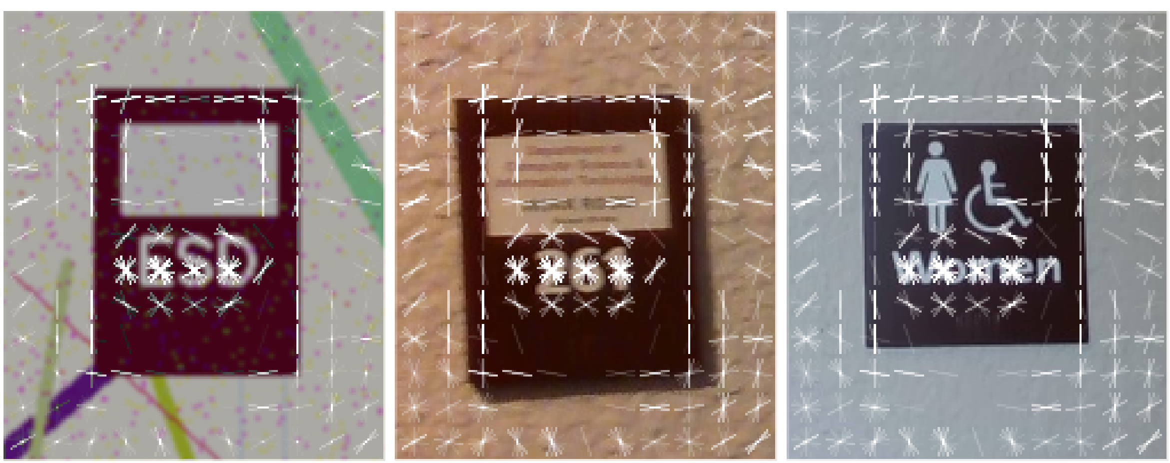

Figure 23: Training sample, room identifying plaque, and the mostly-missed bathroom identifier

5 Text Extraction

5.1 Problem Definition

Public signage, documents, advertisements, and generally any other form of alphanumeric communication is designed to be read by humans. Training a machine system to find and comprehend that text is different from teaching a person to read. Pixels comprising characters written on a plaque have no specific importance compared to all the other pixels in an image. Variations in lighting and camera angle can further frustrate attempts to extract text from an image. Tesseract is a widely used and open-source text extraction library which sits behind a textual region identification system. Assuming a high-fidelity plaque recognition system, I explored two separate methods in order to accurately "discover" the text block in the image. First, applying image transformations and thresholding to select a "box" around the text area. Second, using an implementation of the EAST, or Efficient and Accurate Scene Text detector described by Zhou et al. (2017) to discover the test region. The source data for this exercise was gathered by creating tightly-bound screenshots of the plaques from the real-world images collected from Kutztown University's CSIT (Computer Science and Information Technology) hallway (sample shown in Figure 26).

5.2 Literature Review

Google's Tesseract engine for Optical Character Recognition (OCR), open-source since 2005, is simple to use and there is a profusion of helpful documentation and tutorials available both for the command-line, as well as language-specific ports such as PyTesseract for Python. Described by Smith (2007), it generally works by ingesting a binary image, storing outlines as "blobs", sorting these "blobs" into text lines, then lines into words, and finally detecting the words with an adaptive classifier.

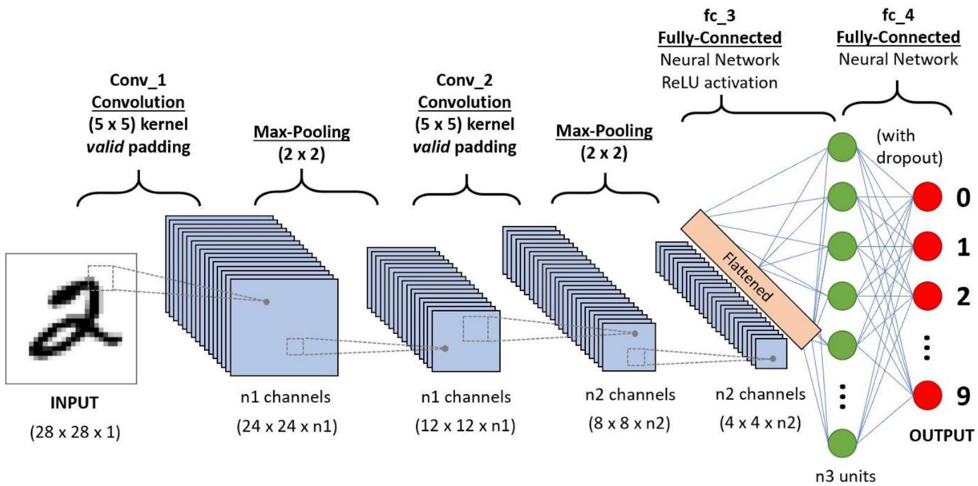

Figure 24: Illustration of convolutional neural network (Saha, 2018)

The EAST detector (Zhou et al., 2017), is a pipeline designed to quickly find regions of text in an image. It uses a convolutional network to extract features from the image. A convolutional neural network (CNN or ConvNet) is a machine learning algorithm which is able to assign learnable weights to different features in an image, illustrated in Figure 24. It achieves this in part by passing convolutional kernels over the source image, performing a matrix multiplication, and passing that output to the next (larger or smaller) layer, with the goal of finding edges and other features. The following "pooling" layer is used to decrease the size of the data (and the computational load) while also reducing noise (Saha, 2018). In the EAST implementation, the resulting feature map is fed over to a merging pipeline at each convolutional step of feature extraction, which allows for both large regions of text (like a billboard, or closeup) to be represented equally with small regions of text (fine print, or far away text). The resulting per-pixel score map and geometry information about the location of the text are thresholded, and those results are fed to a non-maxima suppression (NMS) filter. For each possible discovered result area, all other possible result areas which overlap it are compared; the area with the highest confidence score is kept, and the others are discarded, decreasing computational load while preserving accuracy (Sambasivarao, 2019), The implementation from the paper cited above used a special algorithm which relied on the concept of locality (pixels next to one another should be highly correlated) and merges to decrease runtime complexity.

5.3 Method

5.3.1 Preprocessing

While the actual use of the Tesseract engine is even simpler than its workings, it "... assumes that its input is a binary image with optional polygonal text regions defined" (Smith, 2007). This requires some preprocessing to isolate the text region and apply thresholding to provide optimal input for the Tesseract engine, and the methods used to achieve this are covered in more detail below. Regardless of the method used to find the text region, it still will be passed to the Tesseract engine, and so the image will need to be in an optimal state. In order to understand the effects of different thresholding values and sizes on the performance of the Tesseract OCR engine, a battery of tests were performed, iterating on the size of the image, the timing of the resize, thresholding values, and thresholding methods.

Images start out as BGR (blue-green-red, OpenCV's default color mode, which differs from the standard RGB layout in other tools) and need to be converted to grayscale before the thresholding necessary for Tesseract can be performed. This is done using OpenCV's cvtColor to change the colorspace to grayscale. An additional step which helps to differentiate the foreground (light values, the text in this case) from the background is using skimage's exposure module to rescale the intensity of the grayscale image. This method stretches the highest and lowest values to fit the specified range, in this case 0 to 255 (full black to full white). So an image where the foreground is not very light will have a better contrast after this function, and in the case of multiple images with differing illumination, they will all have a more similar illumination, making it simpler to threshold the images into black and white.

Figure 25: Plain image, converted to grayscale, and with the rescale_instensity. The results below are both thresholded with the same value range (125), but the image in the middle is the result of the rescaled intensity. The bottom image on the right is the result of using Otsu's thresholding on the grayscale image. Note the similarity to the rescaled image.

Another method to simplify binarization of images with different lighting is to use an adaptive threshold. OpenCV provides an implementation of Otsu's thresholding. In "bimodal" images (the value histogram of the image will have two distinct peaks), this mechanism works (in simple words) by finding a threshold value which will sit in between those two peaks (OpenCV, Image Thresholding). Since the source images here are all bimodal (big areas of single intensity values at both the light and dark ends of the spectrum), it works very well in getting properly-separated text.

The results of using Otsu's binarization are very comparable to rescaling the intensity before thresholding as shown in Figure 25. Since ultimately the image must be inverted (text should be black) for the Tesseract engine, and since the exposure rescaling/thresholding combination gave overall better separation of the characters on the plaques, this method was preferred for text processing.

Figure 26: A sampling of the screenshots used to test the character recognition pipelines

Aside from testing different methods of text discovery, the PyTesseract engine was tested on the optimal text boxes with different parameters in order to find the best use for this scenario.

5.3.2 Simple Method

The simple method for finding text areas makes use of the OpenCV library for image manipulation, namely “dilation” and “opening”. Dilation is useful for expanding “foreground” (white) parts of the image. A kernel size is supplied as one of the parameters, which is convolved over the image. At each point of the sliding window, if any of the pixels in the window of the kernel is a lighter value, that value is applied over the whole kernel area. This results in irregular foreground objects (in this case, the text on the plaque) being expanded into a larger, contiguous blob, as shown in Figure 27.

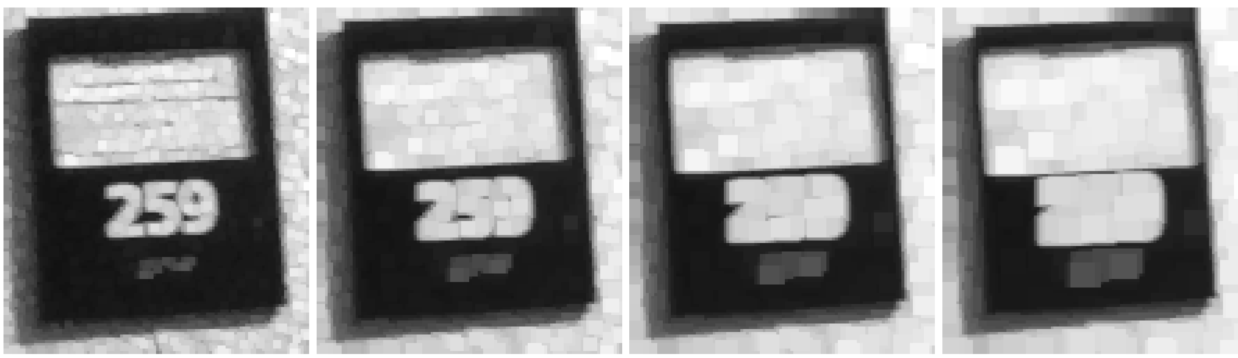

Figure 27: Progression of dilation. Each image represents another iteration of dilation applied over the image.

This allows us to treat the possible test areas as simply another shape in the image.

In this pipeline, the source image is converted to grayscale, the gray image is dilated, and then the dilated image is submitted to thresholding, where contours are found, shown as a green shape in Figure 28.

Figure 28: Image, grayscale, dilated, thresholded, contour, bounding rectangle progression

The bounding rectangle of the contour is then cropped (pink rectangle in Figure 28), and fed to the Tesseract text recognition engine.

5.3.3 EAST Region Of Interest

The implementation used in this project comes from Rosebrook (2017) OpenCV Object Detection tutorial and was implemented based on Zhou et al. (2017). The non-maxima suppression in this implementation is also more straightforward, only calculating and returning which boxes do not overlap. The bounding boxes which are returned by this implementation are used in the project; each box region is cropped, thresholded and inverted, and fed to the Tesseract text recognition engine.

The efficacy of this pipeline was improved after exploring the effects of changing the size of the image being fed to the EAST text detector, as well as border treatments. After disappointing initial results, the EAST text regions were visualized, and there was an obvious trend of the bottom pixels being cut off in the crop. This seemed to be less severe in the larger sizes as in Figure 29, but it was consistent. When reading the code author's notes on the bounding box implementation, it might be possible that a few pixels are being shaved off the return result. To remedy this almost universal phenomenon, 15 pixels were added to the height and width of the cropped region. Since most of the text on the plaques were of a similar size (photographs taken from fairly constant distance and height) this improved the results for the smaller EAST image size settings, shown in Figure 30.

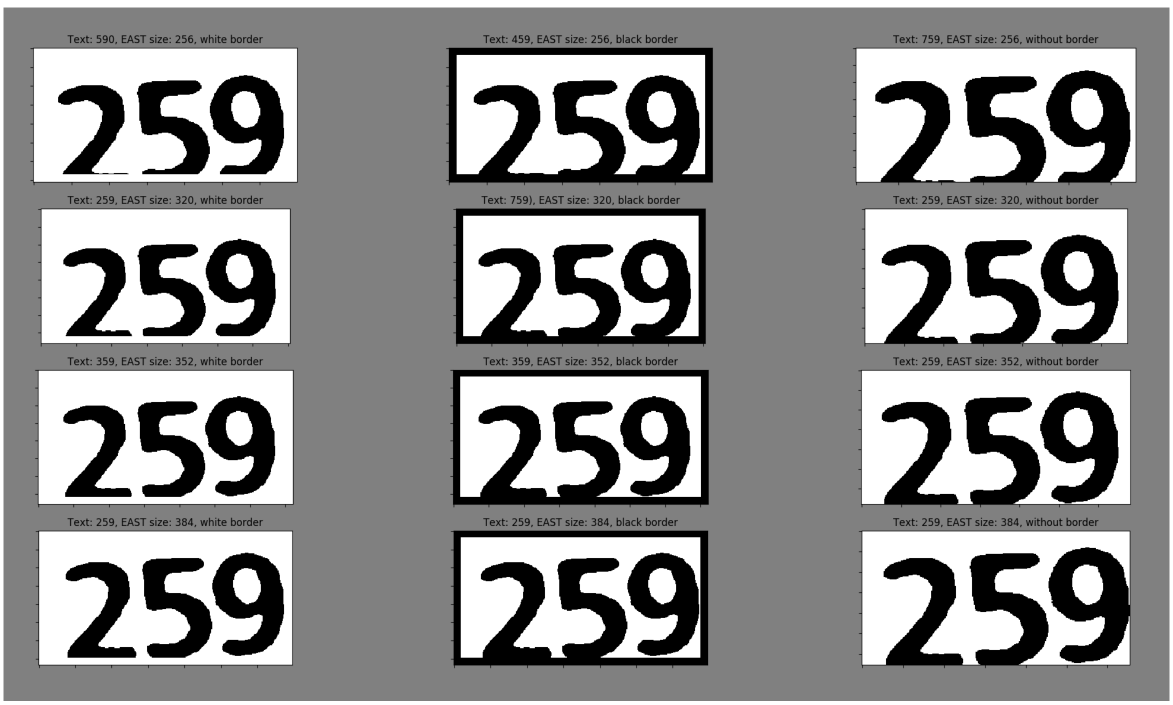

Figure 29: EAST text region crops with no buffer zone, white border, black border, no border.

