Location Reviews Using Cloud-based Landmark Recognition

Leo Alterman, Holly Ho, Aaron Jaffey
Department of Electrical Engineering, Stanford University
{leeoo, hollyho, ajaffey}@stanford.edu

Abstract—We have built a service that allows users to take photos of restaurant facades using a smartphone and receive reviews of said restaurant. The service uses a vocabulary tree trained with a database of restaurant images as its core matching algorithm, further refined using GPS data and RANSAC matching. The training database itself is built from these query photos, so the vocabulary tree is periodically retrained to incorporate new data. Since the training images are automatically collected and may be of dubious quality, we reduce the effect of image noise by weighting descriptors found across many images of the same restaurant more heavily than descriptors that appear in fewer. The result is a self-growing, self-maintaining training set of landmark images that can be used to retrieve rich content like Yelp restaurant reviews.

1. INTRODUCTION

As internet-enabled smartphones and social review websites become increasingly common, it seems natural that one might want to use the former to browse the latter. The motivation for our project stems from the idea of quickly finding reviews of a restaurant without needing to type in a search query. With our service, a hungry user walking down the street can just whip out their phone, snap a quick picture of different restaurants, and immediately receive restaurant reviews.

Prior work on street-level building recognition has been shown to be feasible at large scales using vocabulary trees with reverse indexes, but had the disadvantage of requiring a carefully processed and high quality training set to be built by the authors ahead of time [2]. Further, this training set would need to be manually maintained as its pictured environments changed over time. Stores update their window displays, restaurant faces get renovated, buildings get demolished. It would be nice for these changes to be folded into the training set without manual maintenance by the service’s developers.

Our service, consisting of an Android application and scalable server backend, attempts to realize this vocabulary tree with an “evolving training set” by incorporating the users’ query images into the training set itself after they have been matched by our service and verified by the user. Our algorithm uses a vocabulary tree for initial matching, refines the results with the user’s GPS coordinates and a RANSAC false-positive rejection stage, and then returns the result (with Yelp reviews) to the user. If the match was incorrect the user can report the true location to the server, further improving the training set.

Since we end up with many images of the same thing, we reduce noise (pedestrians occluding the restaurant, window reflections, etc.) by correlating features between images of a location and weighting the more common features more heavily. This method opens up many possibilities to perform more accurate recognition, only a few of which we had time to explore in this project.

In the remaining sections we describe our image recognition pipeline, starting with the client and client-facing web server in section 2.1 and getting into the details of the recognition algorithm in section 2.2. In section 3 we discuss our testing methods and interpret the results taken from those tests, and in section 4 we look at interesting next steps to take with the existing code base. Finally, in section 5 we summarize our work and conclude.

2. PROCESSING PIPELINE

2.1. User client and HTTPS request server

2.1.1. Android application

We created an Android application to provide a convenient user interface for interacting with our system. The application uses the Android camera interface to provide a preview
window when the app runs. When the user presses the identify button, a photo is taken. To provide a real-time experience, we designed the identification process to run quickly. The app takes this bitmap image and transfers it into a buffer, down-sampling by 2 in each direction to save memory and improve performance. The image is then compressed into JPEG format. The app also acquires GPS data. A separate thread is then launched to handle generating and sending an HTTP POST request to upload the image to the server. We send the image and GPS coordinates to a Python webserver.

2.1.2. HTTP front-end

The webserver receives the request. Using the specified GPS data, it retrieves a list of nearby locations from Yelp using their API. These provide the user clickable choices of locations, if the image processing is not able to classify the locations automatically. We add to this list of choices any nearby locations that the user has manually entered (if they were not in Yelp). We use a SQLite3 database (with the Spatialite GIS extension) to manage communication between the webserver and matching daemon, as well as maintain persistent storage. The webserver creates an unmatched entry for the image in the database. It then uses a POSIX message queue to tell the matching daemon to wake up and classify this image. Matching is described in section 2.2. The matching daemon modifies the database entry for the image with its classification, and then this value is read by the webserver and returned to the Android app via a JSON response. If the classification was successful, the app displays data from Yelp about the identified location. Otherwise, the user is prompted to choose the location, which trains the database.

2.2. Image Recognition Pipeline

2.2.1. SIFT keypoints and descriptors

When first adding an image to our database, we extract the SIFT keypoints and descriptors using functions from the VLFeat library; we chose to use SIFT because of its scale and rotation invariance. After extracting information from the image, we discard the image, and store the keypoints and descriptors on disk for later use.

