CHAPTER 4

Matching Local Features

In the previous chapter, we saw several specific techniques to detect repeatable interest points in an image (Section 3.2) and then robustly describe the local appearance at each such point (Section 3.3). Now, given an image and its local features, we need to be able to *match* them to similar-looking local features in other images (e.g., to model images of the specific objects we are trying to recognize). See Figure 4.1. To identify candidate matches, we essentially want to search among all previously seen local descriptors, and retrieve those that are nearest according to Euclidean distance in the feature space (such as the 128-dimensional "SIFT space").

Because the local descriptions are by design invariant to rotations, translations, scalings, and some photometric effects, this matching stage will be able to tolerate reasonable variations in viewpoint, pose, and illumination across the views of the object. Further, due to the features' distinctiveness, if we detect a good correspondence based on the local feature matches alone, we will already have a reasonable measure of how likely it is that two images share the same object. However, to strengthen confidence and eliminate ambiguous matches, it is common to follow the matching process discussed in this chapter with a check for geometric consistency, as we will discuss in Chapter 5.

The naive solution to identifying local feature matches is straightforward: simply scan through all previously seen descriptors, compare them to the current input descriptor, and take those within some threshold as candidates. Unfortunately, however, such a linear-time scan is usually unrealistic in terms of computational complexity. In many practical applications, one has to search for matches in a database of millions of features. Thus, efficient algorithms for *nearest neighbor* or *similarity search* are crucial. The focus of this chapter is to describe the algorithms frequently used in the recognition pipeline to rapidly match local descriptors.[1]

Specifically, in the first section of this chapter we overview both tree-based algorithms for exact near neighbor search, as well as approximate nearest neighbor algorithms (largely hashing-based) that are more amenable for high-dimensional descriptors. Then, in Section 4.2, we describe a frequently used alternative based on *visual vocabularies*. Instead of performing similarity search in the "raw" vector space of the descriptors, the vocabulary-based method first quantizes the feature space into discrete "visual words", making it possible to index feature matches easily with an inverted file.

[1] In fact, while our focus at this stage is on local feature matching for specific object recognition, most of the algorithms discussed are quite general and also come into play for other recognition-related search tasks, such as near-neighbor image retrieval.

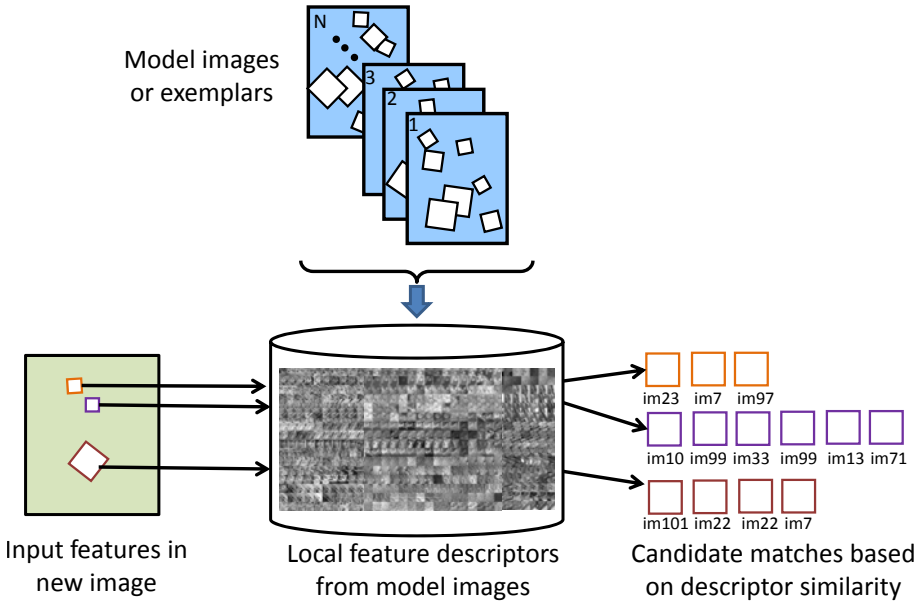


Figure 4.1: The goal when matching local features is to find those descriptors from any previously seen model (exemplar) that are near in the feature space to those local features in a novel image (depicted on the left). Since each exemplar image may easily contain on the order of hundreds to thousands of interest points, the database of descriptors quickly becomes very large; to make searching for matches practical, the database must be mapped to data structures for efficient similarity search.

4.1 EFFICIENT SIMILARITY SEARCH

What methods are effective for retrieving descriptors relevant to a new image? The choice first depends on the dimensionality of the descriptors. For low-dimensional points, effective data structures for exact nearest neighbor search are known, *e.g.*, *kd*-trees [Friedman et al., 1977]. For high-dimensional points, these methods become inefficient, and so researchers often employ *approximate* similarity search methods. This section overviews examples of both such techniques that are widely used in specific-object matching.

