Implementing Visual Search with Vocabulary Trees

Mohamed Harmanani

April 2021

Abstract

Content-based image retrieval is an important application of computer vision, allowing users to search for images in large databases by using image content for computing matches instead of metadata. Nistér and Stewénius' 2006 paper "Scalable Recognition with a Vocabulary Tree" describes a scalable system for image searching and recognition, which builds a vocabulary tree from visual descriptors by using hierarchical clustering, then compares the embedding of two images, which is computed by propagating image features down the tree to form a code vector. The comparison of two images' code vectors gives us insight into their similarity, and allows us to implement rapid matching procedures [1]. The goal of this report is to briefly go over the methods used, and implement the procedure described by Nistér and Stewénius, while hopefully building on some of their ideas and exploring different approaches to the algorithms used in the paper.

Keywords: BRISK, SIFT, k-means, Vocabulary Trees, TF-IDF

1. Feature Extraction

Feature extraction is an extremely important step of any image matching procedure, as it is one of the main ways to extract semantic content from images. In this section we compare two feature detectors, SIFT and BRISK, a feature detector that relies on the more efficient FAST detector to detect keypoints. While the SIFT procedure is one of the most effective feature matching algorithms, it is on the slower side. On the other hand, faster detectors BRISK is less accurate at keypoint detection, as it mainly focuses on description, which SIFT can already very well.

1.1 FAST Detector

The basic idea behind FAST is to extract circular patches of 16 pixels from an image, and for each patch, compute the intensity of the center, I_p. Then, examine a segment of 4 pixels along the border of the circle, and if all of them are brighter than $I_p + t$, or darker than $I_p - t$ for some threshold value t, then it is quite likely that I_p is a keypoint. From the pixels in the circle, we then build a feature vector P [2; 3].

Depending on wether a pixel in the segment is brighter, similar, or darker than p, P is divided into three subsets, P_b, P_s, P_d, for the three states above respectively. We also add a label K_p, which determines wether or not p is a keypoint. Finally, a decision tree classifier is trained to predict the value K_p based on the values of P, P_b, P_s, and P_d [2; 3].

1.2 BRISK Feature Detector

As mentioned above, the BRISK detector extracts keypoints using the FAST procedure, described in **1.1**. After we have our keypoints, we must extract the characteristic direction of each keypoint, which is crucial to implementing a rotation-invariant descriptor [4]. To do so, we define N locations equally spaced on circles concentric with the keypoint [4]. We then apply Gaussian blur to each circular patch centered at some point $\mathbf{p}_i$, with a standard deviation σ_i proportional to the diameter. The gradient $\mathbf{g}(\mathbf{p}_i, \mathbf{p}_j)$ is then estimated using the following equation, where $(\mathbf{p}_i, \mathbf{p}_j)$ represents a sampling-point pair: [4]

$$\mathbf{g}(\mathbf{p}_i, \mathbf{p}_j) = (\mathbf{p}_j - \mathbf{p}_i) \cdot \frac{I(\mathbf{p}_j, \sigma_j) - I(\mathbf{p}_i, \sigma_i)}{||\mathbf{p}_j - \mathbf{p}_i||^2} \tag{1}$$

We then take the set $\mathcal{L}$ of sample-point pairs of cardinality L such that the value $||\mathbf{p}_j - \mathbf{p}_i||$ is larger than some threshold δ_{min}, and we iterate over the set to compute the overall characteristic pattern of the patch using the following equation: [4]

$$\mathbf{g} = \begin{pmatrix} g_x \\ g_y \end{pmatrix} = \frac{1}{L} \cdot \sum_{(\mathbf{p}_i, \mathbf{p}_j \in \mathcal{L})} \mathbf{g}(\mathbf{p}_i, \mathbf{p}_j) \tag{2}$$

Finally, we need to build our descriptor. The first step to do so is to compute the value $\alpha = \arctan2(g_y, g_x)$, which is the direction of the keypoint of interest. Then, we compare the intensities of the points in sampling-point pairs of the set $\mathcal{S}$ such that the value $||\mathbf{p}_j - \mathbf{p}_i||$ is smaller than some threshold δ_{max}. We do so with the equation below, where b is the corresponding bit in the bit vector d_k describing keypoint k, and $(\mathbf{p}_i, \mathbf{p}_j)$ is a sample-point pair in subset $\mathcal{S}$. [4]

$$b = \begin{cases} 1 & \text{if } I(\mathbf{p}_j^\alpha, \sigma_j) > I(\mathbf{p}_i^\alpha, \sigma_i) \\ 0 & \text{otherwise} \end{cases} \tag{3}$$

1.3 SIFT Feature Detector

SIFT starts by detecting characteristic peaks in the image's scale space, which is represented by the Laplacian of Gaussian (LoG). However, since computing the LoG is costly, SIFT approximates the LoG using the Difference of Gaussians, which computes a series of octaves, and a series of scales for each octave. At each step, the DoG is computed by taking the difference of the $G_\sigma(I)$ and $G_{k\sigma}(I)$ for some k [5].

