10 Object Recognition Using Locality-Sensitive Hashing of Shape Contexts

Andrea Frome and Jitendra Malik

At the core of many computer vision algorithms lies the task of finding a correspondence between image features local to a part of an image. Once these features are calculated, matching is commonly performed using a nearest-neighbor algorithm. In this chapter, we focus on the topic of object recognition, and examine how the complexity of a basic feature-matching approach grows with the number of object classes. We use this as motivation for proposing approaches to feature-based object recognition that grow sublinearly with the number of object classes.

10.1 Regional Descriptor Approach

Our approach to object recognition relies on the matching of feature vectors (also referred to here as *features*) which characterize a region of a two-dimensional (2D) or 3D image, where by "3D image" we mean the point cloud resulting from a range scan. We use the term *descriptor* to refer to the method or "template" for calculating the feature vector. There are several lines of work which develop descriptors for use in object recognition. [15] introduced jet-based features; [12] introduced the scale- and rotation-invariant feature transform (SIFT) descriptor for recognition and matching in intensity images; [10] describes the *spin image* descriptor for recognizing objects by shape in 3D range scans; [3] describes a histogram-based descriptor for recognizing objects in 2D images by shape, called the *shape context*, which is extended to the *generalized shape context* in [14]; and [6] presents the *3D shape context*, an extension of the shape context to three dimensions, and experimentally evaluates its performance against the spin image descriptor in difficult range image recognition tasks.

The spin image and shape context descriptors share a *regional* approach to feature calculation; the features incorporate information within a *support region* of the image centered at a chosen *basis point*. The locality of these *regional descriptors* make them robust to clutter and occlusion, while at the same time each feature contains more infor-

mation than purely local descriptors due to their extended support. In some recognition approaches the features are computed at particularly salient locations in the image determined by an *interest operator*, such as in [12]. In other approaches, including the cited works that make use of spin images and shape contexts, the basis points at which features are computed are chosen randomly and are not required to posses any distinguishing characteristics.

Object recognition algorithms typically work by calculating features from a query image and comparing those features to other features previously calculated from a set of *reference* images, and return a decision about which object or image from among the reference set best matches the query image. We consider *full object recognition* to be achieved when the algorithm returns the identity, location, and position of an object occurring in a query image. Our discussion in this chapter focuses on a relaxed version of the full recognition problem where the algorithm returns a *short list* of objects, at least one of which occurs *somewhere* in the image. An algorithm solving this relaxed recognition problem can be used to prune a large field of candidate objects for a more expensive algorithm which solves the full recognition problem. In a more complex system it could be used as an early stage in a cascade of object recognition algorithms which are increasingly more expensive and discriminating, similar in spirit to the cascade of classifiers made popular in the vision community by [16]. A pruning step or early cascade stage is effective when it reduces the total computation required for full recognition and does not reduce the recognition performance of the system. To this end, we want a short-list recognition algorithm which (1) minimizes the number of misses, that is, the fraction of queries where the short list does not include any objects present in the query image, and (2) minimizes its computational cost.

Object recognition algorithms based on features have been shown to achieve high recognition rates in the works cited above and many others, though often in a an easy or restricted recognition setting. We will demonstrate methods for speeding a simple matching algorithm while maintaining high recognition accuracy in a difficult recognition task, beginning with an approach which uses an exhaustive k-nearest-neighbor (k-NN) search to match the *query features* calculated from a query image to the *reference features* calculated from the set of reference images. Using the distances calculated between query and reference features, we generate a short list of objects which might be present in the query image.

It should be noted that the method we examine does not enforce relative geometric constraints between the basis points in the query and reference images, and that most feature-based recognition algorithms do use this additional information. For example, for reference features

centered at basis points p_1 and p_2 and query features centered at basis points q_1 and q_2, if p_1 is found to be a match for q_1, p_2 a match for q_2, and we are considering rigid objects, then it should be the case that the distance in the image between p_1 and p_2 should be similar to the distance between q_1 and q_2. There are many methods for using these types of constraints, [8], RANSAC, and [5] to name a few. We choose not to use these constraints in order to demonstrate the power of matching feature vectors alone. A geometric-based pruning or verification method could follow the matching algorithms described in this chapter.

The drawback of an exhaustive search of stored reference features is that it is expensive, and for the method to be effective as a pruning stage, it needs to be fast. Many of the descriptors listed above are high-dimensional; in the works cited, the scale-invariant feature transform (SIFT) descriptor has 160 dimensions, the spin image has about 200, the 2D shape context has 60 (the generalized version has twice as many for the same number of bins), and the 3D shape context has almost 2000. The best algorithms for exact nearest-neighbor search in such high-dimensional spaces requires time linear in the number of reference features. In addition, the number of reference features is linear in the number of example objects the system is designed to recognize. If we aim to build systems that can recognize hundreds or thousands of example objects, then the system must be able to run in time sublinear in the number of objects.

The goal of this chapter is to present ways to maintain the recognition accuracy of this "short-list" algorithm while reducing its computational cost. Locality-sensitive hashing (LSH) plays a key role in a final approach that is both accurate and has complexity sublinear in the number of objects being recognized. In our experiments we will be evaluating variations on the basic matching method with the 3D shape context descriptor.

10.2 Shape Context Descriptors

We will focus on a type of descriptor called the *shape context*. In their original form, shape context features characterize shape in 2D images as histograms of edge pixels (see [2]). In [14] the authors use the same template as 2D shape contexts but capture more information about the shape by storing aggregate edge orientation for each bin. In [4], the authors developed the notion of *geometric blur* which is an analog to the 2D shape context for continuous-valued images. We extended the shape context to three dimensions in [6], where it characterizes shape by histogramming the position of points in a range scan. In the rest

of this section, we describe the basics of the 2D and 3D shape context descriptors in more detail, and introduce the experimental framework used in the rest of the chapter.

10.2.1 Two-dimensional Shape Contexts

To calculate a 2D shape context feature from an image, first run your favorite edge detector on the image. Next, choose a coordinate in the edge map to be a basis point, and imagine a radar-like template like the one in figure 10.1 laid down over the image, centered at that point. The lines of this pattern divide the image into regions, each of which corresponds to one dimension of the feature vector. The value for the dimension is calculated as the number of edge pixels which fall into the region. This feature vector can be thought of as a histogram which summarizes the spatial distribution of edges in the image relative to the chosen basis point. Each region in the template corresponds to one bin in the histogram, and we use the term *bin* to refer to the region in the image as well as the dimension in the feature vector. Note that if the bins were small enough to each contain one pixel, then the histogram would be an exact description of the shape in the support region.

This template has a few advantageous properties. The bins farther from the center summarize a larger area of the image than those close to the center. The gives a foveal effect; the feature more accurately captures and weights more heavily information toward the center. To accentuate this property of shape context descriptors, we use equally spaced log-radius divisions. This causes bins to get "fuzzy" more quickly as you move from the center of the descriptor.

When comparing two shape context features, even if the shapes from which they are calculated are very similar, the following must also be similar in order to register the two features as a good match:

- orientation of the descriptor relative to the object
- scale of the object

To account for different scales, we can search over scale space, e.g., by calculating a Gaussian pyramid for our query image, calculating query features in each of the down- and upscaled images, and finding the best match at each scale. We could sidestep the issue of orientation by assuming that objects are in a canonical orientation in the images, and orient the template the same way for all basis points. Or, to make it robust to variation, we could orient the template to the edge gradient at the basis point or include in our training set images at different orientations.

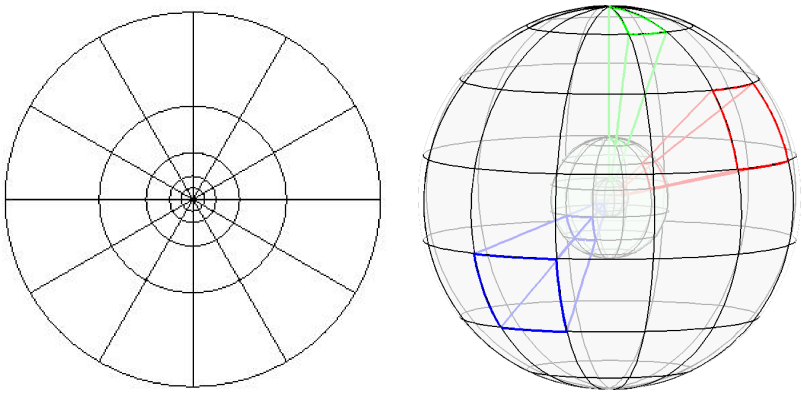


Figure 10.1 Example templates for the shape contexts: (a) for 2D, (b) for 3D. The number of divisions shown are not the same as we used in our experiments.

10.2.2 Three-dimensional Shape Contexts

In order to apply the same idea to range images, we extended the template to three dimensions (see figure 10.1 for a visualization). We use a spherical support volume, and divide the sphere at regular angles along the elevation and azimuth dimensions. Again we use the log-radius division as with the 2D shape contexts. The value for a bin is the count of the number of points in three dimensions from the raw range image that fall into its region.

