1 Introduction

With the unprecedented increase of digital media in the past few decades, the number of photographs in existence has grown exponentially as time has passed. With film photography, each image was precious, taking up valuable space on a roll. But with digital photographs, this limitation is essentially gone, and people can now take a nearly unlimited number of photographs at any moment. Photos come off the camera, and are stored in a hard drive, unclustered and forgotten. Thus, the goal of this project is to make a photo sorter which will be able to group similar, or near-duplicate, images to find that "perfect shot".

1.1 Project Goals

With this project, we hope to design a photo sorter with a simple user interface. The user will be able to select a group of photos as input. With the input, we hope to use object detection to find images containing only specific things (such as people, cars, etc.), and to filter our input based on these. Finally, the program will be able to simply sort the selected images by similarity, organizing the cluttered input into a nice clustered output.

1.2 Approach

In order to build a near-duplicate photo sorter, we decided to use feature detection to match similar images together. We believe that near-duplicate images will have a high number of matching features, and we will be able to have the user decide a threshold on how many matching features will be considered a near-duplicate image.

To detect objects in the images, we plan to use a pre-trained neural network, trained to detect common objects in context. We plan to use a fast model which can be used in real-time image detection, since we want to be able to quickly detect objects for a large number of input images in a short amount of time. We plan find a model which is fairly resource efficient, despite likely sacrificing some accuracy, to help with the overall performance of our

application.

We believe that with this approach we can get decent accuracy without hurting run-time too much. the feature detection will likely work well but depend upon the threshold that we set for what is a match. When this threshold is set intelligently, it should produce results that are fairly accurate with only a bit of error.

2 Results

For 15 images, recomputing both features and the matches:
Runtime Normal: 63.152354764s
Runtime Matrix: 127.497919693s

For the same 15 images, loading the features and recomputing the matches:
Runtime Normal: 39.246707543s
Runtime Matrix: 107.122888122s

For the same 15 images, loading the features and the matches from file:
Runtime Normal: 0.663096192s
Runtime Matrix: 0.562453343s

As the timing results show, the majority of the runtime is spent on the feature matching as loading the features only reduced runtime by 37.85% whereas loading both features and matches reduced it by 98.95% for the normal version of the algorithm. For the matrix version of our algorithm, these were a 15.98% and 99.56% reduction respectively. This shows that the ability to save the number of matches between images is very beneficial to optimizing runtime. Our matrix implementation lets you take advantage of this reduction to half a second runtime when tunign the threshold as it will always be around this runtime despite changing the threshold. With the normal implementation, if you changed the threshold it would take another 107s to recompute the required matches.

