Course Project

CP467: Image Processing & Recognition
Professor: Dr. Zia Ud Din
Due Date: Saturday December 2, 2023. 11:59 p.m.

Chandler Mayberry 190688910
Samson Goodenough 190723380

Table of Contents

Introduction

All the code for this assignment may be run using the main.ipynb file in the source code folder. Running this file will perform the operations and write the resulting images to the *Detected_Objects*, *Keypoints*, *Matches, and Stitches* folder as .png files.

Note:

- The output images in the Detected_Objects, Keypoints, Matches, and Stitches folder are manually resized smaller to reduce the overall .zip file size uploaded to MLS. The original package size is 5.29GB and after resizing is 1.73GB. This does not visually change any of the output but rather only makes the file smaller for grading purposes. If the original output is desired for any reason, please run the main.ipynb file which will overwrite the folders (expect ~15min of runtime). *Original input files have not been modified*.
- No results are shown within the Jupyter Notebook, matplotlib figures were used for testing only.

Past and Current Methods:

Object recognition is one of the most fundamental tasks when it comes to the field of computer vision as being able to identify and classify objects within images and video has a large context of use cases. As we have learned, there are many past and current methods for performing object recognition, all of which use different strategies to detect objects or stitch scenes together. When it comes to past methods, the following process needed to be employed; there needed to be an input image, followed by feature extraction, matching, classification, and then output the results. Feature extraction contains two components, keypoints and keypoint descriptors. Keypoints are the interest points in an image, whereas descriptors are specific data to match those keypoints across different images. Some of the feature detection methods include Harris Corner Detection, Speeded Up Robust Features (SURF), Binary Robust Independent Elementary Features (BRIEF), Features from Accelerated Segment Test (FAST), Oriented FAST and Rotated BRIEF (ORB), and Scale-invariant Feature Transform (SIFT). These feature detection algorithms all share one common goal, to determine and extract keypoint features and descriptors to be used in image matching. Traditionally, these may also run methods such as support vector machines (SVMs) for more specific image classification uses.

However, these past algorithms all share the same problem of needing to perform feature extraction followed by classification in two separate steps, resulting in less optimized and slower runtimes across an application. Many of the new methods used presently are deep learning algorithms such as machine learning algorithms and/or AI to perform image classification tasks. The key difference in using deep learning as opposed to the past model is

that image feature extraction and classification can happen simultaneously when training a model on a data set.

For example, one of the most notable methods currently used is Convolutional Neural Networks (CNNs) which, by the nature of how they function, can perform both feature extraction and classification concurrently end-to-end. With deep learning models and a dataset, not only can features be extracted naively but can intuitively store important data about an object within model features, such as neurons, in a CNN for future recognition. This means in current models there is no need to purely match keypoints between two images naively, but to recognize patterns between many training images and apply better judgment to image classification problems.

Methods and Algorithms:

In the implementation, we chose to go with OpenCV SIFT and Brute-force matcher to perform the feature extraction and matching between scene and object images (*OpenCV: Feature Matching*). This method to perform object detection seemed powerful, accessible, and overall a relatively straightforward operation from the logic behind SIFT. To generate the keypoints and descriptors for every image a SIFT object is used to get these values using the detectAndCompute method. The Brute-Force object is then used to perform the matching of descriptors between the images, followed by a ratio test to filter out matching and non-matching features.

OpenCV SIFT and Brute-force matching seem to functionally work as expected when compared to the logical steps of SIFT (Ud Din). Please see the logical steps of SIFT and feature matching below:

1. Perform scale-space extrema detection. This is the process of repeatedly running Gaussian filters convoluted together to get a set of scale-space images. Adjacent Gaussian results are then subtracted to output difference-of-gaussian images. After each octave is produced, it is sampled down by a factor of two. The purpose of doing this is to identify the potential keypoint locations across the different scales of images to see if there are either local maximums or minimums.

2. Keypoint localization. In this step keypoints with low contrast or poorly localized keypoints along an edge where the r threshold is less than a predefined threshold value are discarded.

3. Orientation assignment. For every keypoint an orientation is assigned to ensure that the keypoints across images are rotationally invariant, the orientation of keypoints is based on the dominant orientation (peaks in a histogram) of the magnitude of a gradient in each keypoint neighborhood.

4. Keypoint descriptor. At this step, SIFT computes the descriptor for each keypoint based on a 16x16 neighborhood and takes the relative orientation and magnitude. In short, it takes the information about the gradient magnitudes and the orientations and assigns them to each keypoint in its neighborhood.

5. Keypoint matching. By doing the above to a source and destination image, SIFT can find matches based on the keypoint descriptors used in both instances. To find the corresponding matches, euclidean distance is used to find keypoints that have minimum distances between them. The threshold applied is 0.8 by default, we chose to go with a threshold of 0.7 based on observational results from two sample scenes and all objects.

To produce the keypoints shown in the images found in the Keypoints and Matching folder, we simply took the keypoints produced by SIFT detectAndCompute and used cv2.drawKeypoints to draw them onto each scene and object image in blue. To show the matching keypoints between the objects and scenes, we used cv2.circle to draw a red circle at the keypoint location of each close match produced by the Brute-Force matcher and ratio test.

