CSC420 DVD Covers Project Report

Katie Datsenko

December 3, 2016

1 Introduction

In this project I am implementing a recognition pipeline for performing image recognition, retrieval, and localization with vocabulary trees. The specific task is to recognize DVD covers in real-world images based on a database of images. The system accepts a query image as the input, and should accurately identify the DVD cover in the database that is same cover, but from a regularized viewpoint. The pipeline has two main stages, the first is to retrieve an initial set of top 10 matches from the database that correspond best to the query image. This is implemented using Nister and Stewenius' technique of building a vocabulary tree by hierarchically clustering image feature descriptors from the entire set of database training images using hierarchical k-means clustering [3]. Features are detected and descriptors for these features are generated using SIFT. The second stage is to prune the top 10 matches down to a single match using the relative geometry of the locations of matched features between the query image and each of the candidates images. This step is implemented via homography estimation with RANSAC. Once the pipeline narrows the search down to one candidate, the homography computed for the finalized database match is used to localize this image in the query image, represented with contour lines outlining the perimeter. In this report I will present my implementation of the DVD-cover recognition pipeline with a few additional sources of improvement.

2 Methods

2.1 Keypoint detection and description with rotation and scale invariance

In this implementation, I use the DoG keypoint detector and the SIFT descriptor implementation from the VLFeat[1] library. As well, I have incorporated a few improvements regarding keypoints detection and description: affine invariant interest point detector and spatial context statistics.

2.1.1 DoG keypoints

The DoG keypoint detection algorithm [2] searches for local 3D extrema in the scale-space pyramid built with Difference-of-Gaussian(DoG) filters. The DoG is a close approximation to the Laplacian-of-Gaussian (LoG), thus it is necessary to understand the LoG operator. Given an input image $I(x, y)$, the scale space representation of the image is obtained by convolving the grayscale image by a variable sigma-scale Gaussian kernel $G(x, y, \sigma)$. At each scale, the LoG image is obtained by calculating a linear combination of second derivatives (Laplacian). Then to find keypoints, local maxima/minima are extracted from the 3D scale space of the LoG function (for instance, a $3 \times 3 \times 3$ neighbourhood). The LoG is a circularly symmetric operator, so it finds change in all directions and thus is naturally invariant to rotation. It is suited for blob detection, and it can do this across a number of scales. The operator response is dependent on the relationship between the size of the blob structures in the image and the size of the smoothing Gaussian kernel. The standard deviation of the Gaussian is used to control the scale by changing the amount of blurring and also to compensate for noise in blob contours. Intuitively, the Laplacian of Gaussian will also detect features at corners of an image because corners have high gradients in at least two different orientations, which correspond to maxima along two different directions in the 2nd derivative (the Laplacian is analog to second derivative), so we have a convex or elliptical point of change, similar to the image of a blob.

The DoG function works similarly by subtracting adjacent scale levels of a Gaussian pyramid with the sigma parameter at each level separated by a factor k, and local scale-space extrema are extracted as features. As well, the scale space is divided into octaves, where the image is downsampled by a factor of 2 after each scale octave. Afterwards rather than taking the center of the $3 \times 3 \times 3$ cell, the final keypoint locations are interpolated based on the DoG response of the surrounding points. The DoG is more efficient than the LoG because it reduces the number of convolutions since there is no need for the computationally-expensive operation of finding second derivatives in the Gaussian images.

2.1.2 SIFT keypoint descriptors

The SIFT[2] algorithm for generating a descriptor for each keypoint is as follows: for the image smoothed to the correct level of Gaussian blur corresponding to the scale of the keypoint, gradient magnitudes and orientations are sampled at each pixel in a 16×16 pixel window around the keypoint, which is further broken down into sixteen 4×4 pixel windows.

A set of gradient orientation histograms is created for each 4x4 window, where each histogram has 8 bins with an interval of 45 degrees. A Gaussian weighting function with σ equal to half the Gaussian sigma for the scale is used to assign weight together with the gradient magnitude of each sample point and gives higher weights to orientations closer to the center of the region.

The orientation angle for every gradient data point is measured relative to the dominant gradient orientation at the keypoint, which is likewise found by building a histogram containing gradient orientations of every pixel in the window segmented by a 10 degree interval, applying a similar weighting strategy, and taking the highest weighted orientation. This gives the SIFT descriptor the property of rotational invariance.