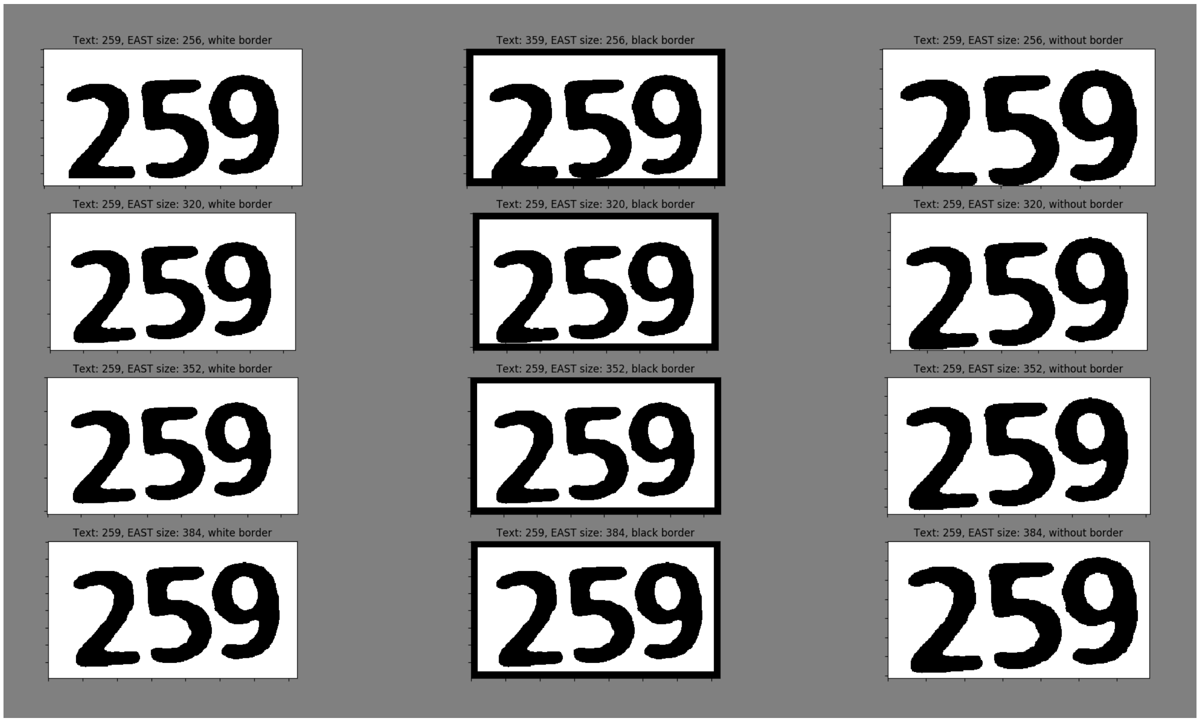

Figure 30: EAST text region crops with 15 pixel buffer, white border, black border, no border. Text reading improves when more of the image is selected.

5.4 Results

The text comprehension was tested on a set of screenshots taken from the real-world dataset. These images were then separated (Figure 31) into three different grades: “normal” for fairly good quality images, “blurry” for images with motion blur, and “dark” from images with low light. There were 33 normal images, 66 blurry images, and 10 dark images.

Figure 31: The three stages of plaque screenshots

5.4.1 Simple Method

The isolated "text only" pipeline was run on the labeled screenshots, using the Tesseract optimizations arrived at via the aforementioned experimentation. Since the dilation of the foreground parts of the image controlled the resulting "text-possibility" areas, when the pipeline was run with only 3 iterations of the dilation step, performance was quite poor with only 11 of the 109 plaques identified correctly. It seemed that the contours being drawn around the images were not accurate, as in Figure 32. Increasing the number of dilation intervals to 5 only exacerbated the problem, with only 6 of the 109 images being labeled correctly. Similarly, decreasing the number of iterations to 2 (Figure 33) only gave an accuracy of 16 out of 109.

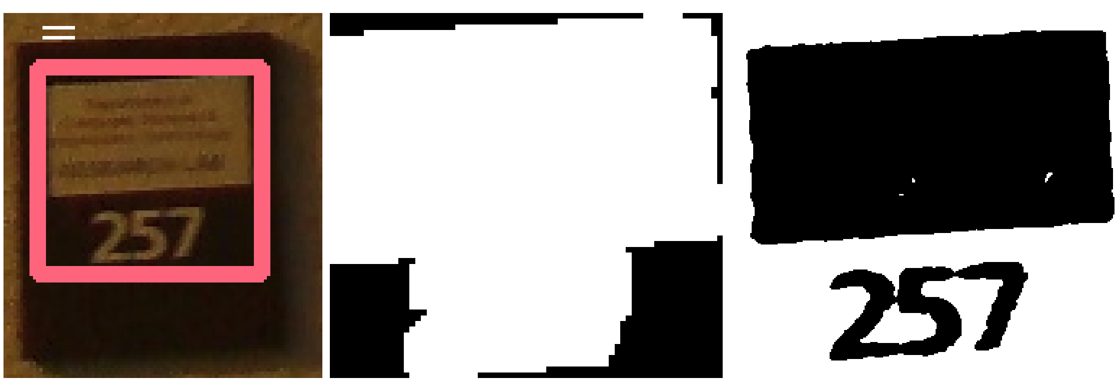

Figure 32: Due to the lighter foreground areas blobbing together in more dilation iterations, the whole area is picked

Figure 33: Less dilation iterations gives better separation

	normal	blurry	dark
No ROI finding	0/33 0%	0/66 0%	0/10 0%
Dilation and contours (3 dilation iterations)	5/33 15.15%	0/66 0%	0/10 0%
Dilation and contours (2 dilation iterations)	10/33 30.3%	0/66 0%	2/10 20%
EAST ROI detection	21/33 63.63%	1/66 1.52%	0/10 0%

Table 5: Results of the text comprehension

5.4.2 EAST region-of-interest

The pipeline utilizing the implementation of the EAST text region of interest detector was run with various configurations on the "screenshot" dataset (already separated plaque images), as well as the entire image frame (wall and all). The addition of a buffer on the bottom and right sides around the region of interest (Figure 34) increased the accuracy of the Tesseract text comprehension. On the sample data, a 25 pixel buffer gave the best results, with diminishing results at 35 pixels and above.

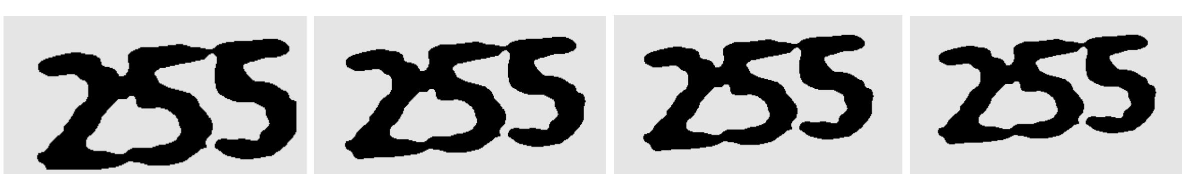

Figure 34: Increased buffers on the bottom and right of the ROI (inverted for effect)

Of the two images which failed with the larger buffer, there was no extra noise or figures caught; it seems that the Tesseract engine can fall down when there is too much white space around the text, as is illustrated in Figure 35. In this specific instance, the text was understood as “>!”, instead of the actual label (251).

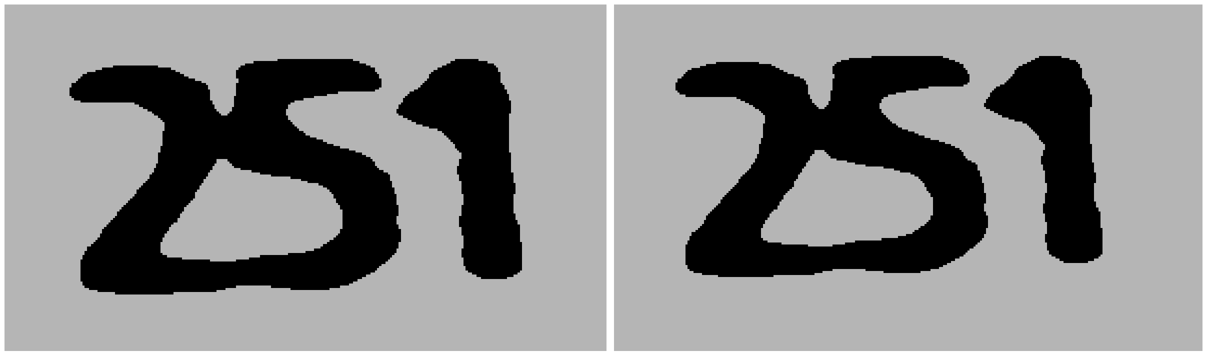

Figure 35: 25 pixel buffer (left), text identified by Tesseract. 35 pixel buffer (right), text missed.

	normal	blurry	dark
EAST on screenshots, no buffer	18/33 54.55%	0/66 0%	0/10 0%
EAST on screenshots, 5 pixel buffer	21/33 63.64%	1/66 1.52%	0/10 0%
EAST on screenshots, 15 pixel buffer	22/33 66.67%	4/66 6.06%	0/10 0%
EAST on screenshots, 25 pixel buffer	25/33 75.76%	4/66 6.06%	0/10 0%
EAST on screenshots, 35 pixel buffer	23/33 69.7%	4/66 6.06%	0/10 0%

Table 6: Results of the text comprehension from EAST text regions

The dark images defied the EAST region of interest detector, which looks at the raw (color) image. To test the efficacy on the thresholded gray image, the grayscale image was converted to the BGR colorspace, without adding any color back to the image. This was then fed through the pipeline. In the dark images, one region of interest was found, but it was not decipherable by the Tesseract engine. In the whole image dataset (367 images from which the plaque screenshots were taken), only 59 plaques were correctly read. This is in contrast with the 142 images in which the model discovered some text ROI. Reviewing the ROI crops, it became obvious that applying a histogram-based threshold on the whole image will not necessarily give the best results when compared to a histogram threshold only applied to the region of interest. The illustration in Figure 36 shows the result of calculating a binary threshold on an image area greater than what is being tested.

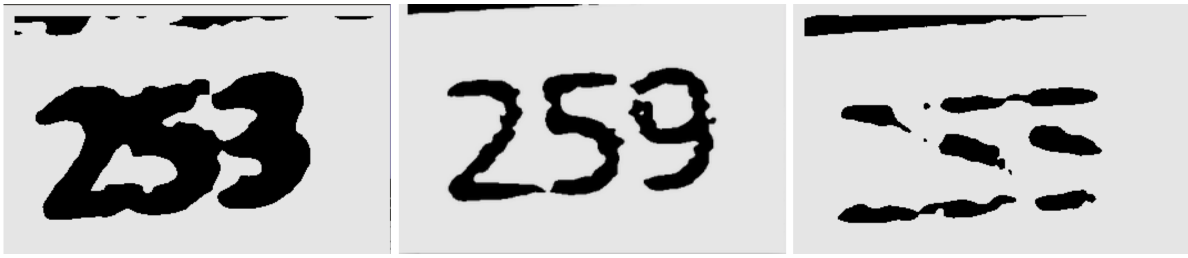

Figure 36: Poor thresholding when applied to the entire image (gray added for effect)

When applying the threshold to only the cropped region of interest, the results were much better. 105 of the 367 total images were read correctly, with ROIs found in 138 images. Those which had an ROI, but no successful text, generally fell into three camps: blurry, incomplete crop, or non-plaque text (Figure 37).

Figure 37: Crops which were not processed by the Tesseract engine, and source images below. From left to right: blurry image, non-plaque text detected, inaccurate ROI

Of the different possible texts to find (['245', '247', '249', '251', '253', '255', '257', '259', '261', '263', 'women'], the names of the rooms), only room 247 was missed completely. All other rooms had at least 1 correct translation, with at least 3 regions of interest being discovered. The completely missed room was also the darkest, nearly unlit, as seen in Figure 38.

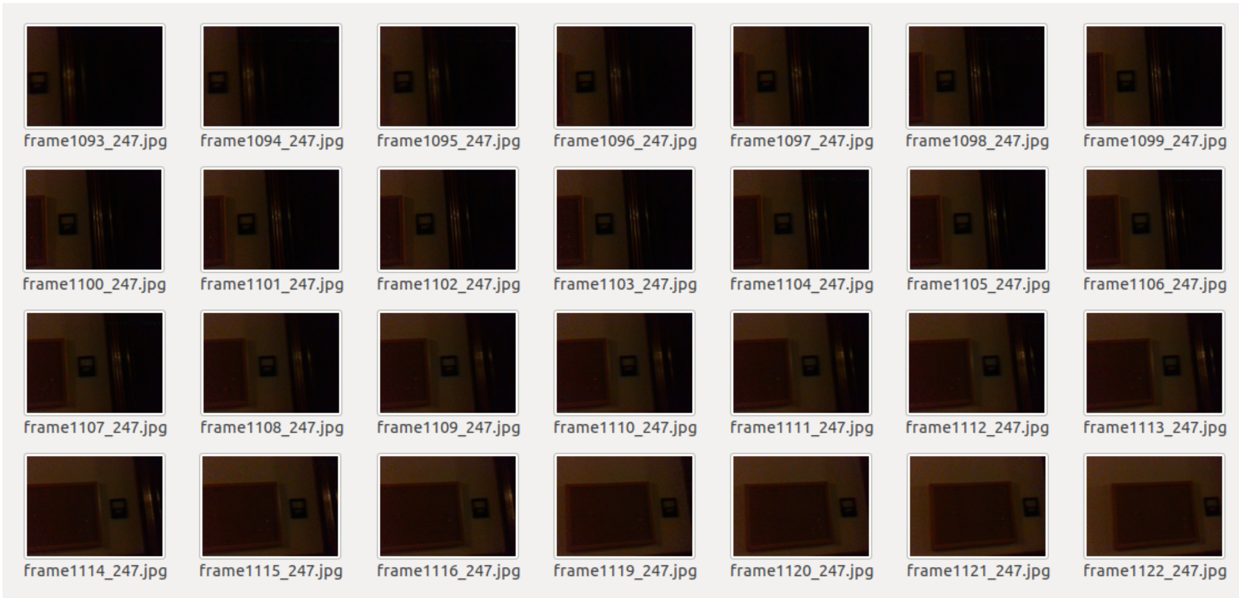

Figure 38: Very dimly lit section of the hallway confounds the EAST ROI implementation

	EAST found plaque text	EAST found other text	EAST Found nothing
No plaque	0	828	742
Whole plaque	125	13	229

Table 7: Results of running EAST on whole images, without cropping the plaque

Plaque text	Plaque present	Plaque roi discovered and labeled	Accuracy
245	54	35	64.81
247	28		0
249	27	5	18.52
251	42	2	4.76
253	31	2	6.45
255	29	1	3.45
257	27	1	3.70
259	28	5	17.86
261	29	7	24.14
263	30	5	16.67
Women	42	42	100.00

Table 8: Efficacy of EAST on whole images, broken down by room number

The excellent accuracy for the 'Women' restroom sign is due to this part of the hallway being well lit, and mostly stationary as it is the beginning of the filmed footage (Figure 39).