For detecting objects in the scene and drawing a box around them we found an example in the OpenCV documentation which used cv2.findHomography and cv2.perspectiveTransform on the object and scene keypoints to draw a box around an object (*OpenCV: Feature Matching + Homography to Find Objects*). This method works by taking the keypoints of both the scene and object to find the perspective transformation of the object in the scene planes using findHomography. The mask that is returned is then used in the perspectiveTransform function to orient and place the object box in the correct location on the scene. To add the text label to the bottom of the box, we found a function in the documentation called cv2.putText that places text on an image providing the destination, text, and location (*OpenCV: Drawing Functions*). For the location to place the text we used the box output position from the cv2.perspectiveTransform and specified it to be the bottom left of the matrix.

To produce the matching output images, we simply took the matching key points found for each respective object and combined the images using matrix manipulation to have the scene on the left and the object on the right. On each run, a raw version of the scene is given to the matching function to only produce scenes with each object individually rather than all together such as a scene in the keypoints folder.

Lastly, for object stitching we use cv2's *Stitcher Class*. The *Stitcher Class* or "High level stitching API" makes use of one of two built-in stitching models, either the Homography model or the Affine model. Using the default Homography model we pass our images through the stitcher which applies pairwise matching and homography estimations to make transformations.

Task 1 - Empirical Detection Results:

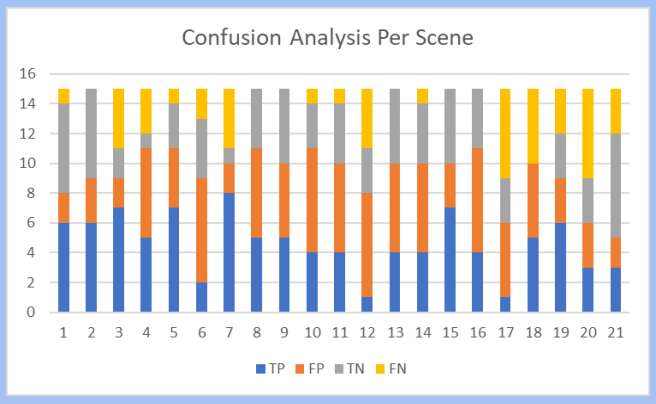
Scene	TP	FP	TN	FN
1	6	2	6	1
2	6	3	6	0
3	7	2	2	4
4	5	6	1	3
5	7	4	3	1
6	2	7	4	2
7	8	2	1	4
8	5	6	4	0
9	5	5	5	0
10	4	7	3	1
11	4	6	4	1
12	1	7	3	4
13	4	6	5	0
14	4	6	4	1
15	7	3	5	0
16	4	7	4	0
17	1	5	3	6
18	5	5	0	5
19	6	3	3	3
20	3	3	3	6
21	3	2	7	3

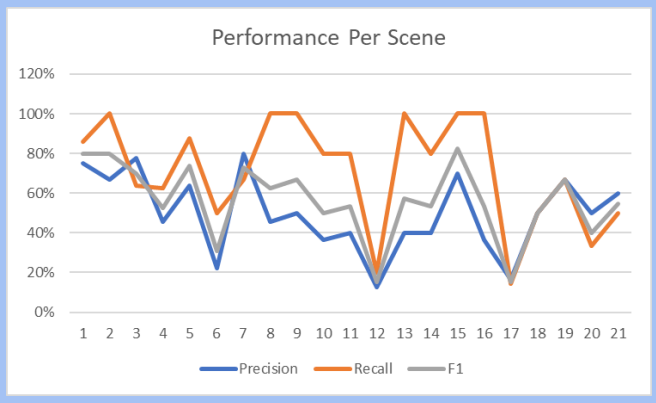
The above was found by stepping through each scene and using the following table.

		REAL	
		POSITIVE	NEGATIVE
PREDICTED	POSITIVE	TP In scene, identified	FP Not in scene, identified
	NEGATIVE	FN In scene, not identified	TN Not in scene, not identified

Analysis of Results:

Scene	Precision	Recall	F1
1	75%	86%	80%
2	67%	100%	80%
3	78%	64%	70%
4	45%	63%	53%
5	64%	88%	74%
6	22%	50%	31%
7	80%	67%	73%
8	45%	100%	63%
9	50%	100%	67%
10	36%	80%	50%
11	40%	80%	53%
12	13%	20%	15%
13	40%	100%	57%
14	40%	80%	53%
15	70%	100%	82%
16	36%	100%	53%
17	17%	14%	15%
18	50%	50%	50%
19	67%	67%	67%
20	50%	33%	40%
21	60%	50%	55%
Average	50%	71%	56%





Note: if most (~70%) of the object is not present in the scene, we consider it ***not*** *in the scene*.

Some scenes worked particularly well compared to others from observation. Looking back through them after performing empirical analysis on the image matching, it is clear that the algorithm performs best when detecting objects with a high amount of detail and that appear in a similar viewing angle to the reference images.