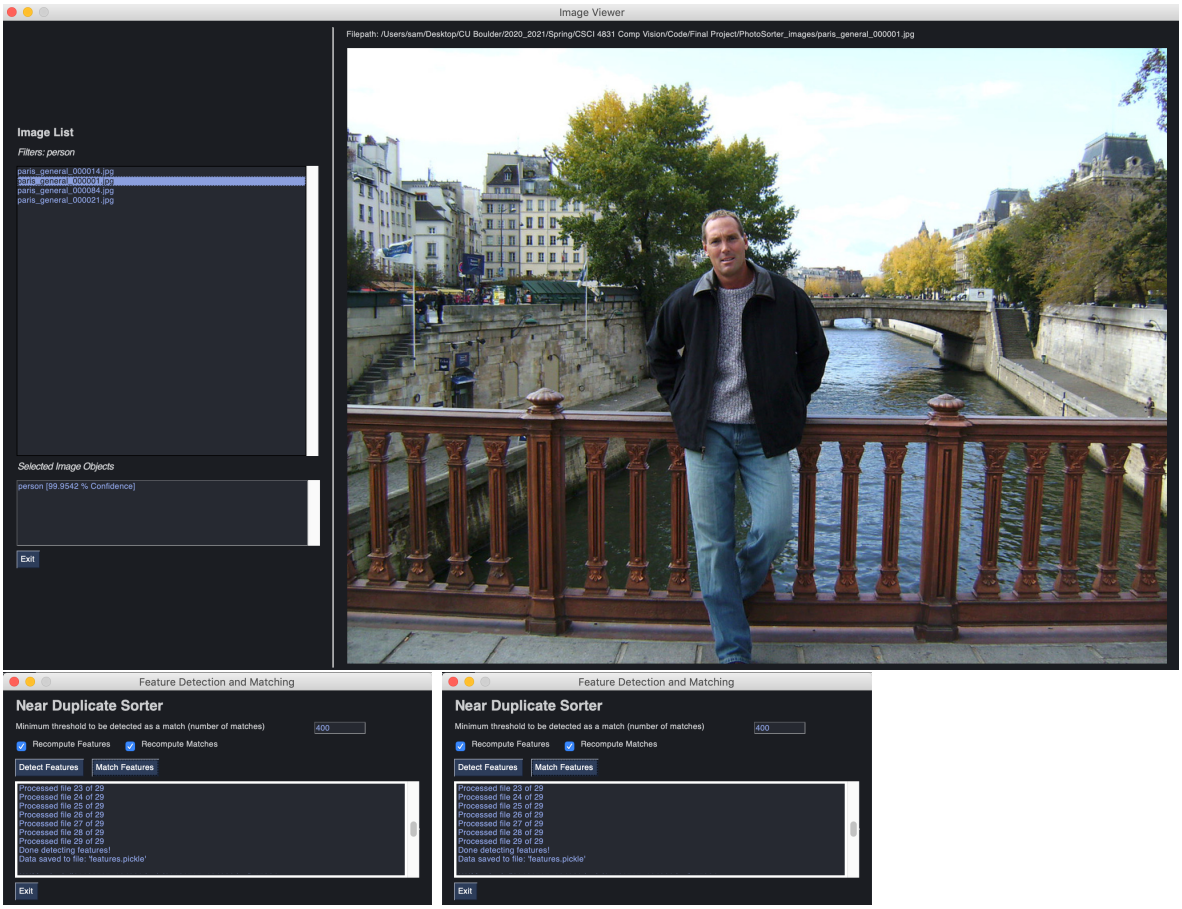
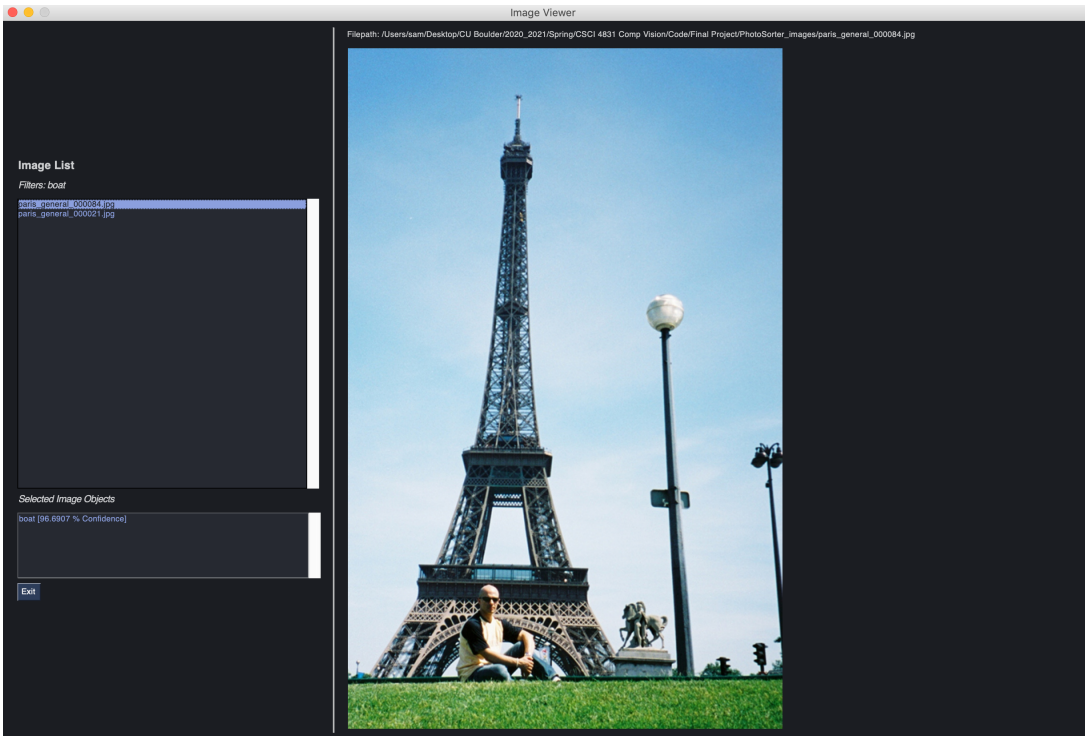


