Adithya Palle

Project 3: Real-time 2-D Object Recognition

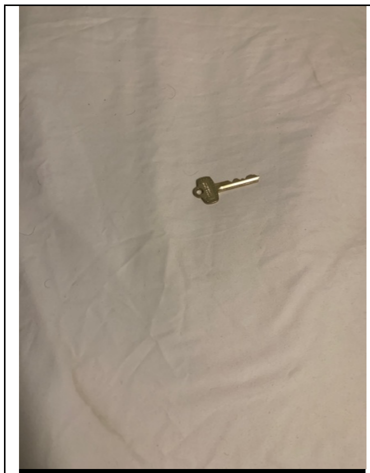
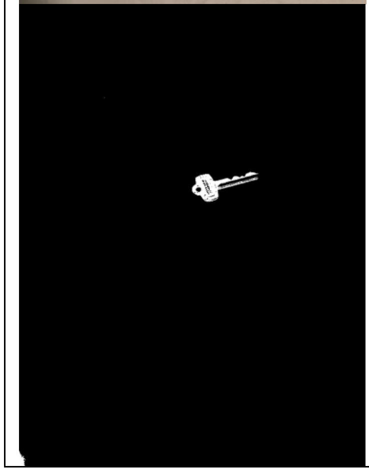
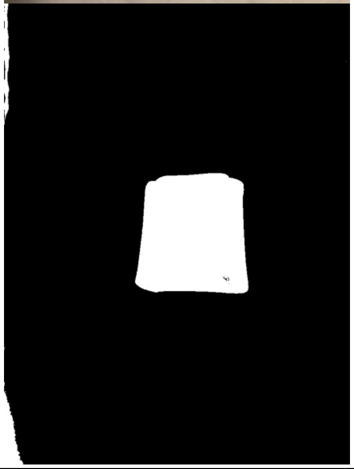
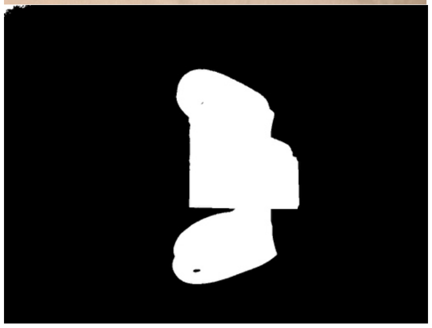
Overview:

In this project, we developed an object recognition system using classical methods. We first create a mask to separate the foreground by darkening saturated areas and applying ISODATA for color thresholding. The mask is refined with morphological operations: one erosion followed by five dilations. We then segment the foreground using a region-growing algorithm, generating a region map for visualization. For classification, we extract features from the primary object in each image, including properties of the oriented bounding box, central moments, area, perimeter, and color. These features form a training set for five objects: a wallet, nail clipper, key, quarter, and Nintendo Switch controller. Each object is represented by a feature vector stored in a database. Unseen images are classified by comparing their feature vectors against the training set using scaled Euclidean distance. We evaluate the system on five unseen images per category and explore alternative distance metrics. A real-time demo showcases the application's performance, and we analyze its strengths and weaknesses. Finally, we optimize the mask cleanup stage by implementing dilation and iteration operations using a Grassfire transform.

Threshold the input video:

I decided to use ISODATA with k = 5 to dynamically determine a threshold by which to do binary thresholding for the images. First, I preprocess the image by darkening pixels in the image that have a saturation above 100. This is done by converting to the HSV space, and then reducing the value channel of such pixels by 99%. Then, I chose 5 clusters as my images included shadows and my background was not perfectly white , so simply choosing 2 centroids often resulted in segmentations where shadows or other darker regions were also included in the segmentation. I instead chose 5 centroids and took the smallest two centroids based on brightness (the max of the values along each channel). I then took the midpoint between these two darkest centroids and used that as a threshold. Below are the results:

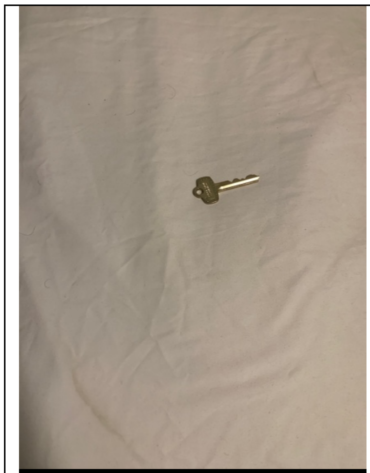
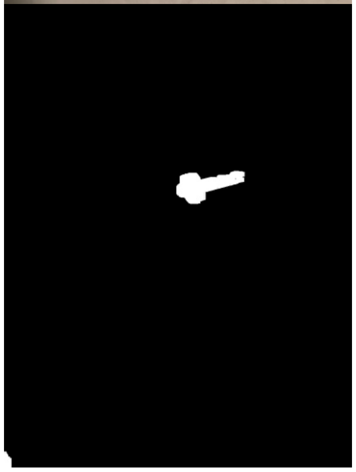
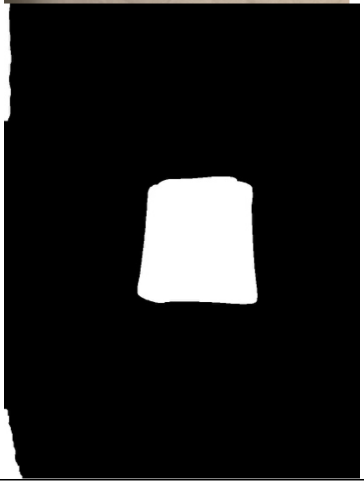
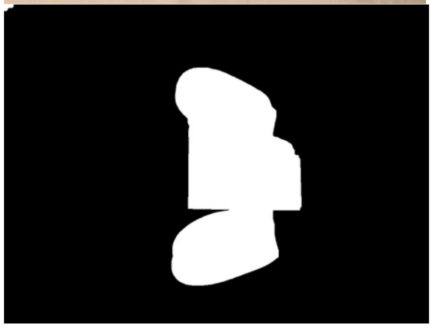
Adithya Palle

Clean up the binary image:

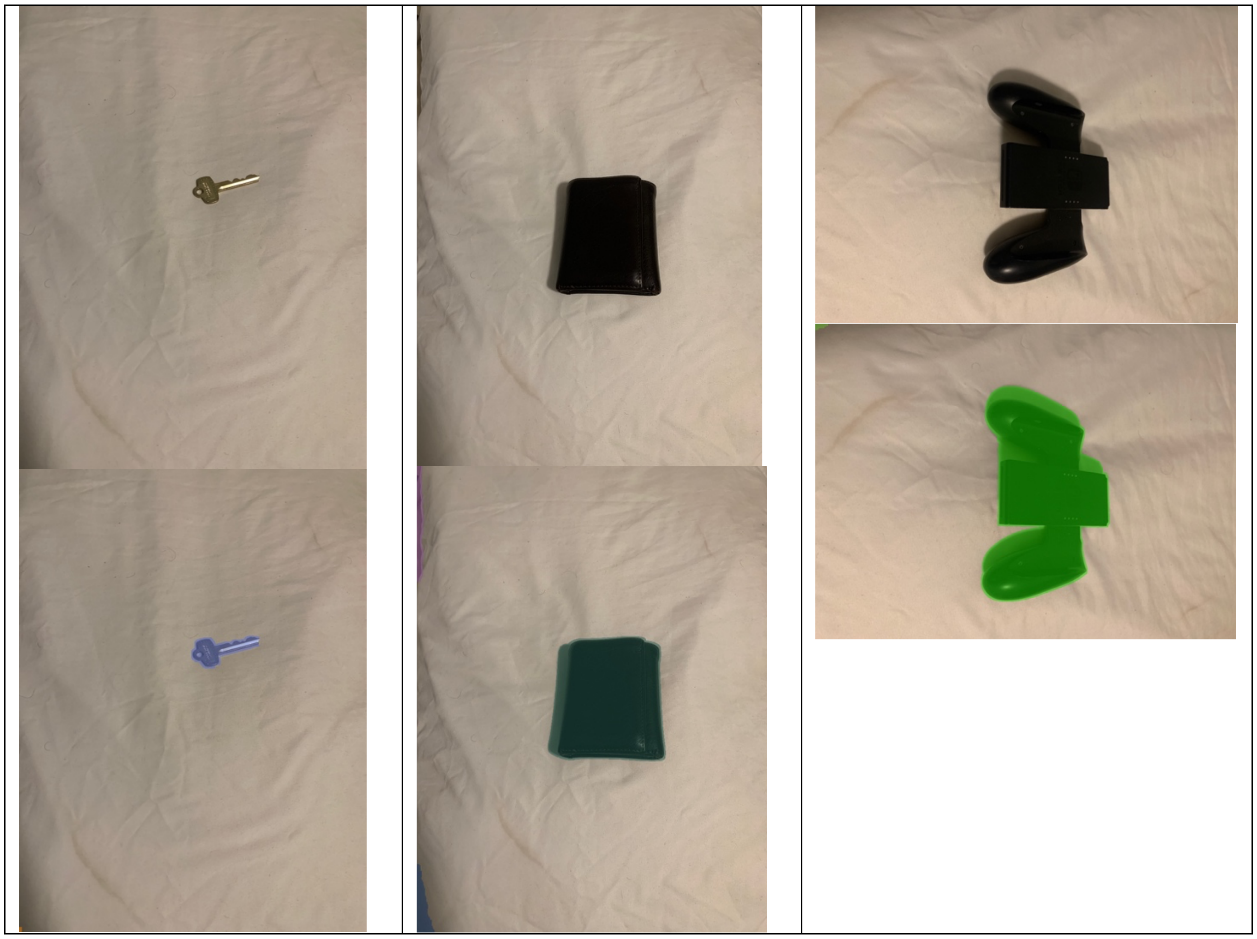
I noticed small holes in some masks, particularly in the key mask. I initially decided to just apply dilation to plug up these holes, but there were unseen white pixels that were expanded as a result, which created more noise. Therefore, I opted for opening, meaning I did 1 iteration of erosion with a 4-connected kernel (to eliminate noise before dilation), and then did 5 iterations of dilation with an 8-connected kernel. Below are the results:

Adithya Palle

Segment the image into regions:

I created a region map for the foreground regions using a simple iterative DFS algorithm (region-growing). I use 8-connectedness to find neighbors in my algorithm. For visualization, I colored each region id with a random, unique color (storing it in a set so we don't repeat it later). I then tinted the original image with the region colors. Below are the results:

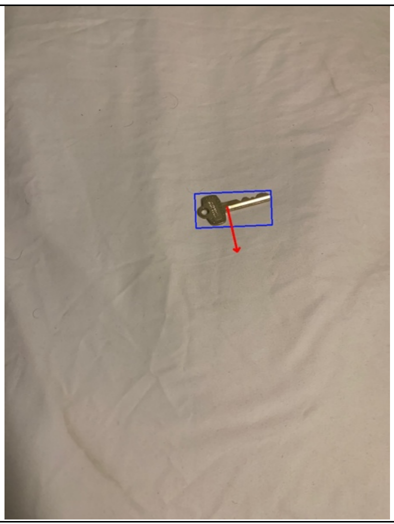
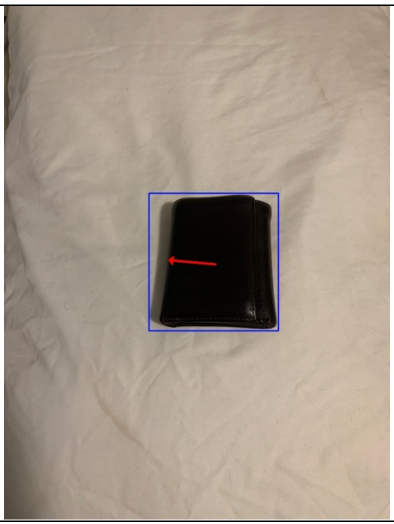
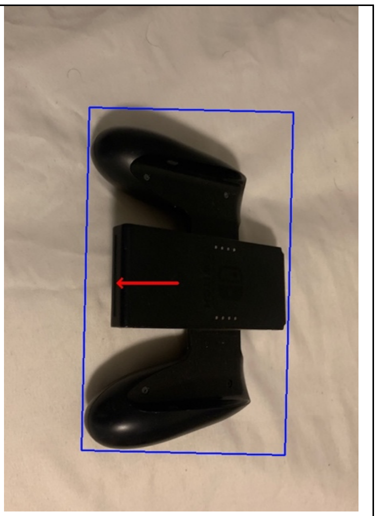


While there aren't many unique regions as there is only a single object in each image, you can see some of the regions on the edge of the wallet image have differing colors than the wallet region.

Computing features for each major region:

For this task, we display the axis of least central moment along with the oriented bounding box as well as calculate some spatially invariant features. The axis of least central moment represents the axis along which the object has the least spread, or in other words, the axis pointing in the direction of the "shorter" side of the object. We compute the oriented bounding box which is basically just a bounding box that is rotated to line up with the axis of least central moment to best fit the object. The oriented bounding box was calculated using OpenCV's in-built function call minAreaRect which takes the contour (outline) of the region and computes the rotated rectangle that best fits it. The axis of least central moment was calculated from the covariance matrix of the 2nd -order moments resulting from cv::moments. I took the eigenvector which corresponded to the minor eigenvalue for this matrix which gave me the least central moment axis.