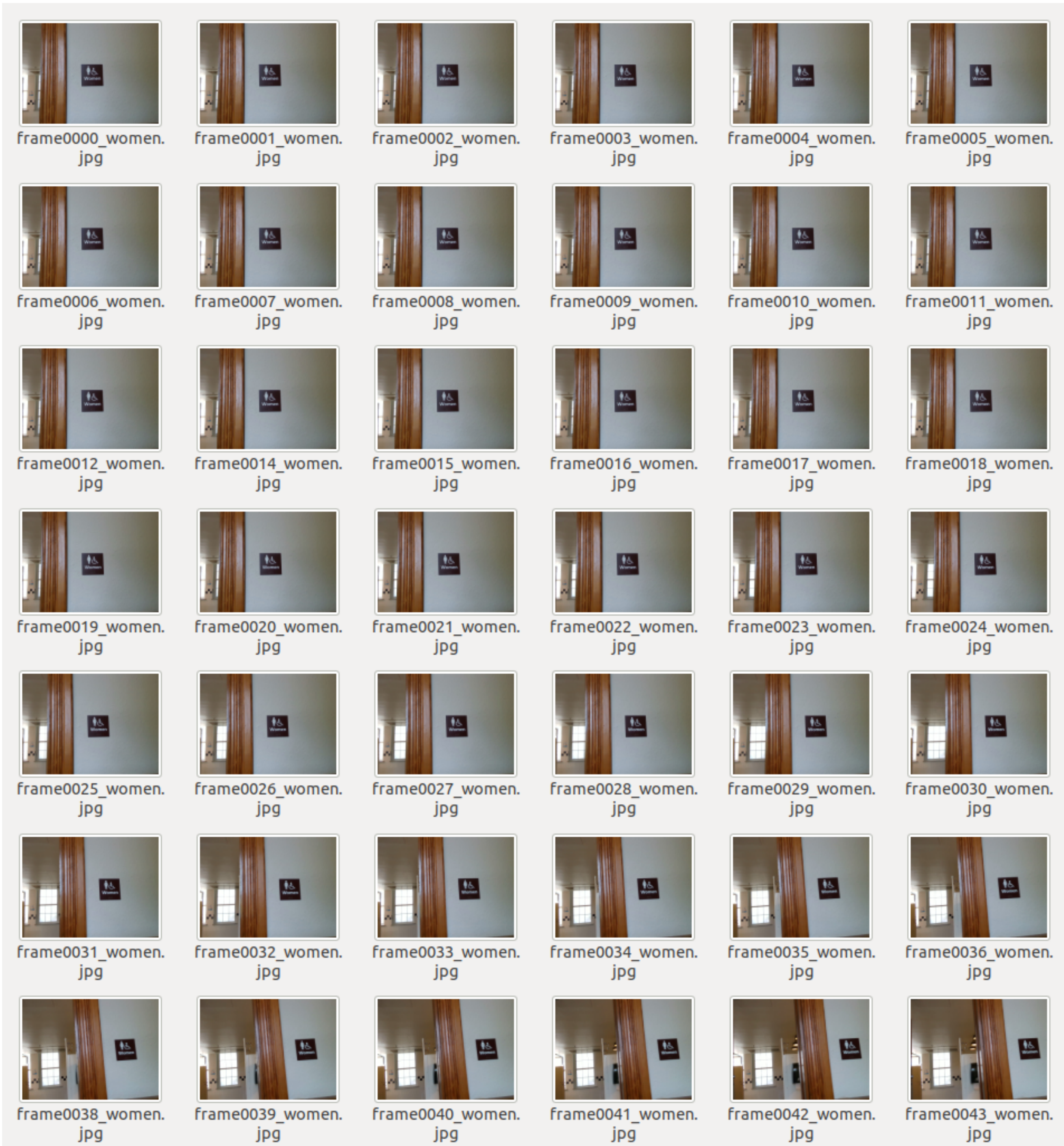

Figure 39: Benefits of being first in line. More clear stationary frames for text detection

5.5 Text Extraction Conclusion

I found that resizing the image before applying binarization greatly increased the quality of the Tesseract results, and the “sweet spot” seemed to be around 80 pixels high (4 times the size of the plaque textual areas, Figure 41). Smaller sizes were mis-labeled and (in some cases) large sizes gave diminishing results; a sampling is shown in Figure 40.

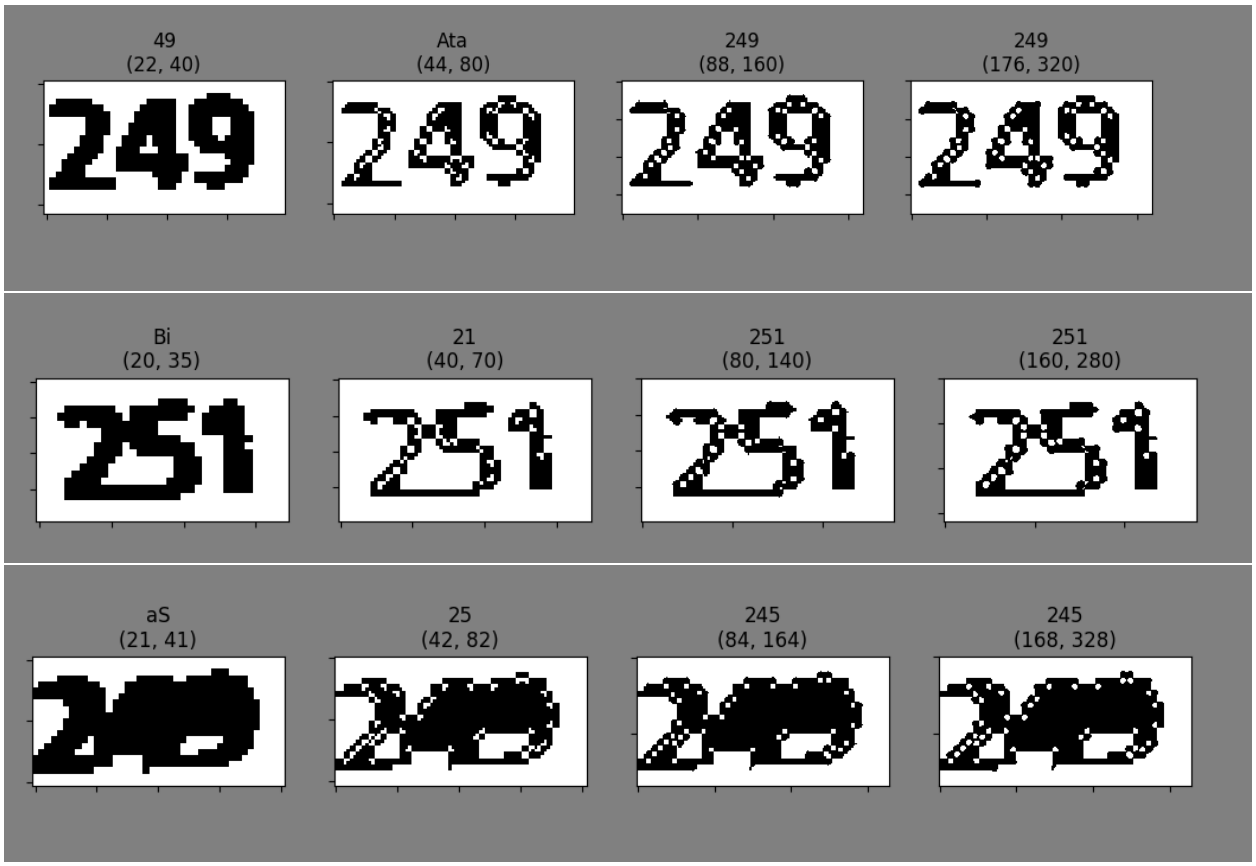

Figure 40: Resizing after thresholding the image. larger seems to be better, but the image quality is quite poor and text is misread

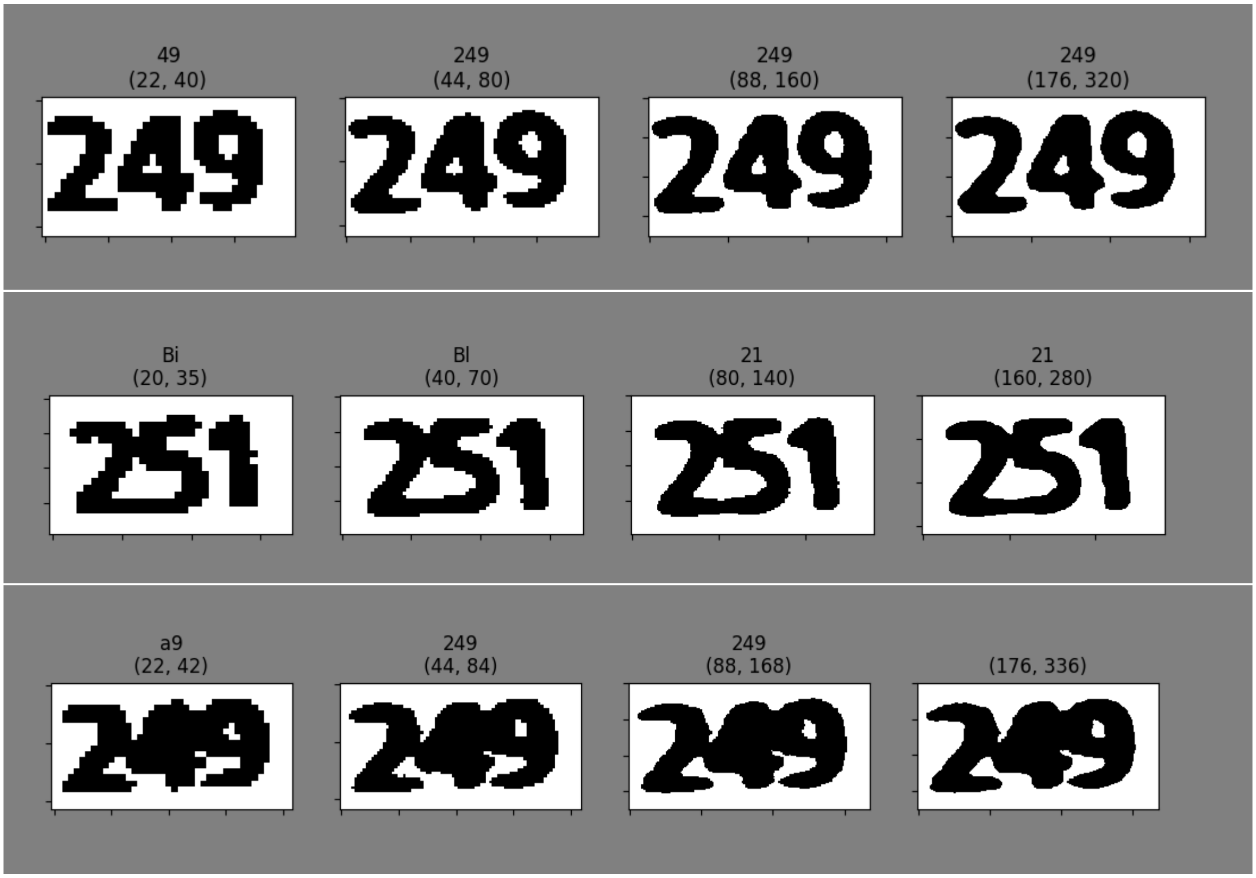

Figure 41: Binarization post-resizing: image quality greatly improves, and generally text is read correctly when possible

The Python extension for the Tesseract engine generally seems to work well for well-cropped and well-aligned text, but can be finicky with the examples encountered in this project. In the EAST pipeline, gray scaling and applying a threshold to the image *after* cropping the ROI gave much better results from the Tesseract engine, and more sensitive preprocessing of the images (conditionally rotating the text, a more nuanced thresholding method) could make these text interpretations more accurate in further work. Having run through the gamut of different modes available for Tesseract, '--psm 9' generated the best results. Additionally, other text reading systems, such as one based on Convolutional Recurrent Neural Network could be used in lieu of a Tesseract implementation.

The system for finding text regions with image manipulation did not perform well on this dataset. Even if there were good results on another dataset (such as dark signs with only a block of light text), it would be completely “overfit”. The “dilation” would need to be reversed depending on the contrast of the plaque (light text on dark, or vise-versa) and any other artifacts which might occur on a room plaque (such as occupancy information, or even designs and visible sign-mounting hardware) would throw off this system’s ability to find a good enough crop of text to send to Tesseract. It also takes longer than the EAST implementation.

The efficacy of this pipeline was improved after exploring the effects of changing the “resize” size of the image being fed to the EAST text detector, as well as border treatments. After disappointing initial results, the EAST text regions were visualized, and there was an obvious trend of the bottom pixels being cut off in the crop. This seemed to be less severe in the larger sizes as in Figure 41, but it was consistent. When reading the code author’s notes on the bounding box implementation, it might be possible that a few pixels are being shaved off the return result. To address this regularly encountered issue, 15 pixels were added to the height and width of the cropped region. Since most of the text on the plaques were of a similar size (photographs taken from fairly constant distance and height) this improved the results for the smaller EAST image size settings.

While some of the images were missed, one benefit of working in an institutional facility is that there is a list of rooms available. So, as long as the pipeline can interpret at least 1 of the frames of the room correctly, it can be registered.

Figure 42: EAST text region crops with white border, black border, no border. Above is no buffer, bottom is 15 pixel buffer. Text reading improves when more image is selected.

The implementation of the EAST text area recognition worked fairly well on the plaque-cropped subset of images, and gave interesting results when run without the “plaque-detection” assumed. If a hallway is well-lit, and the camera is capable of high-quality imaging, and can be transported at a reasonably slow speed, and (most importantly for this scenario) a canonical list of room numbers is available, this method could almost stand alone as a room detector. Now if there are office directories hanging in the hallway, or if the room name shows up as incidental text elsewhere, this could cause issues. There is also another implementation of the EAST text detection pipeline, which allows for rotated bounding boxes and uses a CRNN text-reading network instead of Tesseract. It would be interesting to see how this allowance for rotation, and the possibilities of processing for better text recognition, would compare to the implementation used in this project.

6 Project Conclusion

A good machine learning project starts with a good data set. Collecting images from real-world hallways provided high-quality and labelable data, and using various image manipulation techniques made it possible to expand the size of this set. Elaborating upon these techniques also made it possible to generate a dataset from scratch, allowing for the possibility of more diversity of plaque shapes and designs than may be available when taking photographs by hand.

Discovering the location of the feature in an image, in this case an ADA-compliant room marking plaque, was most successfully accomplished using a Histogram of Oriented Gradients, trained on both the real-world images and the synthesized images. Due to the highly consistent appearance of the different room signs, and the way in which this model utilizes difference in pixel values along edges, most of the plaques were found, even in very low light. A manual method which employed polygon discovery and an area heuristic, performed very poorly and reinforced the usefulness of the HOG method in discovering the identifying plaques.

Two methods, one using image manipulation, the other utilizing the EAST algorithm, were explored in order to extract the text region on a room-identifying plaque. The manual technique relied on thresholding a grayscale version of the image, and applying image manipulation techniques such as dilation and edge detection to “blow out” the lighter text of the plaque image, creating a box from which the text could be cropped. This technique did not perform well, and made some assumptions about the design of the plaque (such as light text on a dark background, consistent lighting for all images, and that the image is a tight crop of the plaque) which required many manual adjustments through trial-and-error.

The EAST (Efficient and Accurate Scene Text detector) technique, after exploring some bounding box irregularities due to implementation, worked well on both the tightly-cropped plaque images and raw images of the hallway. It regularly identified regions of text, even where the text was illegible due to poor lighting or motion blur.

The results of both of these techniques were then “read” using the python extension for Google’s Tesseract text-extraction engine. Various trials were run on some examples of plaque crops in order to discover which arguments generated the highest fidelity. Images with blurry, joined, or rotated text did not perform as well as clearer images, however each sign was read at least once.

7 Future Work

The ideal next step for this project would be to apply the methods for discovering room identifying plaques, and to pair it with a volumetric mapping system like SLAM (Simultaneous Localization and Mapping), facilitating actual mapping of these spaces. If the SLAM can generate metric data about a space, and this pipeline can target room identifiers, then putting these two together will allow directions from one room to another to be generated. These directions could be tailored to the user, either by distance ("walk 20 feet down the hallway and door is on your right") or subjective, based on some user metadata about height or stride ("turn right and take 3 paces"). There are multiple libraries, such as those in Lin (2016) which pair with the open-source Robot Operating System (ROS) (Quigley et al., 2009), allowing for an autonomous hallway-roving robot to make a map of a space.

For the evaluation of the dataset creation, a more robust dataset of real-world images could be gathered, sampling other buildings with differently-designed plaques. These different forms could have associated custom-made synthetic datasets developed to try the multi-object detector method, or a composite image could be developed which would incorporate enough features of the different plaque designs to function well for all of them.