The descriptor is then formed from a vector containing the values of all the orientation histograms entries. Since there are 4×4 histograms each with 8 bins, the feature vector has $4 \times 4 \times 8 = 128$ elements for each keypoint. Finally, the feature vector is normalized to unit length to gain invariance to linear changes in illumination. The vectors values are also thresholded to a maximum value of 0.2, and the vector is again normalized to reduce feature descriptor variance due to non-linear illumination changes such as camera saturation.

2.1.3 Affine Invariant Interest Point Detector

The motivation for this improvement is that large viewpoint changes and perspective distortion of an object such the DVD covers in this project distorts the feature descriptors. For instance, circular regions around keypoints in the regularized front-facing view of the Reference images deform into elliptical regions in a perspective view of the same DVD cover, causing the feature descriptors to change (different gradient responses will be measured in the image patch around the same keypoint in both images).

To improve Nister's method, I also decided to update the keypoint and descriptors to be Harris-Affine via the VLFeat library. The Harris-Affine detector is the implementation of the affine invariant interest point detector outlined in [4]. This detector is based on the Harris-Laplace detector, which performs detection based on Harris corner interest points in a Gaussian scale space. The Laplacian-of-Gaussian is used to find the maxima over the scale and a threshold is used to reject extrema with low cornerness measure. The features detectors as a result are invariant to rotation and scale. An iterative shape adaptation algorithm is applied to estimate the local affine neighbourhood and scale for each point, and localize the Harris interest points. This means that the Harris-Affine detector allows us to identify different projections of the same 3D patch with almost invariant descriptor vectors, in spite of the effects of large viewpoint changes.

2.1.4 Spatial contextual statistics

The other improvement is an optimization on the DoG feature descriptors. A current weakness of Nister's method is that it does not incorporate the consideration of the spatial layout of adjacent features when selecting the top candidates. Instead it leaves spatial verification as a post-processing step over the selected candidates. The result of this is that multiple images could contain very similar word information to a query image, but the spatial relation between words is inconsistent with how they are presented in the query image.

But these false positive are chosen over the correct match in the top 10 candidates because they are great in number.

To deal with this, I add spatial contextual statistics for each local feature descriptor by implementing the SWC statistics in [5]. These statistics include the density of features within a radius-neighbourhood of each feature, their mean scale and mean orientation difference. These statistics are local to the feature and are scale and rotation invariant.

Figure 1: 3 spatial statistics computed in the local neighbourhood of a feature. Figure taken from [5]

$$\begin{aligned} \rho &= |C(\mathbf{f}_0)|, \\ \overline{\Delta s} &= \frac{1}{|C(\mathbf{f}_0)|} \sum_{\mathbf{f} \in C(\mathbf{f}_0)} |s - s_0|, \\ \overline{\Delta \theta} &= \frac{1}{|C(\mathbf{f}_0)|} \sum_{\mathbf{f} \in C(\mathbf{f}_0)} |\theta - \theta_0|, \end{aligned}$$

2.2 Building the Vocabulary Tree

The feature descriptors are hierarchically quantized in a vocabulary tree in order to be able to quickly identify candidate matches from the database of DVD cover images. This step is taken care of in the function `build_vocabulary_tree`. I compute all SIFT descriptors from all Reference images and compute the hierarchical vocabulary tree using the function `vl_hikmeans` provided by the VLFeat[1] Library. The reason I chose to use the library function than implementing it myself is mainly because of the runtime speed, which I would never have achieved myself using MATLAB k-means recursively on the huge training data set. The vocabulary tree is defined with branching factor K=10 and depth L=6. These parameters for the tree are chosen as the optimal parameters for accuracy of retrieval from the results of Nister et al., and are also optimal based on my empirical results.

2.3 Virtual Inverted File Index Generation

I implemented Nister and Stewenius' idea of a virtual inverted file index. This means only the leaf nodes of the vocabulary tree are associated with an explicitly stored inverted file. The inverted files store the ID numbers of the DB images that have a feature represented by the visual word at the leaf, and the frequency m_i of this word's occurrence for each image. This step is taken care of in the function `inverted_file_index` by computing feature points and their descriptors for every image and passing each descriptor through the vocabulary tree down to the leaves and accumulating statistics for the respective image.

A precomputation of the weights of each node using the *TF-IDF* scheme is performed. This is done first by computing the N_i term for each node, or the number of images in the database with at least one descriptor path through the node. This can be combined in the same step as the inverted index generation by incrementing the N_i count for each node i reached while traversing a path in the Vocabulary tree down to the leaves. To avoid over-counting, an image ID number is stored with the current N_i count for each node within the data structure representing the last image this node has seen. If the ID of the current image is greater than the ID that was last seen, the N_i count is incremented. Then it is a simple step to combine with the total number of images N to produce the weights w_i for each node i.

