Scalable Image Matching and Similarity Detection For Localization

Chris Stathis and Yongchen Jiang
EECS E6893 Big Data Analytics, Fall 2014
Columbia University
cgs2135@columbia.edu, yj2338@columbia.edu

***Abstract*—We explore the task of determining the most similar image to a given query image among a database of up to millions of candidates using a bag-of-features clustering approach. Big Data tools and techniques for storage and parallelized processing are leveraged to allow scaling beyond the capacity of a single computer. As a model application, we apply our system to the problem of image localization, where camera imagery is checked against a library of geo-tagged pictures of the environment to determine the location of the photographer. We find that with large datasets, this approach produces good and repeatable results for queries that are highly similar to a database image, and lends itself well to massively-parallel processing.**

I. Introduction

Image matching is a computer vision task with a variety of practical applications. Among them is autonomous mapping and localization by unmanned air and ground vehicles, where landmark recognition from sensor imagery is integrated into the system's knowledge of its environment and its position within [1], [2].

Several robust matching algorithms exist today, the most well-known being the Scale-Invariant Feature Transform (SIFT.) SIFT and similar techniques are typically too computationally intensive to be performed, especially in real-time on embedded platforms such as robots. As a result, scientists have looked to offload such processing to external computers that are not as performance limited. This brings its own host of challenges, and the topic of cloud robotics has become a topic of popular research [3]. This also has application to automatic scene description for a human-facing system such as Flickr.

In this report, we explore the application of "Big Data" distributed storage and processing techniques to image matching for efficient similarity-based retrieval of images from a large database. Using Hadoop and related tools such as Pig, we demonstrate population of a database of image features in a massively-parallel and scalable fashion using the MapReduce framework, and explain how distributed computing systems can be leveraged to keep processing time at acceptable levels as the database scales beyond millions of images. We also discuss what image feature extraction and feature matching algorithms are suited for this task.

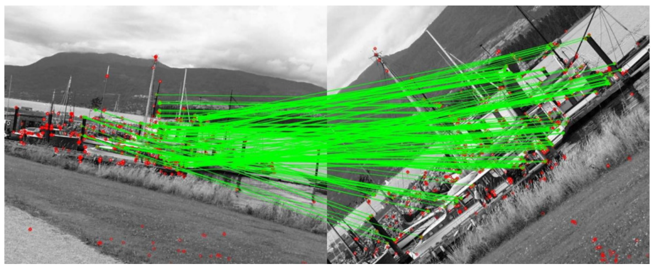


Figure 1. Image of a dock and its rotated counterpart with ORB features and their correspondences, computed and rendered with OpenCV. The red dots represent keypoints, and the green lines connect keypoints in the two images that are similar. This illustrates the orientation invariance of the ORB algorithm. The same example image matching is discussed further in ref. [4].

II. Theory and Algorithms

A. Image Feature Extraction

A zoo of techniques exist to perform image matching. In general, however, matching systems follow the same three steps:

1) Detect differentiating features (i.e. locations of keypoints) from images
2) Compute a descriptor to represent the features in memory in a distinctive way
3) Compare the distance between descriptors according to some distance measure, and look for correspondence by descriptors that are close to each other

The theory of how steps 1 and 2 are done is beyond the scope of this paper, however some discussion on how and why to choose one algorithm over another for our application will be helpful.

The classic SIFT algorithm and its cousin ASIFT (Affine-invariant SIFT) are attractive because their descriptors are invariant to various transformations. This means that images of the same object from different perspectives will likely match well. Unfortunately, such algorithms are complex and are not fast enough for most real-time applications. A large body of work has been done in the last ten years to find more efficient ways to detect keypoints and compute descriptors. Among them are FAST (Features from Accelerated Segment Test), an efficient corner detector, and BRIEF (Binary Robust Independent Elementary Features), a