4.1.1 TREE-BASED ALGORITHMS

Data structures using spatial partitions and recursive hyperplane decomposition provide an efficient means to search low-dimensional vector data exactly. The *kd*-tree [Friedman et al., 1977] is one such approach that has often been employed to match local descriptors, in several variants (e.g., [Beis and Lowe, 1997, Lowe, 2004, Muja and Lowe, 2009, Silpa-Anan and Hartley, 2008]). The *kd*-tree is a binary tree storing a database of k-dimensional points in its leaf nodes. It recursively partitions the points into axis-aligned cells, dividing the points approximately in half by a line

perpendicular to one of the k coordinate axes. The division strategies aim to maintain balanced trees and/or uniformly shaped cells—for example, by choosing the next axis to split according to that which has the largest variance among the database points, or by cycling through the axes in order.

To find the point nearest to some query, one traverses the tree following the same divisions that were used to enter the database points; upon reaching a leaf node, the points found there are compared to the query. The nearest one becomes the "current best". While the point is nearer than others to the query, it need not be the absolute nearest (for example, consider a query occurring near the initial dividing split at the top of the tree, which can easily be nearer to some points on the other side of the dividing hyperplane). Thus, the search continues by backtracking along the unexplored branches, checking whether the circle formed about the query by the radius given by the current best match intersects with a subtree's cell area. If it does, that subtree is considered further, and any nearer points found as the search recurses are used to update the current best. If not, the subtree can be pruned. See Figure 4.2 for a sketch of an example tree and query.

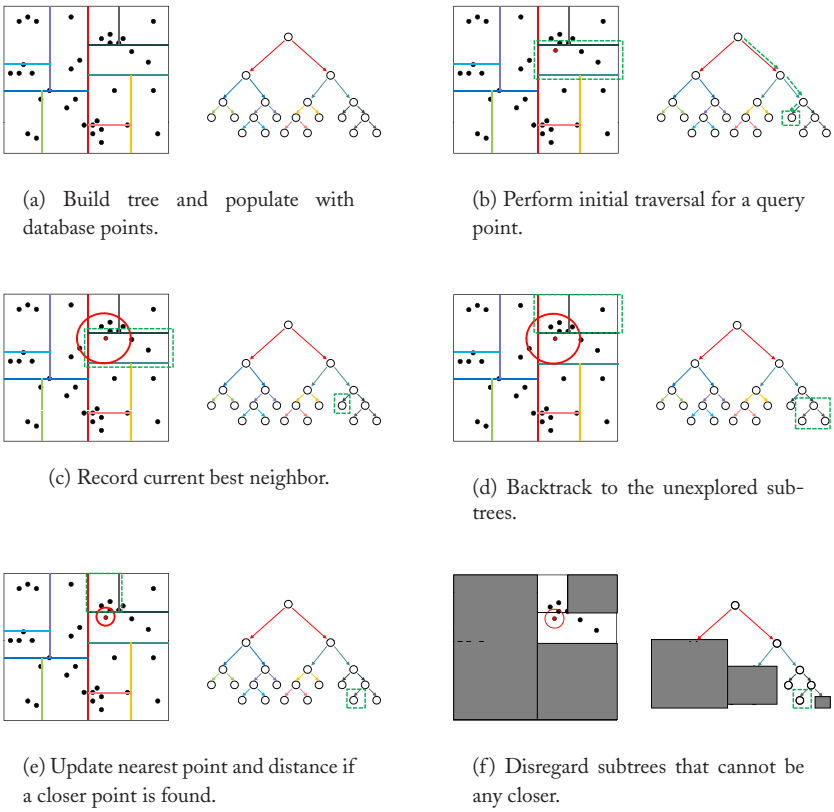
The procedure guarantees that the nearest point will be found.[2] Constructing the tree for N database points (an offline cost for a single database) requires $O(N \log N)$ time. Inserting points requires $O(\log N)$ time. Processing a query requires $O(N^{1-\frac{1}{k}})$ time, and the algorithm is known to be quite effective for low-dimensional data (i.e., fewer than 10 dimensions).

In high-dimensional spaces, however, the algorithm ends up needing to visit many more branches during the backtracking stage, and in general degrades to worst case linear scan performance in practice. The particular behavior depends not only on the dimension of the points, but also the distribution of the database examples that have been indexed, combined with the choices in how divisions are computed.

Other types of tree data structures can operate with arbitrary *metrics* [Ciaccia et al., 1997, Uhlmann, 1991], removing the requirement of having data in a vector space by exploiting the triangle inequality. However, similar to *kd*-trees, the metric trees in practice rely on good heuristics for selecting useful partitioning strategies, and, in spite of logarithmic query times in the expectation, also degenerate to a linear time scan of all items depending on the distribution of distances for the data set.