2.2.2. Vocabulary tree

To produce our initial list of matches we use a vocabulary tree of branch factor and depth 6, with reverse indexes at each leaf. We push each query descriptor down the tree and then observe the list of training images that also had descriptors falling on that leaf. For each image in this list, we increment its “match score”. The amount we increment this score varies for each training image and is described later in section 2.2.5. At the end of this stage, we produce a list of matching training image ID’s sorted by their match score.

2.2.3. GPS filtering

We filter the list of locations mapped to scores by only considering locations as possibilities when they are located within a certain radius of where the image was taken. We chose a value of 1km to use as the radius, but a smaller value would also work well (ideally this would scale with the reported GPS accuracy). Locations not within the radius are removed from the score list because it is impossible that they would be correct (unless the user’s GPS coordinates were reported incorrectly). This helps avoid misclassifying locations that may look very similar, such as two Starbucks cafes.

2.2.4. RANSAC filtering

Once we have filtered by location, we narrow the set to the final location by applying RANSAC to the top five candidates. We first use a FLANN based matcher provided by OpenCV to match the descriptors of the query image and the candidate image. We experimented with two different matching algorithms provided by OpenCV: the FLANN based matcher and the brute force matcher. We ultimately chose to use the FLANN based matcher because of its faster speed. After
computing the matches, we record the minimum and maximum distances of said matches. The matches are then filtered by distance — matches with distances greater than halfway between the minimum and maximum distances (i.e. \( \geq \min_{\text{dist}} + 0.5*(\max_{\text{dist}} - \min_{\text{dist}}) \)) are discarded. The good matches are then used to find a homography between the keypoints of each image. Using the resulting homography mask, we count the number of inliers normalized by the number of good matches. The image with the greatest number of normalized inliers is chosen as the final match.

2.2.5. Training the database

Once we’ve determined the final location of the image, it will be included in the training set the next time we rebuild the vocabulary tree. We rebuild the tree approximately every 30 seconds, though as the database grows this period could be made much longer at the cost of immediate accuracy.

To rebuild the tree, we start by scanning over every location in the database and set the descriptor weights for each photo of the location. To do calculate them, we iterate over every possible pair of images of a location and use OpenCV’s FLANN matcher to correlate descriptors. We record how many times each descriptor finds a match, and then obtain the descriptor weighting by normalizing to the number of images in the location set. This weight essentially tells us how “good” a descriptor is at describing a location: descriptors extracted from things like a restaurant’s sign will show up in many images of the set, while descriptors extracted by transient things such as people walking by or window reflections will only show up once. This weighting is illustrated in Fig. 4, where keypoints are colored based on how common they are across the set of 3 similar images. Note that the introduction of a new object to the scene generates descriptors with exclusively low weights.

![Figure 4. Descriptor weighting. Blue keypoints are "uncommon", red keypoints are "common".](image)

To build the tree, we push each descriptor of each image down the tree and add the descriptor’s weight to the image’s record in the reverse index. We again normalize this value by the sum of all the weights in the image, to prevent “cold” images with many low weights from being overshadowed by “hot” images with many high weights. In the degenerate case that all weights are 1, this reduces to normalizing by the number of descriptors in the image (a common practice in basic vocabulary tree implementation [1]).

2.2.6. Feedback and false positives

In the event that the system produced an incorrect match, the client app provides a ‘Wrong location’ button that presents the user with a list of nearby locations, culled using GPS coordinates. The user selects the location they thought their photo was of, and the image’s location in the database is then changed accordingly. The next time the vocabulary tree is built the image will then be part of the correct location set.

Similarly, if the user takes a photo of a new location, the photo may fail to produce any match. The app will ask the user to explicitly label the location, again using a GPS-culled list, and the image will be included appropriately in the next tree rebuild.

3. TESTING AND OBSERVATIONS

We validated our system using a few different methods. In order to test the accuracy of classifying images, we used a modified cross-validation scheme. We first produced a test dataset of 200 images of restaurants spread across 25 locations. Each image was taken from a different point of view. Some images were more similar to each other, and others were taken from completely different sides of buildings. With this data set, we tried to simulate the behaviors of prospective users. For our tests, we did not experiment with the use of different resolution cameras, but this is unlikely to have a large impact on accuracy because we use SIFT, which is scale-invariant.