For object detection, another interesting system to try would be TensorFlow's object detector (Vladimirov, 2018), which might allow for one detector to run upon multiple different types of plaque. This method would also allow further evaluation of the different datasets explored in this project.

References

Bradski, G., & Kaehler, A. (2000). Opencv. *Dr. Dobb's journal of software tools*, *3*, 2.

Brownlee, J. (2013). *How to prepare for machine learning*. https://machinelearningmastery.com/how-to-prepare-data-for-machine-learning/

Dalal, N., & Triggs, B. (2005). Histograms of oriented gradients for human detection, In *2005 ieee computer society conference on computer vision and pattern recognition (cvpr'05)*. Ieee.

Du, S., Ibrahim, M., Shehata, M., & Badawy, W. (2012). Automatic license plate recognition (alpr): A state-of-the-art review. *IEEE Transactions on circuits and systems for video technology*, *23*(2), 311–325.

Famous people faces dataset. (n.d.). https://www.kaggle.com/caldodepollo/famous-people-faces

Google image search. (n.d.). https://www.google.com/imghp?hl=en

King, D. E. (2015). Max-margin object detection. *arXiv preprint arXiv:1502.00046*.

Kuznetsova, A., Rom, H., Alldrin, N., Uijlings, J., Krasin, I., Pont-Tuset, J., Kamali, S., Popov, S., Malloci, M., Kolesnikov, A., Et al. (2020). The open images dataset v4. *International Journal of Computer Vision*, *128*(7), 1956–1981.

Lin, T. T. (2016). *The list of vision-based slam / visual odometry open source projects, libraries, dataset, tools, and studies*. https://github.com/tzutalin/awesome-visual-slam

Mikołajczyk, A., & Grochowski, M. (2018). Data augmentation for improving deep learning in image classification problem, In *2018 international interdisciplinary phd workshop (iiphdw)*. IEEE.

Mitsa, T. (2019). *How do you know you have enough training data?* https://towardsdatascience.com/how-do-you-know-you-have-enough-training-data-ad9b1fd679ee

Noaa (national oceanic and atmospheric administration data. (n.d.). https://www.ncdc.noaa.gov/cdo-web/datasets

Quigley, M., Conley, K., Gerkey, B., Faust, J., Foote, T., Leibs, J., Wheeler, R., Ng, A. Y., Et al. (2009). Ros: An open-source robot operating system, In *Icra workshop on open source software*. Kobe, Japan.

Reddigari, M., & Vila, B. (2020). *Solved! how high to hang pictures*. https://www.bobvila.com/articles/how-high-to-hang-pictures/

Rosebrook, A. (2017). *How to create a deep learning dataset using google images*. https://www.pyimagesearch.com/2017/12/04/how-to-create-a-deep-learning-dataset-using-google-images/

Saha, S. (2018). A comprehensive guide to convolutional neural networks. 2018. *URL: http://towardsdatascience.com/a-comprehensive-guide-to-convolutional-neural-networks-the-eli5-way-3bd2b1164a53 (visited on 11/10/2019)(cited on page 169)*.

Sambasivarao, K. (2019). *Non-maximum suppression (nms) a technique to filter the predictions of object detectors*. https://towardsdatascience.com/non-maximum-suppression-nms-93ce178e177c

Smith, R. (2007). An overview of the tesseract ocr engine, In *Ninth international conference on document analysis and recognition (icdar 2007)*. IEEE.

Talari, S. (2017). *Crate your own object detector. machine intelligence for humans*. https://www.hackevolve.com/create-your-own-object-detector/

U.s. department of justice, 2010 ada standards for accessible design. (2010).

Van der Walt, S., Schönberger, J. L., Nunez-Iglesias, J., Boulogne, F., Warner, J. D., Yager, N., Gouillart, E., & Yu, T. (2014). Scikit-image: Image processing in python. *PeerJ*, *2*, e453.

Vladimirov, L. (2018). *Tensorflow 2 object detection api tutoria*. https://tensorflow-object-detection-api-tutorial.readthedocs.io/en/latest/

Zhou, X., Yao, C., Wen, H., Wang, Y., Zhou, S., He, W., & Liang, J. (2017). East: An efficient and accurate scene text detector, In *Proceedings of the ieee conference on computer vision and pattern recognition*.

8 Source Code

8.1 ShapeDetection.py

```python
#!/usr/bin/env python3
import CustomImage
import HandyTools as HT
import numpy
from PIL import Image, ImageTk
import cv2
import os
import tkinter as tk
import timeit
import string
from ImageMeta import ImageDetectionMetadata
import logging
ALL_CHARS = string.ascii_letters + string.digits

logging.basicConfig(format='[%(asctime)s] <%(funcName)s> : %(message)s', filename='wholerun.log', level=logging.INFO)
logger = logging.getLogger('wholerun')


def drawSingleContour(image, c, *, text=None, color=(0, 125, 255), to_draw=True):
    '''e-z handle for cv2 implementation of calulating and drawing shape contours'''
    peri = cv2.arcLength(c, True)
    approx = cv2.approxPolyDP(c, 0.04 * peri, True)
    _x, _y, _w, _h = cv2.boundingRect(c)

    M = cv2.moments(c)
    area = float(M['m00'])
    text = area if not text else text
    if area > 50 and to_draw:
        cv2.drawContours(image, [approx], -1, color, 4)
        cv2.putText(image, (f'{text}'), (_x + _w // 2, _y + _h // 2), cv2.FONT_HERSHEY_SIMPLEX, .75, color, 2)
    return area, (_w, _h), (_x, _y)


def catchWeirdShape(width, height):
    try:
        return not HT.betwixt(0.5, width / height, 2)
    except ZeroDivisionError:
```

```
        return True


def actualVsMBRArea(contour_area, minrec_area):
    '''return ratio of area of minimum bounding rectangle to contour's area
        idea is that min bounding rec should be close to contour area if it is a rectangle
    '''
    if contour_area == 0:
        return 0
    ratio = minrec_area / contour_area
    return abs(ratio)


def drawSingleMinRec(image, c, *, doop=None):
    '''draw a min bounding rectangle and return area'''
    minrec = cv2.minAreaRect(c)
    box1 = cv2.boxPoints(minrec)
    bl, tl, tr, br = box1
    height = abs(bl[1] - tl[1])
    width = abs(tl[0] - tr[0])
    weird_shape = catchWeirdShape(width, height)
    min_area = round((width * height), 2)
    box = numpy.int0(box1)
    mid = 0
    if min_area > 50 and not weird_shape:
        if doop:
            for count, item in enumerate(box):
                logger.info(f'#{count}: {item}\n')
                cv2.circle(image, (item[0], item[1]), 10, (mid, 255 - mid, 255), 3)
                mid += 55
            logger.info('~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~')
        cv2.drawContours(image, [box], 0, (100, 0, 255), 2)
        cv2.putText(image, (f'area: {min_area}'), (bl[0], bl[1]), cv2.FONT_HERSHEY_SIMPLEX,
                    .75, (125, 125, 255), 2)
    return min_area, (width, height), (bl, tl, tr, br)


def drawContours(image, contours):
    for c in contours:
        drawSingleContour(image, c)

```

```

def drawBoundingBoxes(image, contours):
    areas = []
    for c in contours:
        areas.append(drawSingleMinRec(image, c))
    return areas


def calibratePlaque(source_image):
    """DEPRECATED"""
    '''sets the area and shape to expect from room marking plaques
        what we need to find is a good size to judge the pother plaques by.
    '''
    # check what we're getting
    if isinstance(source_image, CustomImage.Image):
        image = source_image
    else:
        image = CustomImage.Image(source_image)
    # remove color from image
    gray = CustomImage.Image(image, copy=True)
    gray.gray()
    gray.image = cv2.medianBlur(gray.image, 7)
    # gray.thresh(thresh_num=100)
    contours = canny_edge_and_contours(gray)
    # lets show an image of the contours, they each have a name
    # and a radio button to choose the right one
    areas = {}
    window = tk.Tk()
    window.title("Please Choose Correct Contour")
    window.configure(background='grey')

    PIXEL = tk.PhotoImage(width=1, height=1)

    listbox = tk.Listbox(window)
    listbox.pack(side='right')
    # scrollbar = tk.Scrollbar(listbox)
    # scrollbar.pack(side='right', fill='y')
    chosen = tk.StringVar()
    chosen.trace('w', simpleCallBack)

    def showChoice():
        logger.info(chosen.get())

    def CloseWindow():
```

```python
        logger.info(f"close window!")
        if chosen.get():
            window.destroy()

    numbad = 0
    numgood = 0
    for idx, contour in enumerate(contours):
        # logger.info(f'idx: {idx}, lencont: {len(contour)}\n')
        try:
            label = ALL_CHARS[numgood]
        except Exception as e:
            logger.error(e)
            label = 'TILT'

        areas[idx] = {}
        areas[idx]['label'] = label
        areas[idx]['contour'] = contour
        areas[idx]['contour_area'], (areas[idx]['contour_w'], areas[idx]['contour_h']), (x, y) = drawSingleContour(image.image, contour)
        areas[idx]['minred_area'], mrwh, areas[idx]['bl_tl_tr_br'] = drawSingleMinRec(image.image, contour)
        areas[idx]['ratio'] = actualVsMBRArea(areas[idx]['contour_area'],  areas[idx]['minred_area'])

        if catchWeirdShape(areas[idx]['contour_w'],
                           areas[idx]['contour_h']) or catchWeirdShape(mrwh[0], mrwh[1]):
            areas[idx]['valid'] = False
            numbad += 1
        else:
            areas[idx]['valid'] = True
            drawSingleContour(image.image, areas[idx]['contour'], color=(255, 0, 100), text=str(label))
            if numgood % 10 == 0:
                radioholder = tk.Listbox(listbox)
                radioholder.pack(side='left')
            tk.Radiobutton(radioholder, text=label, padx=20, variable=chosen, command=showChoice, value=str(idx)).pack(side='top')
            numgood += 1

    img = Image.fromarray(image.image)
    img = ImageTk.PhotoImage(img)
    panel = tk.Label(window, image=img)
```

```python
    panel.pack(side='bottom', fill='both', expand='yes')
    window.update()
    tk.Button(window, text="CONFIRM SELECTION", image=PIXEL, command=CloseWindow, compound='c', width=(image.get_width())).pack(side='top')
    window.mainloop()

    logger.info(f"chosen item: {chosen.get()}")
    logger.debug(f"in the result:{areas[int(chosen.get())]}")
    logger.debug(f"just for shits: whole area dictionary: {areas}")
    return areas[int(chosen.get())]


def calibrate_run_with_plaque(source_image_location):
    '''sets the area and shape to expect from room marking plaques
        what we need to find is a good size to judge the other plaques by.
    '''
    # check what we're getting
    image = cv2.imread(source_image_location)
    # lets show an image of the contours, they each have a name
    # and a radio button to choose the right one
    gray = cv2.cvtColor(image, cv2.COLOR_BGR2GRAY)
    # 2) blur and contour
    median_blur = cv2.medianBlur(gray, 9)
    thresh = cv2.threshold(median_blur, 100, 255, cv2.THRESH_BINARY)[1]
    edged = cv2.Canny(thresh, 100, 255)
    contours = cv2.findContours(edged, cv2.RETR_EXTERNAL, cv2.CHAIN_APPROX_SIMPLE)[1]
    areas = {}
    window = tk.Tk()
    window.title("Please Choose Correct Contour")
    window.configure(background='grey')

    PIXEL = tk.PhotoImage(width=1, height=1)

    listbox = tk.Listbox(window)
    listbox.pack(side='right')
    # scrollbar = tk.Scrollbar(listbox)
    # scrollbar.pack(side='right', fill='y')
    chosen = tk.StringVar()
    chosen.trace('w', simpleCallBack)