Since high-dimensional image descriptors are commonly used in object recognition, several strategies to mitigate these factors have been explored. One idea is to relax the search requirement to allow the return of *approximate* nearest neighbors, using a variant of *kd*-trees together with a priority queue [Arya et al., 1998, Beis and Lowe, 1997]. Another idea is to generate *multiple* randomized *kd*-trees (e.g., by sampling splits according to the coordinates' variance), and then process the query in all trees using a single priority queue across them [Silpa-Anan and Hartley, 2008]. Given the sensitivity of the algorithms to the data distribution, some recent work also attempts to automatically select algorithm configurations and parameters for satisfactory performance by using a cross-validation approach [Muja and Lowe, 2009]. Another interesting direction pursued for improving the efficiency and effectiveness of tree-based search involves integrating learning or the matching task into the

[2] Range queries and k-nn queries are also supported.



(a) Build tree and populate with database points.

(b) Perform initial traversal for a query point.

(c) Record current best neighbor.

(d) Backtrack to the unexplored subtrees.

(e) Update nearest point and distance if a closer point is found.

(f) Disregard subtrees that cannot be any closer.

Figure 4.2: Sketch of *kd*-tree processing. (a) The database points are entered into a binary tree, where each division is an axis-aligned hyperplane. (b) Given a new query (red point) for which we wish to retrieve the nearest neighbor, first the tree is traversed, choosing the left or right subtree at each node according to the query's value in the coordinate along which this division was keyed. The green dotted box denotes the cell containing the points in the leaf node reached by this query. (c) At this point, we know the current best point is that in the leaf node that is closest to the query, denoted with the outer red circle. (d) Then we backtrack and consider the other branch at each node that was visited, checking if its cell intersects the "current best" circle around the query. (e) If so, its subtree is explored further, and the current best radius is updated if a nearer point is found. (f) Continue, and prune subtrees once a comparison at its root shows that it cannot improve on the current nearest point. Courtesy of The Auton Lab, Carnegie Mellon University.

tree construction, for example by using decision trees in which each internal node is associated with a weak classifier built with simple measurements from the feature patches [Lepetit et al., 2005, Obdrzalek and Matas, 2005].

4.1.2 HASHING-BASED ALGORITHMS AND BINARY CODES

Hashing algorithms are an effective alternative to tree-based data structures. Motivated by the inadequacy of existing *exact* nearest-neighbor techniques to provide sub-linear time search for high-dimensional data (including the *kd*-tree and metric tree approaches discussed above), randomized *approximate* hashing-based similarity search algorithms have been explored. The idea in approximate similarity search is to trade off some precision in the search for the sake of substantial query time reductions. More specifically, guarantees are of the general form: if for a query point $\boldsymbol{q}$ there exists a database point $\boldsymbol{x}$ such that $d(\boldsymbol{q}, \boldsymbol{x}) \leq r$ for some search radius r, then, with high probability a point $\boldsymbol{x}'$ is returned such that $d(\boldsymbol{q}, \boldsymbol{x}') \leq (1 + \epsilon)r$. Otherwise, the absence of such a point is reported.[3]

4.1.2.1 Locality Sensitive Hashing

Locality-sensitive hashing (LSH) [Charikar, 2002, Datar et al., 2004, Gionis et al., 1999, Indyk and Motwani, 1998] is one such algorithm that offers sub-linear time search by hashing highly similar examples together in a hash table. The idea is that if one can guarantee that a randomized hash function will map two inputs to the same bucket with high probability only if they are similar, then, given a new query, one needs only to search the colliding database examples to find those that are most probable to lie in the input's near neighborhood.[4] The search is approximate, however, and one sacrifices a predictable degree of error in the search in exchange for a significant improvement in query time.

More formally, a family of LSH functions $\mathcal{F}$ is a distribution of functions where for any two objects $\boldsymbol{x}_i$ and $\boldsymbol{x}_j$,

$$\Pr_{h \in \mathcal{F}} \left[h(\boldsymbol{x}_i) = h(\boldsymbol{x}_j) \right] = sim(\boldsymbol{x}_i, \boldsymbol{x}_j), \tag{4.1}$$

where $sim(\boldsymbol{x}_i, \boldsymbol{x}_j) \in [0, 1]$ is some similarity function, and $h(\boldsymbol{x})$ is a hash function drawn from $\mathcal{F}$ that returns a single bit [Charikar, 2002]. Concatenating a series of b hash functions drawn from $\mathcal{F}$ yields b-dimensional hash keys. When $h(\boldsymbol{x}_i) = h(\boldsymbol{x}_j)$, $\boldsymbol{x}_i$ and $\boldsymbol{x}_j$ collide in the hash table. Because the probability that two inputs collide is equal to the similarity between them, highly similar objects are indexed together in the hash table with high probability. On the other hand, if two objects are very dissimilar, they are unlikely to share a hash key (see Figure 4.3). At query time, one maps the query to its hash bucket, pulls up any database instances also in that bucket, and then exhaustively searches only those (few) examples. In practice, multiple hash tables are often used, each with independently

[3] Variants of this guarantee for nearest neighbors (rather than r-radius neighbors) also exist.

