Nearest Neighbor algorithms in High Dimensions

Shannon Pahl and Diaa Fathalla

Overview

In this report, the nearest neighbor search problem is considered. It is fundamentally important in several areas of computer science, including machine learning, pattern recognition and data mining. The most natural category of application for this type of problem is similarity search. For instance, consider a huge database, containing either, text documents, audio files or image files. A nearest neighbor search on such a database could be to find all pairs of closest items. This appears for example in copyright violation detection and duplicate image detection. Many of these applications involve very large data sets (e.g. billions of web pages, or billions of pictures on the web, or hundreds of thousands of pictures on a personal computer). Moreover, the dimensionality of the items is very large. Further, many other problems can be reduced to a nearest neighbor problem, such as MST, matching and clustering. In this report, a brief overview of the problem is described together with some algorithmic approaches to solving the problem, and one area of application, namely, duplicate image detection using search by example.

Definition 1.1 (**Exact Nearest neighbor)**. *Given a set P of points in a d-dimensional space R^d, construct a data structure, and possibly performing some preprocessing to build an index, such that given any query point q, find efficiently the point p with the smallest distance to q.*

The problem is fully specified once the similarity criteria and space is defined. Similarity is specified by a distance measure and a search criteria. The distance measure is induced by an l_p norm, where d(p,q) = $||p-q||^p$ and $||x|| = (\sum_{i=1}^{d}|x_i|_s)^{1/p}$ and is a measure of similarity between objects. Another commonly used metric is on the Hamming metric space, H^d, the space of binary vectors of length *d*, and the Hamming distance measure, $d^h(p,q)$ is the number of bits on which *p* and *q* differ. The search criteria can be a fixed *k* for a *k*-closest search. Another option would be to find points closer than a threshold. Usually, these problems are solved in two steps. In the first step, feature vectors are extracted from objects. Typically, features of objects of interest are represented as points in R^d. For example, in the case of image databases, feature vectors can be defined by fixed sized thumbnails on which images are compared (where each pixel in a thumbnail represents a dimension), or by using more sophisticated feature vectors described later in this report. In the second step, a similarity is defined between the vectors and the similarity search will proceed according to this similarity, where the set of points have been previously indexed into a data structure to aid efficient searching.

A naïve algorithm is as follows: given a query point *q*, compute the distance from *q* to each point in *P*, and return the point with the minimum distance. This linear scan query can be done in *O(dn)* but is very inefficient for very large data sets. There are very efficient exact algorithms known for when the dimensionality, *d*, is low. For small *d*, an index can be built in *O(n)* and searched in *O(logn)* by trading off space for time. These algorithms don't scale well as *d* increases, as they have an exponential dependency on *d*. In particular, such algorithms have a query time of $O(d^{O(1)}logn)$ and space requirements of roughly $n^{O(d)}$, [3]. In [3], a method to speed up the brute force O(dn) search time by a factor of 50 is presented. Traditional nearest neighbor data structures are based on KD-trees, R-trees metric trees and Voronoi diagrams but have a query time which is exponential in *d*. Some reviews of data structures can be found in [1][2].

A kd-tree is constructed as follows: Given a set of *n* points in a *d*-dimensional space, the tree is constructed recursively by first finding a median of the values of the *i*th coordinates of the points (initially, *i* = 1). That is, a value *M* is computed, so that at least 50% of the points have their *i*th coordinate greater or equal to *M*, while at least 50% of the points have their *i*th coordinate smaller than or equal to *M*. The value of *x* is stored, and the set *P* is partitioned into *PL* and *PR*, where *PL* contains only the points with their *i*th coordinate smaller than or equal to *M*. The process is then repeated recursively on both *PL* and *PR*, with *i* replaced by *i* + 1. When the set of points at a node has size 1, the recursion stops. The data structure is a binary tree with *n* leaves, and depth O(log*n)* and the data structure can be constructed in time *O*(*n*log*n*).

For the problem of finding the nearest neighbor in *P* of a given query *q*, the search starts from the root of the tree, and is recursive. At any point in time, the algorithm maintains the distance *R* to the point closest to *q* encountered so far; initially, $R = \infty$. At a leaf node (containing, say, point *p'*) the algorithm checks if $||q - p'|| < R$. If so, *R* is set to $||q - p'||$, and *p'* is stored as the closest point candidate. In an internal node, the algorithm proceeds as follows. Let *M* be the median value stored at the node, computed with respect to the *i*th coordinates. The algorithm checks if the *i*th coordinate of *q* is smaller than or equal to *M*. If so, the algorithm recurses on the left node; otherwise it recurses on the right node. After returning from the recursion, the algorithm then checks whether a ball of radius *R* around *q* contains any point in R^d whose *i*th coordinate is on the opposite side of *M* with respect to *q*. If this is the case, the algorithm recurses on the yet-unexplored child of the current node. Otherwise, the recursive call is terminated. At the end, the algorithm reports the final closest-point candidate.

Other algorithms suffer from a linear query time for high enough dimensions [7]. This has lead researchers to consider approximate nearest neighbor search problems.

