Cooper Union

Masters Thesis

Thesis Title

Author:
David Katz

Supervisor:
Dr. Carl Sable

A thesis submitted in fulfilment of the requirements
for the degree of Masters of Electrical Engineering

in the

Research Group Name
Department or School Name

March 2015

Declaration of Authorship

I, David Katz, declare that this thesis titled, 'Thesis Title' and the work presented in it are my own. I confirm that:

- This work was done wholly or mainly while in candidature for a research degree at this University.
- Where any part of this thesis has previously been submitted for a degree or any other qualification at this University or any other institution, this has been clearly stated.
- Where I have consulted the published work of others, this is always clearly attributed.
- Where I have quoted from the work of others, the source is always given. With the exception of such quotations, this thesis is entirely my own work.
- I have acknowledged all main sources of help.
- Where the thesis is based on work done by myself jointly with others, I have made clear exactly what was done by others and what I have contributed myself.

Signed: _____

Date: _____

"Thanks to my solid academic training, today I can write hundreds of words on virtually any topic without possessing a shred of information, which is how I got a good job in journalism."

Dave Barry

UNIVERSITY NAME (IN BLOCK CAPITALS)

Abstract

Faculty Name
Department or School Name

Masters of Electrical Engineering

Thesis Title

by David Katz

The Thesis Abstract is written here (and usually kept to just this page). The page is kept centered vertically so can expand into the blank space above the title too...

Acknowledgements

The acknowledgements and the people to thank go here, don't forget to include your project advisor...

Contents

List of Figures

List of Tables

Abbreviations

LAH **L**ist **A**bbreviations **H**ere

Physical Constants

Speed of Light $c = 2.997\ 924\ 58 \times 10^8\ \mathrm{ms}^{-\mathrm{S}}$ (exact)

Symbols

a	distance	m
P	power	W (Js^{-1})
ω	angular frequency	rads^{-1}

For/Dedicated to/To my...

Chapter 1

Background

1.1 Nearest Neighbors Search

1.1.1 Overview

One common way to represent a collection of objects is a vector space with a fixed number of dimensions. Each dimension in a vector space represents a different feature of the object. In doing so, the distance between two points in a vector space is analagous to the relatedness of two objects. Nearest neighbors search aims to find the closest points to a query point in a vector space. This type of search has a variety of applications in pattern recognition [1], information retrieval [2] and computer vision [3].

There are a variety of different types of vector spaces such as boolean valued, integer valued and mixed; however, we will focus on real-valued vector spaces. In these spaces, the value of every dimension in a vector can be expressed as a real number. Closeness can be defined by a variety of different distance metrics. Some common distance metrics between two N-dimensional points, x and y, are shown in Table 1.1. In low dimensional

Distance Type	Distance Function
Euclidean	$\sqrt{\sum_{i=1}^{N}(x_i - y_i)^2)}$
Manhattan	$\sum_{i=1}^{N} \lvert x_i - y_i \rvert$
Chebyshev	$\max \lvert x_i - y_i \rvert$

TABLE 1.1: Distance Metrics

spaces, the Euclidean distance is typically used [4], and will be the focus of future sections.

1.1.2 Basic Search Algorithm

The most basic algorithm for a nearest neighbor search performs a linear scan across every element in a set. Such a method computes the distance between a query point and every single point in a dataset, and returns the point with the minimum distance. For a dataset with N points of dimensionality D, the complexity of this operation is O(ND). For large datasets this linear time approach is not feasable, especially if many queries need to be performed.

1.1.3 K-nearest Neighbors

The basic linear search algorithm can easily be extended to support a query which returns the k-nearest neighbors rather than simply the closest. This change requires the use of a priority queue. A priority queue guarantees amortized O(log(N)) insert and delete-max operations and constant time check-max [5]. The priority metric for points will be their distance to the query point. The first K points searched are automatically added to the priority queue. For future points searched, if a point is closer than the furthest of the top K, the delete-max operation can be performed, and the new closer

point can be added to the priority queue ensuring that K elements always remain. If the most recently checked point is not one of the top K, then only the constant time check-max operation needs to be performed. If the point is closer than one of the top K, the delete-max and insert operations must be performed as well. Because both of those operations are logarithmic, the cost of updating the priority queue will at most be O(log(K)). In practice however the priority queue is updated very rarely. Assuming the points are searched in random order, the probability that a point being processed will be one of the current top K encountered is relatively low.