For the feature vectors, I calculated the percent of the oriented bounding box that the object takes up, the aspect ratio of the bounding box (length of the longest side divided by length of shorter side), the circularity of the bounding box (A value between zero and one that tells how circular an object is $C = 4\pi * Area/Perimeter^2$) , and the mean color of the region. These were also calculated using in-build OpenCv functions to find the area , perimeter, and mean color of the contours of the regions that we segmented. It is important to note that while we do use contours as an intermediary value to calculate some features, the features themselves are ultimately properties of the entire region itself. Below are the images with the axis and bounding box drawn on them, and below each image is its feature vector:

Image			
% bbox filled	55.553799	88.183807	69.932594
Aspect ratio	2.216838	1.063830	1.683929
Circularity	0.393732	0.775830	0.452251
Mean Color (BGR)	(84,110,122)	(17,21,24)	(20,27,29)

Collect Training Data

In my system, if the user hits the ‘n’ key on the “Features Image” frame of the “real_time_or” binary, they will be prompted to input a label and then the feature vector calculated from the largest region of the frame that was captured right before they hit ‘n’ will be saved to a file with the path `image_features/<label>.features`. This allows user to seamlessly switch objects in the live video feed and extract features for multiple objects in a single run.

I also implement an executable ("image_dir_or") that extracts features for all the images in a directory. If the image has a basename of the structure <basename>.<extension>, then the feature file is saved at `image_features/<basename>.features`. This makes it easy to extract features from a larger dataset of images.

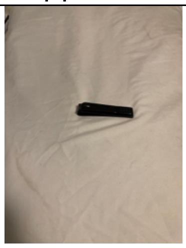
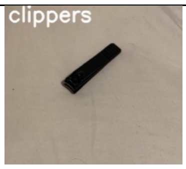
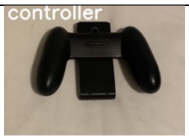
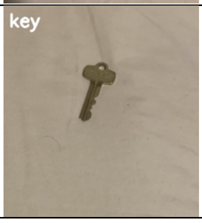
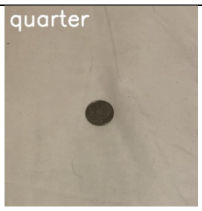
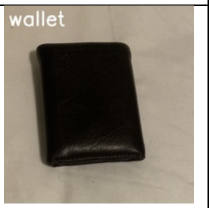
Classify New Images

In this task, we classify unlabeled images using a dataset of five images with pre-extracted features. For each unlabeled image, we compute its feature vector using the same extraction method and determine its closest match in the dataset using the scaled Euclidean distance. The label of the nearest feature vector is then assigned to the unlabeled image. The scaled Euclidean distance is defined as follows:

$$D = \sum \left(\frac{x_i - y_i}{\sigma_{y_i}} \right)^2$$

Here, y represents feature vectors from our database, and x represents those form the unknown objects. The standard deviation is calculated using the database feature vectors, and *i* corresponds to the index of the value in the feature vector.

For our known objects database, we use the 5 images below illustrating the 5 objects we are focusing on, and below it is the "unknown" object corresponding to that category for which the system has no label. The images of unknown objects are annotated with what the system predicted for them.

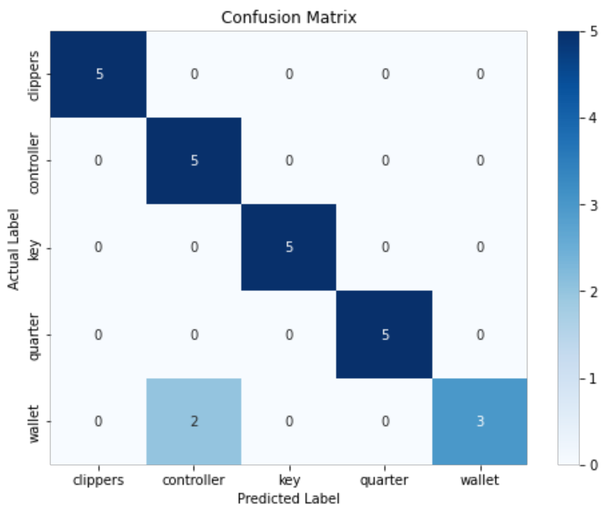
Label	Clippers	Controller	Key	Quarter	Wallet
Known					
Unknown					