We then compute interest points by dividing the image into quadrants, comparing the pixels with their neighbours, and finding all local extrema. We then fine tune the results by applying non-maxima suppression. We then assign an orientation to each keypoint, by building a histogram of gradients, and assigning the dominant orientation to the pixel [5].

Finally, we extract a 16×16 patch around each keypoint, and divide it into 4×4 regions. A 128×1 feature vector is then built by taking the histogram of orientated gradient of pixel in the smaller regions [5].

1.4 Comparing SIFT & BRISK

The SIFT detector is slower than BRISK, but more accurate in many cases [6]. When running both BRISK and SIFT to detect matches between similar images, we observed a very strong performance in similar images, but when comparing them on images that have little in common, we found that SIFT filtered bad matches much better than BRISK. The reason for this is that SIFT is a sparser but more descriptive feature detector. Conversely, BRISK detects many more keypoints than SIFT at the cost of these keypoints being less descriptive. This results in a lot of noise being generated, as a common keypoint that is shared amongst several images in the database can get detected multiple times, and amplified in the prediction, even if that keypoint is not necessarily indicative of a strong match. While testing both descriptors, we observed that using BRISK generated a feature database of over 400,000 descriptors for 265 images, while SIFT produced a database of 200,000 features. For these reasons, we will be using the SIFT detector, but BRISK is a good alternative if accompanied by noise reduction procedures.

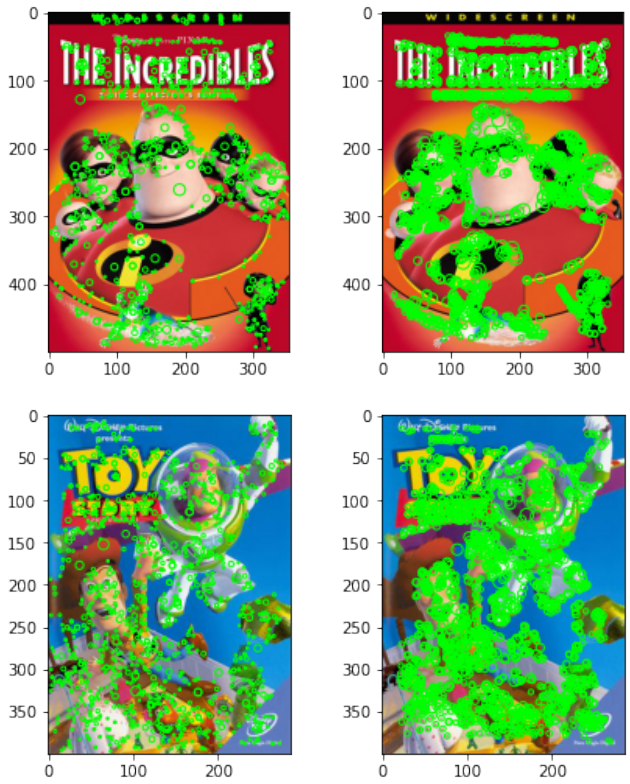

Figure 1: SIFT (left) and BRISK (right) applied to different images

2. Homography Estimation

Homgraphic transforms are an important tool in image matching, as they allow us to compute a warped representation of a given image from a set of matched keypoints, and superpose it onto another image containing all of part of our test image.

2.1 Keypoint extraction

The first step to implement homography estimation for matching one image to another is to extract keypoints using either a method like SIFT, ORB, or BRISK. While we initially started using BRISK, as it is faster than SIFT, we ultimately found it to be less accurate, so we will be using SIFT.