The k-nearest neighbors (k-NN) result is extremely useful for classification and regression on labeled datasets. For a classification task one common way to make a hard decision on a class is to use the class that appears most frequently in the top K. Thus, for datasets where test data is very similar to training data, this simple inference method can perform extremely well. k-NN can also be used for regression. Since the output is continuousin this case, the average of the results in the top K can be used.

The k-nearest neighbors has a few limitations, however [6]. For one, it is an instance based learning technique. This means that it will only perform well when instances are similar to those from training and thus does not generalize well [7]. Another issue is the computational cost of this method. While no training is required, each k-NN query requires linear time with respect to the size of the dataset. Approximate nearest neighbors described in 1.3 attempts to address this. Another issue is that common distance metrics, such as Euclidean distance, weight each dimension equally. Thus, in order to achieve reasonable results one must normalize all dimensions.

1.2 Fixed Radius Search

Another common search type is a fixed radius search [8]. This type of search attempts to find all points within a distance R of a query point. The linear search algorithm can easily be adapted to this type of query. After computing the distance between the query point and each point in the training set, if this distance is found to be less then R that point can be added to the result set.

1.3 Approximate Nearest Neighbors

Computation of the exact nearest neighbors via a linear search algorithm is extremely costly. One way to improve this performance is to create an index. The goal of an index is to increase the speed of a nearest neighbors query at the cost of additional preprocessing time and memory. In low dimensional spaces, one common index is the k-d tree described in more detail in 1.4. While the k-d tree supports average case O(log(N)) queries in low dimensional spaces, no index has beeen found which is guaranteed to return the exact set of neighbors in linear time [9].

However, for many applications it is not important that the result set be perfectly accurate. It may be advantageous to return a set which isn't guaranteed to be exact in significantly less time. For these reasons, approximate nearest neighbors are often computed instead of exact nearest neighbors. The most common index types for approximate nearest neighbor algorithms are constructed out of trees, hash tables, or graphs [10].

1.3.1 Tree Based Indexes

The main concept behind a tree based index is space paritioning. As such, these types of indexes tend to be extremely effective in low dimensionality settings, but do not scale as well to those of higher dimensionality. Generally, at the root of these indexes, the entire search space is present [11]. As the tree splits, space is partitioned and only points which satisfy a split criteria will be present on each subtree. Thus, when searching to the leaf of these trees, one can find the space partition a query point lies in, and by recursively backtracking, similar to the process described for k-d trees in section 1.4 can gradually expand the search radius.

One of the most widely tree based indexes used are k-d trees described in detail in section 1.4. The main advantage to k-d trees are that they are relatively fast to construct, can be easily modified, and have worst case linear space consumption [11].

K-means trees are another type of tree often used in practice [10]. These trees are constructed with a branching factor K. At each node, the k-means clustering algorithm is perfomed, separating remaining points into K clusters [12]. This branching continues until less than K points remain in a node, at which point the node becomes a leaf. To search a tree, one can move to a leaf by moving down the tree selecting the cluster with the closest mean to the query point. Each cluster's center is added to a priority queue, with its priority set as the distance from the query point. When a leaf is reached the algorithm continues the search at the closest center in the priority queue.

K-means trees are more expensive to construct than k-d trees, as the K-means algorithm is not guaranteed to converge quickly. Additionally, since all points are stored in the leafs and only cluster centers are stored at each node, the tree will be larger. K-means

trees however tend to be more effective than k-d trees when high precision is required in the result set.

Quad trees are another algorithm commonly used for nearest neighbor searches [13]. In a two dimensional space, each point in a quad tree splits space into 4 different quadrants, similar to how the origin separates a standard x and y axis into four regions. The tree will be expanded this way and as such has a higher branching factor than k-d trees leading to lower depth. Quad trees can also be expanded into octrees for 3-D space and generalized into similar higher orders for even higher dimensionality [14]. Unfortunitely, in higher dimensional spaces of dimensionality D, each point splits space into 2^D regions. This often leads to many unused pointers since points will not likely lie in all of these regions. As such the memory cost of quad tree variants can become extremely large.