Adithya Palle

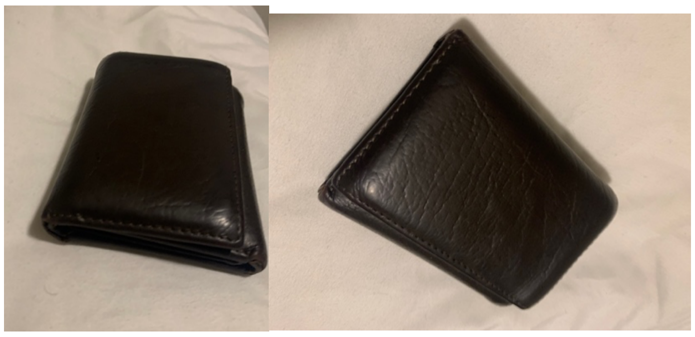
The predictions are correct for all the images shown here, as these are quite simple objects that can be easily discerned by bounding box features, circularity, and color. If the Objects were slightly altered such as the wallet or clippers being opened or the colors of the objects changed, such a simple feature extraction system may not be as effective.

Evaluate the Performance of your System

I generated a dataset of 25 images, 5 of each of the object classes. This dataset contained no overlap with the set of 5 images that are used to create feature vectors to compare against. I ran the classification against these 25 images, saved results in a csv file, and visualized a confusion matrix of the result with the help of a python script.



As seen above, this simple classifier is highly accurate, with only 2 instances of mislabeling for the wallet class. Below are the images that were mislabeled:



Due to the elongation of the wallet caused by the camera angle, it is likely that the aspect ratio of the wallet in this image was very similar to that of the controller, which shares a comparable aspect ratio. Additionally, both the wallet and the controller are dark, almost black in color, making it understandable that this slight change in aspect ratio led the classifier to mistakenly predict the wallet as a controller. This highlights a flaw in using aspect ratio as a feature for classification when camera angles vary, as elongated versions of objects can have significantly different aspect ratios. A potential solution to this issue is incorporating additional features to reduce the reliance on aspect ratio when computing the overall classification distance.

Capture a demo of your system working

A link to a video demo is provided in the README.md file included in the submission. Some key observations from the video include the classifier's persistent misclassification of the wallet. This suggests that its feature vector is too similar to those of other classes, making slight variations between wallet-like objects result in entirely different classifications.

Another notable flaw in our method appears between object placements when my shadow darkens the frame. Since we use IOSDATA thresholding and classify the darker cluster of the image as the foreground, casting a shadow over the object causes the background to darken, leading more of the background to be incorrectly classified as part of the foreground. Additionally, shadows also darken the objects themselves, potentially leading to misclassifications due to differing lighting conditions compared to when the object's features were originally extracted.

A flaw in dynamic thresholding methods such as ISODATA is that they select a threshold even when no dark object is present in the frame. This can cause the system to detect a region due to minor color variations in the background, leading to a false classification. To mitigate this, an additional fixed check could be implemented to verify if the ISODATA separation point falls above a certain threshold—indicating it is primarily detecting the white background. If this condition is met, the system can conclude that no valid regions are present.

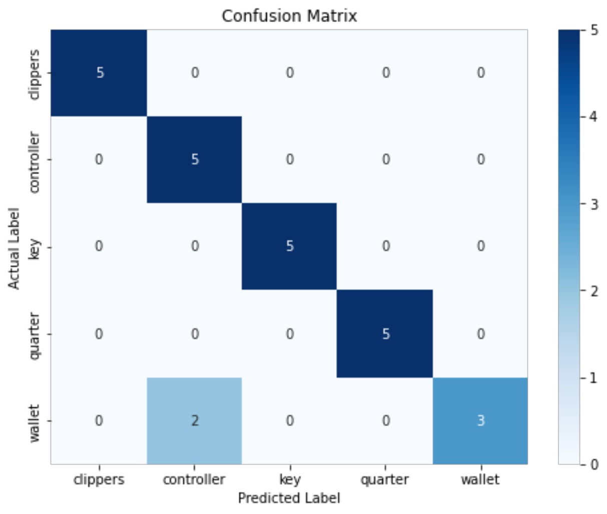
Implement a second classification method

For this task, I chose to use the nearest neighbor matching method as done above, but with different distance metrics.

