Data Dependent Hashing for Approximate Nearest Neighbor Searching

Matthew Kangas
Division of Science and Mathematics
University of Minnesota, Morris
Morris, Minnesota, USA 56267
kanga139@morris.umn.edu

ABSTRACT

Searching for items similar to a query item is a straightforward process when the data being compared has only a few dimensions. This changes as the complexity of the data increases, causing a problem referred to as the curse of dimensionality, where the runtime of a search is linear in both dimension of the data and number of elements being searched. The solution is to approximate a near neighbor instead of finding the nearest neighbor. A modification to locality sensitive hash based methods reaches a less than linear runtime based on the number of items in the dataset being searched. This is an improvement over possible run query times seen with locality sensitive hash based methods for this problem.

1. INTRODUCTION

1.1 Nearest Neighbor Search

The computational problem called the nearest neighbor search is a simple one: find the item in a set S closest to a given query item q. This problem is solved easily in low dimensional spaces. Datasets that reside in one, two, or three dimensions are generally solved with conventional searches, running through the dataset and computing the distance between every item in the dataset s and q. This becomes a problem when the dimension of the data increases, since the cost of performing that distance calculation increases with the dimension of the dataset, and is referred to as the "curse of dimensionality" This curse of dimensionality has provided motivation towards finding more time efficient ways of finding a nearest neighbor, and given incentive to change the problem, creating the "Approximate Nearest Neighbor", or ANN subproblem.

1.2 Approximate Nearest Neighbor

ANN is similar to the standard nearest neighbor problem in that the goal is to find an item s that is close to a query item q, but the qualification that it is the closest item is removed. Instead, the search just needs to return an s that is sufficiently close to the q. The level of accuracy changes based on the characteristics of the dataset, and the requirements of the goal of the search. Trying to compare handwritten characters to those in a database have less reason to be cautious about returning bad matches compared to a system checking the fingerprints of a unknown person. In the latter case, returning nothing would be better than returning something that did not really match.

UMM CSci Senior Seminar Conference, April 2017 Morris, MN.

2. BACKGROUND

2.1 Nearest Neighbor Searching

Nearest neighbor search methods are used when matching a given piece of data up against a larger collection of data that may not exactly match the data that is being matched against. These methods commonly show up in spell checking, where there is a dictionary of possible words, and the goal is to find those words in the dictionary that are closest to the word that is misspelled. Nearest neighbor searching is also used for pattern recognition with textual information, matching up the symbol scanned in with items in a set of possible symbols. [6]

2.2 Approximate Nearest Neighbor

While finding the closest match between a queried item and a set of possible matches is ideal, sometimes this is a computationally expensive task, with costs scaling both in terms of the number of items being searched through, and the dimension. As dimension increases, it is a multiplicative factor of the number if items, since the costs of the comparisons between each item are all more complicated. The answer to this situation is to find an item in the set that is close enough, and is within some acceptable distance of the queried item. Deciding on an acceptable distance for a given problem is a trade-off between the cost of searching, with tighter bounds taking more time to find an acceptable item, and quality of a match, where less strict bounds will return faster results, but with less guarantee about the accuracy of the matching item being returned. [3]

2.3 Hashing

General hash functions are many-to-one functions. Where a larger variable length space of data is mapped onto a smaller fixed length space of data. A larger space is one that takes more space to store any given value, so the space of integers, $(138, 23459, 34776, 439674, ...)$ is larger than the space of integers that are four long $(4727, 8410, 3773, 9344, ...)$. So that for any output of a hash function $f(x)$, there can be multiple values for x. Where x is an item in the larger space, and $f(x)$ is an item in the smaller space. Since the output space is generally smaller than the input space, a *collision* occurs where two input values return the same hash result

for a given hash function. Two of the primary uses of hashes are fast lookup of items in a data set, and verifying integrity of files. Verifying file integrity requires a few special properties that are not needed here, so the hash functions being discussed here will be those used for fast lookup applications. This is normally accomplished though the use of hash tables, where the data to be searched is hashed, and the results are stored in the table. To lookup an item from the hash table, a query item is hashed, and the item is retrieved from the hash table from the location that has that value. [4]