1.3.2 Hash Indexes

Many variations of hash indexes exist; however, the most widely used is locality-sensitive hashing (LSH). The goal of LSH is to use a variety of different hash functions to map similar points into the same buckets. Rather than using cryptographic hash functions which aim map entities into a bucket independent of their state, the hash functions used in LSH aim to match similar points into the same bucket with a high probability, and dissimilar points into the same bucket with a low probability. Formally, each hash function maps a D-dimensional vector v into a R buckets [15]. Many different types of hash functions can be used such as projection, lattice, and quantization based. Different types of hash functions have been studied and evaluated extensively [16].

Thus, to initialize an LSH index, one must pass every point through H hash functions, and store a key to each point within every bucket it falls into. A larger H leads to

more information in the results however requires more processing time and memory consumption. Additionally, since all the information becomes compressed into these hash functions, these types of indexes generalize well to higher dimensional spaces.

To query an LSH index one must pass a query point through all H hash functions, and search each bucket for collisions. The entries that most commonly collide have a smaller hamming distance in the new hash space, and will thus be treated as the most similar points.

While LSH scales extremely well to high dimensional spaces, one disadvantage is that its memory consumption tends to be much larger than that of tree based indexes in low dimensional spaces [15]. Another key disadvantage is that quality of the search queries cannot be changed, as this is dependent on H and R. In other words, LSH indexes have their maximum accuracy constrained during their construction, wheras tree based indexes can have variable levels of accuracy on each query dependent on the number of nodes searched.

1.3.3 Graph Indexes

Graph based indexes tend to be the most expensive type to construct; however, they can support extremely fast queries. One common type is a k-nearest neighbor graph. In this type of graph, each node has exactly K edges, in which each node is connected to its k-nearest neighbors. A variety of different algorithms are available for constructing these types of graphs efficiently [17].

To perform a nearest neighbor query, a very common technique is a greedy traversal of the graph [18]. Given a query point, a randomly chosen node in the graph is chosen as the startpoint. Each neighbor is checked, and the next node traversed to is the one

which is closest to the query point. The algorithm is terminated after a fixed number of moves, where a higher number of moves will have improved results. The K best nodes encountered are returned as the k-nearest neighbors. Often times random resets are incooperated to ensure that different parts of the graph are searched. Another common heuristic is to only search a subset of the the vertices at each node.

From experimental results these indexes tend to perform better than LSH and k-d trees [18]. However, one downside is that the offline construction of the graph is very expensive. Additionally, there is a large amount of randomness in the search algorithm, so there tends to be a large variance in the quality of the results obtained from queries with the same point.

1.4 k-d Trees

1.4.1 Overview

The k-d tree was originally developed as "a data structure for storage of information to be retrieved by associative searches" [19]. k-d trees are efficient both in the speed of associative searches and in their storage requirements. A k-d tree is a binary tree which stores points in a d-dimensiona; space. Each node contains a single d-dimensional point, a split dimension, and up to two children nodes. Each node represents a hyperplane which lies perpendicular to the split dimension and passes through the stored point. The left subtree of a node contains all points which lie to the left of the hyperplane, while the right subtree represents all points which lie to the right of the hyperplane. Thus, each node partitions all nodes below it into two half-spaces. Because only a single split dimension is used at each internal node, each splitting hyperplane is axis-aligned.

This splitting procedure continues on each subtree until each subtree contains only one element. The procedure for axis selection on each split is described in section 1.4.2.

1.4.2 Construction

```
function KDTREE(pointList)
    splitDim = selectAxis()

    medianPoint = selectMedian(pointList, splitDim)
    leftList = select points ≤ medianPoint along splitDim
    rightList = select points > medianPoint along splitDim

    treenode node = new treenode()
    node.splitDim = splitDim
    node.splitPoint = medianPoint
    node.leftChild = kdtree(leftList)
    node.rightChild = kdtree(rightList)

    return node
end function
```

Algorithm 1: Construct k-d tree

The construction of a k-d tree is performed recursively with input parameters of a list of points. Pseudo code is shown in figure 1. Axis selection can be performed in multiple ways. The classical approach is to deterministically iterate between all N dimensions. Another approach, known as spatial median splitting, selects the the longest dimension present in the current pointList to split on [20]. The downside of this method is that a linear traversal is required to select the split dimension. Another popular approach is to randomly select the split dimension with an equal probability of selecting each dimension. This approach is often applied when using multiple k-d trees; because of the additional randomness, trees are likely to be different [10].