This viewing angle discrepancy is because SIFT, while it does ignore rotation and scale, only accounts for skews in out-of-plane rotation up to 60 degrees (Ud Din).

Scene 15

Scene 15 for example, with the highest F1 score at 82%, features many high-detail objects (those that feature text work best) and has a nearly head-on viewing angle of all the detectable objects in the image.

In addition, all objects in the image are in focus, allowing more precision on finer details.

The combination of all these aspects makes this scene a prime example of what this algorithm excels at.

Task 2 - Scene Stitching and Object Detection

To begin stitching we first need to organize all the scenes in our data into groups that have invariant camera positions but variant viewing angles. This will reduce the amount of artifacting and seams created on the image.

Group #	Scenes
Group 1	S1 - S6
Group 2	S7
Group 3	S8 - S14
Group 4	S15 - S19
Group 5	S20 - S21

Looping over each group we use cv2's *Stitcher Class* to stitch each group into a larger image.

Stitched Image of Group 3 (*/Stitched/group_3.png*)

After collecting all of the stitched images we can pass them back into our previous keypoint detection and boxes algorithms to identify the objects in the image.

Object Detected Stitched Image of Group 3 (*Stitched\group_3_detected.png*)

Challenges Faced and Addressed:

During the development of our implementation of SIFT matching, producing boxes around objects, keypoint detection, and image stitching, we encountered a few problems with the most notable below.

1. First, we encountered a problem where there were too many or too few keypoints being discovered using the default threshold in our matching method, causing us to have very specific and skewed results. To rectify this, through observation and graphing the quantity of keypoints for each object on each run, we learned that having a matching threshold of 0.7 compared to SIFT paper default of 0.8 and a minimum keypoint threshold of 28 was the most ideal parameters to reduce this issue in testing.

2. A smaller but notable problem was after SIFT operations the output image was producing a grayscale image rather than an RBG image. We determined this to be the side effect of using SIFT, which in its operation converts the image to grayscale as a prerequisite but does not convert it back during output. To fix this we convert the image back to cv2.COLOR_BGR2RGB in every instance before writing the image to disk.

3. Another challenge was trying to figure out how to put the corresponding object name on the detected object box in the Dectect_Objects operation. While seemingly simple, during operation, it was difficult to grab the name with our initial structure. After some testing, we found that storing all of the object and scene images in a dictionary allowed for quick name lookups using object/scene ID, which was then placed using putText at the bottom of each object box.

4. Lastly, the greatest challenge we faced was poor runtime when running all operations of the code, with each segment on each scene taking 1-3 minutes to run causing total runtime to be well over 30 minutes. To optimize the code, we changed the structure to run all of the methods simultaneously, being careful to only calculate or load in data once. An example of this would be the object keypoints and descriptors which initially were being run again for every scene but converted to run once as a preprocess in the method all_keypoints_on_objects. Overall, this reduced our total runtime for object detection to less than 15 min on one machine and 6 min on the other.

Possible Improvements and Future Work:

Due to our method choice and implementation, there is much room for improvement to make this task better in a few ways. One of the challenges we faced and did not have the time to improve was drawing the boxes around objects in a scene. While drawing the boxes was not perfect, the primary concern was drawing boxes of objects not in the scene. This is the nature of the classification method we chose, as in the implementation would generate a box around an

object even if its a false positive. This comes with the added side effect of having boxes stretched across the scene (visual representations are random lines). This could be fixed in future iterations by either having a better classification model overall or perhaps performing something like clustering/segmentation of objects in a scene to more closely outline keypoints in a neighborhood.

With that being said, another space for improvement would be defining smaller regions for keypoints to cluster. We noticed that in many of the matching scenes, either for true positives or true negatives, there are times when keypoints are very far away from the object or sparsely placed. Having a method to see the density of an area and omit completely sparsed keypoints would reduce the amount of false positives and make the true positives produce better results from observation.

Lastly and as mentioned above, SIFT is more of a past or complementary method of image classification, commonly paired with other methods such as support vector machines. A recurring issue we found when observing our results is that SIFT fails when it comes to illumination changes between images, camera viewpoints changing, and clutter between objects in a scene. This makes it a relatively poor option in the real world and real-time use without time-consuming feature extraction and manual tweaking of values. With the emergence of accessible and understandable machine learning models, something such as a CNN would likely be a much better approach and improvement overall. These models can be taught with less manual tweaking, are intuitive, and in many cases have a much better runtime.

Works Cited

OpenCV: Drawing Functions.

docs.opencv.org/4.x/d6/d6e/group__imgproc__draw.html#ga5126f47f883d730f633d74f07456c576.

OpenCV: Feature Matching. docs.opencv.org/3.4/dc/dc3/tutorial_py_matcher.html.

OpenCV: Feature Matching + Homography to Find Objects.

docs.opencv.org/3.4/d1/de0/tutorial_py_feature_homography.html.

Ud Din, Zia. *Feature Detectors and Descriptors Part 2: Feature Description and Matching*.