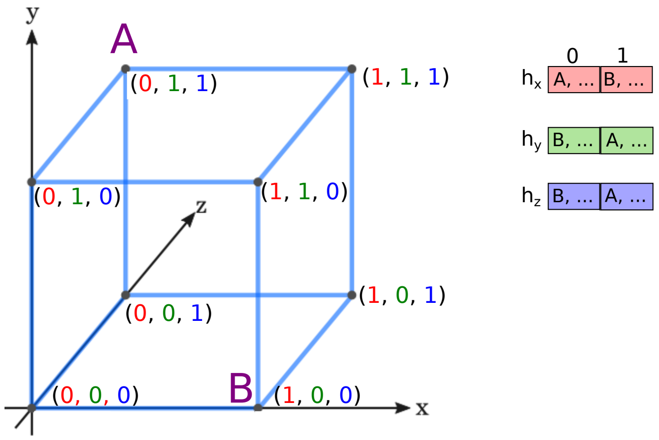


Figure 2. A simple example of locality-sensitive hashing using the Hamming distance for binary strings representing the coordinates of the corners of a unit cube. The coordinates are binned according to their values into hash tables for each of the 3 coordinates.

binary descriptor calculator [5], [6]. As discussed in the next section, binary descriptors are strongly preferred to floating-point vector descriptors because they can be compared much more efficiently. The lack of any sort of transform invariance is a major drawback, however. An algorithm that attempts to fix this is called ORB (Oriented FAST and Rotated BRIEF) [4]. ORB introduces orientation invariance and some performance enhancements to FAST and BRIEF, and provides binary features. Fig. II-A illustrates keypoint correspondence (e.g. a good match) between two example images.

B. Descriptor Matching

Descriptor matching is fundamentally a nearest-neighbor (NN) search problem. Descriptors extracted using preferred techniques are typically of high dimensionality - 128 dimensions or greater in some cases - so the efficiency of the chosen NN algorithm and the efficacy of the distance measure are critical. For matching against small collections of images, conventional space partitioning and clustering methods such as kd-tree and k-means clustering have been found to be effective. However, as sample size increases, these methods quickly become a computational burden and approximate algorithms must be used for real-time applications [7].

In recent work, it has been shown that clustering methods can be used to avoid calculating nearest-neighbor distances against database imagery directly. In this "bag-of-features" representation, a vocabulary is constructed that bins many image descriptors into the same category according to their similarity [8]. Locality-sensitive hashing (LSH) has been shown to be an effective tool for binning image descriptors for such an application [9]. This is the approach taken in our implementation. The mathematics of locality-sensitive hashing are briefly summarized as follows.

Figure 3. Example Google StreetView data used for this project. A script powered by Google's Maps API produces a list of (latitude, longitude) points along a street and captures images pointing at both sides of the sidewalk along the street.

A function family $H = \{h_1, h_2, ..., h_n : S \to U\}$ is called locality sensitive for distance measure D if the following conditions hold [10]:

1) If $D(q, p) \leq r$, $P_H[h(q) = h(p)] \geq p_1$
2) If $D(q, p) > cr$, $P_H[h(q) = h(p)] \leq p_2$

for features q and p, neighbor radius r, and associated probabilities p_1 and p_2. In other words, H is a family of functions such that p and q have a probability of hashing to the same value of: (a) at least p_1 when the features are within a distance r, and (b) no greater than p_2 when the features are a distance greater than cr away.

LSH families for a variety of measures are known [10]. The simplest realization is for the Hamming distance, the measure of the number of bits in two binary strings that differ. It can be shown formally [10] that bit-sampling functions form an LSH family for the Hamming distance - that is, the family of N functions that select the i^{th} bit from a binary string $x = \{0, 1\}^N$

$$h_i(x) = x_i, i \in 1, ..., N \tag{1}$$

Fig. 2 illustrates an example of hashing the coordinates of corners of a unit cube by this approach.

C. Approximate Nearest-Neighbor Search with LSH

To perform nearest-neighbor search with LSH, we construct L hash tables each representing the concatenation of k functions selected uniformly and at random from H.