```

```python
def showChoice():
    logger.info(chosen.get())

def CloseWindow():
    logger.info(f"close window!")
    if chosen.get():
        window.destroy()

numbad = 0
numgood = 0
for idx, contour in enumerate(contours):
    # logger.info(f'idx: {idx}, lencont: {len(contour)}\n')
    try:
        label = ALL_CHARS[numgood]
    except Exception as e:
        logger.error(e)
        label = 'TILT'

    areas[idx] = {}
    areas[idx]['label'] = label
    areas[idx]['contour'] = contour
    areas[idx]['contour_area'], (areas[idx]['contour_w'], areas[idx]['contour_h']), (x, y) = drawSingleContour(image, contour)
    areas[idx]['minred_area'], mrwh, areas[idx]['bl_tl_tr_br'] = drawSingleMinRec(image, contour)
    areas[idx]['ratio'] = actualVsMBRArea(areas[idx]['contour_area'],  areas[idx]['minred_area'])

    if catchWeirdShape(areas[idx]['contour_w'],
                       areas[idx]['contour_h']) or catchWeirdShape(mrwh[0], mrwh[1]):
        areas[idx]['valid'] = False
        numbad += 1
    else:
        areas[idx]['valid'] = True
        drawSingleContour(image, areas[idx]['contour'], color=(255, 0, 100), text=str(label))
        if numgood % 10 == 0:
            radioholder = tk.Listbox(listbox)
            radioholder.pack(side='left')
        tk.Radiobutton(radioholder, text=label, padx=20, variable=chosen, command=showChoice, value=str(idx)).pack(side='top')
        numgood += 1
```

```python

    img = Image.fromarray(image)
    img = ImageTk.PhotoImage(img)
    panel = tk.Label(window, image=img)
    panel.pack(side='bottom', fill='both', expand='yes')
    window.update()
    tk.Button(window, text="CONFIRM SELECTION", image=PIXEL, command=CloseWindow, compound='c', width=(image.shape[1])).pack(side='top')
    window.mainloop()

    logger.info(f"chosen item: {chosen.get()}")
    logger.debug(f"in the result:{areas[int(chosen.get())]}")
    logger.debug(f"just for shits: whole area dictionary: {areas}")
    return areas[int(chosen.get())]


def simpleCallBack(*args):
    logger.info(f'variable changed {args}')


def canny_edge_and_contours(image, *, threshold_1=50, threshold_2=250):
    # its edgin' time
    edged = cv2.Canny(image, threshold_1, threshold_2)
    # fill gaps
    kernel = cv2.getStructuringElement(cv2.MORPH_RECT, (5, 5))
    # closed = CIMAGE(cv2.morphologyEx(edged.image, cv2.MORPH_CLOSE, kernel))
    closed = cv2.morphologyEx(edged, cv2.MORPH_CLOSE, kernel)
    _, contours, _ = cv2.findContours(closed, cv2.RETR_EXTERNAL, cv2.CHAIN_APPROX_SIMPLE)
    return contours


def get_plaques_with_hog(source_image_location, *, hog, save_directory, _debug_mode=False, use_biggest_contour=False, _fileio=True):
    '''
    generates predictions with HOG. for each of these predictions, we crop it out and look for contours.
    those contours are then skewed to fit a rectagnel, and sent along with the data.
    '''
```

```
    # open file and load it up
    image = cv2.imread(source_image_location)
    # dirty_copy = image.copy()
    if image.size < 1:  # or dirty_copy.size < 1:
        # either it is a junk image, or the copy failed.
        logger.debug(f"image not valid: {source_image_location}")
        return []
    logger.debug(f"processing file {source_image_location}")
    source_directory, source_file_name = os.path.split(source_image_location)
    # set up payload
    list_of_plaque_meta_payloads = []
    rgb_image = cv2.cvtColor(image, cv2.COLOR_BGR2RGB)
    predictions = hog.predict(rgb_image)
    logger.info(f"number of predictions: {len(predictions)}")
    for pi, (x, y, xb, yb) in enumerate(predictions):
        # 1) for each prediction, grab the plaque image inside. this will likely be the largest contour.
        cropped_roi = image[y:yb, x:xb, :]
        # single dimension numpy array (junk)
        if cropped_roi.size < 1:
            continue
        gray = cv2.cvtColor(cropped_roi, cv2.COLOR_BGR2GRAY)
        # 2) blur and contour
        median_blur = cv2.medianBlur(gray, 9)
        thresh = cv2.threshold(median_blur, 100, 255, cv2.THRESH_BINARY)[1]
        edged = cv2.Canny(thresh, 100, 255)
        contours = cv2.findContours(edged, cv2.RETR_EXTERNAL, cv2.CHAIN_APPROX_SIMPLE)[1]
        # 3) get the biggest contour
        if use_biggest_contour:
            contour_areas = [cv2.moments(c)['m00'] for c in contours]
            if not contour_areas:
                logger.debug("empty contour areas for biggest contour")
                continue
            logger.debug(f"contour areas: {contour_areas}")
            # could this just use a lambda to get the biggest area without splitting it out?
            location_of_biggest = contour_areas.index(max(contour_areas))
            big_countour = contours[location_of_biggest]
            contours = [big_countour]
```

```
        for ci, c in enumerate(contours):
            approx = cv2.approxPolyDP(c, 0.04 * cv2.arcLength(c, True), True)
            rect_points = numpy.array([x[0] for x in approx])
            logger.debug(f"creating payload for file {source_image_location}, with contour number {ci}")
            payload = ImageDetectionMetadata()
            # take whatever the image may be, and make it a rectangle
            payload.image = HT.four_point_transform(cropped_roi, rect_points)
            payload.contour_area = float(cv2.moments(c)['m00'])
            payload.reference_area = None
            payload.source_image_location = source_image_location
            if _fileio:
                payload.plaque_image_location = os.path.join(save_directory, f"{pi}_{ci}" + source_file_name)
                cv2.imwrite(payload.plaque_image_location, payload.image)
            list_of_plaque_meta_payloads.append(payload)

    if not list_of_plaque_meta_payloads:
        payload = ImageDetectionMetadata()
        payload.source_image_location = source_image_location
        list_of_plaque_meta_payloads.append(payload)
    return list_of_plaque_meta_payloads


def get_plaques_matching_ratio(source_image_location, *, save_directory, good_area, _debug_mode=False, _fileio=False, cutoff_ratio=.30):
    '''
    source_image: CustomImage object
    good_ratio: best ratio for a plaque
    good_area: approximation of a good size for a plaque
    '''
    # open file and load it up
    image = CustomImage.Image(source_image_location)
    source_directory, source_file_name = os.path.split(source_image_location)
    # set up payload
    list_of_plaque_meta_payloads = []

    clean_copy = CustomImage.Image(image)
    dirty_copy = CustomImage.Image(image)
```

```python
    gray = CustomImage.Image(image)
    gray.gray()

    # blur and threshold
    median_blur = cv2.medianBlur(gray.image, 9)
    blur_contours = canny_edge_and_contours(median_blur)
    debug_copy = dirty_copy.image.copy()
    for i, c in enumerate(blur_contours):
        # 0) get contour information
        peri = cv2.arcLength(c, True)
        approx = cv2.approxPolyDP(c, 0.04 * peri, True)
        M = cv2.moments(c)
        contour_area = float(M['m00'])
        # 1) get minimum bounding rectangle
        min_rec_x, min_rec_y, min_rec_w, min_rec_h = cv2.boundingRect(c)
        # 2) compare that area with good area/ratio supplied to function
        ratio_good_to_maybe = min(good_area / contour_area, contour_area / good_area) if good_area != 0 and contour_area != 0 else 0
        # 3) if it is close enough, skew and crop to get proper h/w
        if ratio_good_to_maybe >= cutoff_ratio:

            if _debug_mode:
                cv2.rectangle(debug_copy, (min_rec_x, min_rec_y), (min_rec_x + min_rec_w, min_rec_y + min_rec_h), (10, 0, 225), 2)
                cv2.putText(debug_copy, 'plaque', (min_rec_x + 5, min_rec_y - 5), cv2.FONT_HERSHEY_SIMPLEX, 1.0, (128, 255, 0), 2)

            payload = ImageDetectionMetadata()
            rect_points = numpy.array([x[0] for x in approx])
            payload.image = HT.four_point_transform(clean_copy.image, rect_points)
            payload.contour_area = contour_area
            payload.reference_area = good_area
            payload.source_image_location = source_image_location
            if _fileio:
                payload.plaque_image_location = os.path.join(save_directory, f"{i}_" + source_file_name)
                cv2.imwrite(payload.plaque_image_location, payload.image)
```

```
            list_of_plaque_meta_payloads.append(payload)

    if _debug_mode:
        cv2.imshow(f"points for area {contour_area}", debug_copy)
        cv2.waitKey()
        cv2.destroyWindow(f"points for area {contour_area}")

    if not list_of_plaque_meta_payloads:
        payload = ImageDetectionMetadata()
        payload.source_image_location = source_image_location
        list_of_plaque_meta_payloads.append(payload)
    return list_of_plaque_meta_payloads


def get_plaques_matching_ratio_rigamarole(source_image_location, *, good_area, cutoff_ratio=.30):
    # open file and load it up
    image = cv2.imread(source_image_location)
    source_directory, source_file_name = os.path.split(source_image_location)
    # set up payload
    list_of_plaque_meta_payloads = []
    marked_copy = image.copy()
    gray = cv2.cvtColor(image, cv2.COLOR_BGR2GRAY)
    # 2) blur and contour
    median_blur = cv2.medianBlur(gray, 9)
    thresh = cv2.threshold(median_blur, 100, 255, cv2.THRESH_BINARY)[1]
    edged = cv2.Canny(thresh, 100, 255)
    contours = cv2.findContours(edged, cv2.RETR_EXTERNAL, cv2.CHAIN_APPROX_SIMPLE)[1]
    colors = [
        (255, 125, 0),
        (255,100,255),
        (125,100,255),
        (0,255,0),
        (125,255,0),
        (255,255,0),
    ]
    for i, c in enumerate(contours):
        # 0) get contour information
        peri = cv2.arcLength(c, True)
        approx = cv2.approxPolyDP(c, 0.04 * peri, True)
        M = cv2.moments(c)
```

```python
        contour_area = float(M['m00'])
        # 1) get minimum bounding rectangle
        min_rec_x, min_rec_y, min_rec_w, min_rec_h = cv2.boundingRect(c)
        # 2) compare that area with good area/ratio supplied to function
        ratio_good_to_maybe = min(good_area / contour_area, contour_area / good_area) if good_area != 0 and contour_area != 0 else 0
        # 3) if it is close enough, skew and crop to get proper h/w
        rect_points = numpy.array([x[0] for x in approx])
        (tl, tr, br, bl) = HT.order_points(rect_points)
        polypts = numpy.array([
            [bl[0], bl[1]], [tl[0], tl[1]], [tr[0], tr[1]], [br[0], br[1]],
        ], numpy.int32).reshape((-1,1,2))
        # draw a thin pink contour
        cv2.polylines(marked_copy, [polypts], True, (255,100,255), 1)
        if ratio_good_to_maybe >= cutoff_ratio:
            rect_points = numpy.array([x[0] for x in approx])
            (tl, tr, br, bl) = HT.order_points(rect_points)
            polypts = numpy.array([
                [bl[0], bl[1]], [tl[0], tl[1]], [tr[0], tr[1]], [br[0], br[1]],
            ], numpy.int32).reshape((-1,1,2))
            color = colors.pop()
            cv2.polylines(marked_copy, [polypts], True, color, 3)
            colors.insert(0, color)

    HT.showKill(marked_copy, waitkey=6000)
    cv2.imwrite(os.path.join('/home/johnny/Documents/plaque_only_testing/', source_file_name), marked_copy)


def get_plaques_rigamarole(source_image_location, *, hog):
    '''
    generates predictions with HOG. for each of these predictions, we crop it out and look for contours.
    those contours are then skewed to fit a rectagnel, and sent along with the data.
    '''
    # open file and load it up
    image = cv2.imread(source_image_location)
```

```python

    if image.size < 1:
        # either it is a junk image, or the copy failed.
        logger.debug(f"image not valid: {source_image_location}")
        return []
    logger.debug(f"processing file {source_image_location}")
    source_directory, source_file_name = os.path.split(
    source_image_location)
    rgb_image = cv2.cvtColor(image, cv2.COLOR_BGR2RGB)
    hog_predictions = hog.predict(rgb_image)
    junk_roi = 0
    tval = 100
    marked_copy = image.copy()
    colors = [
        (255, 125, 0),
        (255,100,255),
        (125,100,255),
        (0,255,0),
        (125,255,0),
        (255,255,0),
    ]
    for pi, (x, y, xb, yb) in enumerate(hog_predictions):
        # 1) for each prediction, grab the plaque image inside
        color = colors.pop()
        cv2.rectangle(marked_copy, (x, y), (xb, yb), color, 3)
        colors.insert(0, color)
        cropped_roi = image[y:yb, x:xb, :]
        # single dimension numpy array (junk)
        if cropped_roi.size < 1:
            junk_roi += 1
            continue
        gray = cv2.cvtColor(cropped_roi, cv2.COLOR_BGR2GRAY)
        # 2) blur and contour
        median_blur = cv2.medianBlur(gray, 9)
        thresh = cv2.threshold(median_blur, tval, 255, cv2.
    THRESH_BINARY)[1]
        edged = cv2.Canny(thresh, 100, 255)
        contours = cv2.findContours(edged, cv2.RETR_EXTERNAL, cv2.
    CHAIN_APPROX_SIMPLE)[1]
        contour_areas = [cv2.moments(c)['m00'] for c in contours]
        if not contour_areas:
            continue
        for ci, c in enumerate(contours):
            approx = cv2.approxPolyDP(c, 0.04 * cv2.arcLength(c,
    True), True)
```

```
            rect_points = numpy.array([x[0] for x in approx])
            (tl, tr, br, bl) = HT.order_points(rect_points)

            polypts = numpy.array([
                [bl[0] + x, bl[1] + y],
                [tl[0] + x, tl[1] + y],
                [tr[0] + x, tr[1] + y],
                [br[0] + x, br[1] + y],
            ], numpy.int32).reshape((-1,1,2))
            color = colors.pop()
            cv2.polylines(marked_copy, [polypts], True, color, 1)
            colors.insert(0, color)
            cv2.polylines(marked_copy, [polypts], True, (255,100,255), 2)

    HT.showKill(marked_copy, waitkey=6000)
    cv2.imwrite(os.path.join('/home/johnny/Documents/plaque_only_testing/roi_heuristic_plaques', source_file_name), marked_copy)


def area_plaque_finder(source_image_location, *, good_area, cutoff_ratio=.30):
    # open file and load it up
    start = timeit.default_timer()
    image = cv2.imread(source_image_location)
    source_directory, source_file_name = os.path.split(source_image_location)
    # set up payload
    num_found = 0
    marked_copy = image.copy()
    gray = cv2.cvtColor(image, cv2.COLOR_BGR2GRAY)
    # 2) blur and contour
    median_blur = cv2.medianBlur(gray, 9)
    thresh = cv2.threshold(median_blur, 100, 255, cv2.THRESH_BINARY)[1]
    edged = cv2.Canny(thresh, 100, 255)
    contours = cv2.findContours(edged, cv2.RETR_EXTERNAL, cv2.CHAIN_APPROX_SIMPLE)[1]
    colors = [
        (255, 125, 0),
        (255,100,255),
        (125,100,255),
        (0,255,0),
        (125,255,0),
```

```
        (255,255,0),
    ]
    for i, c in enumerate(contours):
        # 0) get contour information
        peri = cv2.arcLength(c, True)
        approx = cv2.approxPolyDP(c, 0.04 * peri, True)
        M = cv2.moments(c)
        contour_area = float(M['m00'])
        ratio_good_to_maybe = min(good_area / contour_area, contour_area / good_area) if good_area != 0 and contour_area != 0 else 0
        # cv2.polylines(marked_copy, [polypts], True, (255,100,255), 1)
        if ratio_good_to_maybe >= cutoff_ratio:
            num_found += 1
            rect_points = numpy.array([x[0] for x in approx])
            (tl, tr, br, bl) = HT.order_points(rect_points)
            polypts = numpy.array([
                [bl[0], bl[1]], [tl[0], tl[1]], [tr[0], tr[1]], [br[0], br[1]],
            ], numpy.int32).reshape((-1,1,2))
            color = colors.pop()
            cv2.polylines(marked_copy, [polypts], True, color, 3)
            colors.insert(0, color)

    run_data = {'file_name': source_file_name, 'found_something': num_found, 'time': timeit.default_timer() - start}
    if num_found > 0:
        cv2.imwrite(os.path.join('/home/johnny/Documents/plaque_only_testing/area_found', source_file_name), marked_copy)
    return run_data


def hog_plaque_finder(source_image_location, *, hog):
    '''
    generates predictions with HOG. for each of these predictions, we crop it out and look for contours.
    those contours are then skewed to fit a rectagnel, and sent along with the data.
    '''
    start = timeit.default_timer()
    # open file and load it up
    image = cv2.imread(source_image_location)
```

```python
    source_directory, source_file_name = os.path.split(source_image_location)
    if image.size < 1:
        # either it is a junk image, or the copy failed.
        logger.debug(f"image not valid: {source_image_location}")
        return {'file_name': source_file_name, 'found_something': False, 'time': None}

    rgb_image = cv2.cvtColor(image, cv2.COLOR_BGR2RGB)
    hog_predictions = hog.predict(rgb_image)
    marked_copy = image.copy()
    colors = [
        (255, 125, 0),
        (255,100,255),
        (125,100,255),
        (0,255,0),
        (125,255,0),
        (255,255,0),
    ]
    for pi, (x, y, xb, yb) in enumerate(hog_predictions):
        # 1) for each prediction, grab the plaque image inside
        color = colors.pop()
        cv2.rectangle(marked_copy, (x, y), (xb, yb), color, 3)
        colors.insert(0, color)

    run_data = {'file_name': source_file_name, 'found_something': len(hog_predictions), 'time': timeit.default_timer() - start}
    cv2.imwrite(os.path.join('/home/johnny/Documents/plaque_only_testing/specific_made_up_detector_found', source_file_name), marked_copy)

    return run_data


def area_plaque_lean(source_image_location, *, good_area, cutoff_ratio=.30):
    # open file and load it up
    image = cv2.imread(source_image_location)
    source_directory, source_file_name = os.path.split(source_image_location)
    # set up payload

    marked_copy = image.copy()
    gray = cv2.cvtColor(image, cv2.COLOR_BGR2GRAY)
    # 2) blur and contour
```

```
    median_blur = cv2.medianBlur(gray, 9)
    thresh = cv2.threshold(median_blur, 100, 255, cv2.THRESH_BINARY)[1]
    edged = cv2.Canny(thresh, 100, 255)
    contours = cv2.findContours(edged, cv2.RETR_EXTERNAL, cv2.CHAIN_APPROX_SIMPLE)[1]
    for i, c in enumerate(contours):
        # 0) get contour information
        peri = cv2.arcLength(c, True)
        approx = cv2.approxPolyDP(c, 0.04 * peri, True)
        M = cv2.moments(c)
        contour_area = float(M['m00'])
        # 2) compare that area with good area/ratio supplied to function
        ratio_good_to_maybe = min(good_area / contour_area, contour_area / good_area) if good_area != 0 and contour_area != 0 else 0
        # 3) if it is close enough, skew and crop to get proper h/w

        if ratio_good_to_maybe >= cutoff_ratio:
            rect_points = numpy.array([x[0] for x in approx])
            HT.four_point_transform(image, rect_points)


def roi_plaque_lean(source_image_location, *, hog):
    # open file and load it up
    image = cv2.imread(source_image_location)

    if image.size < 1:
        # either it is a junk image, or the copy failed.
        logger.debug(f"image not valid: {source_image_location}")
        return []
    logger.debug(f"processing file {source_image_location}")

    rgb_image = cv2.cvtColor(image, cv2.COLOR_BGR2RGB)
    hog_predictions = hog.predict(rgb_image)

    tval = 100
    for pi, (x, y, xb, yb) in enumerate(hog_predictions):
            continue
        gray = cv2.cvtColor(cropped_roi, cv2.COLOR_BGR2GRAY)
        # 2) blur and contour
        median_blur = cv2.medianBlur(gray, 9)
```

```python
        thresh = cv2.threshold(median_blur, tval, 255, cv2.THRESH_BINARY)[1]
        edged = cv2.Canny(thresh, 100, 255)
        contours = cv2.findContours(edged, cv2.RETR_EXTERNAL, cv2.CHAIN_APPROX_SIMPLE)[1]
        contour_areas = [cv2.moments(c)['m00'] for c in contours]
        if not contour_areas:
            continue
        for ci, c in enumerate(contours):
            approx = cv2.approxPolyDP(c, 0.04 * cv2.arcLength(c, True), True)


def show_multiple_color_images(imlist, num_imgs=0, rows=0, cols=0, name='sample'):
    num_to_show = len(imlist)
    if num_to_show < 1:
        return False
    horizs = []
    # blank =
    blank = numpy.zeros(imlist[0][1].shape, dtype=numpy.uint8)
    for idx in range(0, num_to_show, cols):
        hs = []
        for x in range(cols):  #imlist[idx:idx + cols]:
            try:
                hs.append(imlist[idx+x][1])
            except Exception:
            # if x:
            #     hs.append(x[1])
            # else:
                hs.append(blank)
        hs = numpy.hstack(hs)
        # HT.showKill(hs, waitkey=6000)
        horizs.append(hs)
    for idx in range(0, len(horizs), rows):
        vs = numpy.vstack(horizs[idx:idx + rows])
        cv2.imwrite(f'/home/johnny/Documents/plaque_only_testing/{name}.png', vs)
        # HT.showKill(vs, waitkey=6000)
```