While a linear time algorithm for determining the median of an unordered set is possible [21], a heuristic approach is typically used to approximate the median. A common

heuristic is to take the median of five randomly chosen elements; However many other methods can be used such as the triplet adjust method [22].

At the termination of the algorithm, the root of the k-d tree is returned, and each node contains exactly one point. The runtime of this algorithm is O(Nlog(N)) where N is the number of points in pointList. While the median can be approximated in constant time, partitioning pointList along that median is an O(N) operation. Since the k-d tree is a binary tree in which each node holds one point, assuming it is relatively balanced, its height is O(log(N)).

1.4.3 Nearest Neighbor Query

A simple algorithm exists to apply the k-d tree to a nearest neighbor query. This algorithm is guaranteed to find the single closest point to the search query. Pseudocode for this algorithm is shown in Figure 2.

```
function SEARCHKDTREE(kdTreeNode, searchPoint, currBest)
    If(leaf) kdTreeNode.splitPoint vs currBest; return;
    dim = kdTreeNode.splitDim
    searchDir = searchPoint[dim] < kdTreeNode.splitPoint[dim]
    searchFirst = searchDir ? kdTreeNode.left : kdTreeNode.right
    searchSecond = searchDir ? kdTreeNode.right : kdTreeNode.left
    searchkdtree(searchFirst, searchPoint, currBest)
    if distance(kdTreeNode.splitPoint, searchPoint) < distance(currBest,
searchPoint) then
|       currBest = kdTreeNode.SplitPoint
    end
    if HyperPlaneCheck(searchPoint, currBest, searchSecond) then
|       searchkdtree(searchSecond, searchPoint, currBest)
    end
end function
```

Algorithm 2: Nearest Neighbor Search k-d tree

The first part of the algorithm recursively steps down the tree until a leaf is reached. At each node, a comparison on the split dimension is performed to determine which side of the splitting hyperplane the search point lies so that the search can continue in the

proper half space. When a leaf is reached, the point stored in the leafnode is set as the current closest point. The algorithm then recursively walks back up the tree, and at each node computes the difference between the current node's point and the searchpoint. If this distance is smaller than that of the current best, the current node point becomes the current best.

The algorithm then determines whether a closer point than the current best could potentially exist in the second unsearched subtree. Because all hyperplanes are axis aligned, this computation is very simple. The closest possible point in the halfspace represented by the second subtree could potentially lie within a distance of ϵ from the hyperplane, where ϵ is very small. The lower bound of the distance of the closest point, occuring when ϵ approaches zero, is the absolute value of the difference between the search point and split point along the current split dimension. If this distance is larger than the current best point's distance, then the algorithm does not need to check the second subtree, as there is no possible closer point in that halfspace. If this distance is smaller however, then the algorithm will search down the second subtree following the exact same procedure as before, treating the second child as the root. Because of this comparison however, the worst case run time of this algorithm is O(N), as if all comparisons fail, then the entirity of the tree will be searched. As the dimensionality of the tree becomes larger, this check is more likely to fail, and k-d trees diminish in effectiveness.

This algorithm can be extended to support KNN with the same priority queue procedure described in section 1.1.3. The check as to whether a closer point could potentially exist in the second subtree will then be performed against the furthest point in the top K. This algorithm can also be extended to perform a radius bounded search. Rather than checking the distance to the hyperplane compared to the furthest point in the top K

one can check against a fixed radius R. Thus, this algorithm can also efficiently find all points within R of the query point.

1.4.4 Approximate Nearest Neighbors Query

The k-d tree nearest neighbor search algorithm can be extended into an approximate nearest neighbors search. To do so, a limit on the number of points to search must be applied. The algorithm will follow the exact same steps as shown in Figure 2; however, when the search limit is reached the algorithm terminates. This means that not every possible node that might contain a closer point would necessarily be searched. However, the nodes that do get searched are the more likely to contain the close points. In other words the algorithm will examine the regions of space which are closest to the search query first before expanding outward.