Algorithm 1 LSH Image Matching

```
 1: procedure MATCH
 2:     for queryFeature in queryImage do
 3:         for table in hashTables do
 4:             hash ← table.getHash(queryFeature)                         ▷ Apply the bit mask
 5:             bucket ← table.getBucket(hash)                             ▷ Get bucket of features with hash
 6:             for databaseFeature in bucket do
 7:                 dist ← Hamming(queryFeature, databaseFeature)          ▷ Compute distance
 8:                 if dist ≤ epsilon then
 9:                     record (databaseFeature.ImageID, dist) in FeatureList    ▷ Record neighboring features
10:                 end if
11:             end for
12:         end for
13:         bestMatch = recommend(FeatureList)              ▷ Get name of image with the nearest feature
14:         record bestMatch in MatchList
15:     end for
16:     overallMatch ← recommend(MatchList)                 ▷ Get name of image with most nearest features
17: end procedure
```

Specifically, we construct L functions of the form

$$g_j(q) = (h_{1,j}(q), ..., h_{k,j}(q)), j \in 1, ..., L. \quad (2)$$

In the context of Hamming distance, these L functions g_j are randomly-selected bit masks of size k applied to the query. Strings that are equal after this mask is applied are hashed to the same bucket.

Tuning k and L trades complexity and accuracy. Taking $k = N$ (the size of the queries) and $L = 1$ would hash only exact matches, placing features into the same bucket if and only if they are identical. In the other extreme, if we choose $k = 1$ and large L, then features are hashed based on matches at a single bit, resulting in groups that may be distant from one another. In mathematical terms, by increasing k we increase the difference between the probabilities p_1 and p_2. The authors of FLANN (Fast Library for Approximate Nearest Neighbors Search), a popular open source feature-matching library that is integrated into OpenCV, recommend $10 < L < 30$ and $10 < k < 20$ for most applications [11]. Complexity as a function of these parameters is discussed more formally in [12].

The typical image matching procedure using this approach is summarized in Alg. 1. For each image candidate in the database, all features are extracted and hashed into the hash tables offline. Given a query image, we wish to find the image whose set of features is most similar. We extract a feature from the query image and assemble a list of database images that contain a similar feature. This process is repeated for each feature in the query. The image that is most similar overall ought to be the most commonly-occurring image in the list, although other heuristics are possible. Note that these for-loops can be easily parallelized. Also note that computing distances between features in the same bucket may not be necessary at all - a purely statistical approach, that counts occurrences of images in the matches, may suffice.

III. Bag-of-Features Organization

From an image-centric perspective, one might organize the data as in the Table below:

FeatureID	ImageID	Hash
0	0	0xabcdef...
1	0	0xaecdff...
2	0	0x08bfef...

However, for similarity matching, it is more natural to invert this data according to the hash value. In this representation, the hash values represent words in a vocabulary, and it becomes quite easy to retrieve the list of features that are described by that word. This is illustrated in the following table:

Hash	Bags of Features
0xabcdef...	{(FeatureID=0, ImageID=0), ..
0xaecdff...	{(FeatureID=1, ImageID=0), ...
0x08bfef...	{(FeatureID=2, ImageID=0), ..

For a large dataset, this inversion is an expensive process and is not guaranteed to be space effiicent. This is where Big Data tools such as Pig will become useful.

IV. Data

Since the aim of this project is to perform localization of imagery, we sought out a source of image data with geolocation information. A truly large source of such data is Google StreetView. Using StreetView data, we aim to localize images of the streets of New York City. Unfortunately, one cannot simply download large samples of

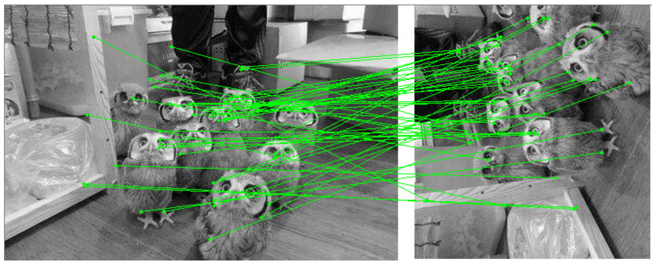


Figure 4. Example output of OpenCV's image matching code illustrating good feature correspondence invariant to rotation. We use these functions to extract keypoints and their descriptors for processing in our database.