Definition 1.2 (**1+ε approximate near neighbor**). *Given a set P of points in a d-dimensional space* R^d*, construct a data structure, such that given any query point q, report any point within a distance at most (1+ε) times the distance from q to p, where p is the closets point in P to q.*

The goal with approximation algorithms is to find a balance between efficiency and accuracy and obtain sub-linear query, and polynomial space requirements. Certain approximate algorithms, however, are not defined for all distance measures. Approximate solutions are data structures that efficiently finds points that are possibly not the closest point to the query point but whose distance differs by at most (1+ε) from the minimal one. Locally sensitive hashing was one of the first indexing schemes to break this curse of dimensionality problem by using randomization techniques to find an approximate solution.

Locally Sensitive hashing

In this section, the results of the paper presented in [4] are reviewed. Locality-Sensitive Hashing (LSH) is defined as a family of hash functions $H = \{ h: U \rightarrow S\}$ where (r_1, r_2, P_1, P_2) – is called sensitive for distance d if for any $q, p \in U$ the following conditions hold:

- If $d(p, q) \le r_1$ then $Pr_H[h(q) = h(p)] \ge P_1$
- If $d(p, q) > r_2$ then $Pr_H[h(q) = h(p)] \le P_2$
- $P_1 > P_2$ and $r_1 < r_2$

To define a nearest neighbor search based on LSH, define the hash functions as follows. For an integer l which is the number of hash tables, choose l subsets $I_1,\ldots,I_l$ of $\{1,\ldots,d\}$ where d is the dimensionality of our space. Each set I_j consists of k elements from $\{1,\ldots,d\}$. The optimal value of k is chosen to maximize the probability that a point p "close" to q will fall into the same bucket as q, and also to minimize the probability that a point p “far away” from q will fall into the same bucket. For the choice of the values of l and k, see [4].

Let $p|I$ denote the projection of vector p on the coordinate set I, i.e., compute $p|I$ by selecting the coordinate positions as per I and then concatenate all these projected bits to create a vector. Essentially, $p|I$ represents a hash bucket for a hash function: denote the bucket of the hash table T_j to be defined as $g_j(p) = p|I_j$. Hence, for the preprocessing phase, store each $p \in P$ in the bucket $g_j(p)$, for $j = 1, \ldots, l$. To query a point q, search all indices $g_1(q), \ldots, g_2(q)$ and output *k* points p_i closest to q; in general, there may be fewer points returned if the number of points encountered is less than *k*.

The algorithm uses $O(dn+n^{1+p})$ space and $O(n^p)$ evaluations of the hash function for each query, where $p=1/(1 + \epsilon)$. The hash function can be evaluated using $O(d)$ operations. Thus, the query time is sub-linear, and the space complexity is sub-quadratic. The preprocessing and query algorithms are as follows:

<u>LSH Preprocessing</u>

- Input: A set of points P, l (number of hash tables),
- Output: Hash tables T_i, $i = 1,\ldots, l$
- Foreach $i = 1,\ldots, l$
    - Initialize hash table T_i by generating a random hash function $g_i(.)$
- Foreach $i = 1, \ldots, l$
    - Foreach $j = 1, \ldots, n$
        - Store point p_j on bucket $g_i(p_j)$ of hash table T_i

<u>LSH Query</u>

- Input: A query point q, K (number of approximate nearest neighbors)
- Access: To hash tables T_i, $i = 1, \ldots, l$ generated by the preprocessing algorithm
- Output: K (or less) approximate nearest neighbors
- $S = \emptyset$
- Foreach $i = 1, \ldots, l$
    - $S = S \cup \{$points found in $g_i(q)$ bucket of table $T_i\}$
- Return the K nearest neighbors of q found in set S

The LSH algorithm complexity estimates are better than exact methods, yet practical implementations appear to require care to avoid the high in memory requirements of the hash tables. Also, clients of the LSH data structure need to accommodate for false positives.

<u>Feature vector extraction</u>

In this section, the results of the paper presented in [5] are reviewed, and a discussion of some schemes for duplicate image detection is presented. A significant effort in defining a nearest neighbor problem is specifying how the objects map into the space R^n where nearest neighbor search techniques can be applied. One obvious approach for an image database is to create fixed size thumbnail of the pictures where each pixel in a thumbnail represents a dimension, and use a simple l_p metric to determine the distance between the thumbnails. Even for small thumbnails, this would require thousands of dimensions. Another approach is to create a histogram of the thumbnail where each bucket defines a dimension. The number of dimension using this approach is much lower. Other techniques include texture and wavelet representations. However, more robust schemes are needed to cater for common transforms, such as changing the scaling, contrast, cropping, partial occlusion and color changes in image duplicates. In this section, an approach is reviewed that is based on distinctive local descriptors which was first advocated in [6] and used in [5] where they used LSH for the approximate similarity search.