2.2 Computing matches

After extracting our keypoints, we need at least 4 matches to compute our homography. To find the best matches, we measure the euclidean distance between the descriptors in the source image and the descriptors in the target image, and build an array of matches sorted in accordance to this distance. We then compute the ratio ϕ of the closest match to the second closest match, given by

$$\phi = \frac{||f_i - f_i'||}{||f_i - f_i''||} \tag{4}$$

If this value is below a certain threshold, we toss the match, because the small difference in distances leads us to believe that this is an ambiguous match. Applying this to one of the test images in our dataset, we get

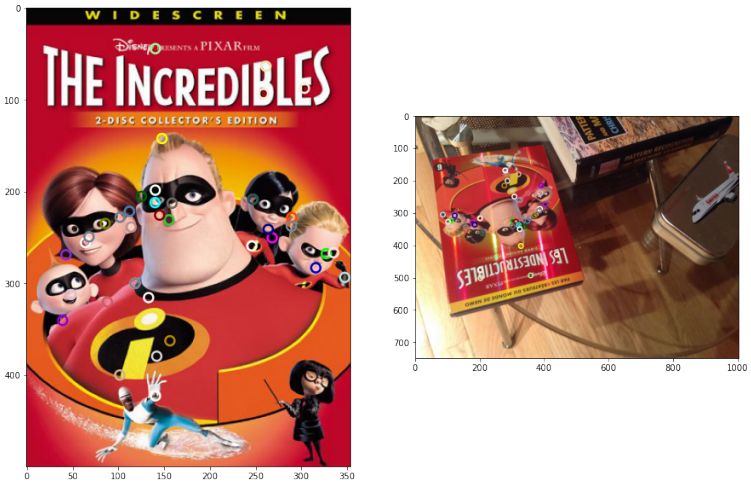

Figure 2: Result of matching two images using SIFT

2.3 Homography estimation using RANSAC

After computing our matches, we take each pair of matched points, $\mathbf{p}$ and $\mathbf{q}$, and setup the following linear system:

$$\begin{bmatrix} p_x^{(1)} & p_y^{(1)} & 1 & 0 & 0 & 0 & -q_x^{(1)}p_x^{(1)} & -q_x^{(1)}p_y^{(1)} & -q_x^{(1)} \\ 0 & 0 & 0 & p_x^{(1)} & p_y^{(1)} & 1 & -q_x^{(1)}p_x^{(1)} & -q_y^{(1)}p_y^{(1)} & -q_y^{(1)} \\ & & & & \vdots & & & & \\ p_x^{(n)} & p_y^{(n)} & 1 & 0 & 0 & 0 & -q_x^{(n)}p_x^{(n)} & -q_x^{(n)}p_y^{(n)} & -q_x^{(n)} \\ 0 & 0 & 0 & p_x^{(n)} & p_y^{(n)} & 1 & -q_x^{(n)}p_x^{(n)} & -q_y^{(n)}p_y^{(n)} & -q_y^{(n)} \end{bmatrix} \mathbf{h} = \mathbf{0} \tag{5}$$

One we've computed the vector $\mathbf{h}$, we can get $\mathbf{H}$ by rearranging its elements into

$$\mathbf{H} = \begin{bmatrix} h_0 & h_1 & h_2 \\ h_3 & h_4 & h_6 \\ h_7 & h_8 & h_9 \end{bmatrix} \tag{6}$$

Moreover, given a very large number of matches, we can detect the best 4 points to compute our homography by using the RANSAC algorithm. The main idea is to sample 4 points at random, compute a homography from these points, and count how many inliers we have among our points. We can repeat this step about 5000 times for a 99% probability of identifying the homography with the maximum number of inliers.

Figure 3: Using homography to warp a database image onto test image 5

3. Efficient index retrieval

3.1 Building a feature database

The first step towards building our vocabulary tree is to compile all the features across our image database and compile them into a convenient format for parsing. To do so, we go over every image in our database, extract its descriptors using SIFT, and store them in a matrix of size $|F| \times 128$, where F is our feature space across all images in the dataset.

3.2 Building the vocabulary tree

Now that we have our database, we build the vocabulary tree using the hierarchical k-means clustering algorithm. We supply two parameters to the tree, K, the branch factor and number of clusters to be used in k-means, and L, the maximum depth of the tree. We perform L clustering operations, one for each level, and at each iteration, we identify the centroid of each cluster, and make it the root of its respective subtree. Then, each subtree gets added to an array, and connected to the already existing tree.

We experimented with multiple values of K and L, and ultimately found that the values of K and L which yielded the best results were $K = 4$ and $L = 10$.

3.3 Generating image indices

To generate an image's encoding, we take each of its feature and propagate it down the tree. The basic idea is to search each level for a descriptor value that is closest to it, add that node's index to the computed path, then repeat the process above for that node's children, until we reach a leaf node. Node indices are unique, and are computed by generating a list of integers in the range $(0, |F|)$, and choosing the minimum value for an index each time a new node is created. The chosen index is then removed from the array. We store these values in an array u [1].