When working with 3D range scans, we do not need to consider differences in scale since the scanner measurements in both the query and reference scans are reported in real-world dimensions. In three dimensions there are 2 degrees of freedom in the orientation of the template. We solve half of the problem by aligning the north pole with the surface normal calculated at the basis point. However, this still leaves a free rotation in the azimuth dimension. We account for that freedom with sampling; if we divide the azimuth into twelve sections, then we include in the reference set twelve discrete rotations of the feature vector. Since we are rotating the reference features, we do not need to rotate the query features. We could just as easily rotate the query features instead, but it should become clear why we rotate the reference features when we discuss our use of LSH later in the chapter.

Spin images, another descriptor used for 3D object recognition presented in [10], is very similar to the 3D shape context. It differs primarily in the shape of its support volume and its approach to the azimuth degree of freedom in the orientation: the spin image sums the counts over changes in azimuth. See [6] for a direct comparison between spin images and 3D shape contexts in similar experiments.

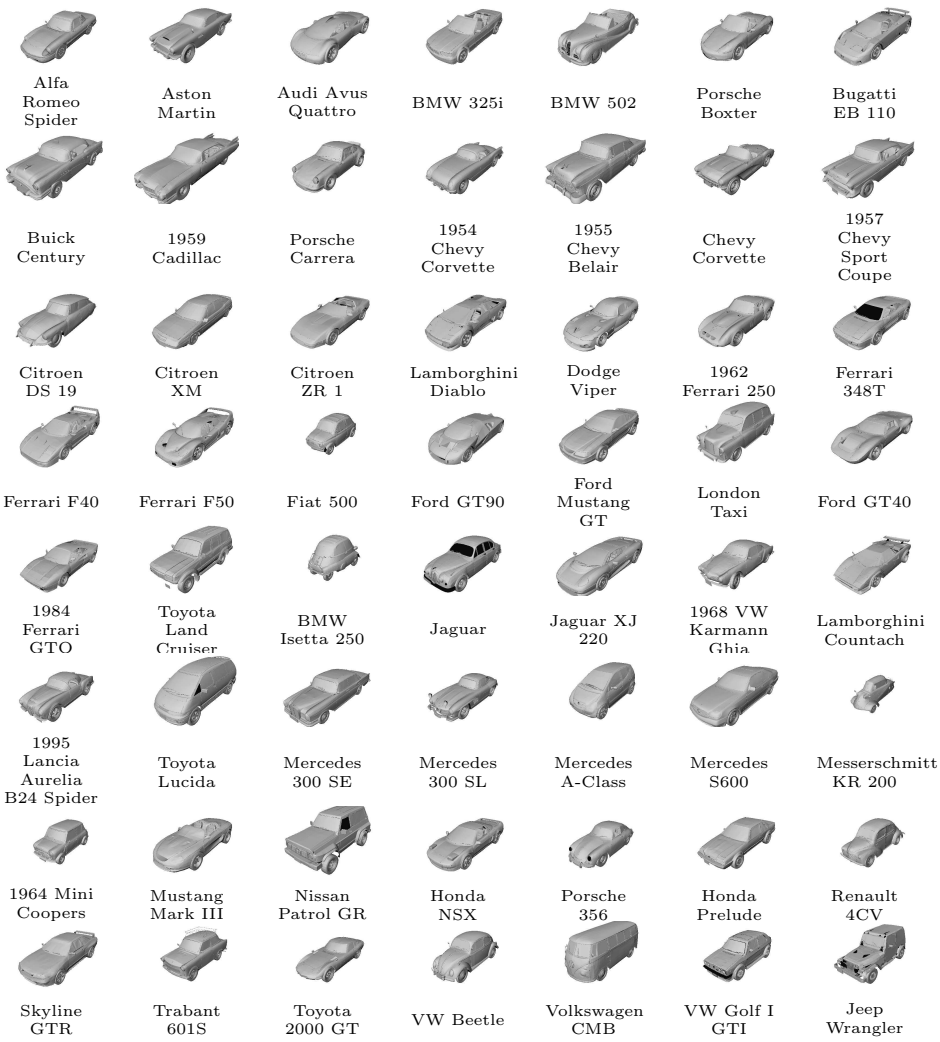


Figure 10.2 The fifty-six car models used in our experiments.

10.2.3 Experiments with Three-dimensional Shape Contexts

In this subsection, we introduce the data set that we use throughout the chapter to evaluate recognition with 3D shape contexts. The 3D shape contexts we calculate are the same as those used in [6]: they have twelve azimuth divisions, eleven elevation divisions, and fifteen radial divisions. These values were chosen after a small amount of experimentation with a similar data set.

The range scans from which we calculate the features are simulated from a set of fifty-six 3D car models, and are separated into reference scans (our training set) and query scans. The full models are shown in figure 10.2. The reference scans were generated from a viewpoint at 45 degrees elevation (measured from the horizon) and from four different azimuth positions, spaced 90 degrees apart around the car, starting from an angle halfway between the front and side views of

(a) (b) (c)

Figure 10.3 The *top row* shows scans from the 1962 Ferrari 250 model, and the *bottom* scans are from the Dodge Viper. The scans in column *(a)* are the query scans at 30 degrees elevation and 15 degrees azimuth with $\sigma = 5$ cm noise, and those in *(b)* are from the same angle but with $\sigma = 10$ cm noise. With 10 cm noise, it is difficult to differentiate the vehicles by looking at the 2D images of the point clouds. Column *(c)* shows the reference scans closest in viewing direction to the query scans (45 degrees azimuth and 45 degrees elevation).

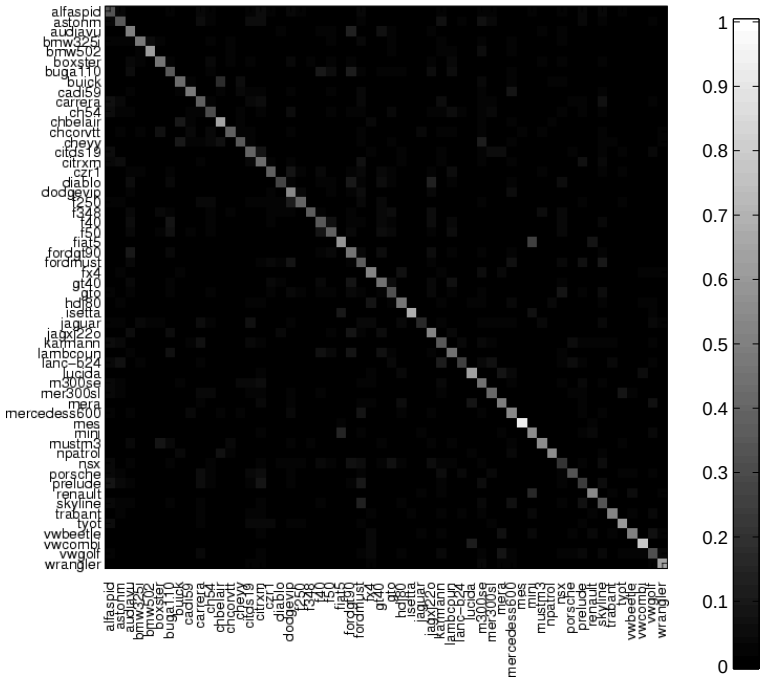
the vehicle. The query scans were generated from a viewpoint at 30 degrees elevation and at one azimuth position 15 degrees different from the nearest reference scan. We also added Gaussian noise to the query scans along the viewing direction, with either a 5 cm or 10 cm standard deviation. This amount of noise is comparable to or greater than the noise one could expect from a quality scanner. An example of the noisy query scans next to the nearest reference scan for two of the car models is shown in figure 10.3.

From the reference scans, we calculated normals at the points, and calculated 3D shape context features at basis points sampled uniformly over the surface, an average of 373 features per scan. For each noisy query scan, we calculated the normals, then calculated features at 300 randomly chosen basis points. Now we can describe our first experiment.