[4] Locality sensitive hashing has been formulated in two related contexts—one in which the likelihood of collision is guaranteed relative to a threshold on the radius surrounding a query point [Indyk and Motwani, 1998], and another where collision probabilities are equated with a similarity function score [Charikar, 2002]. We use the latter definition here.

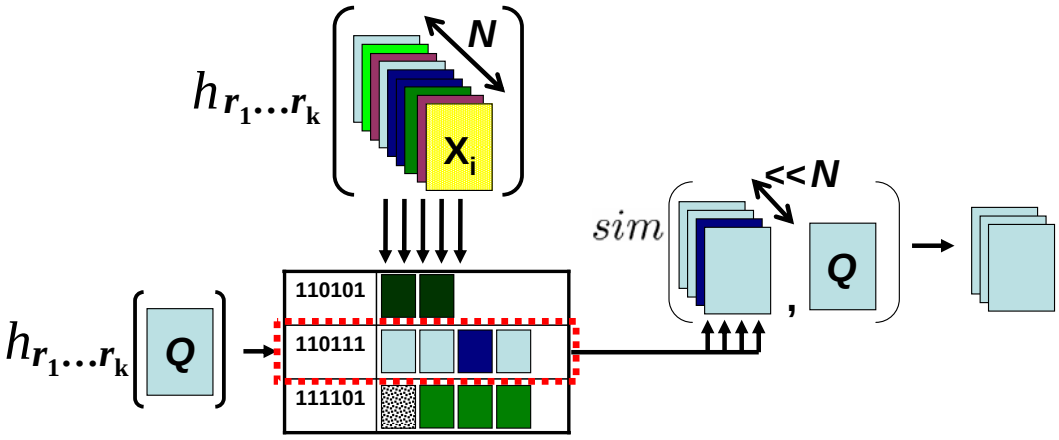


Figure 4.3: Overview of locality-sensitive hashing. If hash functions guarantee a high probability of collision for features that are similar under a metric of interest, one can search a large database in sub-linear time via locality sensitive hashing techniques [Charikar, 2002, Indyk and Motwani, 1998]. A list of k hash functions $h_{r_1}, \ldots, h_{r_k}$ are applied to map N database images to a hash table where similar items are likely to share a bucket. After hashing a query $\mathbf{Q}$, one must only evaluate the similarity between $\mathbf{Q}$ and the database examples with which it collides to obtain the approximate near-neighbors. From Kulis et al. [2009]. Copyright © 2009 IEEE.

drawn hash functions, and the query is compared against the union of the database points to which it hashes in all tables.

Given valid LSH functions, the query time for retrieving $(1+\epsilon)$-near neighbors is bounded by $O(N^{1/(1+\epsilon)})$ for the Hamming distance and a database of size N [Gionis et al., 1999]. One can therefore trade off the accuracy of the search with the query time required. Early LSH functions were developed to accommodate the Hamming distance [Indyk and Motwani, 1998], inner products [Charikar, 2002], and ℓ_p norms [Datar et al., 2004]. These methods were quickly adopted by vision researchers for a variety of image search applications [Shakhnarovich et al., 2006].

Since meaningful image comparisons for recognition often demand richer comparison measures, work in the vision community has developed novel locality-sensitive hash functions for additional classes of metrics. For example, an embedding of the normalized partial matching between two sets of local features is given in [Grauman and Darrell, 2007*a*] that allows sub-linear time hashing for the pyramid match kernel (see Section 10.2.1.1 below). A related form of hashing computes *sketches* of feature sets and allows search according to the sets' overlap [Broder, 1998]; this "Min-Hash" framework has been demonstrated and extended for near-duplicate detection and image search in [Chum et al., 2008]. Most recently, a kernelized form of LSH (KLSH) is proposed in [Kulis and Grauman, 2009], which makes it possible to perform locality-sensitive hashing for arbitrary kernel functions. Results are shown for various kernels relevant to object recognition, in-

cluding the χ^2 kernel that is often employed for comparing bag-of-words descriptors (to be defined below).

Aside from widening the class of metrics and kernels supportable with LSH, researchers have also considered how to integrate machine learning elements so that the hash functions are better suited for a particular task. For object recognition, this means that one wants hash functions that are more likely to map instances of the same object to the same hash buckets, or, similarly, patch descriptors from the same real-world object point to the same bucket. For example, Parameter Sensitive Hashing (PSH) [Shakhnarovich et al., 2003] is an LSH-based algorithm that uses boosting to select hash functions that best reflect similarity in a parameter space of interest. *Semi-supervised hash functions* make it possible to efficiently index data according to learned distances [Jain et al., 2008a, Kulis et al., 2009, Strecha et. al., 2010, Wang, Kumar and Chang, 2010]. Typically, supervision is given in the form of similar and dissimilar pairs of instances, and then while the metric learning algorithm updates its parameters to best capture those constraints, the hash functions' parameters are simultaneously adjusted. While most methods assume that all supervision is available in "batch" at the onset, online metric learners that accumulate constraints over time together with hash tables that can be updated incrementally have also been developed [Jain et al., 2008b].