Image Viewer
Filepath: /Users/sam/Desktop/CU Boulder/2020_2021/Spring/CSCI 4831 Comp Vision/Code/Final Project/PhotoSorter_images/paris_general_000001.jpg
Image List
Filters: person
paris_general_000014.jpg
paris_general_000001.jpg
paris_general_000084.jpg
paris_general_000021.jpg
Selected Image Objects
person [99.9542 % Confidence]
Exit

Feature Detection and Matching
Near Duplicate Sorter
Minimum threshold to be detected as a match (number of matches)
400
Recompute Features
Recompute Matches
Detect Features
Match Features
Processed file 23 of 29
Processed file 24 of 29
Processed file 25 of 29
Processed file 26 of 29
Processed file 27 of 29
Processed file 28 of 29
Processed file 29 of 29
Done detecting features!
Data saved to file: 'features.pickle'
Exit

Feature Detection and Matching
Near Duplicate Sorter
Minimum threshold to be detected as a match (number of matches)
400
Recompute Features
Recompute Matches
Detect Features
Match Features
Processed file 23 of 29
Processed file 24 of 29
Processed file 25 of 29
Processed file 26 of 29
Processed file 27 of 29
Processed file 28 of 29
Processed file 29 of 29
Done detecting features!
Data saved to file: 'features.pickle'
Exit



3 Computer Vision Methods

3.1 Near Duplicate Matching

We attempted using two different feature detection algorithms in our project. First was Scale-Invariant Feature Transform (SIFT), and the other was Oriented FAST and Rotated BRIEF (ORB). We tried using ORB as it was branded as an "efficient alternative to SIFT or SURF". Both are implemented in the code but only SIFT is actively used due to issues encountered with ORB.

In the end, our near duplicate photo sorting algorithm uses Scale-Invariant Feature Transform (SIFT) to do keypoint and description generation for the features in the image. In OpenCV's SIFT implementation, feature descriptors are a matrix of neighbors to the feature pixel. These can then be matched using a nearest neighbors algorithm to determine feature matches.

OpenCV has a brute force matcher (BFBatcher), and a Fast Library for Approximate Nearest Neighbors matcher (FLANN). Our program's feature matching uses a FLANN based matcher to compute feature descriptor matches due to the increased speed that FLANN

provides on large data sets with high dimensional features. We are using FLANN with the FLANN_INDEX_KDTREE algorithm and 5 trees. FLANN runs k nearest neighbors with k=2 on the descriptors for the two images it is passed and returns a list (in this case of the best 2) descriptors that matched for each descriptor from a feature in the first image.

From this list of feature matches, they get narrowed down using a ratio test, eliminating all but the best feature matches with a ratio of 0.7 (ie: our best match must be sufficiently different from the second match to keep it). This helps to prevent ambiguous matching of features, improving accuracy.

3.2 Object Detection

3.3 Limitations and Future Extensions

One major limitation of our implementation of the near duplicate matching is in its run time. Despite using the faster FLANN based matcher for kNN, the algorithm still takes a long time to run for checking each image against another (which has to be done many times). This means that the more images it must check, the longer the program will need to run to do the matching for each image.

In addition, when attempting to use the apparently more efficient algorithm ORB, we hit issues with it not producing enough feature matches to effectively match images together. It produces feature matches numbering in the 20-150 range, whereas when matching an image using SIFT features, it typically had between 600-50,000 feature matches depending on the image. This meant that accuracy with ORB was a problem and lead to a lot of miss-matching of images when doing the nearest neighbor matching.

To attempt to fix this issue we implemented 2 different versions of the matching algorithm, one that used a dictionary of matched images and the other that used a matrix. The dictionary was fast as it would eliminate checking an image against any others once it had been matched once. Though this had the trade off that if the cutoff threshold for a match was changed, all the data would have to be recomputed as that threshold determined when images were eliminated from the algorithm. The matrix method had a couple qualities that we liked despite it having much worse performance in terms of processing time. In the matrix method, we compute an upper triangular matrix of all the numbers of matches between the different combinations of images. This allows us to re-sore/filter the data when changing the match threshold without needing to recompute any of the values in the matrix. Because

our algorithm was designed to save the data to a pickled file once it was computed, this had the advantage of speeding up tuning of that threshold and future sorting at the expense of a drastically increased initial run-time.

Our biggest improvement we could make in the future would be to multi-thread the matrix method. Since each row of the matrix is independent, they can be run individually then combined afterwards. Adding this multi-threading to the matrix method would drastically increase run-time performance while giving the benefits of not needing to recompute data.

4 Analysis

4.1 Individual Contribution

4.2 Unsuccessful Attempts

4.3 Lessons Learned

4.4 Advice for Future Students

References