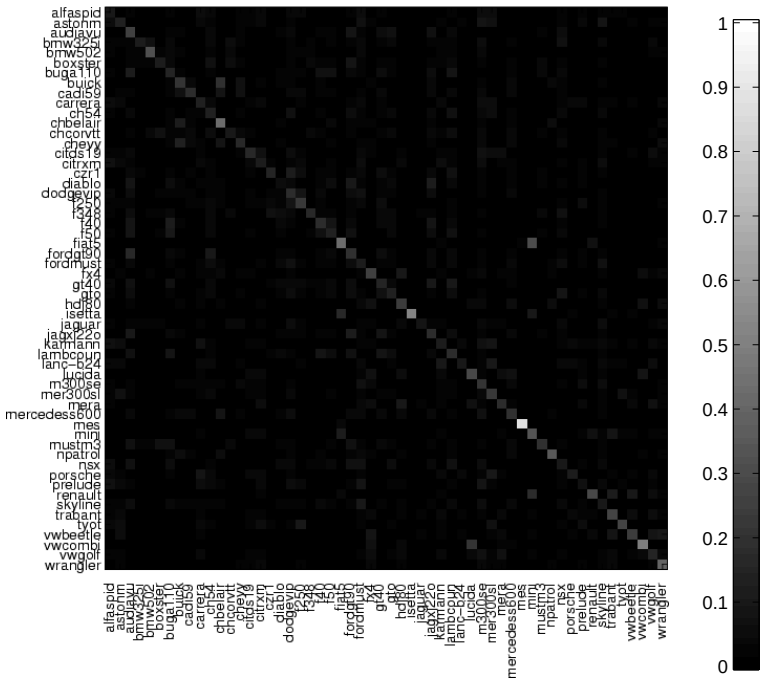
10.3 Basic Matching Experiment

Experiment 1

Given a query scan, we want to return the best match from among the reference scans. Each of the 300 query features from the query scan casts a "vote" for one of the fifty-six car models, and the best match to the query scan as a whole is the model which received the most votes. We determine a query feature's vote by finding its nearest neighbor from among the reference features, and awarding the vote to the model that produced that reference feature. We could also give the n best matches by ordering the models by the number of votes received, and returning the top n from that list. We run this procedure for all fifty-six query scans and calculate the recognition rate as the percentage of the fifty-six query scans which were correctly identified.



(a)



(b)

Figure 10.4 Confusion matrices for experiment 1 for *(a)* 5 cm and *(b)* 10 cm noise queries. Each row corresponds to one query and each column to one reference model. A square in the matrix represents the percentage of votes for the column's reference models by the row's query, where each row sums to 100%. The scale at the far right maps the colors to numbers. The strong diagonal in *(a)* means that most of the votes for each 5 cm noise query went to the correct corresponding model, giving us 100% recognition in the top choice. There was more confusion in the 10 cm query, with fifty-two of the fifty-six models correctly identified in the top choice, and 100% recognition within the top four choices.

The results we get are shown as confusion matrices in figure 10.4 for the 5 cm and 10 cm queries. Each row corresponds to the results for one query scan, and each column to one car model (four reference scans). Each square is a color corresponding to the number of votes that the query gave for the model. If every query feature voted for the correct model, then the matrix would have a dark red diagonal and otherwise be dark blue. Perfect recognition is achieved when the diagonal has the largest number from each row, which is the case here for the 5 cm noise data set. In the 10 cm experiment, we got fifty-two out of fifty-six queries correct, giving a recognition rate of 92.86%. The correct model is always in the top four matches, so if we are want a short list of depth four or greater, then our recognition is 100%.

10.3.1 Complexity and Computation Time

Take

- m to be the number of reference images (assume one object per reference image),
- n_r the number of features calculated per reference image,
- n_q the number of features calculated per query image,
- d the dimensionality of the features,
- p the number of pixels or points in the query scene, and
- s the number of scales over which we need to search.

Let us first look at the cost of computing each query feature. For the 2D shape context, we need to compute edge features at all the pixels that may lie in one of the descriptors' support, and then count the number of edge pixels in each bin. This gives us a preprocessing cost of $O(p)$ and a computation cost of $O(p)$ for each query feature, for a total of $O(p) + O(p \cdot n_q)$ for the query image as a whole.

For the 3D shape context, we do not need to preprocess all the points in the scan, just the neighborhood around the basis point to get the normal at that point. We still need to look through the points in the scene to calculate the bin contents, giving a cost of $O(p \cdot n_q)$.

Once we have the query features, we need to search through the $m \cdot n_r$ reference features. If we are performing an exact nearest-neighbor search as in experiment 1, we need to calculate the distance between each of those reference features and each of the query features. The cost for that is $O(m \cdot n_r \cdot n_q \cdot d)$. If we are working with 2D shape contexts, then we may also have to search over scale, increasing the cost to $O(m \cdot n_r \cdot n_q \cdot d \cdot s)$.

For the 3D shape contexts, this gives us a total cost of $O(p \cdot n_q) + O(m \cdot n_r \cdot n_q \cdot d)$. In experiment 1, $n_q = 300$, $m = 224$, $n_r = 4476$

(average of 373 features per reference scan times the twelve rotations through the azimuth for each), and $d = 1980$ ($11 \times 12 \times 15$), so the second term sums to 5.96×10^{11} pairs of floating point numbers we need to examine in our search. On a 1.3 GHz 64-bit Itanium 2 processor, the comparison of 300 query features to the full database of reference features takes an average of 3.3 hours, using some optimization and disk blocking. The high recognition rate we have seen comes at a high computational cost.

The rest of this chapter focuses on reducing the cost of computing these matches, first by reducing n_q using the *representative descriptor method* and then by reducing n_r using LSH. The voting results for $n_q = 300$ using exact nearest neighbor provides a baseline for performance, to which we will compare our results.

10.4 Reducing Running Time with Representative Descriptors

If we densely sample features from the reference scans (i.e., choose a large n_r), then we can sparsely sample basis points at which to calculate features from query scans. This is the case for a few reasons.

- Because the features are fuzzy, they are robust to small changes due to noise, clutter, and shift in the center point location. This makes it possible to match a feature from a reference object and a feature from a query scene even if they are centered at slightly different locations on the object or are oriented slightly differently. This also affects how densely we need to sample the reference object.
- Since regional descriptors describe a large part of the scene in fuzzy terms and a small part specifically, few are needed to describe a query scene well.
- Finally, these features can be very discriminative. Even with the data set we use below where we are distinguishing between several very similar objects, the features are descriptive enough that only a few are enough to tell apart very similar shapes.

We make use of these properties via the *representative descriptor* method. The method was originally introduced in [13] as *representative shape contexts* for the speeding search of 2D shape contexts, and were renamed in [6] to encompass the use of other descriptors such as spin images. Each of the few features calculated from the query scene is referred to as a *representative descriptor* or *RD*. What we refer to as the *representative descriptor method* really involves four aspects:

1. Using a reduced number of query points as centers for query features

2. A method for choosing which points to use as representative descriptors

3. A method for calculating a score between an RD and a reference object

4. A method for aggregating the scores for the RDs to give one score for the match between the query scene and the reference object

In our experiments, we try a range of values for the number of RDs and find that for simple matching tasks (e.g., low-noise queries), few are needed to achieve near-perfect performance. As the matching task becomes more difficult, the number required to get a good recognition rate increases.

We choose the basis points for the RDs uniformly at random from the 300 basis points from the query scans. This is probably the least sophisticated way to make the choice, and we do so to provide a baseline. Instead, we could use an interest operator such as those used with SIFT descriptors.

We take the score between one RD and a particular car model to be the smallest distance between the RD and a feature from one of the four reference scans for the model. To calculate the score between the query scene as a whole and the model, we sum the individual RD scores for that model. The model with the smallest summation is determined to be the best match. We have found this summation to be superior to the "voting" method where we take a maximum over the scores; the individual distances give a notion of the quality of the match, and summing makes use of that information, whereas taking a maximum discards it.

10.4.1 Experiment and Results

Experiment 2

Calculate n_q features from the query scan, which will be our RDs. Find the nearest neighbors to each of the RDs from each of the models, and calculate the scores. The model with the smallest score is the best match. Repeat for all queries and calculate the recognition rate as the percentage of query models that were correctly matched. Repeat the experiment several times with different randomly chosen sets of n_q features, and report the average recognition rate across these runs. Perform the experiment for different values of n_q.

The graphs in figure 10.5 show the results. Note that the number of comparisons increases linearly with the number of RDs. For example, if the voting method with 300 query features required n comparisons, then using thirty RDs requires $n \times \frac{30}{300}$ comparisons. With the 5 cm

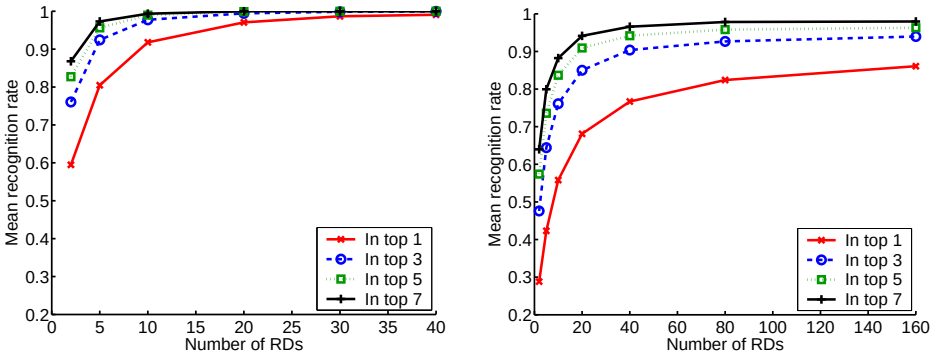


Figure 10.5 Results from experiment 2, shown as the number of RDs vs. mean recognition rate for the *(a)* 5 cm noise and *(b)* 10 cm noise queries. While our performance has dropped when considering only the top match, our recognition within a short list of matches is still very good, while we are performing a fraction of the feature comparisons. Note that the number of feature comparisons increases linearly with the number of RDs.

queries, we achieve 100% recognition with thirty descriptors if we consider the top seven matches. If we use forty RDs, we achieve 99.9% in the top two matches and 100% in the top three. The performance on the 10 cm noise query degrades quickly with fewer RDs. Because of the noise, fewer of the original 300 query points are useful in matching, so we randomly choose more RDs in the hopes that we will get more of the distinctive query features. With the 10 cm queries, we achieve 97.8% mean recognition in the top seven results using eighty RDs. The mean recognition within the top seven with 160 RDs is 98%.