1.4.5 Modification

One key advantage of k-d trees is that they are very easy to modify. Inserting a node requires a traversal to a leaf node following the same procedure as described in the first part of Figure 2. This search takes approximate O(log(N)) time if the tree is balanced. Once a leaf node is reached, a single comparison along the split dimension needs to be performed in order determine whether the new node should be added as the left or right child. If random points are inserted the tree will remain relatively balanced as there will be an equal probability of being placed on each leaf. However heuristics can be used to help ensure that a tree remains balanced in a dyanmic environment [23].

Deletion on k-d trees can also be performed with relative ease. Again a downward traversal is performed until the target node is encountered. If the target node is a leaf,

it can be removed trivially by removing the connection from its parent. If the target node is not a leaf, lazy deletion can be performed. The node will not be removed and will still serve to partition space; however it will not store any actual split point against which comparisons can be made. If many nodes are lazy deleted, performance will slowly degrade, and reconstruction of the tree may be advisable.

1.5 Randomized k-d Tree Forests

With one k-d tree, one runs the risk of potentially very bad queries occuring when one of the earlier splitting hyperplanes lies close to the point of interest. By having a forest of multiple k-d trees made from randomized split dimensions and splitpoints, the effects of this can be minimized. [24] has shown that k-d tree forests can lead to increased performance. It is important to note that when searching multiple trees, the same heap must be used, and as such this object needs to properly be mutex locked if the search is to occur in parallel. In doing so, as better results are found in each tree, more hyperplane checks will pass and less time will be wasted searching parts of the tree where better results do not lie. When searching trees sequentially, rather than in parallel the performance was significantly weaker [24].

Another approach to improve performance is the rotation of dimensions along different trees [24]. This allows the splitting hyperplanes to no longer be constrained to axis alignment, and can introduce a greater amount of variety in the forest. From experimental results the optimal amount of trees was generally less than 20, though the number varied greatly on different datasets [9].

Chapter 2

Related Work

2.1 Approximate Nearest Neighbor Frameworks

Many ANN frameworks exist for fast computation. Since no exact nearest neighbors algorithms are faster than linear time in high dimensional spaces, a variety of different ANN algorithms must be applied. The Fast Library for Approximate Nearest Neighbors (FLANN) makes use of many of these algorithms including k-d trees, k-means trees, and locality sensitive hashing [9]. However, each of these types of indexes have different properties, and some may be suited to some datasets more than others. Thus, one difficult task this framework can perform is automatic selection of the index type and parameters. This is done by estimating the overall cost of an index in terms of memory consumption, index generation time, and query speed to achieve a given accuracy. The user of the system can also put weights on each of these costs to raise the relative importance of one ore more of these factors. By benchmarking on a small subset of a dataset, the framework can effectively select the most efficient index type and apply the simplex optimization algorithm [25] to optimize its parameters.

Also of importance with FLANN and other frameworks is that they are optimized for real world performance. This means that that benchmarks are taken in terms of real time, and memory consumption is minimized. For example, when possible bit array keys are used rather than pointers in order to save as much memory as possible.

Chapter 3

System Description

3.1 Overview

As mentioned in subsection 1.1.3, before applying a nearest neighbors algorithm, one must normalize dimensions proportional to their relevance. Conventionally, if the relevance of a dimension or a set of dimensions were to be changed, one must perform a linear transformation on the every single point in the search space. When using an index type described in section 1.3, this linear transformation requires a total reconstruction of the index for optimal nearest neighbors search performace, as the distance between all pairs of points in the index is now different.

The goal of this system is to support queries of dynamic dimension relevance in low dimensional spaces. Dynamic dimension relevance means that requester of a given query must provide both a search point, and the relevance of each dimension in the query. The system will then compute the ANNs using a modified Euclidean distance metric in which the distance in each dimension is weighted proportionately to the relevance. This metric is described formally in section 3.1.2.