4.1.2.2 Binary Embedding Functions

Embedding functions are a related mechanism that are used to map expensive distance functions into something more manageable computationally. Either constructed or learned, these embeddings aim to approximately preserve the desired distance function when mapping to a low-dimensional space that is more easily searchable with known techniques. Informally, given an original feature space $\mathcal{X}$ and associated distance function $d_{\mathcal{X}}$, the basic idea is to designate a function $f : \mathcal{X} \rightarrow \mathcal{E}$ that maps the inputs into a new space $\mathcal{E}$ with associated distance $d_{\mathcal{E}}$ in such a way that $d_{\mathcal{E}}(f(\boldsymbol{x}), f(\boldsymbol{y})) \approx d_{\mathcal{X}}(\boldsymbol{x}, \boldsymbol{y})$, for any $\boldsymbol{x}, \boldsymbol{y} \in \mathcal{X}$. Often the target space for the embedding is the Hamming space. Such binary codes have the advantage of requiring minimal memory; they also permit fast bit counting routines for the Hamming distance, and can be indexed directly using the computer's memory addresses.

Work in the vision and learning community has developed useful embedding functions that aim to preserve a variety of similarity metrics with simple low-dimensional binary codes. For example, the BoostMap [Athitsos et al., 2004] and Boosted Similarity Sensitive Coding (BoostSSC) [Shakhnarovich, 2005] algorithms learn an embedding using different forms of boosting, combining multiple weighted 1D embeddings so as to preserve the proximity structure given by the original distance function. Building on this notion, more recent work develops Semantic Hashing algorithms that train embedding functions using boosting or multiple layers of restricted Boltzmann machines [Salakhutdinov and Hinton, 2007, Torralba et al., 2008]; results show the impact for searching Gist image descriptors [Torralba et al., 2008]. Embeddings based on random projections have also been explored for shift-invariant kernels, which includes a Gaussian kernel [Raginsky and Lazebnik, 2009].

Such methods are related to LSH in the sense that both seek small "keys" that can be used to encode similar inputs, and often these keys exist in Hamming space. However, note that while hash functions also typically map the data to binary strings (the "hash keys"), in that case the codes serve to insert instances into buckets, whereas technically the embedding function outputs are treated as a new feature space in which to perform the similarity search.

4.1.3 A RULE OF THUMB FOR REDUCING AMBIGUOUS MATCHES

When matching local feature sets extracted from real-world images, many features will stem from background clutter and will therefore have no meaningful neighbor in the other set. Other features lie on repetitive structures and may therefore have ambiguous matches (for example, imagine an image containing a building with many identical windows). Hence, one needs to find a way to distinguish reliable matches from unreliable ones. This cannot be done based on the descriptor distance alone, since some descriptors are more discriminative than others.

An often-used strategy (initially proposed by Lowe [2004]) is to consider the ratio of the distance to the closest neighbor to that of the second-closest one as a decision criterion. Specifically, we identify the nearest neighbor local feature originating from an exemplar in the database of training images, and then consider the second nearest neighbor that originates from a different object than the nearest neighbor feature. If the ratio of the distance to the first neighbor over the distance to the second neighbor is relatively large, this is a sign that the match may be ambiguous. Similarly, if the ratio is low, it suggests that it is a reliable match. This strategy effectively penalizes features that come from a densely populated region of feature space and that are therefore more ambiguous. By comparing the probability density functions of correct and incorrect matches in quantitative experiments, Lowe arrives at the recommendation to reject all matches in which the distance ratio is greater than 0.8, which in his experiments eliminated 90% of the false matches while discarding less than 5% correct matches [Lowe, 2004].

4.2 INDEXING FEATURES WITH VISUAL VOCABULARIES

In this section, we overview the concept of a *visual vocabulary*—a strategy that draws inspiration from the text retrieval community and enables efficient indexing for local image features. Rather than preparing a tree or hashing data structure to aid in direct similarity search, the idea is to *quantize* the local feature space. By mapping the local descriptors to discrete tokens, we can then "match" them by simply looking up features assigned to the identical token.

In the following, we first describe the formation of visual words (Sections 4.2.1 through 4.2.3), and then describe their utility for indexing (Section 4.2.4). Note that we will return to this representation later in Section 8.1 in the context of object categorization, as it is the basis for the simple but effective "bag-of-words" image descriptor.