When we consider only our top match, our performance has dropped significantly with both query sets. However, we are primarily interested in getting a *short list* of candidates, and for the 5 cm queries we can reduce the number of computations required by 87% to 90% (depending on the length of our short list) by using the RD method over voting. And for almost all choices of the forty RDs, we find the correct match in the top five returned. With the 10 cm set, we can reduce our computation by 47% to 73%. Also keep in mind that these are recognition rates averaged across 100 different random selections of the RDs; for many choices of the RDs we are achieving perfect recognition.

10.5 Reducing Search Space with a Locality-Sensitive Hash

When comparing a query feature to the reference features, we could save computation by computing distances only to the reference features that are nearby. Call this the "1, 2, 3, many" philosophy: the few close ones play a large role in the recognition; the rest of the features

have little meaning for the query. One way to achieve this is to use an algorithm for approximate k-NN search that returns a set of candidates that probably lie close to the query feature. The method we will look at is LSH, first introduced in [9].

We use a version of the simple LSH algorithm described in [7]. To create a hash, we first find the range of the data in each of the dimensions and sum them to get the total range. Then choose k values from that range. Each of those values now defines a cut in one of the dimensions, which can be visualized as a hyperplane parallel to that dimension's axis. These planes divide the feature space into hypercubes, and two features in the same hypercube hash to the same bucket in the table. We represent each hypercube by an array of integers, and refer to this array as the *first-level hash* or *locality-sensitive hash*. There are an exponential number of these hashes, so we use a standard second-level hash function on integer arrays to translate each to a single integer. This is the number of the bucket in the table, also called the *second-level hash* value. To decrease the probability that we will miss close neighbors, we create l tables, independently generating the k cuts in each. In most of our experiments in this section, we will use twenty tables. We will use the notation $b = h_i(\cdot)$ to refer to the hash function for the ith table which takes a feature vector and returns a second-level hash, or bucket, number. $T_i(b_i)$ will refer to the set of identifiers stored in bucket b_i in the ith table.

To populate the ith hash table, we calculate $b_i = h_i(f_j)$ for each feature f_j in the set of features calculated from the reference scans, and store the unique integer identifier j for the feature f_j in bucket b_i. Given a query feature q, we find matches in two stages. First, we retrieve the set of identifiers which are the union of the matches from the l tables: $\mathbb{F} = \bigcup_{i=1}^{l} T_i(h_i(q))$. Second, we retrieve from a database on disk the feature vectors for the identifiers, and calculate the distances $\text{dist}(q, f_j)$ for all features f_j where $j \in \mathbb{F}$.

The first part is the LSH query overhead, and in our experiments this takes 0.01 to 0.03 second to retrieve and sort all the identifiers from twenty tables. This is small compared to the time required in the second step, which ranges from an average of 1.12 to 2.96 seconds per query feature, depending upon the number of matches returned. Because the overhead for LSH is negligible compared to the time to do the feature comparisons, we will compare the "speed" of our queries across methods using the number of feature comparisons performed. This avoids anomalies common in timing numbers due to network congestion, disk speed, caching, and interference from other processes.

As we mentioned earlier, we are storing in the hash tables the azimuth rotations of the reference features instead of performing the rotations on the query features. If LSH returns only the features that are most

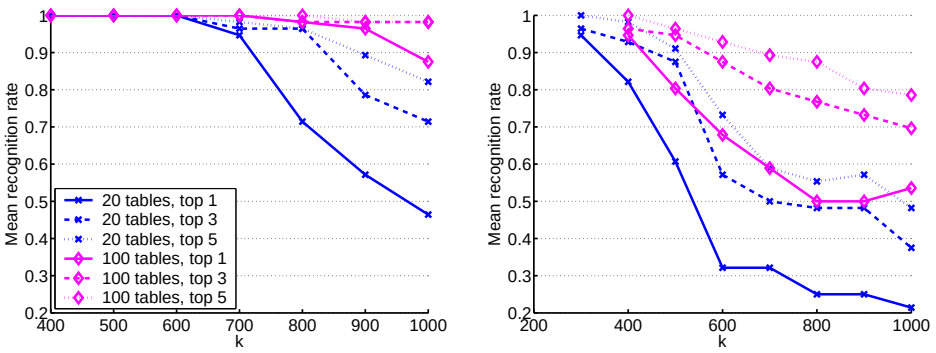


Figure 10.6 Results for experiment 3 using the voting method with 300 query features. The graph shows the recognition rate vs. the number of hash divisions (k) for 20 and 100 tables and for short lists of length one, three, and five (the legend applies to both graphs). The left and right graphs show results for the 5 cm and 10 cm noise queries, respectively. In general, as the number of hash divisions increases for a given number of tables, the performance degrades, and if the number of tables is increased, for a given value of k, performance increases. To see how the same factors affect the number of comparisons performed, see figure 10.7. To visualize the tradeoff between the number of comparisons and recognition rate, see section 10.8.

similar to a query q, it will effectively select for us the rotations to which we should compare, which saves us a linear search over rotations.

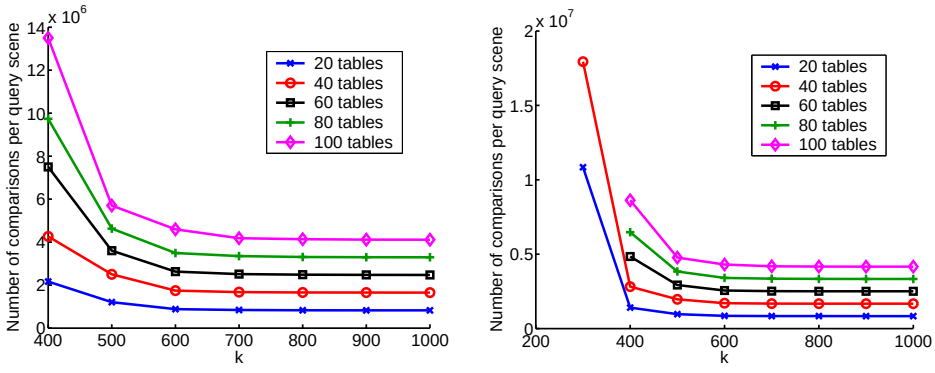
10.5.1 LSH with Voting Method

We first examine the performance of LSH using the voting method from subsection 10.2.3 to provide a comparison with the strong results achieved using exact nearest neighbor.

Experiment 3

Given the number of hash divisions k and the number of LSH tables l, perform LSH search with 300 features per query, and tabulate the best matches using the voting scheme, as we did in experiment 1. Perform for 5 cm and 10 cm noise queries.

We created 100 tables, and ran experiments using all 100 tables as well as subsets of 20, 40, 60, and 80 tables. In figure 10.6, we show how the recognition rate changes with variations in the number of hash divisions for the 5 cm and 10 cm queries. We show results for experiments with 20 and 100 tables, and show the recognition rate within the top choice, top three choices, and top five choices. In the 5 cm experiment, we maintain 100% recognition with up to 600 hash divisions when using twenty tables, and up to 800 hash divisions if we use 100 tables and consider a short list of length five. Notice that when using twenty tables, recognition degrades quickly as k increases, whereas recognition is better maintained when using 100 tables.



5 cm noise 10 cm noise

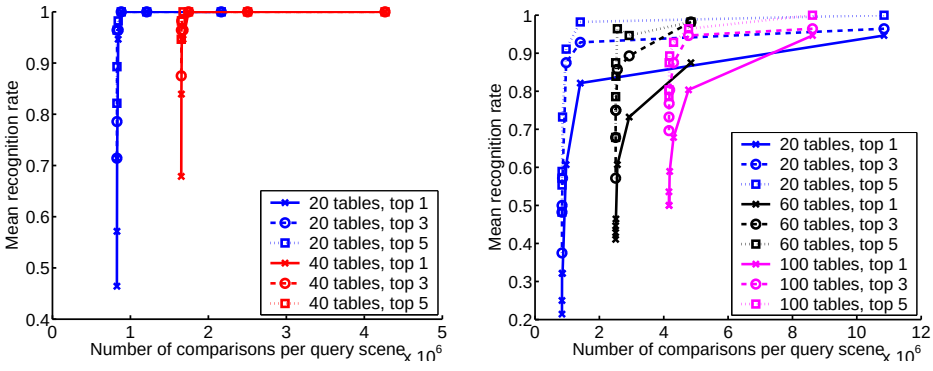
Figure 10.7 Results for experiment 3, showing the mean number of comparisons per query scene vs. the number of hash divisions (k), using 20, 40, 60, 80, or 100 tables. The left and right graphs show results for the 5 cm noise and 10 cm noise queries, respectively. The scale of the y-axis in the 10 cm graph is larger than in the 5 cm graph to accommodate the $k = 300$ results, though the number of comparisons required for each k and l combination is fewer with the 10 cm queries.