One last thing that should be addressed is the precomputation and normalization of the database image vectors. In this implementation, I decided to improve the space requirement by not precomputing the database vectors wasting $\mathcal{O}(\texttt{nnodes}\cdot\texttt{Nimgs})$ space to store all of them, and instead implementing the technique of Bottom-Up concatenation/union of inverted files of the leaf nodes to score database image vectors using Equation 5 of [3]. This implementation decision also does not affect the runtime as will be discussed in the next section on Hierarchical scoring. This keeps the implementation in the spirit of virtual inverted files by not forcing the use of Forward files or explicit inverted files for inner nodes.

However, it is not possible to implement the bottom-up technique only knowing the m_i for each database image of the leaf nodes (provided by the inverted file index) because the database vector terms should be

normalized to unit magnitude. Thus, the **norms** for each database image vector are precomputed and stored in space $\mathcal{O}(\texttt{Nimgs})$. This is done by noticing the trick that the L_1-norm for a database image vector can be computed as the sum over sum of weights along the path to each leaf node multiplied by the m_i of the database image at the leaf node.

$$\|d\|_1 = \sum_{leaf\ i} \left(\sum_{node\ j\ on\ path\ to\ i} w_j \right) m_i \tag{1}$$

It takes $\mathcal{O}(\texttt{nnodes})$ time and space to compute the sum of weights for each leaf node using a top down procedure, and then $\mathcal{O}(K^L \cdot \texttt{Nimgs})$ where K is the branching factor and L is the number of levels of the vocabulary tree and K^L represents the number of leaves in the tree. This is in the order of $\mathcal{O}(\texttt{nnodes} \cdot \texttt{Nimgs})$, not more than the runtime for a precomputation of all normalized database vectors, but a big improvement in the space complexity.

2.4 Hierarchical Scoring

The first thing we do is compute the normalized query vector by passing its image descriptors through the Vocabulary Tree and accumulating the n_i at each node i and then weighting the vector with the node weights precomputed in the previous method `inverted_file_index` and finally normalizing. This produces a query vector of length `nnodes`.

Then we do a post-order tree traversal on query nodes, only visiting the subtrees of nodes with a non-zero query vector entry value. To find d_i at each node, while traversing we perform the concatenation of the inverted files starting at the leaf nodes to compute inverted files at inner nodes as described in [3]. This is done by accumulating m_i at the parent node from the K children nodes. The inverted files of the children nodes are stored in a list `mi_accumulator` indexed by nodes. The concatenation here is actually a union to avoid redundant addition operations (regarding accumulation of m_i at a node for a each image) since multiple children of a parent may have an m_i value under identical IDs (the same database image has descriptors passing through more than one child). Algorithm 1 shows how the database scores are accumulated via a post-order traversal of the inverted file tree structure.

Algorithm 1 pseudocode of post-order tree traversal of virtual inverted file structure

```
1: procedure TreeTraversal
2:     process_parent ← false
3:     while current_node > 0 do                                   ▷ stop at 1 (before root)
4:         processed ← false                                       ▷ for every new node
5:         if q_vector(current_node) ≠ 0  then                     ▷ We can do this node
6:             if process_parent == false then
7:                 if children are leaf nodes then
8:                     build inverted file for current_node
9:                     for each leaf l do
10:                        for each database img j in leaf inverted file do
11:                            normalized_diff_scores(j) += |q_l − d_lj| − |q_l| − |d_lj|
12:                    processed ← true
13:                else
14:                    skip while loop iter, go to first child of current_node
15:            else                                                ▷ process_parent == true
16:                process_parent ← false
17:                build inverted file for current_node
18:                delete all children inverted files
19:                processed ← true
20:        if  processed == true then
21:            for each database img j in current_node i inverted file do
22:                normalized_diff_scores(j) += |q_i − d_ij| − |q_i| − |d_ij|
23:        if right sibling node exists then
24:            current_node ← sibling
25:        else
26:            process_parent ← true
27:            current_node ← parent
```

However, a union of two inverted files may take quadratic time in $\mathcal{O}(\texttt{Nimgs}^2)$ thus to combat this problem we have a hashtable-like data structure where it's possible to access the m_i score of an image in the current node inverted file in constant time, making a union operation for the sets of images and their scores in all child nodes of a parent a trivial operation: $\mathcal{O}(\texttt{Nimgs})$. To do this, we have a vector mapping image ids to indexes in the Inverted File of the current node, `img_indx_table`. Since image ids are mapped from 1 to total number of images N, this vector is in the size of number of images N. A variable storing the size of the current inverted file is reinitialized to zero for every new node and used to render old mappings of image ids to indexes obsolete in the context of a new node. This is to avoid incurring the $\mathcal{O}(N)$ runtime to reinitialize the entire vector.