2.4 Locality Sensitive Hashing

Locality Sensitive Hashes (LSH) are a variant of a standard hash function, with some key differences. LSH functions are designed in ways such that when two inputs are within a distance r_1, there is a probability, p_1, that they will have the same LSH output. These values, p_1 and r_1, are picked when the LSH is made, and are chosen by those that create the LSH. The exact values chosen are dependent on how accurate matches have to be given the context of the search. This collision should ideally only happen on values that are considered close to each other, so there should be at most a small probability, p_2, that values that are far apart, r_2, will collide. However, instead of using r_1 and r_2, we will use r and cr, where $c > 1$. These functions are defined over a set of data, S, and will be accompanied with a function that can measure distance between two items in this set. [5]

For example, given a LSH function f for the set of all five letter strings, where $p_1 = .5$, $p_2 = .1$, $r = 2$, and $c = 1$. The distance function in this example just measures matching letters, so "jolly" and "joker" would be three away. The values of $r = 2$ and $p_1 = .5$ mean that f will generate a collision in 50% of possible strings that are within two characters. So there is a 50% probability that "jolly" will collide with "moldy". Remember that even strings that are not words will have valid results in f, so this can be used to possibly find the correct spelling of a word based on a misspelling such as "jooli".

2.5 Applied Locality Sensitive Hashing

These collisions can be used to quickly find possible close matches to the query value in a already existing dataset. Since LSH functions are designed in such a way that close items are more likely to have collisions than items that are far apart, by presorting the dataset based on the results of the hash functions used, the number of items that have to be directly checked for closeness is reduced. However, each hash function generated only has a chance of creating a collision between the query item and any given item that is close to it. This is based on that probability, p_1, defined in the hash function. So any given point within r_1 of the query point only has a chance to collide with any given LSH function. To compensate for this, and increase the chance that we will find a close match between the query point and the items in the dataset, we will make multiple attempts at creating collisions with the hash functions. This is done in the LSH search method by making multiple different LSH functions. The functions are randomly generated from a family of LSH functions with a series of linear combinations. Since there are many functions that will meet the requirements to be a LSH for a given set, a large family of hash functions that meet the requirements can be generated. [3]

Table 1: My caption

f_1(jooli)

f_1	
Key	**Value**
f_1(jolly)	jolly
f_1(jibed)	jibed
f_1(qubit)	qubit
f_1(squib)	squib

f_3(jooli)

f_3	
Key	**Value**
f_3(squib)	squib
f_3(jolly)	jolly
f_3(qubit)	qubit
f_3(jibed)	jibed

Table 2: An illustration of the LSH search data structure

To make the query process efficient, the dataset to be searched is preprocessed into a set of hash tables. Each hash table in the set is assigned a hash function that was generated from the family of hash functions. Then for each table, the entire dataset that is being preprocessed is hashed with each of the hash functions. The results of these hashes are used as the keys in the hash table. There will be L different LSH functions in the set for a given dataset such that $1 < L < n$. In the best case scenario, L will be n^p. With p being defined as the ratio between the log of the true collision probability and the log of the false collision probability, or $r = \frac{log(p_1)}{log(p_2)}$. This will be $0 < p < 1$. When this is complete, for a dataset with n items, there will be around nL items being stored. Then after the hashing is finished, all unallocated positions in the hash tables are removed. Giving this method a space efficiency of $O(nL)$. The LSH method has a time requirement of $O(nLKT)$. n and L represent the same values from above. K represents a measure of the complexity of the hash functions being generated from the family of LSH functions. A larger K will add more variation to the functions being used to preprocess the data and larger values are required for datasets with higher dimensions, but will increase the time required to run the preprocessing. T is the computational time requirements of each LSH function call. There are methods of reducing the dimension of data, at the expense of the accuracy of that data. [3]

For example, there is a dataset W which contains a collection of words. This dataset would have to be preprocessed before any searching can happen, this takes place before the LSH method can be used for querying. When a word w is given, it is hashed with each of the randomly generated LSH functions tied to each hash table in the structure. The values that are associated with the keys returned by the hash results are then returned for the next stage. Only these words, which have a matching hash in one of the tables, are going to be compared directly with the query word w. The time saved is a trade off between both reliability and accuracy. If L is too small, there is a possibility that no close matches will be found, and if L is too large, the efficiency gains are lost to excessive computations. [3]

The data after preprocessing will be in L hash tables of length n as shown in figure 2. Most of these hash tables will have at least one element in box and boxes that are empty are trimmed for space efficiency. Each hash table is also pared with a locality sensitive hash function from the LSH family G denoted g_0 through g_l. These are all randomly created to help increase the distribution of points that will get matched up in the hash table and therefore increase the chance that a good match is found for the query point. To then query a point, the hash functions g_0 through g_l are applied to the point, and the corresponding values are pulled