In the 10 cm experiments, we only achieve 100% recognition looking at the top five and using 300 or 400 hash divisions, with recognition declining quickly for larger values of k. Also notice that recognition falls with increasing k more quickly in the 10 cm experiments. As the queries become more difficult, it is less likely we will randomly generate a table with many divisions that performs well for many of our query features.

The recognition rate is only one measure of the performance. In figure 10.7, we show the mean number of comparisons per query scene vs. the number of hash divisions. Here we show results for 20, 40, 60, 80, and 100 tables. In both the 5 cm and 10 cm queries, we see a decline in the number of comparisons with increasing k, though the decline quickly becomes asymptotic. We also see a linear increase in the number of computations with a linear increase in the number of tables used.

For the 10 cm query, we tried using 300 hash divisions, but for more than forty tables, the queries were computationally too expensive. The range on the y-axis is larger in the 10 cm graph than in the 5 cm graph due to the jump at $k = 300$, but the number of computations performed for all other combinations of k and l are fewer in the 10 cm experiments. This seems to indicate that in general, the 10 cm query features lie farther away from the reference features in feature space than the 5 cm query features.

We see that as k decreases or the number of tables increases, the recognition improves, but the number of comparisons increases. To evaluate the trade off between speed and accuracy, we show in figure 10.8 the number of comparisons vs. the recognition rate, varying k along



5 cm noise 10 cm noise

Figure 10.8 Results for experiment 3, where we vary the value of k along each line to show the tradeoff between the number of comparisons performed and the mean recognition rate. The ideal point is in the upper-left corner where the number of comparisons is low and recognition is perfect. Exact nearest neighbor is off the graph in the upper-right corner, and would lie at $(3.0 \times 10^8, 1)$ if it were plotted.

each line. The ideal point would be in the upper-left corner, where the recognition rate is high and the number of comparisons is low. Exact nearest neighbor gives us a point at $(3.0 \times 10^8, 1)$, off the graph in the far upper-right corner. In the 5 cm graph, the leftmost point still at 100% recognition is from the experiment with 600 divisions and twenty tables. We can see that there is little to gain in increasing the number of divisions or the number of tables. The rightmost points in the 10 cm graph correspond to the experiments with 300 divisions, showing that the high recognition comes at a high computational cost. The points closest to the upper-left corner are from experiments using twenty tables and either 400 or 500 hash divisions. Unless we require perfect recognition for all queries, it makes little sense to use fewer than 400 divisions or more than twenty tables.

Lastly, while we are still achieving 100% mean recognition with $k = 600$ on the 5 cm queries using the voting method, the confusion matrix in figure 10.9 shows that we are not as confident about the matches relative to the confusion matrix for exact nearest neighbor (see figure 10.4). The RD method depends upon having several distinguishing query features, so if we combine LSH with RD, we expect a decrease in the number of comparisons but also a further degradation in recognition performance.

10.5.2 Using RDs with LSH

Experiment 4

Perform LSH search with varying numbers of RDs, values of k, and numbers of tables. Using the model labels returned with each feature,

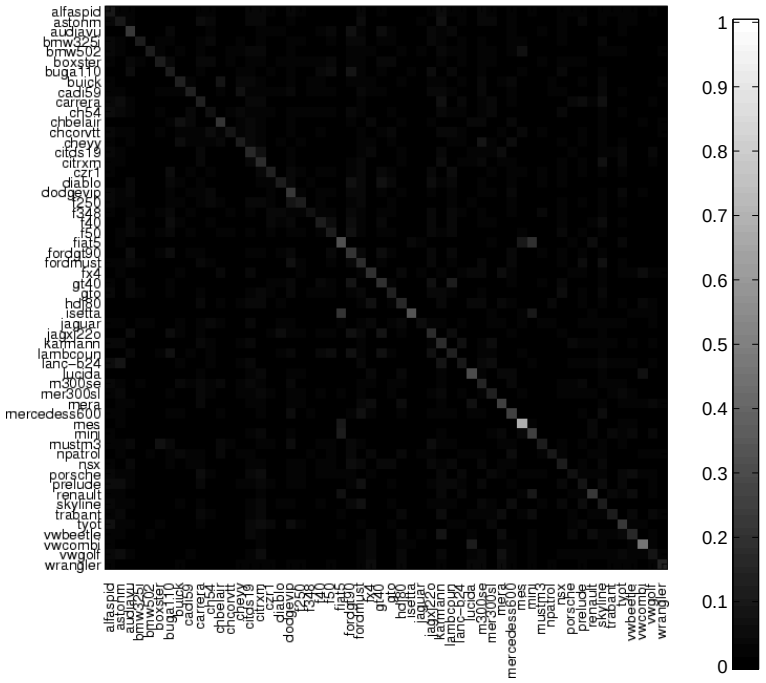
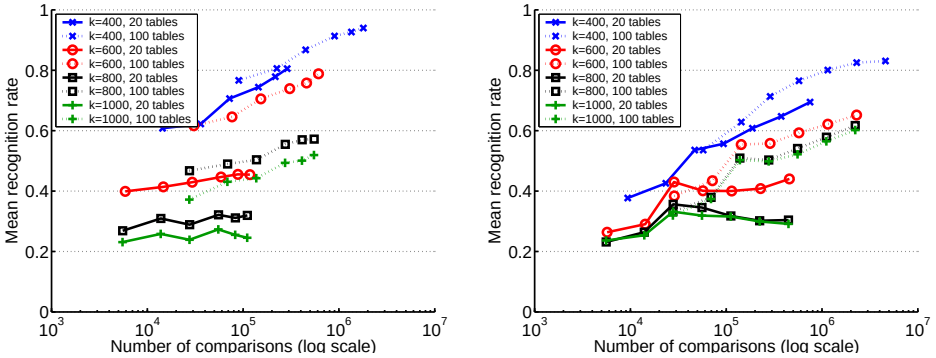


Figure 10.9 Confusion matrix showing the results for the 5 cm query for $k = 600$ from experiment 3. While we achieved 100% recognition with the top choice, comparing this matrix to the one from experiment 1 using exact nearest neighbor (see figure 10.4) shows that we are less certain of our choices.

tabulate scores as we did in experiment 2, with one exception: it is possible that LSH does not return any matches corresponding to a particular model, and in that case, we substitute for the RD model score a number larger than any of the distances as a penalty.

The tradeoff for experiment 4 between number of comparisons and mean recognition rate is shown in figure 10.10. Along each line we varied the number of RDs, and show results for combinations of 400, 600, 800, and 1000 divisions and 20 and 100 tables. For the 5 cm experiment, we show recognition in a short list of five, and show results within the top seven for the 10 cm experiment. The mean recognition in the 5 cm experiment using 400 divisions and twenty tables reaches 80%, which is much worse than before. With $k = 600$ and twenty tables, which demonstrated a good tradeoff when we using the voting method with 300 query features, only a 45% mean recognition rate is achieved. We do see however, that increasing the number of tables has helped us; using $k = 400$ with 100 tables yields a mean recognition rate of 94%. We see similar degradation with the 10 cm experiments, achieving only 83% mean recognition withing the top seven results using 400 divisions and 100 tables.

Recognition performance is decreased when using LSH because LSH misses many of the nearest neighbors to the query points, resulting in a heavy penalty. We can improve performance by being more “forgiv-



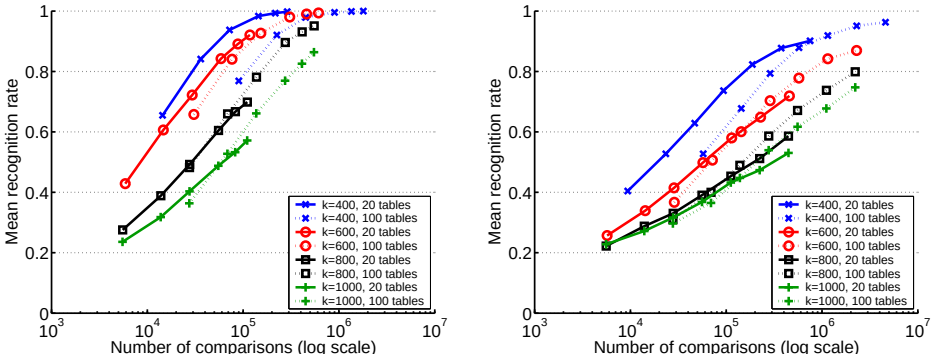
5 cm noise (top 5) 10 cm noise (top 7)

Figure 10.10 Results for experiment 4, showing number of comparisons vs. recognition rate with varying numbers of RDs along each line. We ran the experiment for different values of k and different numbers of tables. In the left graph we show the recognition within the top five results for the 5 cm queries, and in the right graph we show recognition within the top seven results for the 10 cm queries.

ing" and including in the RD sum only the closest x percent of the RD model matches, hopefully discarding the large values that arise because LSH unluckily misses good matches. If we are using twenty RDs and we are summing the top 50%, then for a given model, we would search for the model's closest reference features to each of the twenty RDs, and include in the sum only the ten of those which are closest.