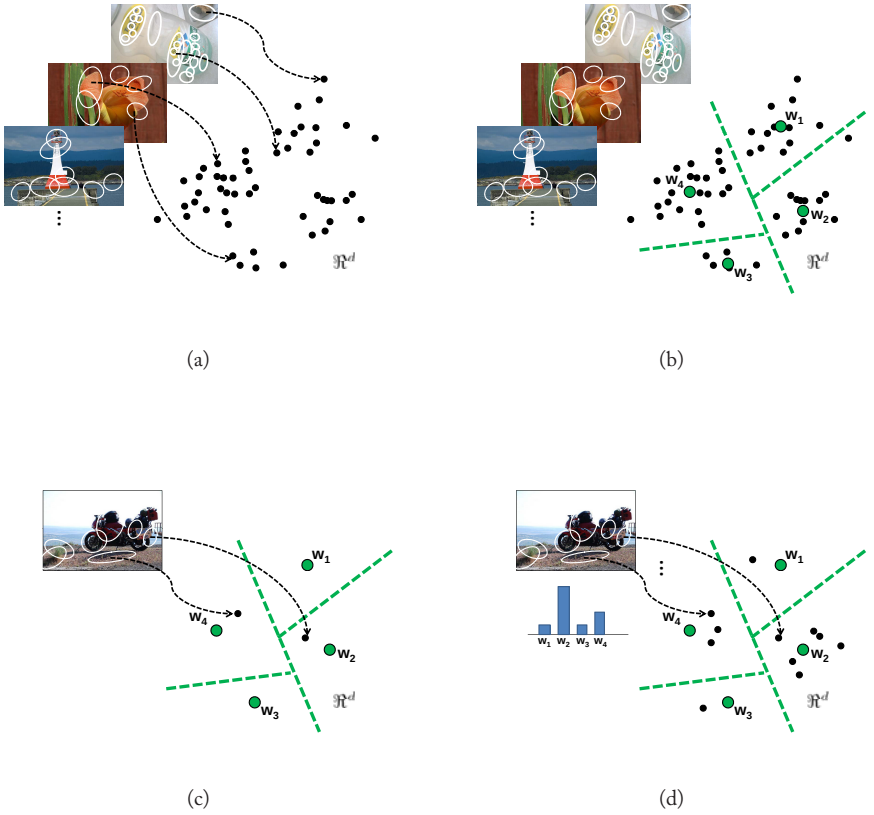


Figure 4.4: A schematic to illustrate visual vocabulary construction and word assignment. (a) A large corpus of representative images are used to populate the feature space with descriptor instances. The white ellipses denote local feature regions, and the black dots denote points in some feature space, *e.g.*, SIFT. (b) Next, the sampled features are clustered in order to quantize the space into a discrete number of visual words. The visual words are the cluster centers, denoted with the large green circles. The dotted green lines signify the implied Voronoi cells based on the selected word centers. (c) Now, given a new image, the nearest visual word is identified for each of its features. This maps the image from a set of high-dimensional descriptors to a list of word numbers. (d) A bag-of-visual-words histogram can be used to summarize the entire image (see Section 8.1). It counts how many times each of the visual words occurs in the image.

4.2.1 CREATING A VISUAL VOCABULARY

Methods for indexing and efficient retrieval with text documents are mature, and effective enough to operate with millions or billions of documents at once [Baeza-Yates and Ribeiro-Neto, 1999]. Documents of text contain some distribution of words, and thus can be compactly summarized by their word counts (known as a bag-of-words). Since the occurrence of a given word tends to be sparse across different documents, an index that maps words to the files in which they occur can map a keyword query directly to potentially relevant content. For example, if we query a document database with the word "car", we should immediately eliminate the many documents that never mention the word "car".

What cues, then, can one take from text processing to aid visual search? An image is a sort of document, and (using the representations introduced in Chapter 3) it contains a set of local feature descriptors. However, at first glance, the analogy would stop there: text words are discrete "tokens", whereas local image descriptors are high-dimensional, real-valued feature points. How could one obtain discrete "visual words"?

To do so, we must impose a quantization on the feature space of local image descriptors. That way, any novel descriptor vector can be coded in terms of the (discretized) region of feature space to which it belongs. The standard pipeline to form a so-called "visual vocabulary" consists of (1) collecting a large sample of features from a representative corpus of images and (2) quantizing the feature space according to their statistics. Often simple *k-means clustering* is used to perform the quantization; one initializes the k cluster centers with randomly selected features in the corpus, and then iterates between updating each point's cluster membership (based on which cluster center it is nearest to) and updating the k means (based on the mean of the points previously assigned to each cluster). In that case, the visual "words" are the k cluster centers, and the size of the vocabulary k is a user-supplied parameter. Once the vocabulary is established, the corpus of sampled features can be discarded. Then a novel image's features can be translated into words by determining which visual word they are nearest to in the feature space (*i.e.*, based on the Euclidean distance between the cluster centers and the input descriptor). See Figure 4.4 for a diagram of the procedure.

Drawing inspiration from text retrieval methods, Sivic and Zisserman proposed quantizing local image descriptors for the sake of rapidly indexing video frames with an inverted file [Sivic and Zisserman, 2003]. They showed that local descriptors extracted at interest points could be mapped to visual words by computing prototypical descriptors with k-means clustering, and that having these tokens enabled faster retrieval of frames containing the same words. Furthermore, they showed the potential of exploiting a *term frequency-inverse document frequency* weighting on the words, which de-emphasizes those words that are common to many images and thus possibly less informative, and a *stop-list*, which ignores extremely frequent words that appear in nearly every image (analogous to "a" or "the" in text).