3.1.1 Normalization

Each dimension in the vector space representation of a dataset must be normalized to ensure that each dimension is weighted equally. The normalization scheme used performs a linear transform to set the min of a dimension to zero and the max to one. After finding the min and max of each dimension, one must normalize every dimension of every point following equation 3.1.

$$d_{normalized} = \frac{d - d_{min}}{d_{max} - d_{min}} \tag{3.1}$$

This normalization technique is ideal for datasets which follow a relatively uniform distribution. However, the presence of outliers could greatly skew this normalization scheme. An alternative normalization strategy for these cases is to normalize the mean of each dimension to zero, and the variance to one. This can be achieved by following the linear transformation outlined in equation 3.2. This procedure is equivalent to finding the standard score or z-score of each dimension [26].

$$d_{normalized} = \frac{d - d_{mean}}{d_{stdev}} \tag{3.2}$$

3.1.2 Dimension Relevance

After normalization described in section 3.1.1, each dimension in the dataset is said to have equal relevance, and would have an equivalent contributribution to a standard Euclidean distance. As described in section 3.1 each query requires both a search point, and a dimension relevance vector.

The dimension relevance vector (DRV) must contain the same number of dimensions as all points in a datset. Each element in the DRV contains a weight on each dimension. The DRV is then normalized to have a sum of one. Using the DRV v and two D dimensional points x and y, the modified Euclidean distance metric is shown in equation 3.3.

$$distance = \sqrt{\sum_{i=1}^{D}((x_i - y_i) \times v_i \times D)^2)} \tag{3.3}$$

For the case in which all dimensions are weighted equally, each element in $v_i = 1/D$. Thus, in this special case, the standard Euclidean distance is computed. This distance metric is also equivalent to transforming each dimension via multiplying by $v_i \times D$, and computing the standard Euclidean distance. By using this metric instead, this transformation does not need to be explicitly performed but is inherent in the distance calculation.

3.1.3 Motivation for k-d Trees

why other indexes suck.

3.1.4 Split Probability Matching

A common heuristic for determining the split dimension on k-d tree indexes is spatial median splitting CITETEETE. The motivation for this technique is that each hyperrectangle should be as close to a cuboid as possible. In doing so, the check of whether or not a ball of radius R around a point intersects with an enclosing hyperplane is more likely to result in no intersection. When no intersection occurs the k-d tree search algorithm

described in section 1.4.3, can eliminate a subtree without searching it. It is important to note that k-d trees can still function, even if the partitions are not square. However, because of the inability to prune subtrees as effectively, ANN searches will not be as efficient and the quality of the result set will suffer. It is also important to note that on a vector space in which each dimension has been normalized, selecting a random dimension to split on with equal probability accomplishes a similar effect to spatial median splitting. On average, regions will tend to be close to cuboids, and the advantage of this method is less offline computation in index construction and more variety in trees, which is advantageous when searching multiple trees.

When applying dimension weights via a DRV to a k-d tree search, the distance metric acts on a transformed space. As such, if the split dimension was selected with equal probability the regions would no longer be cuboids on average. In order to combat this, I introduce the heuristic of split probability matching (SPM). The goal of this heuristic is to adjust the probability of splitting on each dimension to account for the fact that in the transformed space the dimensions are no longer normalized to the same weight. Therefore, by selecting split dimensions with probabilities equal to the weight of each dimension, the regions will tend to approach a cuboid shape in the transformed space.

Figure blablidy blah shows blabidy blah this shit is better

3.1.5 Tree Quality Metric

While matching the split dimension probabilities of a k-d tree a DRV results in greatly improved ANN performance, this method is not directly feasible in practice on a system which supports specifying a DRV at query time rather than during tree construction. As shown in section 1.4.2 the cost of generating a k-d tree is NLog(N), while the cost of

a standard linear query is N. Thus, generating new trees to directly match the DRV of each query is not practical, as the tree construction cost is very high.

To work around this, we opted to construct a large set of k-d trees with a variety of different split probabilities. On each query, our index can then select the tree or subset of trees whose split probabilities best match the queries's DRV. In doing so, the set of trees search will be close to the optimal split, leading to high performance for ANN queries.

The quality metric used is the modified Euclidean distance metric from equation 3.3. Rather than comparing two points however, the two entities compared are the DRV and the split probabilities of the tree, both of which have been normalized to have a sum of one. The set of trees with the highest quality can then be searched in parallel. Additionally, based on the result of this tree quality metric, the parallel search between multiple trees can be skewed towards the trees of highest quality.