Google StreetView without overrunning daily limits placed on Google's API. We found special difficulty in downloading batches from StreetView due to the stealth way that Google blocks requests - if the download limit has been reached, the query still returns an image, but it is a picture of an error icon, which is hard to recognize during script execution. As a result, we were only able to sample a small section (a few hundred) of the streets surrounding Morningside campus using Javascript and Google's HTTP StreetView API. The javascript code interpolates (latitude, longitude) locations along particular streets and queries for two images at each point, corresponding to the two headings at those locations that point perpendicular to the road. An example of this is depicted in Fig 3.

To produce a dataset worthy of the title "Big Data", we looked to other sources. INRIA (French Institute for Research in Computer Science and Automation) provides an image database several GB in size to accompany their paper on a similar topic [8]. The addition of these thousands of images (and resulting millions of descriptors) increases our dataset size to levels where conventional image matching algorithms are untractable on a desktop PC. Since their data is not of New York City but contains some urban environments, it provides a good noise floor for test queries.

V. Implementation

A. Database Construction

For the mechanics of image feature extraction and descriptor construction, we leverage the open-source computer vision platform OpenCV. Extracting a matrix of ORB descriptors using OpenCV is an easy task thanks to the comprehensive library of feature extraction and descriptor computation algorithms included (see Fig V. OpenCV's matching algorithms are powered by FLANN, which includes a locality-sensitive hashing implementation; however, FLANN's implementation does not lend itself well to this project for practical reasons. As a result, we chose to create our own small implementation of LSH for the Hamming distance. Since this algorithm would be executed millions or billions of times on a large database, speed was considered very important, so we implemented this as a Python extension written in C++. This extension statically allocates the bit masks for some number of hash tables, and provides functions to iterate through the descriptor matrix produced by OpenCV functions and produce a set of hashes for each feature. The database is processed with a Python script that iterates through the image files, extracts descriptors using built-in OpenCV image processing functions, and then invokes our LSH functions to produce the hash values. The result of this process is a table whose columns include: unique feature ID, an ID for the image the feature was taken from, and hash value for each of the tables in the LSH family.

B. Bag-of-Features Generation

As discussed previously, this is all the information we need to do a similarity search, but it is not in the desired format. In order to speed up matching at large scales, a global sequential search process should be converted to a separable parallelized one. To invert this table, which consists of millions of entries, we take advantage of MapReduce relational database processing in Hadoop Pig to generate a vocabulary for each hash table. This is done using the following Pig Latin commands for each hash table:

```
masterDB = LOAD 'master' using PigStorage(,)
        as (FeatureID:int, ImageID: int,
        Hash1:int, Hash2:int, ...
table1_raw = foreach masterDB generate
        FeatureID, ImageID, Hash1;
table1_grouped = group table1_raw by Hash1;
```

The result is a bag of features for each hash value populated in the table, for example:

```
(hash:int, featurebag:{(FID, IID), ...})
```

The *featurebag* field represents all the images that have a feature which is described by the given hash, and a reference to the feature in question so it can be retrieved if necessary. If this form is used, the neighbor retrieval action is reduced to a simple database query to fetch a certain bag of features, which is much faster.

C. Query Processing in Pig

The search algorithm (Alg. I) that operates on these bags of features can be recast as relational database operations and prototyped entirely in Pig Latin. This takes full advantage of distributed storage of the database in HDFS and parallel processing using the MapReduce framework (however, the reliance on MapReduce incurs significant overhead so this likely is not the most appropriate way to perform queries on a single image.) The only instance where custom code in the form of user-defined functions (UDFs) is required is where feature Hamming distance calculation would be performed, however we argue this direct calculation of nearness is unnecessary when the dataset is large.

The basic operation for matching is finding the entries in the database whose hashes correspond to the query image features. First, we group the features with the same hash value into one bag. This pre-treatment of the database is done offline, which saves time for finding all fitted featuers if we query the database. Since there may be multiple hash tables (we use 20 in our implementation), the result of this grouping is split into multiple tables. Next, in the online matching process, we input the query image features and join it with the database features by hash value. A list of values shared between query and database features is filtered out of the database. Then, we regroup the list by term of $ImageID$, so that we can clearly see who in the database shares these hash values, and how many hash collisions there are for each. Afterwards, by counting the number of collisions and sorting, the image who owns the most similar features is found. The whole query process consists of the following steps:

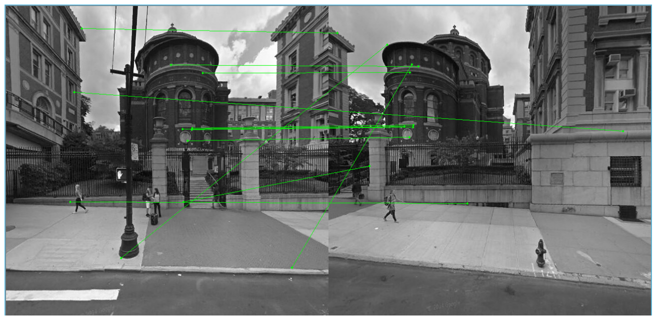
1) Extract the table of features for the query image, using the same method as was used to construct the database
2) Generate the bag-of-features vocabulary for the image
3) For each word in the query vocabulary, get the list of image IDs who have features that are also described by that word
4) Calculate the image ID with the most entries in the list

The output of steps 1 and 2 are as previously discussed. In Step 3, we produce an unsorted list of (ImageID, matchCount) pairs, and in Step 4 the image that has the most words in common with the query is selected as the match.

VI. Experiment Results

For initial testing, we built a small database containing 98 images and selected one of them as a query image. The query image is transformed into a bag-of-features representation and matched to the database. In this trivial example we expected to see the query image match perfectly with the original copy that resided in the database, which is precisely what we saw. The output of the MapReduce job showed a consensus on the correct image ID, and the number of similar features in all hash tables was equal to the total number of features extracted from the image. This demonstrated that our database generation and querying mechanisms were functional.

For more practical testing, we manually obtained pictures from Google StreetView that were very similar to a database image - ones that are clearly of the same scene, but not pixel-perfect identical. Then we populated our database with approximately 15 GB of more images from our datasets for a total of millions of database features. Notably, the time taken to construct the database using Pig with this many millions of features was only marginally longer than the time taken to construct the dataset with only 100 images on a six-core Intel i7 desktop PC. This is representative both of the highly-parallel nature of MapReduce (even though on a single desktop computer we cannot take full advantage of it) and the gargantuan overhead that comes with using it for small tasks.



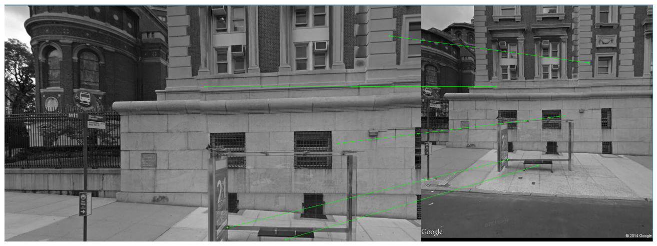


Figure 5. Example matches to query images. The image on the left represents the query image, and the matched image is on the left. In both cases the exact match was not present in the database, this represents best effort. The correspondences for features are drawn with a green line. Only "good" feature correspondences (those that are truly close neighbors in descriptor space) are drawn here for clarity.

Again in these cases, our matching query reached a consensus of the correct image. The detailed results varied by hash table, as expected. For example, in some tables, there were incorrect images in the database that matched nearly as well, while in others there was a clear winner. This highlights the statistical nature of checking only for neighbors within the bag of features and paying no attention to the distances between features themselves. The fact that the algorithm arrived at a consensus without any direct calculation of nearest-neighbor distance confirms our expectation that it is not necessary when the number of hash tables is sufficiently large. Figures VI and VI illustrate example results. The image on the left is the query image, and on the right the match selected from the database is shown. Feature correspondence is rendered using OpenCV.

VII. Conclusion

In this project, we have demonstrated a proof-of-concept for applying locality-sensitive hashing and "bag of features" large-scale image matching techniques to an image database 10s of GBs in size. We used Hadoop Pig to construct the database in a manner suited for distributed storage and perform batch queries in the MapReduce framework, such that the system could reliably be scaled up beyond millions of images, and then demonstrated batch processing of queries via manipulation of the relational database with Pig Latin.