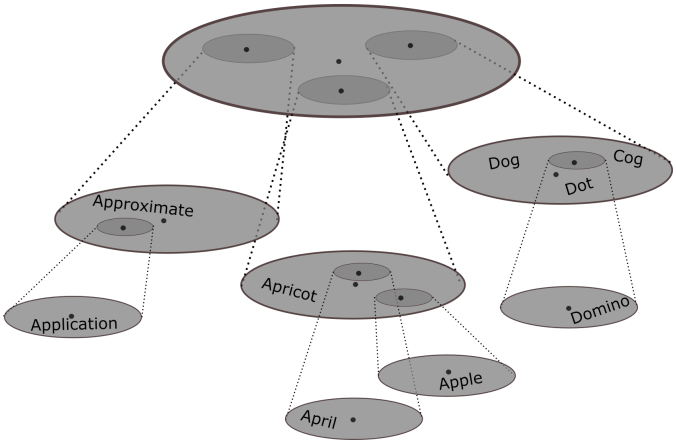

Figure 1: An illustration of the spherical LSH partitioning when applied to a dataset of words

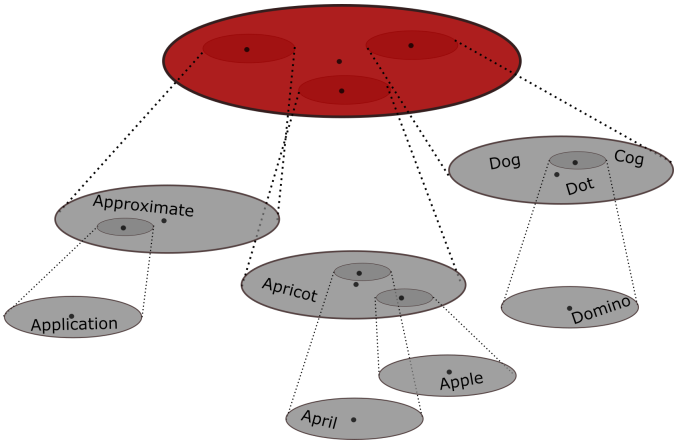

Figure 2: Searching for a near neighbor to "apble"

from each hash table for a final distance comparison to see if any of them are actually close.

3. METHOD

3.1 Data Dependent Hashing

The data dependent hashing DDH method was introduced in [1] and was improved upon in [2]. This method strays away from the LSH method discussed above in that the structure used to hold the processed dataset before it is queried is determined by the properties of the data. In an LSH search, the data structure is always a set of hash tables, no matter what the dataset looks like. In DDH the dataset takes the shape of a recursively constructed tree, that is determined by the properties of the dataset.

3.2 Spherical LSH Partitioning

[NOTE: This section needs to be rewritten already, it was written before the section after it, and contains information that belongs there and not here]

To bring the dataset being searched into a uniform shape, the data is normalized to fit on a d-dimensional sphere with a radius of 1. The dimension used here is normally smaller than the original dataset, but it can be the same dimension. The normalization step can introduce some small distortions to the dataset. To partition the dataset into discreet groups, first, a random value, g from a normal distribution is chosen. Then every point within $\sqrt{2} - o(1)$ is added into a subset. This value ends up being slightly smaller than the hypotenuse of a right triangle with the two shorter sides being 1. That will put every item that is currently not in another subset, and on this half of the sphere into this subset. This continues until every item in the dataset is in a subset. This can be a computationally expensive process since there is a random element added in what points get picked to partition on, and the costly distance measurements that have to be done between each item in the set and the item that they are being grouped with. However once the preprocessing is done for a dataset, it reduces the work that needs to be done to query from the dataset. [2]

In figure 1, each circle is an area that the spherical LSH found a partition of points. The areas between these circles have points but they were included for sake of clarity, and would actually be partitions as small as a single point. This structure is constructed recursively at each level by the preprocessing, with one level being made a time before passing the contents of any of the partitions into the spherical LSH to construct the next level down of the structure.