What will a visual word capture? The answer depends on several factors, including what corpus of features is used to build the vocabulary, the number of words selected, the quantization algorithm used, and the interest point or sampling mechanism chosen for feature extraction. Intuitively, the

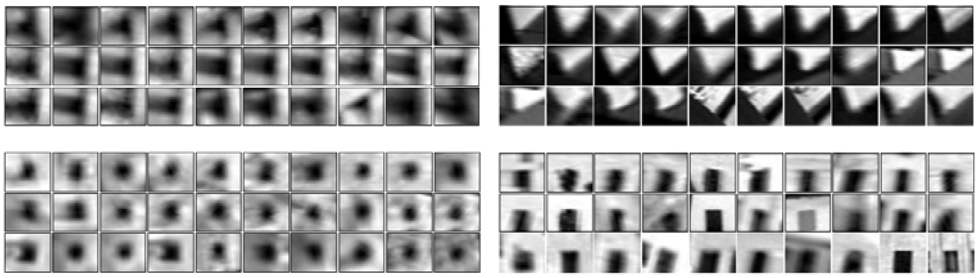


Figure 4.5: Four examples of visual words. Each group shows instances of patches that are assigned to the same visual word. From Sivic and Zisserman [2003]. Copyright © 2003 IEEE.

larger the vocabulary, the more fine-grained the visual words. In general, patches assigned to the same visual word should have similar low-level appearance (see Figure 4.5). Particularly when the vocabulary is formed in an unsupervised manner, there are no constraints that the common types of local patterns be correlated with object-level parts. However, in later chapters we will see some methods that use visual vocabularies or codebooks to provide candidate parts to a part-based category model.

4.2.2 VOCABULARY TREES

The discussion above assumes a flat quantization of the feature space, but many current techniques exploit hierarchical partitions [Bosch et al., 2007*a*, Grauman and Darrell, 2005, 2006*a*, Moosmann et al., 2006, Nister and Stewenius, 2006, Yeh et al., 2007]. In particular, the *vocabulary tree* approach [Nister and Stewenius, 2006] uses hierarchical k-means to recursively subdivide the feature space, given a choice of the branching factor and number of levels. Vocabulary trees offer a significant advantage in terms of the computational cost of assigning novel image features to words—from linear to logarithmic in the size of the vocabulary. This in turn makes it practical to use much larger vocabularies (*e.g.*, on the order of one million words).

Experimental results suggest that these more specific words (smaller quantized bins) are particularly useful for matching features for specific instances of objects [Nister and Stewenius, 2006, Philbin et al., 2007, 2008]. Since quantization entails a hard-partitioning of the feature space, it can also be useful in practice to use multiple randomized hierarchical partitions, and/or to perform a soft assignment in which a feature results in multiple weighted entries in nearby bins.

4.2.3 CHOICES IN VOCABULARY FORMATION

An important concern in creating the visual vocabulary is the choice of data used to construct it. Generally, researchers report that the most accurate results are obtained when using the same data source to create the vocabulary as is going to be used for the classification or retrieval task. This

can be especially noticeable when the application is for specific-level recognition rather than generic categorization. For example, to index the frames from a particular movie, the vocabulary made from a sample of those frames would be most accurate; using a second movie to form the vocabulary should still produce meaningful results, though likely weaker accuracy. When training a recognition system for a particular set of categories, one would typically sample descriptors from training examples covering all categories to try and ensure good coverage. That said, with a large enough pool of features taken from diverse images (admittedly, a vague criterion), it does appear workable to treat the vocabulary as "universal" for any future word assignments.

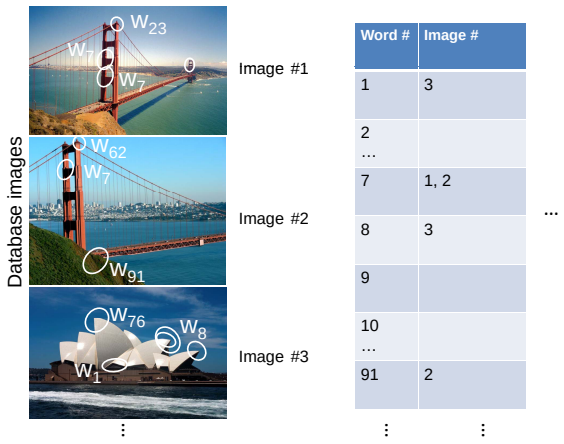
Furthermore, researchers have developed methods to inject supervision into the vocabulary [Moosmann et al., 2006, Perronnin et al., 2006, Winn et al., 2005], and even to integrate the classifier construction and vocabulary formation processes [Yang et al., 2008]. In this way, one can essentially learn an application-specific vocabulary.