Our implementation followed a path of least resistance that could be improved upon in future work in many ways. For example, other binary descriptors besides that of ORB could be explored, and those descriptors could be hashed using a more sophisticated distance measure than the simple Hamming distance used in our prototype. Furthermore, our recommendation algorithm could be expanded upon, by leveraging known improvements to LSH such as multi-probe LSH for example.

In terms of computational efficiency, we have not begun to stress the limits of the Hadoop platform for this task, in terms of database size or query complexity. All the work seen here was conducted on a single machine, running either in local mode or on HDFS as both the namenode and the datanode. Performance has not been tested on multiple nodes, in a distributed system.

Finally, an area of special interest that we did not have time to explore here is that of low-latency single image queries. While Pig operations are great for processing large quantities of data at once, we have seen that the overhead imposed by MapReduce for a single query is intractable. Given the efficient organization of the data, it should be possible to construct a query response without so much overhead.

A. *Work Split*

Below is a rough summary of the work distribution between the two project members for each component of the project.

- Overall algorithmic approach: jointly 50/50
- Initial testing of OpenCV feature extraction, algorithm evaluation in Python: by Yongchen Jiang
- Final image feature extraction algorithm implementation using OpenCV: done jointly, 50/50
- Theory of similarity search and LSH algorithm development, and C++ Python extension implementation: by Chris Stathis
- Javascript and HTML to mine images from Google StreetView: by Chris Stathis
- Development of Pig Latin scripts for database and query construction: jointly 50/50
- Writing of this report: Chris Stathis (80%) and Yongchen Jiang (20%)

Acknowledgements

The authors would like to thank first and foremost the developers of the open-source projects that made development and testing of this project possible, specifically: Hadoop, Pig, OpenCV, and FLANN. We also thank and acknowledge Professor Ching-Yung Lin for his time and efforts as professor in this Big Data Analytics course, and the TAs that supported him.

References

[1] G. Conte and P. Doherty, "An integrated uav navigation system based on aerial image matching," in *Aerospace Conference, 2008 IEEE*. IEEE, 2008, pp. 1–10.

[2] T. Lemaire, C. Berger, I.-K. Jung, and S. Lacroix, "Vision-based slam: Stereo and monocular approaches," *International Journal of Computer Vision*, vol. 74, no. 3, pp. 343–364, 2007.

[3] G. Hu, W. P. Tay, and Y. Wen, "Cloud robotics: architecture, challenges and applications," *Network, IEEE*, vol. 26, no. 3, pp. 21–28, 2012.

[4] E. Rublee, V. Rabaud, K. Konolige, and G. Bradski, "Orb: an efficient alternative to sift or surf," in *Computer Vision (ICCV), 2011 IEEE International Conference on*. IEEE, 2011, pp. 2564–2571.

[5] E. Rosten and T. Drummond, "Machine learning for high-speed corner detection," in *Computer Vision–ECCV 2006*. Springer, 2006, pp. 430–443.

[6] M. Calonder, V. Lepetit, C. Strecha, and P. Fua, "Brief: Binary robust independent elementary features," in *Computer Vision–ECCV 2010*. Springer, 2010, pp. 778–792.

[7] M. Muja and D. G. Lowe, "Scalable nearest neighbor algorithms for high dimensional data," in *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 36, no. 11, November 2014.

[8] H. Jegou, M. Douze, and C. Schmid, "Hamming embedding and weak geometric consistency for large scale image search," in *Computer Vision–ECCV 2008*. Springer, 2008, pp. 304–317.

[9] B. Kulis and K. Grauman, "Kernelized locality-sensitive hashing for scalable image search," in *Proceedings of the IEEE International Conference on Computer Vision*, October 2009.

[10] A. Gionis, P. Indyk, and R. Motwani, "Similarity search in high dimensions via hashing," in *Proceedings of the 25th International Conference on Very Large Data Bases*, ser. VLDB '99, 1999, pp. 518–529.

[11] M. Muja and D. G. Lowe, "Fast approximate nearest neighbors with automatic algorithm configuration," in *VISAPP International Conference on Computer Vision Theory and Applications*, 2009, pp. 331–340.

[12] A. Andoni and P. Indyk, "Near-optimal hashing algorithms for approximate nearest neighbor in high dimensions," in *Communications of the Association for Computing Machinery*, vol. 51, no. 1, 2008.