There is of course a tradeoff between the size of the set of k-d trees, and system performance. More trees will result in high quality matches for a more DRVs, and thus improved accuracy. However, each additional tree is linear in memory consumption, so the number of these trees must be kept within reason. Additionally, each test of the tree quality metric is computationally equivalent to a comparison between two points. As such, there is a cost associated with computing the quality metric of each tree which must be accounted for in performance benchmarks.

3.2 Detailed Implementation Overview

3.2.1 Index Construction

The initial dataset input into the system is an unordered set of N points containing D dimensions. The first step is to normalize all points in this set along each dimension using the scheme specified in equation 3.1 or 3.2, per selection of the user.

The user must also specify two index size parameters: the deterministic dimension depth (DDD), and the number of random trees. Both of these parameters impact the amount of trees which will be generated. The use of the DDD allows the system to perform well on DRVs which put very heavy weight on a low number of dimensions, which is likely to be common in practice. To do so, a tree is generated with every possible subset of dimensions containing less than or equal to the DDD. In each of these trees the split weights are equal in all of the selected dimensions, and as such are equivalent to $1/D$. The number of trees needed to satisfy a DDD of R on a dataset with dimensionality D is shown in equation 3.4. This number scales factorially with both D and R, so a very small R must be used if D is large to ensure that the number of trees is reasonable. A large R however has the advantage of high quality tree matches on queries with R or less dimensions.

$$ntrees = \sum_{i=1}^{R} \binom{D}{i} \tag{3.4}$$

The number of random trees directly controls the number of trees generated with uniform random split probabilities. Each set of split of tree split probabilities is generated by selecting a number from U(0,1) on each dimension, and normalizing the results to a sum of one. A larger number of random trees will result in overall better matches on

randomly selected DRVs at the cost of memory consumption. Other types of dimension split priors other than uniform could be used instead if additional information was known about the distribution of DRVs used.

The total number of trees generated is the number of deterministic trees as specified by equation 3.4 and the number of random trees. Each tree is generated following the algorithm in section 1.4.2 with the modification that the split dimension is selected weighted from the specified tree split probabilities rather than from a uniform distribution. In our test environment, if the number of trees exceeds that which can be held in memory, they will be written to disk. In practice, this would not be an acceptable solution as the cost of reading a tree from memory is linear, and disk reads are significantly slower than a linear seek performed in memory [27]. The evaluation metric however considers only the quality of results against the number of nodes search, so this solution is acceptable for testing purposes. Details about implementation in a live, distributed system will be discussed in (REF FUTURE WORK SECTION).

The last step of construction after each tree is generated is to generate an index on the tree split probabilities. Without an index, in order to determine the best tree(s) on each query a linear seek across each would need to be performed comparing each to the DRV. This adds an overhead of one comparison operation for each tree on every single ANN query. To avoid this the set of split probabilities are cast into a vector space. After doing so, a k-d tree index is constructed on the split probabilities following the standard procedure in section 1.4.2. At query time, this index can be applied to heuristically select the best set of trees via an ANN search against the DRV in sublinear time.

3.2.2 ANN Query

Appendix A

Appendix Title Here

Write your Appendix content here.

Bibliography

[1] Thomas Cover and Peter Hart. Nearest neighbor pattern classification. *Information Theory, IEEE Transactions on*, 13(1):21–27, 1967.

[2] Christopher D Manning, Prabhakar Raghavan, and Hinrich Schütze. *Introduction to information retrieval*, volume 1. Cambridge university press Cambridge, 2008.

[3] Oren Boiman, Eli Shechtman, and Michal Irani. In defense of nearest-neighbor based image classification. In *Computer Vision and Pattern Recognition, 2008. CVPR 2008. IEEE Conference on*, pages 1–8. IEEE, 2008.

[4] Per-Erik Danielsson. Euclidean distance mapping. *Computer Graphics and image processing*, 14(3):227–248, 1980.

[5] Peter van Emde Boas, Robert Kaas, and Erik Zijlstra. Design and implementation of an efficient priority queue. *Mathematical Systems Theory*, 10(1):99–127, 1976.

[6] Kevin Beyer, Jonathan Goldstein, Raghu Ramakrishnan, and Uri Shaft. When is nearest neighbor meaningful? In *Database TheoryICDT99*, pages 217–235. Springer, 1999.