Consistent with the cloud-based training process that our system provides, we performed incremental tests as follows: Start with one \( l \times p \) matrix \( O \), where \( l \) is the number of locations in the dataset, and \( p \) is the number of photos in each location. Another \( l \times n \) matrix \( T \) will hold the training data.

For \( n := 1 \) to max number of samples per location tested
1. For each location (row) in \( O \), pick one element randomly, remove it from \( O \), and add it to the same row in \( T \). \( O \) becomes \( l \times (p-n) \), and \( T \) is \( l \times n \).
2. Train the classifier (vocabulary tree) on \( T \).
3. Classify all of the images in \( O\backslash T \), and report accuracy.

The above process was repeated 12 times for our data, and we computed averages across each \( n \) value for the different runs. This gave us estimated classification accuracies for different dataset sizes. We ran the entire process for different configuration parameters, as well, to compare accuracy when not using descriptor weighting, or not using RANSAC. For these test data, we did not use the location filter because all of the buildings were nearby each other. In actual use, buildings would likely be found in clusters by location in the database. In this case, the accuracy of a classification would likely reduce to the accuracy of classifying within a smaller space given by the number of buildings in the area. We discovered
that our various different configuration parameters had minimal effect on the classification accuracy. The number of images in the training set for each location had a much stronger positive correlation with accuracy (Fig. 5). Although with our test data, classification accuracy tended toward 100% as the number of training images in each location increased, we expect that in a real scenario, accuracies will be lower, due to variations in lighting, poor photo taking, etc.

In addition to verifying the accuracy of our classification scheme, we also tested our system integration by performing numerous queries using an Android phone, and ensuring that the platform worked together quickly and reliably.

Figure 5. Results from testing location recognition accuracy.

4. FUTURE WORK

We only began to explore the variety of things one could do with a self-growing vocabulary tree and image sets. Given more time and a larger dataset, we would have liked to try some of the following improvements/extensions:

- As locations change over time, older images will tend to produce less strong matches and eventually might even cause misidentifications. By bounding the size of a location’s image set, older images will automatically get thrown out as new images come in and the server can avoid storing redundant or, worse, inaccurate data. This would also keep processing times down.

- Photos of the same location can vary greatly when taken at different times of day. There are a number of ways to compensate for this, the simplest being to separate a location’s image set into discrete ‘day’ and ‘night’ sets and retraining the tree using one or the other based on the current time of day. More advanced algorithms could attempt to find correlations between these two sets in order to use their data at all times of day.

- At the moment, our weighting algorithm just looks for correlations among descriptors but ignores geometric/keypoint data. We found that further filtering with RANSAC didn’t improve our weights particularly much and was too computationally intensive to run periodically on each image pair. Nonetheless, some sort of geometric correlation could produce better descriptor weights if optimized correctly.

On the application side of the interface, we would have liked to provide information from multiple location databases, such as user reviews from Google Maps and the Zagat restaurant guide. These services could be easily layered on top of our existing infrastructure and greatly enrich the amount of information users could retrieve from their photos.

Furthermore, we believe this could become a very useful and widely adopted Android application, especially with the extensions and improvements stated above. If we were to release this to the Google Play Store, our application icon would look like Fig. 6.

Figure 6. Android application icon.

5. CONCLUSION

We have developed a service that allows users to photograph restaurants “on the street” and be served reviews retrieved from Yelp. We perform location recognition on custom-developed server software that uses a Python HTTP server to interface with clients and a constantly running C++ daemon to maintain and query a training image database. Our recognition pipeline uses a vocabulary tree built from previous query images and refines its results with GPS coordinates and RANSAC inlier counting. We introduce the idea of descriptor weighting across sets of images of the same location to improve the quality of the user-build training set database. While our descriptor weighting prototype provides no particular improvement in recognition quality, we believe that it opens up a variety of approaches that might yield better results and certainly provide better scalability.

6. ACKNOWLEDGEMENTS

The authors of this paper would like to thank David Chen for all of his help and advice, and Professor Bernd Girod for his enthusiasm and support.
7. REFERENCES


8. DISTRIBUTION OF WORK

Leo Alterman - Vocabulary tree matcher, descriptor weighting algorithm, C++ daemon infrastructure

Holly Ho - RANSAC match filter, test data acquisition

Aaron Jaffey - Android application, python HTTP interface, GPS match filter, test design