8.2 CustomImage.py

```
#!/usr/bin/env python3
# Custom image class wrapper to simplify image classification and parsing/ playin' around

import os
import math
import cv2
import random
import numpy as np
import platform
import CustomErrors as cerr
from skimage import exposure

class Image(object):
    '''Base class for custom images. Trying hand at pythonic polymorphism.
        can initalize and save image.
    '''
    def __init__(self, image, *, path=None, fileName=None,
                 extension=None, copy=False, seed=42, color=None, percentage=None):
        '''initializer for base image class.
        Args:
            image: cv2 image or np array
            path: path to save to (default is pwd)
            fname: filename (default is temp)
            extension: filetype (default is .png)
            copy: used as token for deep copy constr
            color: let us know if we are using color or not
            percentage: for scaling, in terms of percentage of one
        '''
        if isinstance(image, Image) or copy:
            self.image = np.copy(image.image)
        elif isinstance(image, str):
            # TODO: fix path and filename stuff
            self.file_name = image
            self.image = cv2.imread(self.file_name)
        else:
            self.image = image

        self.path = '' if not path else path
        self.file_name = 'temp' if not fileName else fileName
        self.extension = 'png' # if not extension else extension
```

```
        self.image_path = ''.join(self.path + self.file_name + '.' + self.extension)
        self.seed = seed
        self.percentage = percentage if percentage else 1

        self.shape = self.image.shape
        self.height = self.shape[0]
        self.width = self.shape[1]
        self.dimensions = 0 if len(self.shape) < 3 else self.shape[2]
        self.possible_x = None
        self.possible_y = None
        if self.dimensions is not 3:
            self.color = False
        else:
            self.color = True

    def copy(self):
        '''returns a Custom Image object identical to this one'''
        return Image(self.image)

    def get_size(self):
        '''returns size (height, width, dimensions) of the image'''
        return (self.height, self.width, self.dimensions)

    def get_width(self):
        return self.width

    def get_height(self):
        return self.height

    def get_shape(self):
        return self.height, self.width, None

    def get_dimensions(self):
        return self.dimensions

    def resize(self, *, percentage=None, vertical=None, horizontal=None):
        ''' resize image.
        Args:
            percentage: what percent size the image should be from the original
            vertical: desired vertical. horizontal will be scaled.
```

```
            horizontal: same but vis-versa
        '''
        h, w = self.image.shape[:2]

        if vertical is not None and horizontal is None:
            factor = vertical / h
        elif vertical is not None and horizontal is not None:
            self.image = cv2.resize(self.image, (vertical, horizontal), interpolation=cv2.INTER_AREA)
            return
        elif horizontal is not None and vertical is None:
            factor = horizontal / w
        elif percentage is not None:
            factor = percentage
        else:
            factor = self.percentage

        self.image = cv2.resize(self.image,None,fx=factor, fy=factor)

    def show(self, *, title=None, pause=None):
        '''
        Show the image in a window. will wait for kill signal.
            checks to see if running windows or linux to fix a bug fixed by
            the getwindowproperty, where closing the image with the 'x' button
            would cause ipython to block, and the window was not there to receive a
            weaitkey signal.
            did not occur on windows, and the window property does not work the same way
            (i think) on windows system.

            pause allows a window to automaatically close after a length of time
            pause is limited to minimum 1000 msec as lower calues can cause weird behavior
                plus it should be safe!
        '''
        if pause is not None:
            pause = 1000 if pause < 1000 else pause
            cv2.imshow(title, self.image)
            cv2.waitKey(pause)
            cv2.destroyWindow(title)
```

```python
        else:
            title = title if title else "image"
            status = 1
            if platform.system() == 'Windows':
                # print("[INFO] using Windows")
                cv2.imshow(title, self.image)
                cv2.waitKey()
                cv2.destroyWindow(title)
            else:
                # print(f"[INFO] using system {platform.system}")
                # assume we're running linux
                try:
                    cv2.imshow(title, self.image)
                    while status > 0:
                        ks=cv2.waitKey(1000)
                        try:
                            # this does not work for windows like it does for linux.
                            # TODO: check system first
                            status = cv2.getWindowProperty(title, cv2.WND_PROP_VISIBLE)
                        except Exception as e:
                            status = -1
                            break
                        if ks > 0:
                            break
                    cv2.destroyWindow(title)
                except Exception as e:
                    print("error occured: {}",e)
                    raise

    def rectangle(self, top_left, bottom_right, value = 120, thickness = 3):
        '''Draw a rectangle at coordinates
        Args:
            p1, p2: edges of rectangle
            value: greyscale value
            thickness: how thick a line. -1 for filled.
        '''
        self.image = cv2.rectangle(self.image, top_left, bottom_right, value, thickness)

    def line(self, pt1,pt2, value = 120, thickness = 3):
        '''Draw a line at coordinates
        Args:
```

```
            p1, p2: points of line
            value: greyscale value
            thickness: how thick a line. -1 for filled.
        '''
        self.image = cv2.line(self.image,pt1,pt2,value,thickness)

    def thresh(self,*, type=None, thresh_num=170):
        '''simpler handle for cv2 threshold.'''
        if type == 'OTSU':
            ret2,img = cv2.threshold(self.image,0,255,cv2.THRESH_BINARY+cv2.THRESH_OTSU)
        elif not type:
            ret, img = cv2.threshold(self.image, thresh_num, 255, cv2.THRESH_BINARY)
        self.image = img

    def addColor(self):
        '''make gray image BGR compatible'''
        self.image = cv2.cvtColor(self.image, cv2.COLOR_GRAY2BGR)
        self.color = True

    def gray(self):
        '''make the image grayscale. pretty straighforward! overwrites original.'''
        self.image = cv2.cvtColor(self.image, cv2.COLOR_BGR2GRAY)
        self.image = exposure.rescale_intensity(self.image, out_range=(0,255))
        self.color = False

    def save(self, *, file_path=None, file_name=None):
        '''Saves image to file.
        TODO: allow for changing file name'''
        if file_name:
            pass
        if file_path:
            self.image_path = file_path
        cv2.imwrite(self.image_path, self.image)

    def blur(self, *, kernel=(3, 3), blur_type='GAUSS'):
        '''function to encapsulate blurring activity.
        Args:
            kernel: size of kernel to apply blurring
            blur_type: gaussian or average or median,
                keywords 'GAUSS', 'AVG', 'MEDIAN'
        '''
```

```
        if blur_type is 'GAUSS':
            self.image = cv2.GaussianBlur(self.image, kernel, 0)
        elif blur_type is 'AVG':
            self.image = cv2.blur(self.image, kernel)
        elif blur_type is 'MEDIAN':
            self.image = cv2.medianBlur(self.image, kernel[0])
        else:
            print("{} is not implemented. Blurring with Gauss.".format(blur_type))
            self.image = cv2.GaussianBlur(self.image, kernel, 0)

    def isolate(self, xRange, yRange):
        '''isolate section of an image.
        Args:
            xRange: tuple. grabs horiz bounds,
                i.e. 100:250
            yRange: tuple. grabs vert bounds,
                i.e. 100:250
        '''
        if len(self.image.shape) is 3:
            return self.image[xRange[0]:xRange[1], yRange[0]:yRange[1], :]
        else:
            return self.image[xRange[0]:xRange[1], yRange[0]:yRange[1]]

    @classmethod
    def add_many(cls, image_list):
        '''creates a big image from many and shows it.
            not smart so dont make an image that is too big!
            also not smart and can only take even swquares!
        '''
        numImages = len(image_list)
        x = math.ceil(math.sqrt(numImages))
        if x == math.sqrt(numImages):
            y = int(x)
            # add images together by y
            horizStrips = []
            for i in range(0, numImages+1, x):
                if x <= numImages:
                    horizStrips.append(cv2.hconcat([image.image for image in image_list[i:x]]))
                x = x+y
            print("\n\nDEBUG: length of horizStrips: {}\n\n".format(len(horizStrips)))
```

```
            full = cv2.vconcat([image for image in horizStrips])
            fullImage = cls(full)
            fullImage.resize(horizontal= 2048)
            return fullImage
        else:
            raise cerr.DumbProgramError("Can only accept even squares!")


    @classmethod
    def open(cls, filename):
        '''simple implementation to open a file and return an image object.
        simplementation.
        '''
        try:
            img = cv2.imread(filename)
        except Exception as e:
            print(f"{e}. Filename: {filename}\n")
            raise

        image = cls(img)
        return image


class GeneratedImage(Image):
    '''images that are created. Inherits init from parent Image'''
    def __init__(self, image, *, path=None, fileName=None,
                extension=None, copy=False, seed=42, color=None, percentage=None):
        super().__init__(image, path=path, fileName=fileName, extension=extension, copy=copy, seed=seed, color=color, percentage=percentage)
        self.possible_x = [n for n in range(self.width)]
        self.possible_y = [n for n in range(self.height)]

    def copy(self):
        '''returns a Custom Image object identical to this one'''
        return GeneratedImage(self.image)

    def rotate(self, degree, *, center=None):
        '''rotate an image'''
        # img = cv2.imread('messi5.jpg',0)
        # rows,cols = img.shape
```

```
        matrix = cv2.getRotationMatrix2D((self.height/2,self.width/2),degree,1)
        self.image = cv2.warpAffine(self.image, matrix, (self.width,self.height), borderMode=cv2.BORDER_REPLICATE)

    def skew(self, four_points):
        '''apply perspective tranformation to image
            takes either simple list or npfloats
            tl, tr, br, bl order
        '''
        if not isinstance(four_points, np.ndarray):
            four_points = np.float32(four_points)

        dest_points = np.float32([[0,0],[self.height,0],[0, self.width], [self.height, self.width]])
        matrix = cv2.getPerspectiveTransform(four_points, dest_points)
        self.image = cv2.warpPerspective(self.image, matrix, (self.height, self.width))

    def salt_and_pepper(self, seasoning=0.007, seed=None):
        '''creates a sprinkling of salt and pepper on an image.
        Args:
            seasoning: how much salt and pepper to add
            seed: random seed
        '''
        if seed is None:
            seed = self.seed
        np.random.seed(seed)
        shapeinfo = self.image.shape
        row=shapeinfo[0]
        col=shapeinfo[1]
        s_vs_p = 0.5
        # out = np.copy(image) # don't need to make a copy, image itself is modified
        # Salt mode
        num_salt = np.ceil(seasoning * self.image.size * s_vs_p)
        coords = [np.random.randint(0, i - 1, int(num_salt))
                    for i in self.image.shape]
        self.image[coords] = 255
        # Pepper mode
        num_pepper = np.ceil(seasoning* self.image.size * (1. - s_vs_p))
        coords = [np.random.randint(0, i - 1, int(num_pepper))
                    for i in self.image.shape]
```

```
        self.image[coords] = 0

    def random_lines(self, *, seed=None, num_lines=2):
        '''add random lines
        Args:
            seed: seed for randomint
            num_lines: how many lines to draw
        '''
        if seed is None:
            seed = self.seed
        random.seed(seed)


        for n in range(num_lines):
            top_left = (random.randint(0, self.image.shape[0]),
                        random.randint(0, self.image.shape[1]))
            bottom_right = (random.randint(0, self.image.shape[0]),
                        random.randint(0, self.image.shape[1]))
            value = random.randint(0,255)
            if self.color:
                val2 = random.randint(0,255)
                val3 = random.randint(0,255)
                value = (value, val2, val3)
            thickness = random.randint(1,10)
            self.line(top_left, bottom_right, value, thickness)

    def random_rectangles(self, *, seed=None, num_recs=2, zona_peligrosa_x=None, zona_peligrosa_y=None, rec_w=8, rec_h=12):
        '''add random rectangles
        Args:
            seed: seed for randomint
            num_lines: how many lines to draw
            zona peligrosa: areas on x or y that cannot be drawn upon, a set
            assuming that rec_h and rec_w will only be used if the clear space parameters are included (11-13)
        '''
        if seed is None:
            seed = self.seed
        random.seed(seed)
        # must account for width of rectangle!
        ok_x = [n for n in self.possible_x if (n + rec_w) not in zona_peligrosa_x] if zona_peligrosa_x else self.possible_x
```

```
        ok_y = [n for n in self.possible_y if n not in zona_peligrosa_y] if zona_peligrosa_y else self.possible_y

        for n in range(num_recs):
            # each of these is a rectangle dummy!
            top_left = (random.choice(ok_x), random.choice(ok_y))
            bottom_right = (top_left[0] + rec_w, top_left[1] + rec_h)

            value = random.randint(180, 255)
            if self.color:
                val2 = random.randint(180, 255)
                val3 = random.randint(180, 255)
                value = (value, val2, val3)
            thickness = random.randint(-10, 10)
            self.rectangle(top_left, bottom_right, value, thickness)
```