[7] David W Aha, Dennis Kibler, and Marc K Albert. Instance-based learning algorithms. *Machine learning*, 6(1):37–66, 1991.

[8] Matthew T Dickerson and R Scot Drysdale. Fixed-radius near neighbors search algorithms for points and segments. *Information Processing Letters*, 35(5):269–273, 1990.

[9] Marius Muja and David G. Lowe. Fast approximate nearest neighbors with automatic algorithm configuration. In *International Conference on Computer Vision Theory and Application VISSAPP'09)*, pages 331–340. INSTICC Press, 2009.

[10] Marius Muja and David G. Lowe. Scalable nearest neighbor algorithms for high dimensional data. *Pattern Analysis and Machine Intelligence, IEEE Transactions on*, 36, 2014.

[11] Peter N Yianilos. Data structures and algorithms for nearest neighbor search in general metric spaces. In *SODA*, volume 93, pages 311–321, 1993.

[12] John A Hartigan and Manchek A Wong. Algorithm as 136: A k-means clustering algorithm. *Applied statistics*, pages 100–108, 1979.

[13] Raphael A. Finkel and Jon Louis Bentley. Quad trees a data structure for retrieval on composite keys. *Acta informatica*, 4(1):1–9, 1974.

[14] Hanan Samet. An overview of quadtrees, octrees, and related hierarchical data structures. In *Theoretical Foundations of Computer Graphics and CAD*, pages 51–68. Springer, 1988.

[15] Mayur Datar, Nicole Immorlica, Piotr Indyk, and Vahab S Mirrokni. Locality-sensitive hashing scheme based on p-stable distributions. In *Proceedings of the twentieth annual symposium on Computational geometry*, pages 253–262. ACM, 2004.

[16] Loïc Paulevé, Hervé Jégou, and Laurent Amsaleg. Locality sensitive hashing: A comparison of hash function types and querying mechanisms. *Pattern Recognition Letters*, 31(11):1348–1358, 2010.

[17] Michael Connor and Piyush Kumar. Fast construction of k-nearest neighbor graphs for point clouds. *Visualization and Computer Graphics, IEEE Transactions on*, 16 (4):599–608, 2010.

[18] Kiana Hajebi, Yasin Abbasi-Yadkori, Hossein Shahbazi, and Hong Zhang. Fast approximate nearest-neighbor search with k-nearest neighbor graph. In *IJCAI Proceedings-International Joint Conference on Artificial Intelligence*, volume 22, page 1312, 2011.

[19] Jon Louis Bentley. Multidimensional binary search trees used for associative searching. *Communications of the ACM*, 18(9):509–517, 1975.

[20] Kun Zhou, Qiming Hou, Rui Wang, and Baining Guo. Real-time kd-tree construction on graphics hardware. In *ACM Transactions on Graphics (TOG)*, volume 27, page 126. ACM, 2008.

[21] Nimrod Megiddo. Linear programming in linear time when the dimension is fixed. *Journal of the ACM (JACM)*, 31(1):114–127, 1984.

[22] Sebastiano Battiato, Domenico Cantone, Dario Catalano, Gianluca Cincotti, and Micha Hofri. An efficient algorithm for the approximate median selection problem. In *Algorithms and Complexity*, pages 226–238. Springer, 2000.

[23] Warren Hunt, William R Mark, and Gordon Stoll. Fast kd-tree construction with an adaptive error-bounded heuristic. In *Interactive Ray Tracing 2006, IEEE Symposium on*, pages 81–88. IEEE, 2006.

[24] Chanop Silpa-Anan and Richard Hartley. Optimised kd-trees for fast image descriptor matching. In *Computer Vision and Pattern Recognition, 2008. CVPR 2008. IEEE Conference on*, pages 1–8. IEEE, 2008.

[25] John A Nelder and Roger Mead. A simplex method for function minimization. *The computer journal*, 7(4):308–313, 1965.

[26] Chris Cheadle, Marquis P Vawter, William J Freed, and Kevin G Becker. Analysis of microarray data using z score transformation. *The Journal of molecular diagnostics*, 5(2):73–81, 2003.

[27] John Ousterhout and Fred Douglis. Beating the i/o bottleneck: A case for log-structured file systems. *ACM SIGOPS Operating Systems Review*, 23(1):11–28, 1989.