3.3 Preprocessing

The efficiency gained with data dependent hashing for approximate nearest neighbor search come from the way that the data is preprocessed before it can be queried. The preprocessing operates on a dataset containing n elements. This dataset is assumed to be of dimension $\theta(\log\, n \cdot \log \log n)$, if the dataset is not in this dimension, it can be normalized with small distortions. Where LSH based methods build a set of hash tables for querying purposes, the DDH based method will build a tree with recursive partitions of the space. This starts by splitting the dataset into a number of dense data partitions using the spherical partitioning. Since not every point in the dataset will be in one of these clusters, an extra partition will contain every point that isn't in another partition. These dense partitions are limited to a chosen radius, so there will be items that are in the dataset, but are not in something that can be a considered a dense partition. From this point, this process is repeated on each dense partition recursively, by mapping the partition onto a sphere of radius 1. The set of items that is not considered a dense partition is transformed and then also preprocessed like the original dataset. This in the end leaves a tree where all of the items that are in a dense partition at the lowest level of the tree are similar enough that they count as near neighbors for each other. This preprocessing can be done in near linear time, $O(n^{o_c(1)})$, where $o_c(1)$ is some small quantity that approaches 0 as c increases. c being the parameter that defines what is a far neighbor versus a near neighbor in conjunction with r.

In figure 3.3 is a representation of the preprocessing being done on a 2-dimensional space. This is much lower than the spaces generally used, and would not actually be a good application of data dependent hashing. In the figure, there are partitions of points, these are found using the spherical LSH algorithm described above, and are stored along with another partition of all of the points that are not included in

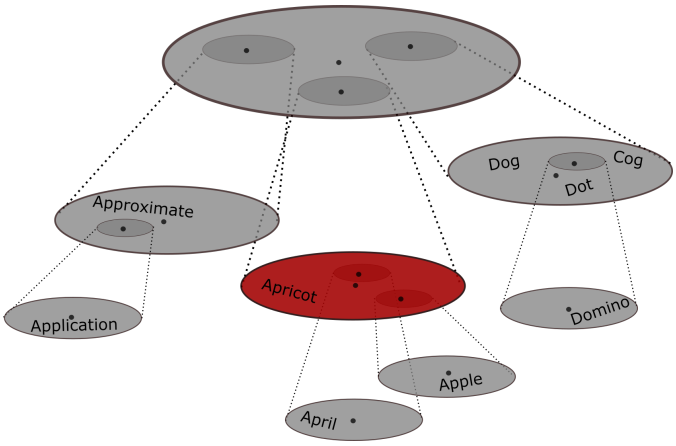

Figure 3: A partition that is close to the query point "apble" but not close enough

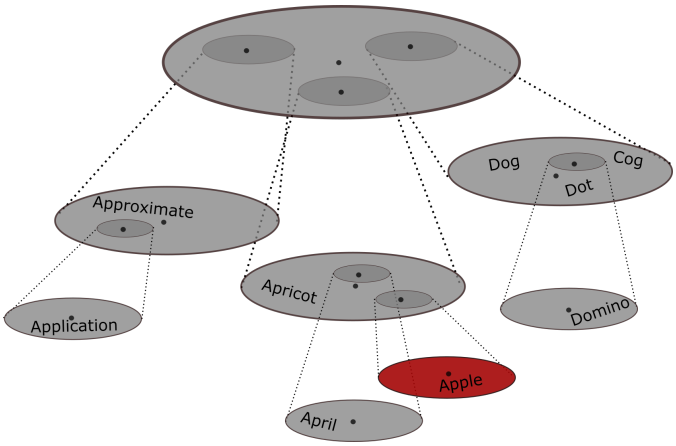

Figure 4: The partition whose center point is considered close enough th "apble" to qualify as a near neighbor

these partitions. Normally this process will then be repeated inside each partition, but for the sake of image clarity, this process was left out. [2]

3.4 Querying

Once preprocessing is complete, the dataset can be queried. To query for a near neighbor to the point q, the first step is to recursively query the recursively structured partitions described above. If the point is found in the top level, the search stops there. The neighbor to q is found when a value s is found within r of q in dataset. This distance is computed between q and the origin of the current partition. If the point is not found at the current level, the recursive call is made on partitions that are approximately as far away from the current partition's origin, as q was found to be with the distance function call. These are the partitions that are likely to contain q, or to contain the partition that contains q.

Looking at figure 3.3, To query for a point that's close to the misspelled word "apble", first the method finds the distance between "apble" and the center point in the top partition. If the distance between these points is less than r, which is the maximum distance for a close match, then that center point is returned as the match. If the distance is greater than r, the method computes the distance between the query value "apble" and the centers of the partitions that are inside the current partition marked in red.