The choice of feature detector or interest operator will also have notable impact on the types of words generated. Factors to consider are (1) the invariance properties required, (2) the type of images to be described, and (3) the computational cost allowable. Using an interest operator (*e.g.*, a DoG detector) yields a sparse set of points that is both compact and repeatable due to the detector's automatic scale selection. For specific-level recognition (*e.g.*, identifying a particular object or landmark building), these points can also provide an adequately distinct description. A common rule of thumb is to use multiple complementary detectors; that is, to combine the outputs from a corner-favoring interest operator with those from a blob-favoring interest operator. See Section 8.1.4 of Chapter 8 for a discussion of visual word representations and choices for category-level recognition.

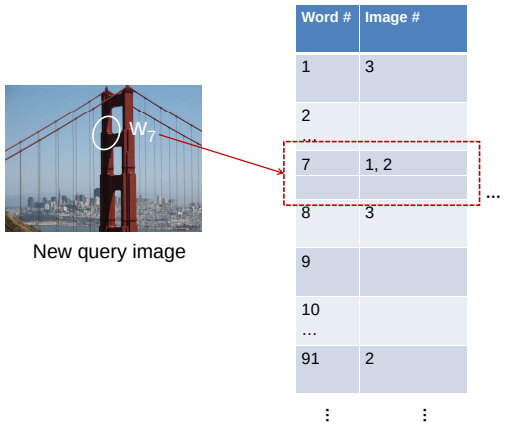
4.2.4 INVERTED FILE INDEXING

Visual vocabularies offer a simple but effective way to index images efficiently with an *inverted file*. An inverted file index is just like an index in a book, where the keywords are mapped to the page numbers where those words are used. In the visual word case, we have a table that points from the word number to the indices of the database images in which that word occurs. For example, in the cartoon illustration in Figure 4.6, the database is processed and the table is populated with image indices in part (a); in part (b), the words from the new image are used to index into that table, thereby directly retrieving the database images that share its distinctive words.

Retrieval via the inverted file is faster than searching every image, assuming that not all images contain every word. In practice, an image's distribution of words is indeed sparse. Since the index maintains no information about the relative spatial layout of the words per image, typically a spatial verification step is performed on the images retrieved for a given query, as we discuss in detail in the following chapter.



(a) All database images are loaded into the index mapping words to image numbers.



(b) A new query image is mapped to indices of database images that share a word.

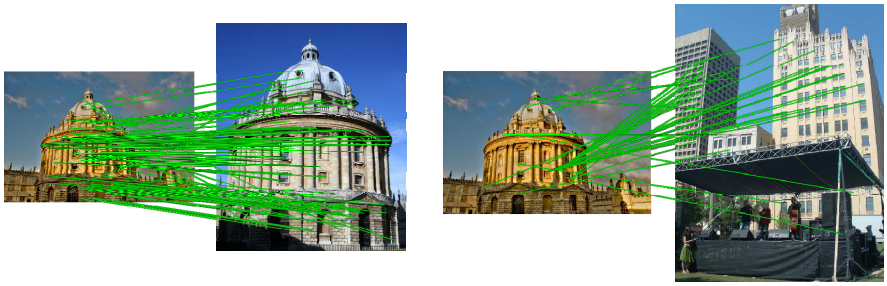
Figure 4.6: Main idea of an inverted file index for images represented by visual words.

4.3 CONCLUDING REMARKS

In short, the above methods offer ways to reduce the computational cost of finding similar image descriptors within a large database. While certainly crucial to practical applications of specific object recognition based on local features (the focus of this segment of the lecture), they are also commonly used for other search problems ranging from image retrieval to example-based category recognition, making this section also relevant to generic category algorithms that we will discuss starting in Chapter 7.

Which matching algorithm should be used when? The tree or hashing algorithms directly perform similarity search, offering the algorithm designer the most control on how candidate matches are gathered. In contrast, a visual vocabulary corresponds to a fixed quantization of a vector space, and lacks such control. On the other hand, a visual vocabulary approach has the ability to compactly summarize all local descriptors in an image or window, allowing a fast check for overall agreement between two images. In general, the appropriate choice for an application will depend on the similarity metric that is required for the search, the dimensionality of the data, the available online memory, and the offline resources for data structure setup or other overhead costs.

At this point, we have shown how to detect, describe, and match local features. Good local feature matches between images can alone suggest a specific object has been found; however, to discount spurious matches or to recognize an object from very sparse local features, it is important to also perform a geometric verification stage (see Figure 4.7). Thus, the following chapter closes our discussion of specific object recognition with techniques to verify spatial consistency of the matches.



(a) Matched features alone do not ensure a confident object match.

(b) Candidate matches must next be verified for geometric consistency.

Figure 4.7: The candidate feature matches established using the methods described in this chapter may strongly suggest whether a specific object is present, but are typically verified for geometric consistency. In this example, the good appearance matches found in the top right example can be discarded once we find they do not fit a geometric transformation well, whereas those found in the top left example will check out in terms of both appearance and geometric consistency. Courtesy of Ondrej Chum.