Once the inverted file of a parent node has been computed, the inverted files of children that are inner nodes can be discarded. This way, the space required by the data structure storing inverted files is upper-bound by the space required for the original virtual inverted file index, and only decreasing from that point on (because of the unions). Algorithm 2 presents pseudocode for uniting the inverted files of the children for a node.

Algorithm 2 pseudocode to build inverted file for current node from its children

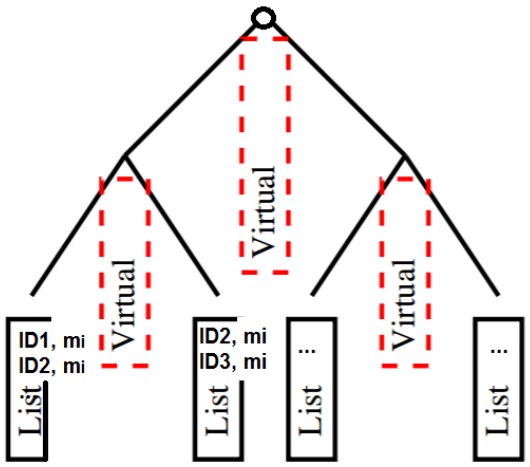
```
1:  procedure BuildInvertedFile
2:      ▷ nnodes is number of nodes witinh the vocabulary tree (except root)
3:      ▷ Globally available variable, scope not just within this block of code
4:      ▷ Nimgs is number of DB images
5:      inverted_files ← cell(1, nnodes)
6:      img_index_table ← zeros(1, Nimgs)
7:      parent_inverted_file ← zeros(2, Nimgs)                    ▷ columns are entries
8:      img_index_hash_size ← 0
9:      for child_node=first_child:last_child do
10:         child_inverted_file ← inverted_files[child_node]
11:         for entry in child_inverted_file do:
12:             id ← entry.imageID
13:             m_i ← entry.miScore
14:             i ← all_img_indx(id);
15:             ▷ Cases:
16:             ▷ i == 0: allocate entry for entirely new img
17:             ▷ i > img_indx_size: detected obsolete info from previous node
18:             ▷ id ≠ parent_inverted_file(i).imageID: detected obsolete info
19:             ▷ i < img_indx_size ∧ id ≠ parent_inverted_file(i).imageID: not possible
20:             if i > img_indx_size ∨ i == 0 ∨ id ≠ parent_inverted_file(i).imageID then
21:                 i ← img_index_hash_size + 1
22:                 img_index_table(id) ← i
23:                 img_index_hash_size += 1
24:                 new_entry.imageID ← id
25:                 new_entry.miScore ← m_i
26:                 parent_inverted_file(i) = new_entry
27:             else
28:                 parent_inverted_file(i).miScore += m_i
29:             delete inverted_files[child_node];
        inverted_files[current_node] ← parent_inverted_file(:, 1:img_index_hash_size)
```

When all of a node's children are processed, then scores for each image in the inverted file for the node can be updated using the same scheme as in Equation 5 of [3] (Algorithm 1). We are computing a score update between every query vector node entry q_i which is non-zero by construction of traversal, and every database image vector node entry with $d_i \neq 0$ (which is implied based on the images gathered in the inverted file - database images with $d_i \neq 0$ are not even listed among them). We assume every leaf node has at

least one database image descriptor assigned to it based on the dense training set of image descriptor data, making bottom up post-order traversal worthwhile.

Thus the implementation successfully incorporates the efficient scoring technique with virtual inverted files as outlined in Nister et al. The runtime for hierarchical scoring with precomputed database vectors would be $\mathcal{O}(\texttt{nnodes} \cdot \texttt{Nimgs})$. The runtime with efficient scoring is $\mathcal{O}(\texttt{nnodes} \cdot \texttt{Nimgs} - \text{all pairs } i|qi = 0 \vee di = 0)$. This will speed up our runtime when we have many database images with skewed database vector trees. Then after scoring by traversal, I take the top 10 matches by repeatedly computing the minimum among all database image scores which is faster than doing a sort on the entire set of image scores.