However, this scheme inadvertently favours noisy features, which occur very frequently in each image without telling us anything descriptive about the image's content. An example of this is the root node of the tree, which occurs in every single image, and thus, should have a low representational weight. The next logical step is thus to weigh our path computation such that lower nodes have more weight than higher ones, since they are ultimately more descriptive. We use TF-IDF weighting for this. The basic idea behind this technique is to assign a weight equal to the logarithm of the quotient of the total size of the database and the term's frequency in the database (Equation 7) [1; 7].

$$\text{tf-idf} = \log \frac{N}{N_i} \tag{7}$$

Finally, we can compute our path vector as follows, by applying the weights computed above

$$v_i = w_i \cdot u_i = \log \frac{N}{N_i} \cdot u_i \tag{8}$$

3.4 Searching by scoring

After computing the final representation for each image, we must store them in a data structure for quick and efficient retrieval times. We can use a relational database for larger dataset sizes, but for ours, a simple hash table will do. We store $< T, v >$ mappings inside the table, where T is the image's name, and v is its unique code vector.

Now if we want to search for some input image's closest match in our database, we simply compute its code vector using the procedure above, and we end up with some vector u.

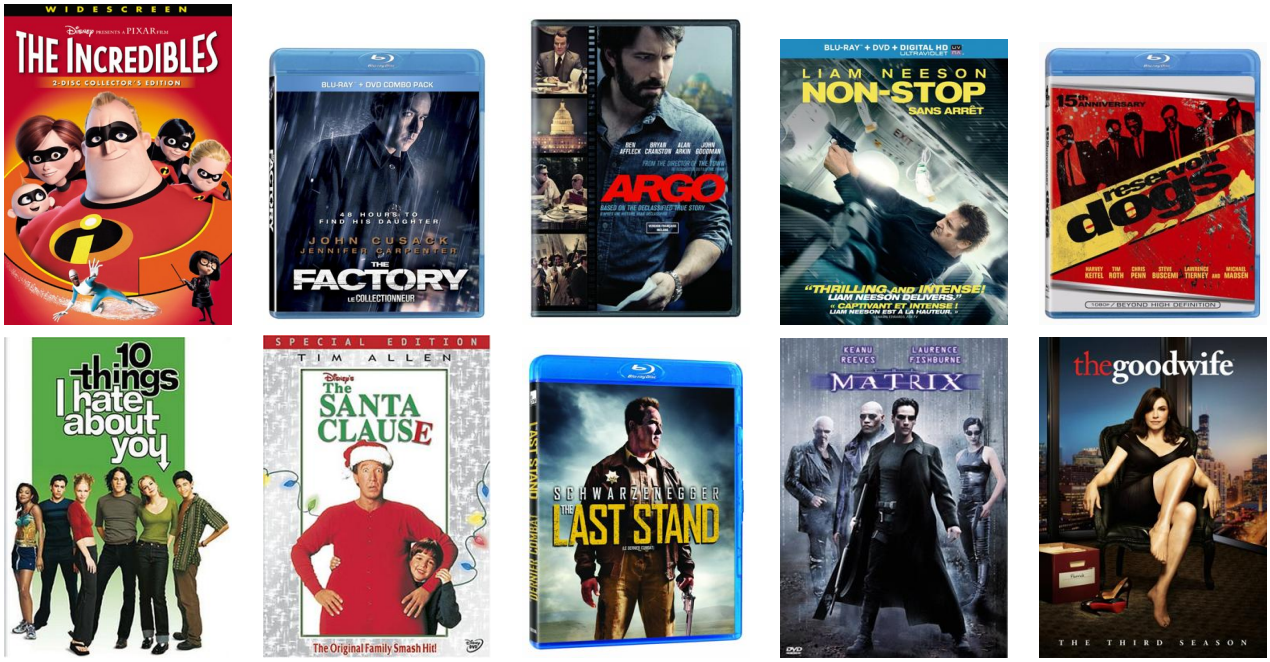

Figure 4: Top 10 matches for test image 5

We then iterate over our pre-computed code vectors stored in our table, and compute the cosine similarity of u and v, which is shown below. The 10 images with the highest cosine similarity will then be shown to the user. The results for test image 7 are shown above in Figure 4.

$$\cos \text{ similarity}(u, v) = \frac{u \cdot v}{||u|| \cdot ||v||} \tag{9}$$

3.5 Filtering good matches

Finally, once we've assmbled our top 10 matches, we can iterate over each image and compute the maximum number of inliers we can get from each homography estimation. We can then safely say that the image whose homography estimation has the highest number of inliers is the target image, as can be seen in Figure 5 and Table 1.