Experiment 5

We perform LSH search with varying numbers of RDs, values of k, and numbers of tables. We tally the RD scores by including in the sum the distances from only the best 50% of the RD model matches.



5 cm noise (top five) 10 cm noise (top seven)

Figure 10.11 Results from experiment 5, where we use the RD method but sum only the top half of the RD scores. The graphs show the number of comparisons vs. the mean recognition rate, with the number of RDs varying along each line. In the left graph we show the recognition within the top 5 results for the 5 cm queries, and in the right graph we show recognition with the top 7 results for the 10 cm queries. Note the logarithmic scale along the x-axis.

The results for experiment 5 in figure 10.11 show that this method improved performance significantly within the top five results for 5 cm and top seven for 10 cm. In the 5 cm experiments, our sweet spot appears to be forty RDs, 400 divisions, and twenty tables with a mean recognition rate of 99.8% within the top five matches (and 95.3% with the top match; 99.4% within the top three, not shown). In the 10 cm experiments we reach 96% mean recognition with 160 RDs, 400 divisions, and 100 tables within the top seven matches (and 93.6% in the top five, not shown). We reach 90% mean recognition with 160 RDs, 400 divisions, and twenty tables within the top seven matches, which requires less than one-sixth the number of comparisons as with the same settings except with 100 tables.

The key to further improving performance lies primarily with getting better results from our approximate nearest-neighbor algorithm. In the next section, we examine the quality of the LSH results relative to exact nearest neighbor, and use this to motivate the need for algorithms that provide better nearest-neighbor performance.

10.6 Nearest-Neighbor Performance of LSH

In this section, we look at the performance of LSH as an approximate nearest-neighbor algorithm, independent of any recognition procedure. In most work on approximate nearest-neighbor algorithms, the performance is measured using the *effective distance error* or a similar measure [11, 7, 1], defined for the nth nearest neighbor as

$$E = \frac{1}{Q} \sum_{q \in Q} \left(\frac{d_{alg,n}}{d^*_n} - 1 \right), \tag{10.1}$$

where Q is the set of query features, d^*_n is the distance from the query q to the nth true nearest neighbor, and $d_{alg,n}$ is the distance from q to the nth best feature returned from the approximate algorithm. The effective distance error with increasing rank depth n is shown for the 5 cm and 10 cm queries in the first row of figure 10.12. Each line of the graphs represents one LSH query with a different number of hash divisions (k).

The effective distance error does not capture whether an approximate nearest-neighbor algorithm is returning the *correct* nearest neighbors, only how close it gets to them. In a recognition setting, the *identity* of the features returned is of primary interest, so we suggest a better measure would be *error by rank*. If we want the n nearest neighbors, the error by rank is the percentage of the true n nearest neighbors that were missing in the list of n returned by the approximate algorithm.

The graphs in the second row of figure 10.12 show the error by rank with increasing n for the 5 cm and 10 cm queries.

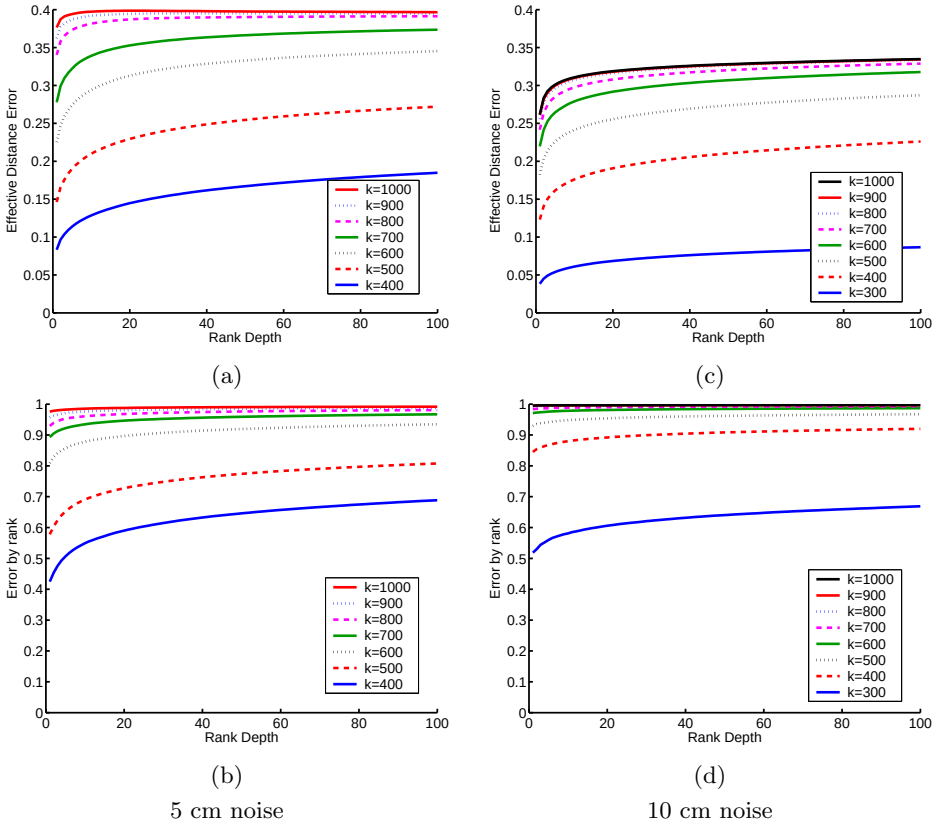


Figure 10.12 LSH performance, relative to exact nearest neighbor. The graphs in the first column show the performance on the 5 cm queries, using effective distance error in (a) and error by rank in (b). The second column shows results for the 10 cm query, with (c) showing effective distance error and (d) showing error by rank. All results are for twenty tables.

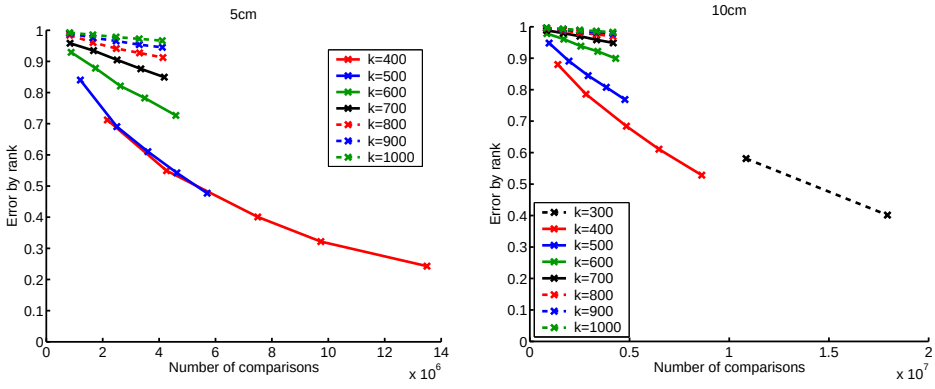
In the first column of figure 10.12 we see that for the 5 cm query, the effective distance error reaches a maximum at 40% for 800, 900, and 1000 hash divisions, but the same LSH results show almost 100% error by rank, meaning that almost never are any of the correct nearest neighbors returned. The second column of the figure shows results for the 10 cm queries. Notice that, relative to the 5 cm queries, the ceiling on the effective distance error is actually lower; the 900 and 1000 hash division LSH queries level off around 0.32, and all queries except LSH with 400 and 500 hash divisions are actually performing better by this measure than in the 5 cm query. However, we know from our recognition results that this should not be the case, that recognition results for the 10 cm queries were worse than the 5 cm queries for the same LSH settings. Indeed, we can see in the error-by-rank graph that the 10 cm

queries are performing much worse than the 5 cm queries for all LSH settings.

As an aside, the lines on these graphs are monotonically increasing, which does not have to be the case in general. If an approximate nearest-neighbor algorithm misses the first nearest neighbor, but then correctly finds every nearest neighbor of deeper rank, than the error by rank would decrease with increasing rank depth, from 100% to 50% to 33%, etc. It is also true that the effective distance error need not increase with increasing rank depth. It is a feature of LSH that we get fewer correct results as we look further from the query, which means that we cannot expect to increase our recognition performance by considering a greater number of nearest neighbors.