Below are the 5-distance metric I compare, along with the confusion matrix on the 25-image dataset that I used in a previous task. This was implemented by creating an interface for distance metrics defining the distance function and implementing this interface for each separate method using the below formulas. The feature vectors used for distance comparison are still the same as previous tasks as well (5 image dataset).

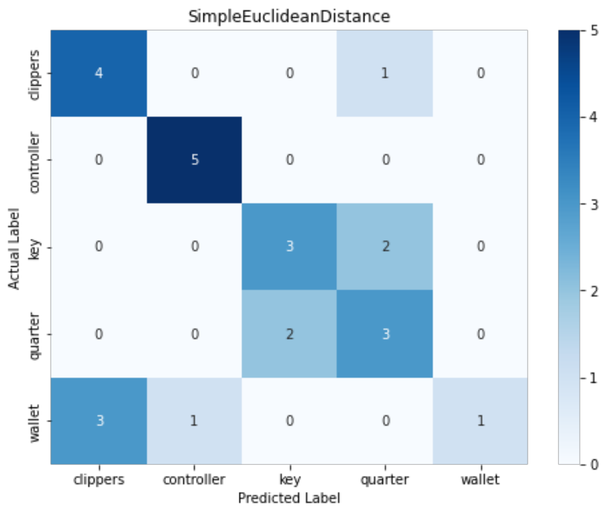
1. Scaled Euclidean Distance

$$D = \sum \left(\frac{x_i - y_i}{\sigma_{y_i}} \right)^2$$



2. Simple Euclidean Distance (Sum Squared Distance)

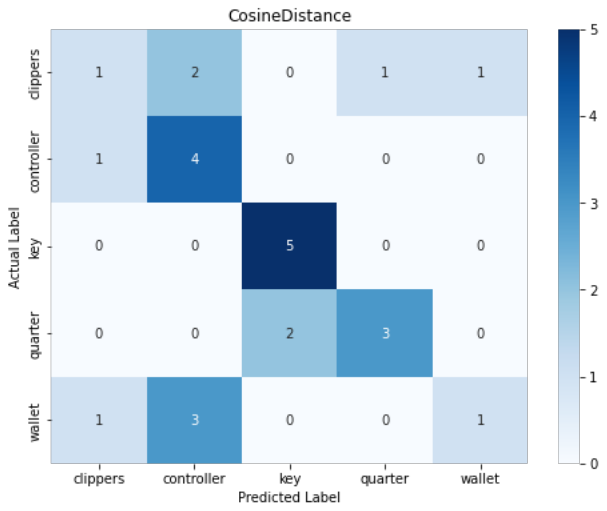
$$D = \sum (x_i - y_i)^2$$



3. Cosine Distance

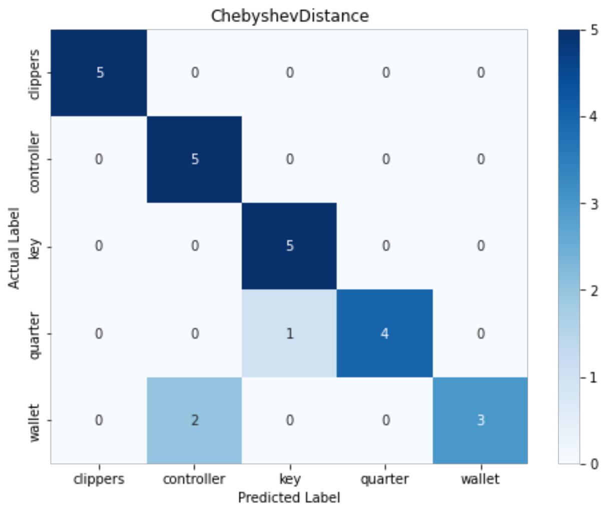
$$D = 1 - \left(\frac{X \cdot Y}{|X||Y|} \right)$$