In figure 3.3 the method is currently looking in the partition marked in red, though at this stage the method would be checking all partitions that could reasonably contain a close match, this would be determined by checking the distance of the query value against the center of each partition, if the distance minus the radius of the partition is larger than r, the partition is ignored, since all elements in that partition will be too far away to return a possible near neighbor. At this stage, the process recursively repeats the steps from the top level.

The method will continue to recurse down the tree structure until it either finds a suitable item or it runs out of items to check. In figure 3.3 the method is checking a partition of the space that happens to contain "apple" as the center point of the partition. Assuming that "apple" is considered close enough to "apble" to be a match, the method will return "apple" and this run will be complete. However, if "apple" wasn't the center item in that partition, and was instead just contained by that partition. Then a different word such as "able" might be returned even when the intended word was "apple". [2]

4. RESULTS

4.1 Efficiency

The space requirements of the preprocessing and query steps are on the scale of $O(n^{1+o_c(1)})$. This is just slightly above linear due to the restricted number of times that the preprocessing will recurse on itself for a given dataset. Most elements in the original dataset will only be stored in one location in the new query tree, and the overlap of elements is minimal. The Query is actually has sub linear time costs, at $O(n^{o_c(1)})$. This is mostly explained by the query traversing a tree with a relatively high branching factor in a near depth first manner. The depth first properties of the search doesn't open this up to a bad worst case, since any possible paths

where a match could be found, are in the direction that is being searched. The areas that are ignored are far enough away from the query target that there is very little chance a match will be found there. These two factors, the reasonable item storage and fast query time, mean that this is a good candidate for searching high dimension space. Earlier methods for approximate nearest neighbor search had query times around $O(dlog(n))$ and or $O(dn^{1/c}log^2n\ log\ log\ n)$. These will both have significantly longer query times compared to the data dependent hashing query time in most cases. There is a case where on specifically designed data, and a purposefully picked set of query points, the data dependent hashing will perform worse. However given that both of the older methods have a dimension component, and this new one does not, the datasets that the others might be able to perform better in, are probably the datasets that would not be best handled by algorithms like this.

5. CONCLUSION

In conclusion, using this data dependent hashing scheme for approximate nearest neighbor searching in high dimensionality space is going to improve the cost of preprocessing, down from $O(LNKT)$ in classical LSH to $O(n^{o_c(1)})$ with data dependent hashing, the space requirements, from $O(LN)$ to $O(n^{1+o_c(1)})$, and the query time requirements, from $O(dlog(n))$ to $O(n^{o_c(1)})$. These are significant improvements that can be applied to spell checking, image recognition, plagiarism detection, and computer vision applications, among others. The benefits gained by moving from a list of hash tables to a new tree build using spherical locality sensitive hashing reduces the space required to store the dataset, decreases the time required to pre-process the dataset, and speeds up the query process by limiting the search space to only the areas that are likely to contain matches.

6. REFERENCES

[1] A. Andoni, P. Indyk, H. L. Nguyen, and I. Razenshteyn. Beyond locality-sensitive hashing. In *Proceedings of the Twenty-fifth Annual ACM-SIAM Symposium on Discrete Algorithms*, SODA '14, pages 1018–1028, Philadelphia, PA, USA, 2014. Society for Industrial and Applied Mathematics.

[2] A. Andoni and I. Razenshteyn. Optimal data-dependent hashing for approximate near neighbors. In *Proceedings of the Forty-seventh Annual ACM Symposium on Theory of Computing*, STOC '15, pages 793–801, New York, NY, USA, 2015. ACM.

[3] P. Indyk and R. Motwani. Approximate nearest neighbors: Towards removing the curse of dimensionality. In *Proceedings of the Thirtieth Annual ACM Symposium on Theory of Computing*, STOC '98, pages 604–613, New York, NY, USA, 1998. ACM.

[4] Wikipedia. Hash function — Wikipedia, the free encyclopedia. `https://en.wikipedia.org/wiki/Hash_function`, 2017. [Online; accessed 1-May-2017].

[5] Wikipedia. Locality-sensitive hashing — Wikipedia, the free encyclopedia. `https://en.wikipedia.org/wiki/Locality-sensitive_hashing`, 2017. [Online; accessed 29-March-2017].

[6] Wikipedia. Nearest neighbor search — Wikipedia, the free encyclopedia. `https://en.wikipedia.org/wiki/Nearest_neighbor_search`, 2017. [Online; accessed 29-March-2017].