The interest point detection in [5] uses Lowe's Difference in Gaussian detector [6]. One of the key concepts in keypoint extraction is using Gaussian filters. The image is repeatedly convoluted with a Gaussian filter to produce a set of images. Adjacent Gaussian images are subtracted to produce a difference of Gaussian image. The process is then repeated after scaling the image down by a factor of 2. Then, the local minima and maxima for each difference of Gaussian image is computed to produce regions over keypoints. Further thresholding and computation is required to come up with the final feature vector for each keypoint in the image. One image can contain hundreds or even thousands of keypoints. In [8], some algorithms to help prune the keypoint set are discussed.

Images are therefore represented as a set of keypoints and the nearest neighbor database now contains keypoints as data values. A search is performed by querying the database for each keypoint in an image. In order to make this more efficient, the duplicate image detection query is submitted as a batch of keypoint values, and a file table and keypoint table is used to map back from the database query result onto the set of image file names. The batching is done by precomputing the hash bins that need to be visited for the batch and sorting them, and then sequentially visiting each bin. In this way, the disk access are sequential and reduces disk head movement. A similar approach is used in [8] but additional pruning is done to reduce the number of keypoint queries for one duplicate image query.

There are three structures. The first is a table describing the filenames of the images. So for instance, one column would be the file id, another would be the filename. Another table, the keypoint table, has columns for the keypoint id, the file id and the keypoint feature vector for the keypoint. A hashtable in the LHS implementation stores keypoint id, hashed by the keypoint feature vector. The algorithm first indexes all the images in the database using the following algorithm:

- For each image in the gallery
- Find the keypoints for the image using the DoG detector and compute the keypoint feature vectors for each keypoint in the image.
- Add an entry in the file table for the image
- For each computed keypoint in the image
    - Add an entry in the keypoint table using the file id for this image.
    - Add the keypoint id to each LSH hashtable by hashing on the keypoint feature vector

A query is computed using the following algorithm:

- Find the keypoints for the query image using the DoG detector and compute the keypoint feature vectors.
- For each keypoint feature vector, compute the LSH hashes, and sort the resulting bucked ids, and scan the hashtables sequentially to get the keypoint ids.
- Sort the returned keypoint ids and scan the keypoint table linearly, building up a histogram of hit counts of file ids.
- Possibly do some more thresholding of the resulting computed filename set
- Return the final set of image file names.

Results of experiments in [5] show very high accuracy. It is noted that the approach smoothes over false positives in that even though the approximate nearest neighbor search using LSH can sometimes miss a query, that because an image comprises hundreds of feature vectors, the accuracy is further improved as compared to just a single LSH query. Further, the query time was found to be two orders of magnitude faster than a exhaustive linear search. A dramatic speed up is observed because of the efficient batching of queries. An additional benefit of this approach is that matching can be done on sub images within the image, and with user specifies context, it can use used to effective provide multi feature vector indexing. However, the effectiveness of this approach has not yet been evaluated against huge database of billions of images.

Some remaining open problems in NN search

- Currently, no scheme takes advantage of the structure of the data set.
- It would be interesting to investigate hybrid data structures by merging tree based and hashing based approaches.
- Nearest neighbor in normed spaces and for other metrics
- Currently there are various lower bound conjectures for exact solutions related to the curse of dimensionality. There is no proof that there is no exact solution that can simultaneously have polynomial storage and polynomial search time.
- It is not known how LSH performs for huge dataset

References

[1] S. Berchtold and D.A. Kleim. High-dimensional index structures. In proceeding of SIGMOD, 1998, p. 501.

[2] H Samet. The design and analysis of special data structures. Addison-Wesley, Reading, M.A., 1989.

[3] Laura Heinrich-Litan, Exact $L_{infinity}$ nearest neighbor search in high dimensions. Phd thesis, Freie Universität Berlin, November 2002. See http://www.diss.fu-berlin.de/cgi-bin/zip.cgi/2003/80/Fub-diss200380.zip

[4] A. Gionis, P. Indyk, and R. Motwani. *Similarity Search in High Dimensions via Hashing*. In Proc 25th VLDB Conference, 1999. See http://theory.lcs.mit.edu/~indyk/vldb99.ps

[5] Ke, Y., Sukthankar, R., Huston, L., "Efficient Near-duplicate Detection and Sub-image Retrieval," IRP-TR-04-03, 2004. See www.cs.cmu.edu/~yke/retrieval/mm2004-retrieval.pdf

[6] D. Lowe. Distinctive image features from scale-invariant keypoints. *International Journal of Computer Vision*, 2004. See http://www.cs.ubc.ca/~lowe/papers/ijcv04.pdf

[7] Piotr Indyk *High-Dimensional Computational Geometry, University of Washington Colloquia series.* February 17, 2000. See http://norfolk.cs.washington.edu/htbin-post/unrestricted/colloq/details.cgi?id=2153

[8] Foo, J.J., Sinha, R. Prunning SIFT for scalable near duplicate image matching. See http://crpit.co/confpapers/crpitv63foo.dpf

[9] Y. Ke, R. Sukthankkar. PCA-SIFT: A more distinctive representation for local image descriptors. See http://www.cs.cmu.edu/~yke/pcasift