Figure 2: Representation of the virtual inverted file structure, L=2 and K=2. Taken from [3]



2.5 Image Retrieval and Verification

Since the previous step to extract the top 10 candidates lacks any spatial criteria, the final step is to implement a post spatial verification step. We spatially verify the geometric structure of planar matching images by fitting a Homography estimation with RANSAC. First, I find feature matches between a candidate image and the query image. For this I use the VLFeat[1] function `vl_ubcmatch` to find the correspondences for speed.

This library function implements Lowe's method for feature matching, which involves finding the closest feature in the candidate image for each feature in the query image, and discarding any matches where the ratio of the distances between first best match and second best match is above some designated threshold (the match should be strong enough).

Once the matches are computed, I run 1000 iterations of RANSAC. The algorithm steps consist of picking 4 distinct matches in each iteration, computing a best-approximation of the Homography transformation matrix that maps the candidate image into a new space via SVD based on the 4 matches, and then counting inliers based on a threshold applied to the euclidean distance of transformed points from their equivalent matches in the query image. The candidate image among the 10 that has the greatest number of inliers is chosen.

3 Main challenges

One of the main challenges I faced was minimizing the runtime of various steps of the pipeline. Most the runtime is incurred in the training stage of the pipeline, where I found that it takes 27 seconds to train the Vocabulary Tree and 37 seconds to generate the Inverted File Index (explicit storage at the leaves). To

separate the time for training from the testing stage, I save and load trained models from disk such as the vocabulary tree, and intermediate structures such as the virtual inverted file index, the node weights of the tree, and the database vector L_1-norms in preparation for query testing. The runtime for hierarchical k-means clustering for the vocabulary tree is very fast, approximately 10 seconds due to the library implemented function `vl_hikmeans` which incorporates sophisticated optimization techniques beyond a pure MATLAB implementation of the same recursive procedure. In terms of training the inverted file index and the statistics for weighting and normalizing vectors, I found that the $\mathcal{O}(\texttt{nnodes}\cdot\texttt{Nimgs})$ runtime cannot be helped, however the space complexity is greatly improved via the idea of virtualization of the inverted files for inner nodes. The query testing runtime is reduced via the optimization of only accumulating scores for database image candidates at the non-zero query and database entries q_i and d_i.

The main challenge I faced here was implementing the ideas in the Nister Paper, where the outline of the method is vague and I found a mathematically inaccuracy in the statement that the the database vector dimensions d_i can be fragmented into independant parts d_{ij} coming from the children nodes since the weights applied to the database vector dimensions are not linearly related. Therefore, in the implementation the m_i are actually the only fragmented statistics. As well, this is not possible in an virtual inverted file scheme without calculating the database vectors norms in a precomputation step. Once implemented successfully, the optimizations founded in this paper allows the scoring algorithm to potentially skip large subtrees of the database vector trees when performing operations at each node. As a result of the optimizations, the scoring computation part of the pipeline takes under a second.

4 Results and discussions

The vocabulary tree defined with branching factor K=10 and depth L=6 and the scoring algorithm with DoG SIFT descriptors gave me near perfect results for each of the testing image sets E63, Droid, Palm and Canon. Figure 3 demonstrates an example of a correctly classified test case.

Figure 3: Correctly identified and located Beautiful Mind DVD cover

In fact there were two test cases from the entire data set where a misclassification occurred, image 88 from the E63 dataset, and image 100 from Canon. Image 100 (Figure 4) was in general very hard to classify possibly because of the result of the glare on the DVD cover from the lighting in the scene which presents a case of misclassification due to noise since in the other datasets the correct reference cover is chosen among the top ten candidates. However, in the other 3 datasets the Mad Men (Canon 100) DVD cover had a lower score (8 in one case) and this can be explained due to the fact that the Mad Men cover has very little distinctive features, it is mostly all black with some text, which contributed to the misclassification result as well.

The misclassification of the other case with Avatar (E63 88) can be partly attributed due to the affine transformation effect visible when comparing the reference to the test image: the test cover appears stretched

Figure 4: Misclassification of Madmen: incorrectly identified as Hitchhiker's DVD

and is also viewed on an angle contributing to its elongated affine distortion. Thus to demonstrate the results of my improvements, I focus on the misclassification of image E63-88 which has identifiable distinctive features as well as evidence towards improving the overall score of correctly classified image cases among the 10 candidates, which is relevant in terms of larger scale databases where there may be many more images with similar feature and color (gradient) schemes competing with the correct candidate for the top place.