8.3 DataGenerator.py

```
''' This generates test data for the image capturing system.
    600 by 600 images are created and some have the correct numbered square in the image somewhere.
    others will have other shapes or noise.
'''
import random
import cv2
import numpy as np
import string
import math
import ADA
import HandyTools as HAT

PLAQUE_SHAPES = {'circle': 0, 'rectangle': 1, 'ellipse': 2, 'triangle': 3}

class ImageGenerator(object):

   def __init__(self, IMAGECLASS, resolution, *, size=(600,600,3), bgValue=(237,245,247), randSeed=42,
               plaqueValue=(42,5,102), plaqueSize=None, plaqueShape='rectangle',
               fontFace=cv2.FONT_HERSHEY_SIMPLEX):
       '''initializer for generator class that produces images.
       Args:
           IMAGECLASS: the image class for objects being created.
           resolution: how many pixels per inch (conceptually)
           size: image size. 600X600 BGR by default
           bgValue: desired backgound value for image creation.
           randSeed: seed for random lines drawing.
           plaqueValue: desired color for room plaque
           plaqueSize: desired plaque size. default is a little more than 10% of image. will convert to an int
           fontFace: desired font for plaques.
       '''
       # set internal image class
       self._imgclass = IMAGECLASS
       self.res = resolution
       # set up initial size

       self._size = size

       # set background value. default is beige.
```

```
        self._bgv = bgValue
        #set random seed
        self._rands = randSeed
        random.seed(self._rands)
        #set plaque grayscale value. default is maroon.
        self._pqv = plaqueValue
        #set plaque size
        if plaqueSize is None:
            self._pqs = int(math.sqrt((self._size[0]*self._size[1])*0.01))
        else:
            self._pqs = int(plaqueSize)
        # will plaque be rectangle, ellipse, or other shape?
        self._plaque_shape = PLAQUE_SHAPES[plaqueShape]
        #set font typeface
        self._font = fontFace
        #set font color, high contrast is key
        if len(self._pqv) is 3:
            self._fontv = tuple( HAT.hiLow255(n) for n in self._pqv)
        else:
            self._fontv = HAT.hiLow255(self._pqv)
        #number of chars on plaque
        self._strlen = 3
        # are we doing color for these?
        self._color = len(self._size) is 3

        # print("\nDEBUG:\nBGV:{}\nPQV:{}\nPQS:{}\nFONTV:{}\nCOLOR:{}\nEND ~~"
        #         .format(self._bgv,self._pqv,self._pqs,self._fontv,self._color))

    def __str__(self):
        pass

    def create_canvas(self):
        '''create a base image object'''
        image = self._imgclass(np.full((self._size), self._bgv,np.uint8),
                               color=self._color, seed=random.randint(0, 255))
        return image

    def make_hallway(self, *, res=None, txt='358B', papers=None, posters=None):
```

```
        '''Make a 'hallway' with a sign and a door. Keep the sign coordinates.
        this hallway will then be chopped up and skewed to create a better dataset.
        should be long.
            according to ADA guidelines, the baseline of raised signage should
            be between 48 and 60 inches from floor. treating 10 pixels as inches.
        Args:
            res: ratio of pixels to inches.
        Assumptions:
            Door opening is 80" by 32", 3" trim around
            Plaque height is 60" at top left corner
            plaque is 2" from door, and 7" wide/ high
            ceiling is 10'

        REturns:
            image: hallway image object
            plaqueTL: plaque location top left coords
            plaqueBR: plaque location bottom right coords
        '''
        if res is None:
            res = self.res
        TRIM = 3
        DR_HT = (ADA.DOOR_HT +TRIM) * res
        DR_WD = (ADA.DOOR_WD+2*TRIM) * res

        PQ_DIM = 8*res
        PQ_MGN = .5*res
        PQ_2_DR = 2*res
        HL_CEIL = ADA.CEIL_HT * res
        PQ_WALL_HT = HL_CEIL-(ADA.PQ_HT * res)
        HL_WD = 2*HL_CEIL
        FONT = cv2.FONT_HERSHEY_DUPLEX
        FONT_BS = 22
        # create canvas for our beautiful painting
        hallway = self._imgclass(np.full((HL_CEIL,HL_WD,3),(250,250,250),
                                        dtype=np.uint8),color=self._color,
                                        seed=random.randint(0,255))

         # now add some rectangles as papers and billboards in an area where there is no plaque or door
        paper_size_h = res*11
```

```
    paper_size_w = res*8
    poster_size_h = random.randint(res*12, res* 36)
    poster_size_w = random.randint(res*12, res* 36)
    # this is the zone where we should not be drawing anything
    zona_peligrosa_x = []
    # zona_peligrosa_y = [n for n in range(min(Dy1, Py1), max(Dy2, Py2))]
    # additionally, add restrictions for height (so things are only where people would see them)
    # assume most things hang between 80" and 36"
    vis_top = HL_CEIL-res*80
    vis_bottom = HL_CEIL-res*36
    # not sure if it would be faster to build bigger list and then slice but my guess is the list
    # comprehension is pretty integral so going with that
    # need to include the size of the paper or poster in the danger zone, thus the subtraction of poster_size
    zona_peligrosa_y = [n for n in range(HL_CEIL) if n < vis_top or n > vis_bottom-poster_size_h]
    # print(len(zona_peligrosa_y))
    # now use the random square placement to drop a random number of papers, posters on the clear space
    if posters:
        hallway.random_rectangles(seed=random.randint(0,1000), num_recs=posters,
                                                    zona_peligrosa_x=zona_peligrosa_x,
                                                    zona_peligrosa_y=zona_peligrosa_y,
                                                    rec_w=poster_size_w,
                                                    rec_h=poster_size_h)
    if papers:
        hallway.random_rectangles(seed=random.randint(0,1000), num_recs=papers,
                                                    zona_peligrosa_x=zona_peligrosa_x,
                                                    zona_peligrosa_y=zona_peligrosa_y,
                                                    rec_w=paper_size_w,
                                                    rec_h=paper_size_h)



    # generate text info
    # figure font size
    try:
```

```
        fontInches = ADA.get_font_size(txt, PQ_DIM/res)
    except Exception as e:
        print("ERROR: {}".format(e.message))
        raise
    FSPx = fontInches*res
    FSCALE = FSPx/FONT_BS
    # now to generate coords for the plaque
    # TODO: rn this is hardcoded. should be dynamic
    txtbx = cv2.getTextSize(txt,FONT,FSCALE,1) # get size of box bounding text
    # print("DEBUG TEXT BOX SIZE: {}".format(txtbx))
    (wt,ht),bs = txtbx
    self._pqs = wt+30 # 10px margin around at least
    # find a random spot for the plaque to be
    Px1 = random.randint(0,HL_WD-self._pqs)
    Py1 = PQ_WALL_HT
    # add the plaque
    (_, _), (Px2, Py2) = self.draw_room_sign(hallway, (Px1,Py1), self._pqs)
    # add text
    self._draw_room_number(hallway, Px1, Py1+ht*2, text=txt)
    # its time for the door. will add on right if space, otherwise on left
    if Px1 < DR_WD + PQ_2_DR:
        # not enough space on left of sign
        Dx1 = Px2 + PQ_2_DR
    else:
        Dx1 = Px1 - (DR_WD+PQ_2_DR)
    Dy1 = HL_CEIL-DR_HT
    Dx2 = Dx1+DR_WD
    # Dy2 = Dy1+DR_HT
    Dy2 = HL_CEIL
    # add the door
    self.draw_door(hallway, Dx1, DW=DR_WD, DH=DR_HT)
    # # now add some rectangles as papers and billboards in an area where there is no plaque or door
    # paper_size_h = res*11
    # paper_size_w = res*8
    # poster_size_h = random.randint(res*12, res* 36)
    # poster_size_w = random.randint(res*12, res* 36)
    # # clear space didn't work, need to make forbidden zone
    # # it is min of door left or plaque left, and max or door right and plaque rt

```

```
    # zona_peligrosa_x = [n for n in range(min(Dx1, Px1), max(Dx2, Px2))]
    # # zona_peligrosa_y = [n for n in range(min(Dy1, Py1), max(Dy2, Py2))]
    # # additionally, add restrictions for height (so things are only where people would see them)
    # # assume most things hang between 80" and 36"
    # vis_top = HL_CEIL-res*80
    # vis_bottom = HL_CEIL-res*36
    # # not sure if it would be faster to build bigger list and then slice but my guess is the list
    # # comprehension is pretty integral so going with that
    # zona_peligrosa_y = [n for n in range(HL_CEIL) if n < vis_top or n > vis_bottom]
    # print(len(zona_peligrosa_y))
    # # now use the random square placement to drop a random number of papers, posters on the clear space
    # if posters:
    #     hallway.random_rectangles(seed=random.randint(0,1000), num_recs=posters,
    #                               zona_peligrosa_x=zona_peligrosa_x,
    #                               zona_peligrosa_y=zona_peligrosa_y,
    #                               rec_w=poster_size_w,
    #                               rec_h=poster_size_h)
    # if papers:
    #     hallway.random_rectangles(seed=random.randint(0,1000), num_recs=papers,
    #                               zona_peligrosa_x=zona_peligrosa_x,
    #                               zona_peligrosa_y=zona_peligrosa_y,
    #                               rec_w=paper_size_w,
    #                               rec_h=paper_size_h)
    # a little seasoning
    hallway.salt_and_pepper()
    return hallway, (Px1,Py1), (Px2, Py2)

# def add_paper_and_posters(self, num_posters, num_papers, top_left, bottom_right):
    # pass


def add_stuff(self, image, stuffScale = 2):
```

```python
        '''adds other shapes and lines to image
        Args:
            image: image class instance
            stuffScale: scale of 1 to 10, how much stuff is in the image
        '''
        image.random_lines(seed=random.randint(0,1000),
                           num_lines = stuffScale*2)
        image.random_rectangles(
            seed=random.randint(0,1000),
            num_recs=stuffScale,
            rec_h=random.randint(0,170),
            rec_w=random.randint(0,280)
        )
        return image

    def _draw_room_number(self, image, x, y, *, FSCALE=.75, text=None):
        '''Helper function. Draws room number/letter on the plaque.
            Args:
                self: instance
                image: image object to draw on
                (x,y): origin of plaque
        '''
        if text is None:
            text = self._gen_plaque_text()
        # cv2.putText(img, text, origin, fontFace, fontScale, color[, thickness[, lineType[, bottomLeftOrigin]]])
        cv2.putText(image.image, text, (x,y), self._font, FSCALE, self._fontv, 2)
        return image

    def draw_door(self, image, x_coord, value=(7,30,56), *, DH=None, DW=None, CH=None):
        '''draw a door on the image.
        Args:
            DH: door height
            DW: door width
            CH: ceiling height
        '''
        if DH is None:
            DH = ADA.DOOR_HT*self.res
        if DW is None:
            DW = ADA.DOOR_WD*self.res
```

```
        if CH is None:
            CH = ADA.CEIL_HT*self.res
        p1=(x_coord,(CH-DH)) # top left
        p2=(x_coord+DW,CH) # bottom right
        image.rectangle(p1,p2,value,-1)

    def _gen_plaque_text(self):
        '''Thanks to https://stackoverflow.com/a/2257449 for the text/number generation
            generates random 3-char string of numbers and uppercase letters
        '''
        text = ''.join(random.choices(string.digits + string.ascii_uppercase,
                                 k=self._strlen))
        return text

    def draw_room_sign(self, image, top_left=None, width=75):
        '''places a numbered room sign somewhere on image,
            marks filename as having room sign
            Args:
                image: image object
                top_left: coordinates for placement of top left
                of plaque. if None, randomly place.
                    should be (point1x,point1y)
            Returns:
                top_left: x, y coordinate of top left of rectangle
                bottom_right: x, y coordinate of bottom left of rectgl
        '''
        #create random starting point within boundaries
        if top_left is not None:
            point1x,point1y = top_left
        else:
            point1x = random.randint(0, (self._size[0]-self._pqs))
            point1y = random.randint(0, (self._size[0]-self._pqs))
        point2x = point1x + width
        point2y = point1y + width
        top_left = (point1x, point1y)
        bottom_right = (point2x, point2y)
        # adding possible scenarios for elliptical or triangular palques. not implemented yet.
        if self._plaque_shape is 1:
            image.rectangle(top_left, bottom_right, self._pqv, -1)
        elif self._plaque_shape is 2:
```

```
            pass
        return top_left, bottom_right

    def draw_special_room_sign(self, image, top_left=None, width=75, height=105):
        '''places a numbered room sign somewhere on image,
            marks filename as having room sign
            Args:
                image: image object
                top_left: coordinates for placement of top left
                of plaque. if None, randomly place.
                    should be (point1x,point1y)
            Returns:
                top_left: x, y coordinate of top left of rectangle
                bottom_right: x, y coordinate of bottom left of rectgl
        '''
        #create random starting point within boundaries
        if top_left is not None:
            point1x, point1y = top_left
        else:
            point1x = random.randint(0, (self._size[0]-width))
            point1y = random.randint(0, (self._size[0]-height))
        top_left = (point1x, point1y)
        bottom_right = (point1x + width, point1y + height)
        # adding possible scenarios for elliptical or triangular palques. not implemented yet.

        image.rectangle(top_left, bottom_right, self._pqv, -1)
        # draw another rectangle to look like the plaques, 13 px from top, 10 px in from the sides, 33 px tall, 57 wide
        top_left = (top_left[0] + 10, top_left[1] + 13)
        bottom_right = (top_left[0] + 57, top_left[1] + 33)
        image.rectangle(top_left, bottom_right, (240,240,240), -1)
        bottom_right = (bottom_right[0], bottom_right[1] + 5)
        # and then to put the text, it hosuld be under this new rectangle

        return (top_left[0] + 5, top_left[1] + 5 + 33 + 20)

    def make_false_image(self, num_randos=4, seasoning = 0.02, *, blur = None):
        '''generate an image without a room sign.
        Args:
            num_randos: how many random lines/recs to add
```

```
            seasoning: how much salt and pepper
            blur: optional, overrides defualt blur amount
        '''
        image = self.create_canvas()
        image = self.add_stuff(image, num_randos)
        image.salt_and_pepper(seasoning)
        if blur is not None:
            image.blur(blur)
        else:
            image.blur()
        return image

    def make_true_image(self, num_randos=4, seasoning=0.02, *,
    blur=None, special=True):
        '''generate an image with a room sign.
        Args:
            num_randos: how many random lines/recs to add
            seasoning: how much salt and pepper
            blur: optional, overrides defualt blur amount
        '''
        image = self.create_canvas()
        image = self.add_stuff(image, num_randos)
        if special:
            (px,py) = self.draw_special_room_sign(image)
        else:
            (px,py), (px2,py2) = self.draw_room_sign(image)
        image = self._draw_room_number(image, px, py)
        image.salt_and_pepper(seasoning)
        if blur is not None:
            image.blur(blur)
        else:
            image.blur()
        return image
```