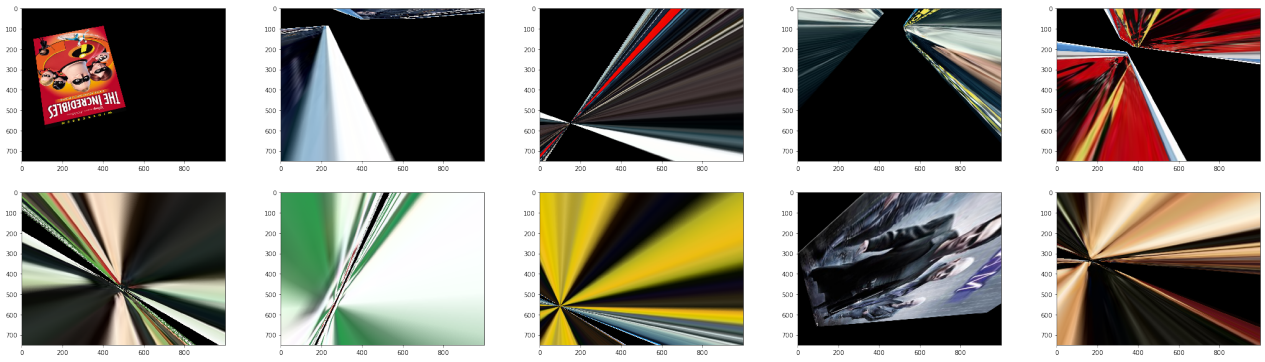

Figure 5: Homography mappings for each retrieved image

Image file name	Inliers
the_incredibles.jpg	96
the_factory.jpg	8
argo.jpg	7
non_stop.jpg	8
reservoir_dogs.jpg	7
10_things_i_hate_about_you.jpg	7
the_santa_clause.jpg	8
the_last_stand.jpg	7
matrix.jpg	8
the_good_wife_season_3.jpg	6

Table 1: Number of inliers for each homography

4. Issues & Discussion

4.1 Feature extraction

While this report relied mainly on traditional feature detectors like SIFT and BRISK, it would have been wiser to use some of the advances in deep learning technologies to extract more meaningful features, which would have also alleviated the problem with high levels of noise observed with this approach. While we experimented with detectors like VGG, the logistics of these detectors proved to be somewhat troublesome. Moreover, the very small size of our dataset leads us to theorize that training a new model from scratch would be just as hard, as it would likely overfit the data.

4.2 Time and space complexity

Building the vocabulary tree is a very computationally costly process, which takes a long time to run, and uses a lot of storage space. We observe a building runtime of $O(K^L)$, which means that in order to be on the faster side, K needs to be larger, and L smaller. This makes this model extremely difficult to build and test, and there seems to be no reason to use this type of method over more modern techniques like DCNNs.

4.3 Tree geometry

While our optimal K and L worked for our purposes, varying the size of the dataset also changes the optimal parameters to a large extent. This makes experimenting with different dataset sizes very difficult, and it's probably best to identify an ideal dataset from the start, otherwise, finding new values of K and L will prove very costly, as was mentioned above.

References

[1] D. Nistér, H. Stewénius, "Scalable Recognition with a Vocabulary Tree, " *2006 IEEE Computer Society Conference on Computer Vision and Pattern Recognition*. New York, NY, USA, 2006.

[2] E. Rosten, T. Drummond, "Machine Learning for High-Speed Corner Detection," *2006 European Conference on Computer Vision*. Berlin, Heidelberg, 2006.

[3] E. Rosten, R. Porter, T. Drummond, "Faster and better: a machine learning approach to corner detection, " *IEEE Trans. Pattern Analysis and Machine Intelligence*. 2010.

[4] S. Leutenegger, M. Chli, R. Y. Siegwart, "BRISK: Binary Robust invariant scalable keypoints, " *2011 International Conference on Computer Vision*. Barcelona, Spain, 2011.

[5] D.G. Lowe, "Distinctive Image Features from Scale-Invariant Keypoints, " *International Journal of Computer Vision 2004*. 2004.

[6] S. A. K. Tareen, Z. Saleem, "A comparative analysis of SIFT, SURF, KAZE, AKAZE, ORB, and BRISK, " *2018 International Conference on Computing, Mathematics and Engineering Technologies – iCoMET 2018*. Sukkur, Pakistan, 2018.

[7] E. Uriza, F. Gómez-Fernández, M. Rais, "Efficient Large-scale Image Search With a Vocabulary Tree, " *Image Processing On Line*. 2018.