As mentioned in the Method section, to handle cases of error due to heavy affine transformation I introduced the Harris-Affine [4] feature descriptors as an affine-invariant alternative to the native DoG SIFT descriptors, which have only the scale and rotation invariant property. While not as powerful as the original SIFT descriptors as it introduces many more misclassifications, this improvement results in the correct classification of image E63-88-Avatar (Figure 5).

Figure 5: Harris-Affine detector resulted in the correct classification of E63-88-Avatar

Harris-Affine is also evidenced to improve the score within the 1-10 candidates for other heavily affine transformed cases, shown in Figure 6.

The other direction of improvement I took was to include some kind of spatial verification as part of candidate selection process. While it is infeasible to perform verification with a Homography estimation via RANSAC for every database image due to the runtime cost, I added additional features to the DoG descriptors representing quickly estimated spatial statistics (adapted from [4]) within the neighbourhood of the feature. This resulted in improved scores for images 89 from E63 and 96 from Canon test sets (Figure 7). These cases represent images that can be qualitatively described as containing a set of common features shared between a multitude images (i.e. they don't stand out to our eyes). In these cases, an enforcement of correspondence of spatial relationships between features is necessary to prune the candidate set of the false

Figure 6: Affine Invariance: Titanic (E63, 45) rank raised from an 8 to a 1, The Prestige (Canon, 97) raised from 4 to 2

positives.

Figure 7: Spatial Stats: Escape (Canon, 96) rank raised from a 6 to a 1, America (E63, 89) raised from 9 to 4

5 Conclusion and future work

An implementation of a method outlined in [3] for recognizing DVD covers from a training set of 100 images using a vocabulary tree that hierarchically quantizes SIFT descriptors from image keypoints has been presented. The strength of this implementation includes the runtime optimization for scoring database images as described in [3] which can be summarized as scoring via a bottom-up post order partial tree traversal using a virtual inverted file index structure. I found that on the current training data set, I receive near perfect results, however a few directions for improvement exist. An improvement can be made in the runtime of the spatial verification step involving Homography estimation, and in particular the feature matching step. The current implementation uses the top 100 best feature matches based on Euclidean distance computed using an exhaustive search between all the feature pairs. A faster alternative can be proposed such as nearest neighbour matching using kd-trees [6].

Another improvement would be to continue to incorporate local and global spatial context as part of the scoring mechanism for the initial selection of the candidates. With a larger training set of DVD covers, I predict that features in their designs will begin to overlap heavily and as well in a full scale image database

robustly checking a selected percentage of 10% of the whole database is infeasible. Currently I simply include a few local spatial-context statistics as part of the descriptors used to build and traverse the words (nodes) in the vocabulary tree. A goal would be to ideally fully implement the ideas in [5] in the weighing portion of the scoring and compare with the existing results.

The VLFeat library [1] implementation of Affine-invariant feature descriptors [4] produces sub-optimal results and is slow, so another direction for improvement would be to work on an implementation of my own, as well as explore the research direction of affine-invariant features which are shown to produce very good results in the context of applications where object surfaces in images can be approximated by planar patches, such as our training set of DVD covers.

References

[1] Vedaldi, Andrea, and Brian Fulkerson. "VLFeat: An open and portable library of computer vision algorithms." Proceedings of the 18th ACM international conference on Multimedia. ACM, 2010.

[2] Lowe, David G. "Object recognition from local scale-invariant features." Computer vision, 1999. The proceedings of the seventh IEEE international conference on. Vol. 2. Ieee, 1999.

[3] Nister, David, and Henrik Stewenius. "Scalable recognition with a vocabulary tree." 2006 IEEE Computer Society Conference on Computer Vision and Pattern Recognition (CVPR'06). Vol. 2. IEEE, 2006.

[4] Mikolajczyk, Krystian, and Cordelia Schmid. "Scale affine invariant interest point detectors." International journal of computer vision 60.1 (2004): 63-86.

[5] Wang, Xiaoyu, et al. "Contextual weighting for vocabulary tree based image retrieval." 2011 International Conference on Computer Vision. IEEE, 2011.

[6] Muja, Marius, and David G. Lowe. "Scalable nearest neighbor algorithms for high dimensional data." IEEE Transactions on Pattern Analysis and Machine Intelligence 36.11 (2014): 2227-2240.