8.4 HandyTools.py

```python
#!/usr/bin/env python3

import os
import cv2
import argparse
import numpy
import math
import matplotlib.pyplot as plt


def getFilesInDirectory(directory, fileType):
    return [os.path.join(directory, item) for item in os.listdir(directory) if item.lower().endswith(fileType)]


def resize_files_in_directory(rs_factor, directory, outdir):
    files = getFilesInDirectory(directory, 'jpg')
    for f in files:
        img = cv2.imread(f)
        rs = cv2.resize(img, (img.shape[1]//rs_factor, img.shape[0]//rs_factor), interpolation=cv2.INTER_AREA)
        fn = os.path.join(outdir, os.path.split(f)[1])
        cv2.imwrite(fn, rs)


def crop_image(image, x, y, xb, yb):
    copy = image.copy()
    return copy[y:yb, x:xb, :]


def show(image):
    cv2.imshow("image", image)
    cv2.waitKey()
    cv2.destroyWindow("image")


def hiLow255(num):
    return 0 if num > 122 else 255


def showKill(image, title=None, waitkey=0):
    '''takes cv2 image and shows it.
```

```
        if something goes wrong and window is clicked closed, it will recover.
    '''
    title = title if title else "image"
    status = 1
    try:
        cv2.imshow(title, image)
        while status > 0:
            ks=cv2.waitKey(waitkey)
            try:
                status = cv2.getWindowProperty(title,cv2.WND_PROP_VISIBLE)
            except Exception:
                status = -1
                break
            if ks > 0:
                break
        cv2.destroyWindow(title)
    except Exception as e:
        print("error occured: {}", e)
        raise


def betwixt(less_num, target, great_num):
    '''true if target falss between less_num and great_num'''
    return(less_num < target and target < great_num)


def add_prefix_to_file(filepath, prefix):
    '''
    sets prefix in front of a filename and returns amended path
    sample filepath: 'train/plaques/002999.png'
    sample prefix: '0_'
    '''
    directory, file_name = os.path.split(filepath)
    file_name = prefix + file_name
    changed_path = os.path.join(directory, file_name)
    return changed_path


def str2bool(word):
    '''
    from 'Maxim's response to https://stackoverflow.com/questions/15008758/parsing-boolean-values-with-argparse
    will evaluate a string as a true or false
```

```
    '''
    if word.lower() in ('yes','true','y','t','yep','1','ok'):
        return True
    elif word.lower() in ('no','false','n','f','nope','0','nah','fuck you'):
        return False
    else:
        raise argparse.ArgumentTypeError('Boolean value expected. Very disappointed')


def distill_list(list_of_elements):
    '''
    takes a list of many items and removes all adjacent duplicates.
    '''
    new_list = []
    cur_idx = 0
    now_val = list_of_elements[cur_idx]
    for index, value in enumerate(list_of_elements):
        if cur_idx == len(list_of_elements)-1:
            new_list.append(now_val)
            break
        if list_of_elements[cur_idx+1] is now_val:
            cur_idx += 1
        elif list_of_elements[cur_idx+1] is not now_val:
            new_list.append(now_val)
            cur_idx += 1
            now_val = list_of_elements[cur_idx]
    return new_list


def four_point_transform(image, pts):
    # https://www.pyimagesearch.com/2014/08/25/4-point-opencv-getperspective-transform-example/
    # obtain a consistent order of the points and unpack them
    # individually
    rect = order_points(pts)
    (tl, tr, br, bl) = rect
    # compute the width of the new image, which will be the
    # maximum distance between bottom-right and bottom-left
    # x-coordiates or the top-right and top-left x-coordinates
    widthA = numpy.sqrt(((br[0] - bl[0]) ** 2) + ((br[1] - bl[1]) ** 2))
```

```
    widthB = numpy.sqrt(((tr[0] - tl[0]) ** 2) + ((tr[1] - tl[1]) ** 2))
    maxWidth = max(int(widthA), int(widthB))
    # compute the height of the new image, which will be the
    # maximum distance between the top-right and bottom-right
    # y-coordinates or the top-left and bottom-left y-coordinates
    heightA = numpy.sqrt(((tr[0] - br[0]) ** 2) + ((tr[1] - br[1]) ** 2))
    heightB = numpy.sqrt(((tl[0] - bl[0]) ** 2) + ((tl[1] - bl[1]) ** 2))
    maxHeight = max(int(heightA), int(heightB))
    # now that we have the dimensions of the new image, construct
    # the set of destination points to obtain a "birds eye view",
    # (i.e. top-down view) of the image, again specifying points
    # in the top-left, top-right, bottom-right, and bottom-left
    # order
    dst = numpy.array([
        [0, 0],
        [maxWidth - 1, 0],
        [maxWidth - 1, maxHeight - 1],
        [0, maxHeight - 1]], dtype = "float32")
    # compute the perspective transform matrix and then apply it
    M = cv2.getPerspectiveTransform(rect, dst)
    warped = cv2.warpPerspective(image, M, (maxWidth, maxHeight))
    # return the warped image
    return warped


def order_points(pts):
    # https://www.pyimagesearch.com/2014/08/25/4-point-opencv-getperspective-transform-example/
    # initialzie a list of coordinates that will be ordered
    # such that the first entry in the list is the top-left,
    # the second entry is the top-right, the third is the
    # bottom-right, and the fourth is the bottom-left
    rect = numpy.zeros((4, 2), dtype = "float32")
    # the top-left point will have the smallest sum, whereas
    # the bottom-right point will have the largest sum
    s = pts.sum(axis = 1)
    rect[0] = pts[numpy.argmin(s)]
    rect[2] = pts[numpy.argmax(s)]
    # now, compute the difference between the points, the
    # top-right point will have the smallest difference,
    # whereas the bottom-left will have the largest difference
    diff = numpy.diff(pts, axis = 1)
```

```python
    rect[1] = pts[numpy.argmin(diff)]
    rect[3] = pts[numpy.argmax(diff)]
    # return the ordered coordinates
    return rect


def _plot_multiple_images(labels_and_images, num_imgs=36, rows=6, cols=6):
    total_number = len(labels_and_images)
    # num_imgs = 36
    num_iterations = math.ceil(total_number / num_imgs)
    # rows = math.ceil(math.sqrt(numgs))
    # cols = math.ceil(numgs / rows)
    # rows = 6
    # cols = 6
    for n in range(num_iterations):
        fig = plt.figure(facecolor='gray')
        for idx, title_img_tup in enumerate(labels_and_images[n * num_imgs:n * num_imgs + num_imgs]):
            # print(title_img_tup)
            sp = fig.add_subplot(cols, rows, idx + 1)
            # image = cv2.resize(title_img_tup[1], (title_img_tup[1].shape[1]//3, title_img_tup[1].shape[0]//3), interpolation=cv2.INTER_AREA)
            image = title_img_tup[1]
            image = cv2.cvtColor(image, cv2.COLOR_GRAY2RGB)
            plt.imshow(numpy.array(image, dtype=float))
            sp.set_title(title_img_tup[0])
            sp.set_yticklabels([])
            sp.set_xticklabels([])
        # fig.set_size_inches(numpy.array(fig.get_size_inches()) * numgs)
        fig.set_size_inches(numpy.array(fig.get_size_inches()) * 20)
        plt.show()


def plot_result_images(results):
    labels_and_images = []
    for meta in results:
        if meta.text and meta.thresheld_image:
            labels_and_images.extend([(meta.text[n], meta.thresheld_image[n]) for n in range(len(meta.text))])
    _plot_multiple_images(labels_and_images)

```

```

def normalize_image_illumination(image):
    max_dim = max(image.shape[:2])
    y, cr, cb = cv2.split(cv2.cvtColor(image, cv2.COLOR_BGR2YCrCb)
    )
    sigma = 5 * max_dim // 300
    gaussian = cv2.GaussianBlur(y, (0, 0), sigma, sigma)
    y = (y - gaussian + 100)
    return cv2.cvtColor(cv2.merge([y, cr, cb]), cv2.
    COLOR_YCrCb2BGR)
```

8.5 ADA.py

```
'''collection of ADA requirements.
ALL IN INCHES
'''
import CustomErrors as CER

def get_font_size(text, signWidth):
    '''returns sign spec from ADA.
    in INCHES!
    usage= sizeChart[signWidth][charnum]
    '''
    charnum = len(text)
    if charnum > 26:
        raise CER.PlaqueFontError(1)
    elif signWidth > 19:
        raise CER.PlaqueFontError(2)

    sizeChart= {
        4:{5: 0.625, 4: 0.75, 3: 1},
        6:{7: 0.625, 6: 0.75, 5: 0.875, 4: 1.25, 3: 1.5},
        8:{11: 0.625, 9: 0.75, 8: 0.875, 7: 1, 5: 1.25, 4: 1.5},
        10:{14: 0.625, 11: 0.75, 10: 0.875, 9: 1, 8: 1.25, 7: 1.5},
        12:{18: 0.625, 14: 0.75, 12: 0.875, 11: 1, 8: 1.25, 7: 1.5},
        18:{25: 0.625, 21: 0.75, 18: 0.875, 16: 1, 13: 1.25, 11: 1.5}
    }
    keys = list(sizeChart.keys())
    chartKey = min(keys, key=lambda x: abs(x-signWidth))
    chartKeyKeys = list(sizeChart[chartKey])
    fontKey = min(chartKeyKeys, key = lambda x: abs(x-charnum))
    return sizeChart[chartKey][fontKey]

def toGray(B,G,R):
    '''converts color value to grayscale via the Limunosity method.
    Args:
        (B,G,R): blue, green, and red colorspace
    '''
    return (R*0.21 + G*0.72 + B*0.07)

DOOR_HT = 80
DOOR_WD = 32
```

```python
VEIL_HT = 120
PQ_HT = 50
CEIL_HT = 120

AdriansEastROIDetector.py
# code from https://www.pyimagesearch.com/2018/08/20/opencv-text-detection-east-text-detector/
from imutils.object_detection import non_max_suppression
import numpy as np
import argparse
import time
import cv2


def main(args):
    detect_ranges_with_east(args["image"], args["width"], args["height"], args["east"], args["min_confidence"])


def detect_ranges_with_east(image, width, height, east, min_confidence):
    """(h and w should be multiple of 32)"""
    # load the input image and grab the image dimensions
    # image = cv2.imread(image)
    orig = image.copy()
    (H, W) = image.shape[:2]

    # set the new width and height and then determine the ratio in change
    # for both the width and height
    (newW, newH) = (width, height)
    rW = W / float(newW)
    rH = H / float(newH)

    # resize the image and grab the new image dimensions
    image = cv2.resize(image, (newW, newH))
    (H, W) = image.shape[:2]

    # define the two output layer names for the EAST detector model that
    # we are interested -- the first is the output probabilities and the
    # second can be used to derive the bounding box coordinates of text
    layerNames = [
```

```
    "feature_fusion/Conv_7/Sigmoid",
    "feature_fusion/concat_3"]

# load the pre-trained EAST text detector
# print("[INFO] loading EAST text detector...")
net = cv2.dnn.readNet(east)

# construct a blob from the image and then perform a forward pass of
# the model to obtain the two output layer sets
blob = cv2.dnn.blobFromImage(image, 1.0, (W, H), (123.68, 116.78, 103.94), swapRB=True, crop=False)
start = time.time()
net.setInput(blob)
(scores, geometry) = net.forward(layerNames)
end = time.time()

# show timing information on text prediction
# print("[INFO] text detection took {:.6f} seconds".format(end - start))

# grab the number of rows and columns from the scores volume, then
# initialize our set of bounding box rectangles and corresponding
# confidence scores
(numRows, numCols) = scores.shape[2:4]
rects = []
confidences = []

# loop over the number of rows
for y in range(0, numRows):
    # extract the scores (probabilities), followed by the geometrical
    # data used to derive potential bounding box coordinates that
    # surround text
    scoresData = scores[0, 0, y]
    xData0 = geometry[0, 0, y]
    xData1 = geometry[0, 1, y]
    xData2 = geometry[0, 2, y]
    xData3 = geometry[0, 3, y]
    anglesData = geometry[0, 4, y]

    # loop over the number of columns
```

```
    for x in range(0, numCols):
        # if our score does not have sufficient probability, ignore it
        if scoresData[x] < min_confidence:
            continue

        # compute the offset factor as our resulting feature maps will
        # be 4x smaller than the input image
        (offsetX, offsetY) = (x * 4.0, y * 4.0)

        # extract the rotation angle for the prediction and then
        # compute the sin and cosine
        angle = anglesData[x]
        cos = np.cos(angle)
        sin = np.sin(angle)

        # use the geometry volume to derive the width and height of
        # the bounding box
        h = xData0[x] + xData2[x]
        w = xData1[x] + xData3[x]

        # compute both the starting and ending (x, y)-coordinates for
        # the text prediction bounding box
        endX = int(offsetX + (cos xData1[x]) + (sin * xData2[x]))
        endY = int(offsetY - (sin * xData1[x]) + (cos * xData2[x]))
        startX = int(endX - w)
        startY = int(endY - h)

        # add the bounding box coordinates and probability score to
        # our respective lists
        rects.append((startX, startY, endX, endY))
        confidences.append(scoresData[x])

# apply non-maxima suppression to suppress weak, overlapping bounding
# boxes
boxes = non_max_suppression(np.array(rects), probs=confidences)
```

```
    regions = []
    drawn_images = []
    # loop over the bounding boxes
    for (startX, startY, endX, endY) in boxes:
        # scale the bounding box coordinates based on the respective
        # ratios
        startX = int(startX * rW)
        startY = int(startY * rH)
        endX = int(endX * rW)
        endY = int(endY * rH)

        # draw the bounding box on the image
        cv2.rectangle(orig, (startX, startY), (endX, endY), (0, 255, 0), 2)
        regions.append([(startX, startY), (endX, endY),
                        (startX, endY), (endX, startY)])
        drawn_images.append(orig)
    return regions, drawn_images


if __name__ == "__main__":
    # construct the argument parser and parse the arguments
    ap = argparse.ArgumentParser()
    ap.add_argument("-i", "--image", type=str, help="path to input image")
    ap.add_argument("-east", "--east", type=str, help="path to input EAST text detector")
    ap.add_argument("-c", "--min-confidence", type=float, default=0.5, help="minimum probability required to inspect a region")
    ap.add_argument("-w", "--width", type=int, default=320, help="resized image width (should be multiple of 32)")
    ap.add_argument("-e", "--height", type=int, default=320, help="resized image height (should be multiple of 32)")
    args = vars(ap.parse_args())
    main(args)
```

8.6 Detector.py

```python
import dlib
import cv2


class ObjectDetector(object):
    """
    from https://www.hackevolve.com/create-your-own-object-detector/
    https://github.com/saideeptalari/Object-Detector

    """
    def __init__(self, options=None, loadPath=None):
        # create detector options
        self.options = options
        if self.options is None:
            self.options = dlib.simple_object_detector_training_options()

        # load the trained detector (for testing)
        if loadPath is not None:
            self._detector = dlib.simple_object_detector(loadPath)

    def _prepare_annotations(self, annotations):
        annots = []
        for (x, y, xb, yb) in annotations:
            annots.append([dlib.rectangle(left=int(x), top=int(y), right=int(xb), bottom=int(yb))])
        return annots

    def _prepare_images(self, imagePaths):
        images = []
        for imPath in imagePaths:
            image = cv2.imread(imPath)
            image = cv2.cvtColor(image, cv2.COLOR_BGR2RGB)
            images.append(image)
        return images

    def fit(self, imagePaths, annotations, visualize=False, savePath=None):
        annotations = self._prepare_annotations(annotations)
        images = self._prepare_images(imagePaths)
        self._detector = dlib.train_simple_object_detector(images, annotations, self.options)
```

```python

        # visualize HOG
        if visualize:
            win = dlib.image_window()
            win.set_image(self._detector)
            dlib.hit_enter_to_continue()

        # save detector to disk
        if savePath is not None:
            self._detector.save(savePath)

        return self

    def predict(self, image):
        boxes = self._detector(image)
        preds = []
        for box in boxes:
            (x, y, xb, yb) = [box.left(), box.top(), box.right(), box.bottom()]
            preds.append((x, y, xb, yb))
        return preds

    def detect(self, image, annotate=None):
        rgb_image = cv2.cvtColor(image, cv2.COLOR_BGR2RGB)
        preds = self.predict(rgb_image)
        for (x, y, xb, yb) in preds:
            # draw and annotate on image
            cv2.rectangle(image, (x, y), (xb, yb), (0, 0, 255), 2)
            if annotate and isinstance(annotate, str):
                cv2.putText(image, annotate, (x + 5, y - 5), cv2.FONT_HERSHEY_SIMPLEX, 1.0, (128, 255, 0), 2)
        cv2.imshow("Detected", image)
        cv2.waitKey(0)
        return image
```