In figure 10.13, we show the tradeoff for different numbers of tables and hash divisions (l and k). Each line corresponds to a fixed number of divisions, and we vary the number of tables along the line, with the largest number of tables at the rightmost point on each line. As expected, with a greater number of tables we see better performance but we also perform more comparisons.

In general, recognition performance should increase as error by rank decreases, though to what degree will depend upon the recognition algorithm and data set. Next we introduce a variant of LSH which will find the same or more of the nearest neighbors as LSH, but at a computational cost between LSH and exact nearest neighbor.



5 cm noise 10 cm noise

Figure 10.13 Nearest-neighbor performance of LSH, shown as the tradeoff between the number of comparisons and the error-by-rank for the 5 cm and 10 cm query sets. The lower-right corner of the graph is the ideal result, where the number of comparisons and the error by rank are low. The number of tables used is varied from 20 to 100 along each line. With 400 divisions, we drive down the error by rank, but also dramatically increase the number of comparisons required.

10.7 Associative LSH

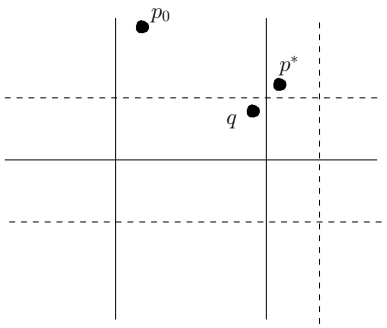


Figure 10.14 A 2D LSH example showing the space divided into bins by axis-parallel lines. The solid lines represent the divisions from one hash table, and the dashed lines represent divisions from another. Note that although p^* is the nearest neighbor to q, they do not occupy the same bin in either of the tables. It is the case, however, that p^* can be reached from q: q and p_0 are binmates in the solid-line table and p_0 and p^* are binmates in the dashed-line table.

In order to improve the error-by-rank and recognition performance, we introduce a variation which we will refer to as *associative LSH*. This algorithm begins with the results returned from LSH, and then uses the LSH tables to further explore the neighborhood around the query feature.

Consider the situation in figure 10.14 where we have a query q and the closest point to it, p^*, where for all tables i, $h_i(q) \neq h_i(p^*)$. It may be the case that there exists a point p_0 such that for two different tables i and j, $h_i(q) = h_i(p_0)$ and $h_j(p_0) = h_j(p^*)$. This suggests that we could use p_0 to find p^*.

First, a word about the data structures necessary. We will need the l LSH hash tables. To speed the algorithm we will also use precomputed l reverse hashes $b_i = R_i(j)$, which take an integer feature identifier and return the bucket in the ith table in which it is stored. Note that this is the reverse of the $T_i(b_i)$ function. Note that these reverse hashes are not necessary since we could retrieve the feature f_j from disk and calculate $h_i(f_i)$.

Results will be written to a structure $\mathcal{R}$ that for each match found so far stores the integer feature identifier j and the distance to the query, $\text{dist}(q, f_j)$, sorted by distance. This is the same structure we used for results when performing LSH queries. We will keep lists of the numbers of the buckets we have visited, one for each of the tables. Call the ith of these lists $\mathcal{B}_i$. We will also have a set of integer identifiers $\mathcal{A}$ which is initially empty.

The algorithm takes as input a rank depth r and a query feature q and outputs the results structure $\mathcal{R}$. Notice that the first three steps below are identical to the original LSH algorithm as described earlier, with the exception of the use of $\mathcal{B}_i$ for record-keeping.

1. For all i, calculate $b_i = h_i(q)$. Add b_i to $\mathcal{B}_i$ so that we do not visit the bucket b_i in the ith table again.
2. Calculate $\mathbb{F} = \bigcup_{i=1}^{l} T_i(b_i)$.
3. For all $j \in \mathbb{F}$, calculate $\text{dist}(q, f_j)$ and add to the results list R.
4. Find a feature identifier that is within the top r results in $\mathcal{R}$ and that is not in the set $\mathcal{A}$, call it a. If such a feature does not exist, then terminate.
5. Add a to the set $\mathcal{A}$. This, with the check above, ensures that we do not iterate using this feature again.
6. For all i, find $b_i = R_i(a)$, the bucket in which a is stored in the ith table.
7. For all i where $b_i \notin \mathcal{B}_i$ (i.e., we have not already looked in bucket b_i in table i), retrieve $\mathbb{F} = \bigcup_i T_i(b_i)$, the identifiers in the buckets in which a resides.
8. For all i, add b_i to $\mathcal{B}_i$.
9. For each identifier $j \in \mathbb{F}$ that is not already in $\mathcal{R}$, calculate $\text{dist}(q, f_j)$ and store the result in $\mathcal{R}$.
10. Go to step 4.

This algorithm requires only one parameter, r, that LSH does not require. In our experiments, we did not tune r, setting it only to two. Setting it higher would result in more comparisons and perhaps better results. The data structures for $R_i(\cdot)$ are l arrays, each with an element for each reference feature stored in the LSH tables. This roughly doubles the amount of memory required to hold the LSH tables, though it does not need to be stored on disk as it can quickly be generated when the tables are loaded from disk. Note that any variation on LSH that randomly generates the hash divisions can be used with this method as well.

The running time of the algorithm is dependent upon the number of associative iterations performed and the number of features retrieved on each iteration. The additional bookkeeping required for associative LSH over regular LSH adds a negligible amount of overhead. Step 4 requires a $O(r)$ search through the results list and comparison with the hashed set $\mathcal{A}$, but r will be set to a small constant (two in our experiments). Steps 8 and 9 require additional bookkeeping using the structure $\mathcal{B}_i$, but the complexity in both cases is $O(l)$ if we make $\mathcal{B}_i$ a hashed set.

In figure 10.15 we show the tradeoff between the number of comparisons performed and the error by rank for our associative LSH queries. We see a drop in the error by rank over regular LSH, especially when comparing results using the same number of hash divisions, but we see a corresponding increase in the number of comparisons.

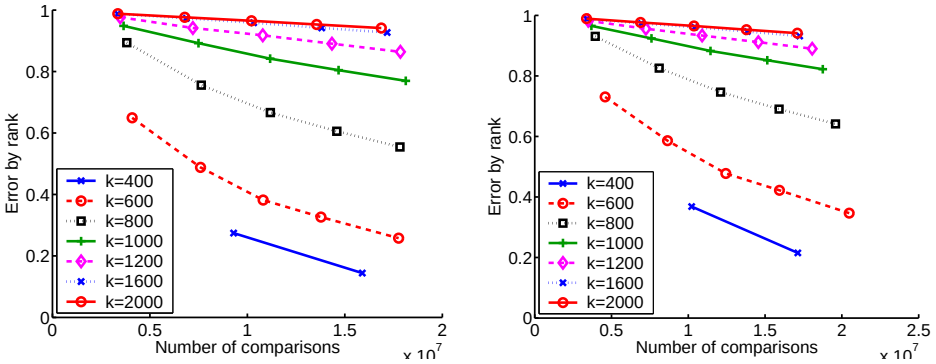


Figure 10.15 Nearest-neighbor performance of associative LSH, shown as the tradeoff between the number of comparisons and the error by rank for the 5 cm and 10 cm query sets. Compare these graphs to those in figure 10.13 showing nearest-neighbor performance of LSH. The number of tables used is varied from 20 to 100 along each line.

In figure 10.15 we show the tradeoff between comparisons and error by rank using associative LSH. Comparing to the results for LSH in figure 10.13 we see that we achieve a better tradeoff. For example, in the 5 cm experiments using 400 divisions, associative LSH achieves a slightly lower error and about the same number of comparisons using twenty tables as LSH does using eighty tables. Similarly, using 600 divisions, associative LSH achieves 65% error in 5×10^6 comparisons using twenty tables, whereas LSH reaches only 72% error in the same number of comparisons using 100 tables. From these results we can see that our search using associative LSH is more focused; we are finding a comparable number of nearest neighbors with associative LSH but with fewer comparisons. In the 10 cm experiments, this effect is more dramatic as associative LSH is able to achieve much lower error rates with a comparable number of comparisons.

Another important difference is that associative LSH is much less sensitive to the choices of k and the number of tables. With LSH, error changes dramatically with a change in the number of tables, and we see a quick degradation with an increase in the number of divisions.

10.8 Summary

In this chapter, we have performed an analysis of methods for performing object recognition on a particular data set, with a focus on the tradeoff between the speed of computation and the recognition performance of the methods. We made use of LSH for improving the speed of our queries, and demonstrated ways in which it could be made more robust.