where X and Y are the two feature vectors as a whole



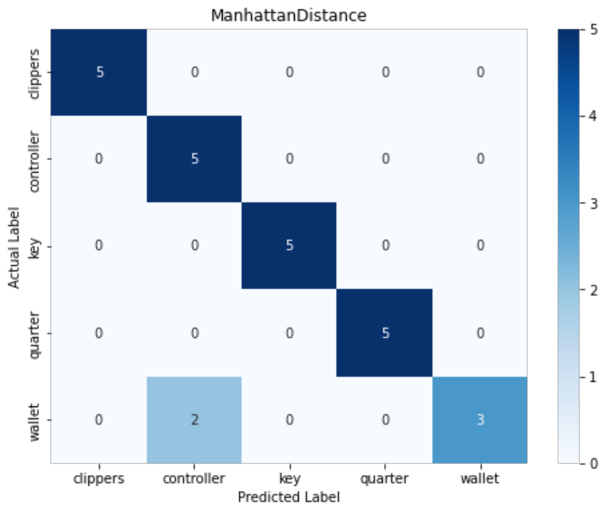
4. Scaled Chebyshev Distance (L-inf norm)

$$D = \max \left| \frac{x_i - y_i}{\sigma_{y_i}} \right|$$



5. Scaled Manhattan Distance (L1 norm)

$$D = \sum \left| \frac{x_i - y_i}{\sigma_{y_i}} \right|$$



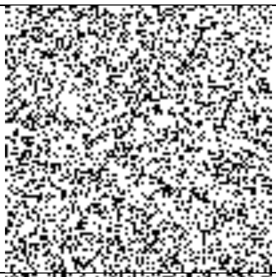
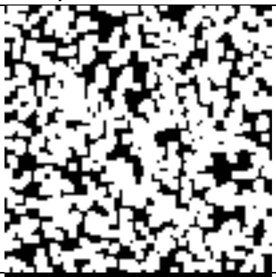
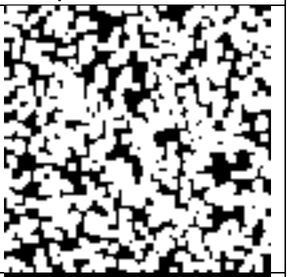
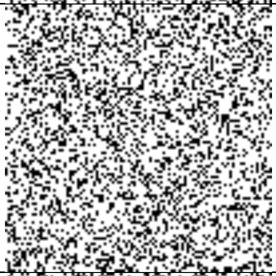
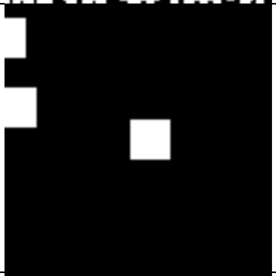
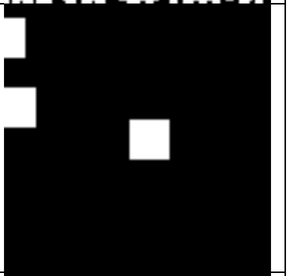
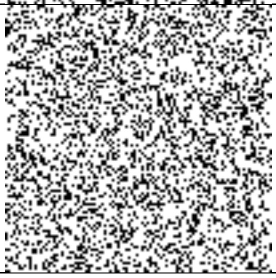
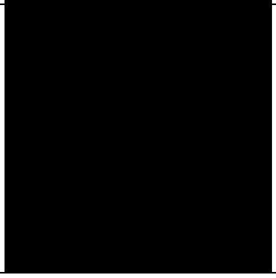
Scaled Euclidean Distance and Manhattan Distance both performed the best, producing identical confusion matrices. This makes sense because the two metrics are quite similar, except that Euclidean distance squares the differences before summing. The fact that both performed the same suggests that the feature differences were not particularly large—otherwise, the squaring in Euclidean distance would have amplified those differences, potentially leading to different classification results. Simple Euclidean distance, however, performed poorly. This is likely due to the heavy weighting of the %bbox filled and color features, which have significantly larger values than the other two features. As Euclidean distance is sensitive to feature magnitude, this disproportionate scaling likely led to suboptimal classification. The strong performance of Chebyshev distance (22/25 correctly classified) suggests that objects in this feature space can be effectively distinguished by the maximum feature difference. This implies that a difference in just one or a few key features is enough to discern objects correctly. However, this also suggests that our classification task is relatively simple—if a single feature per comparison is sufficient, adding more objects and increasing intra-class variation could degrade performance. The relatively poor performance of cosine distance was surprising at first, but upon reviewing its formula, this outcome makes sense. Cosine similarity is affected by uneven feature weighting, as it relies on the dot product, which involves multiplying feature values and summing them. As a result, features with smaller magnitudes, such as circularity and aspect ratio, are overshadowed by larger-valued features like %bbox filled and color. This imbalance likely caused cosine similarity to perform worse than expected. This article suggests adjusting the cosine similarity by the mean as a potential resolution to this issue.