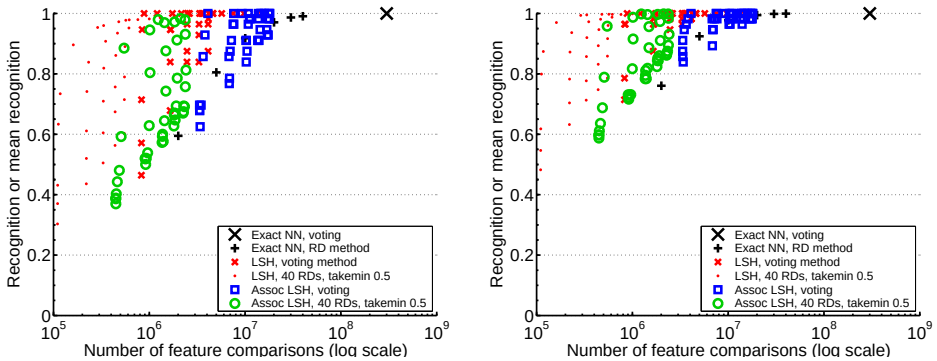
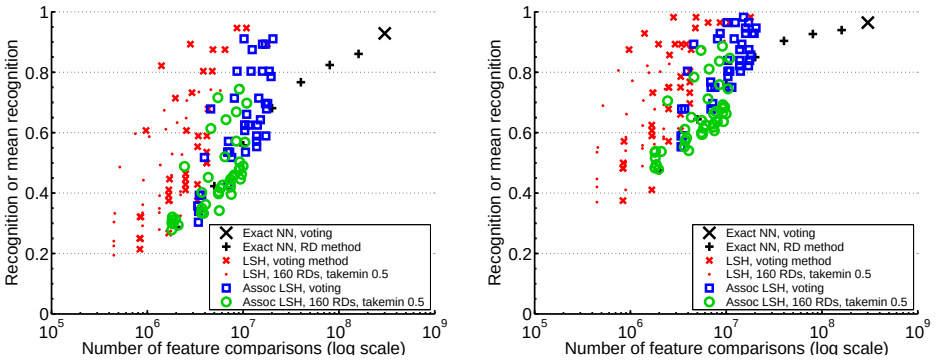


Figure 10.16 Summary of results for various methods on the 5 cm noise data set. For each method, we show points for all the variations of number of RDs, number of hash divisions, and number of tables discussed earlier in the chapter.

In figure 10.16 we display as a scatterplot results from the different methods discussed earlier in the chapter on the 5 cm query data set. For each method, we show points for all the variations of number of RDs, number of hash divisions, and number of tables. In general, results for associative LSH using voting lie between LSH and exact nearest neighbor using voting, with the same true for all three methods using RDs. Looking at the left graph showing results using the top choice, the best associative LSH results using RDs is close both in recognition and speed to LSH results using voting. If we can accept finding the match in the top three results returned, all the methods presented can get us to 100% recognition, with LSH with RDs achieving the lowest number of comparisons by a small margin over LSH with voting and associative LSH with RDs. We note again that the range of k using in the associative LSH experiments is much larger than in the LSH experiments, showing that we can achieve similar performance with less precise tuning of the parameters.

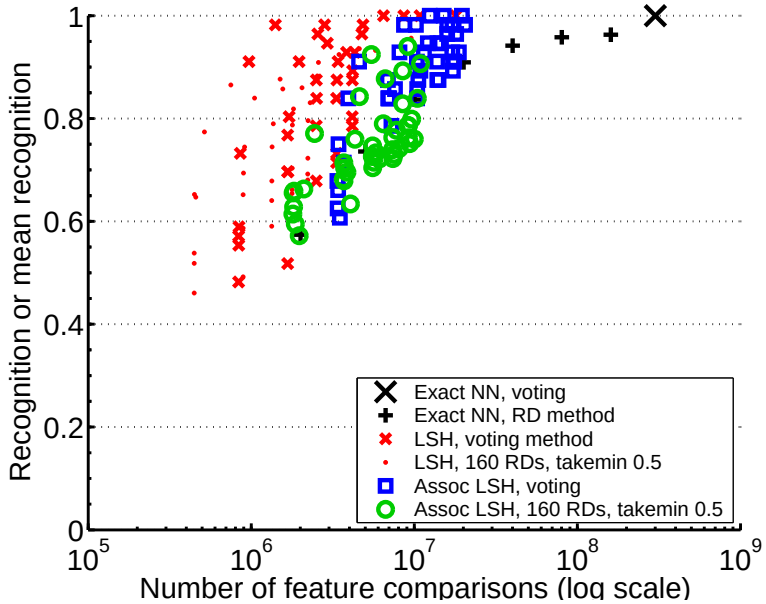
In figure 10.17 we give a scatterplot of the results for the various 10 cm noise experiments. Again we see that the associative LSH results lie between LSH and exact nearest neighbor, though as we see in the first plot, LSH using 300 divisions and the voting method shows a slightly higher recognition rate and lower comparisons than associative LSH. In general, however, associative LSH yields a higher recognition rate than LSH, though by performing more comparisons. We also note that when using the voting method, the results for associative LSH are more tightly packed than the LSH results, despite using a wider range of parameters for associative LSH in the experiments. This indicates that associative LSH can yield similar results on this data set with less tuning of the parameters.

In conclusion, we have found that LSH is an effective method for speeding nearest-neighbor search in a difficult object recognition task,



Top choice

Top three choices



Top five choices

Figure 10.17 Summary of results for various methods on the 10 cm noise data set, showing results for the top choice, top three, and top five choices. For each method, we show points for all the variations of number of RDs, number of hash divisions, and number of tables discussed earlier in the chapter.

but at the cost of some recognition performance. We have touched upon the connection between the reduction in recognition performance and the performance of LSH as a nearest-neighbor algorithm, and have presented a variation, associative LSH, which gives an improvement in nearest-neighbor performance on our data set. This increase in nearest-neighbor performance translates only roughly into recognition performance, showing small gains in recognition performance on this data set for an additional computational cost.

References

1. S. Arya, D.M. Mount, N.S. Netanyahu, R. Silverman, and A.Y. Wu. An optimal algorithm for approximate nearest neighbor searching fixed dimensions. *Journal of the ACM*, 45(6):891–923, November 1998.
2. S. Belongie, J. Malik, and J. Puzicha. Matching shapes. In *Eighth IEEE International Conference on Computer Vision*, volume 1, pages 454–461, July 2001.
3. S. Belongie, J. Malik, and Jan Puzicha. Shape matching and object recognition using shape contexts. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 24(4):509–522, April 2002.
4. A. Berg and J. Malik. Geometric blur for template matching. In *Proceedings of IEEE Conference on Computer Vision and Pattern Recognition*, pages 607–614, 2001.
5. A.C. Berg, T.L. Berg, and J. Malik. Shape matching and object recognition using low distortion correspondence. Technical Report UCB//CSD-04-1366, University of California Berkeley, Computer Science Division, Berkeley, December 2004.
6. A. Frome, D. Huber, R. Kolluri, T. Bülow, and J. Malik. Recognizing objects in range data using regional point descriptors. In *Proceedings of the European Conference on Computer Vision*, volume 3, pages 224–237, May 2004.
7. A. Gionis, P. Indyk, and R. Motwani. Similarity search in high dimensions via hashing. In *Proceedings of Twenty-Fifth International Conference on Very Large Data Bases (VLDB)*, pages 518–529, 1999.
8. D.P. Huttenlocher and S. Ullman. Recognizing solid objects by alignment with an image. *International Journal of Computer Vision*, 5(2):195–212, November 1990.
9. P.Indyk and R.Motwani. Approximate nearest neighbor—towards removing the curse of dimensionality. In *Proceedings of the Thirtieth Symposium on Theory of Computing*, pages 604–613, 1998.
10. A.E. Johnson and M. Hebert. Using spin images for efficient object recognition in cluttered 3D scenes. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 21(5):433–449, 1999.
11. T. Liu, A.W. Moore, A. Gray, and K.Yang. An investigation of practical approximate nearest neighbor algorithms. In *Advances in Neural Information Processing Systems*, December 2004.
12. D.Lowe. Object recognition from local scale-invariant features. In *Proceedings of the International Conference on Computer Vision*, pages 1000–1015, September 1999.
13. G. Mori, S. Belongie, and J. Malik. Shape contexts enable efficient retrieval of similar shapes. In *Proceedings of IEEE Conference on Computer Vision and Pattern Recognition*, volume 1, pages 723–730, 2001.
14. G. Mori and J. Malik. Recognizing objects in adversarial clutter: Breaking a visual captcha. In *Proceedings of IEEE Conference on Computer Vision and Pattern Recognition*, 2003.
15. C.Schmid and R.Mohr. Combining greyvalue invariants with local constraints for object recognition. In *Proceedings of IEEE Conference on Computer Vision and Pattern Recognition*, pages 872–877, June 1996.
16. P. Viola and M.J. Jones. Robust real-time face detection. *International Journal of Computer Vision*, 57(2):137–154, May 2004.