Extension

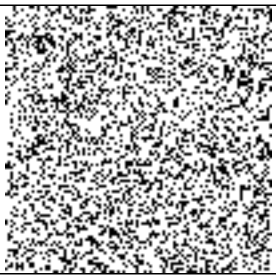
One of the extensions mentioned is to write more than two of the stages of the system from scratch. I wrote both the thresholding algorithm (including means implementation) and the connected components analysis (with region growing) from scratch. However, this is not my primary extension.

For my main extension, I want to take an alternative approach to my cleanup algorithm (which involves multiple iterations of erosion followed by dilation) using the grassfire transform. I implement this from scratch in the "cleanupGrassfire" function and kept my simple implementation that relies on OpenCV functions in the "cleanupSimple" function. Below, I compare the two algorithms on arbitrary region masks in terms of accuracy and runtime.

First, I illustrate the correctness of my implementation, by running "cleanupSimple" and "cleanupGrassfire" on the same random masks and verifying that the outputs are the same via a pixelwise comparison. I created a testing script for this in `testGrassfire.cpp` and below I will share the visualizations of some cleaned up 100x100 masks.

# Erosion Iterations	# Dilation Iterations	Original Mask	cleanupSimple Output	cleanupGrassfire Output
1	1			
3	7			
4	4			

Adithya Palle

1	5			

With the above results and the more thorough tests in the code, we can be assured that both functions have the same outputs. I ran both algorithms 100 times on the same random 100x100 mask and with 1 erosion iteration and 5 dilation iterations and found that it took 0.00721817 seconds for “cleanupSimple” and 0.376827 seconds for “cleanupGrassfire” to complete 100 executions each. Initially, I was surprised as I expected the distance transform optimization to be much faster than the naive approach of repeatedly applying multiple iterations of erosion and dilation. However, after running more tests at increasing mask sizes, it is apparent that the cv::erode and cv::dilate operations used in cleanupSimple are heavily optimized (likely with some sort of hardware acceleration). Below are the results over 100 executions for different mask sizes with 1 erosion iteration and 5 dilation iterations.

Mask Size	cleanupSimple time (s)	cleanupGrassfire time (s)
10 x 10	0.00151633	0.0041905
100 x 100	0.00721817	0.376827
1000 x 1000	0.0830504	37.2029
5000 x 5000	1.6816	926.95

We would expect the runtime of these algorithms to scale linearly with the area of the mask, as both have a complexity of O(A), where A is the area of the image. This holds true for cleanupGrassfire, but cleanupSimple grows significantly slower than O(A), at least until the jump from 1000x1000 to 5000x5000. Clearly, some advanced optimization is occurring under the hood that allows it to perform this efficiently.

Furthermore, my grassfire implementation wasn’t really intended to be a high-performance replacement for cleanupSimple; rather, I wanted to explore distance transforms further. This is why I didn’t apply certain optimizations, such as avoiding the use of .at, in order to maintain simplicity. I believe that if I had implemented cleanupSimple from scratch using a naive approach, it likely would have been slower. This is because grassfire is a one-pass algorithm, whereas the number of passes for cleanupSimple grows with the sum of the erosion and dilation iterations (one pass per iteration).

Ultimately, I decided not to use “cleanupGrassfire” due to performance issues, but implementing the algorithm and seeing it in action was still an insightful experience. It clearly illustrates the usefulness of distance transforms and a dynamic programming approach in optimizing code performance.

Reflection

This project taught me a great deal about simple object recognition. Before working on it, I assumed that recognizing objects was a task so complex that it was reserved for deep convolutional neural networks like YOLO. However, I now understand that both classical and deep learning-based approaches ultimately reduce to the same core idea: generating a feature vector for an image and using it to determine the most similar object. I enjoyed implementing ISODATA, region growing, and Grassfire distance transforms from scratch, as they required interesting algorithms like depth-first search (DFS) and dynamic programming. Additionally, this project reinforced the importance of selecting good features and ensuring invariance in feature representations—both of which are crucial for reliable image comparison. Finally, I was challenged to write performant, clean code to ensure the algorithms ran efficiently on a real-time video feed. This experience not only deepened my understanding of object recognition but also strengthened my software engineering and C++ skills.

Acknowledgements

For this project, the sources I consulted were Bruce Maxwell’s code demos and lectures, OpenCV documentation, and a couple of Stack Overflow posts which were helpful in debugging certain implementation issues.