An Appreciation of Locality Sensitive Hashing

John Mount*

November 21, 2011

Abstract

We share our admiration for a set of results called "locality sensitive hashing" by demonstrating a greatly simplified example that exhibits the spirit of the techniques.

1 Introduction

Locality sensitive hashing was invented by Rajeev Motwani and Piotr Indyk[IM98, IG99] and rapidly further developed.[SC08, MNP06, AI08] Many of the methods and ideas were already anticipated[BCFM98, Lub96] but locality sensitive hashing is awe inspiring in its originality, simplicity, beauty and effectiveness. In addition locality sensitive hashing is a remarkable technique as it works even when drastically abridged and simplified (as we do here).

It is our goal to give a description of conditions where the technique works and a heuristic argument why it works (using only elementary math).

Paper hosted at:
http://www.win-vector.com/blog/2011/11/an-appreciation-of-locality-sensitive-hashing/
and example code at https://github.com/WinVector/Locality-Sensitive-Hashing-Example.

2 Definitions

Suppose we have a large set S of vectors in $\{0,1\}^n$. Further suppose we are interested in finding near neighbors. That is we would like to find vectors $x, y \in S$ such that x and y are similar. By similar we mean the number of positions they disagree in is small. Let $x \triangle y$ denote the set $\{i | x_i \neq y_i\}$ and $|x \triangle y|$ the size of that set. We want to find x, y such that $|x \triangle y|$ is small, or x, y that are near neighbors.

A couple of obvious methods are worth mention.

We could find near neighbor x, y by trying all pairs and retaining pairs such that $|x \triangle y|$ is small. If the size of S (denoted as $|S|$) is m then this technique requires us to examine $\binom{m}{2} = m(m-1)/2$ pairs, which can be a lot of work if m is large.

We could also find near neighbor x, y by entering all vectors from S in a fast lookup table (like a hash table) and then for each vector $x \in S$ try all changes of up to d coordinates of x, checking if any of these variations are in our lookup table. This would require $\binom{n}{d} m$ lookups which becomes very large as d grows. This method can be improved by ideas like "search to the middle" where instead of searching distance d from each vector and looking for other vectors we search distance $d/2$ from each vector and enter all new vectors formed into our lookup table while looking for collisions (reducing the number of pair inspections to $\binom{n}{d/2} m$ at the cost of some additional storage).

What can we do if m is large and the minimum distance between vectors is not small? One thing we can do is appeal to locality sensitive hashing.

*email: mailto:jmount@win-vector.com

3 Locality Sensitive Hashing

The core ideas of locality sensitive hashing are very simple and quite brilliant. The papers are quite technical because they include a number of important improvements and techniques to ensure that locality sensitive hashing works in a wide set of situations. But we can call out the key ideas and demonstrate them on specially chosen easy data. Easy data allows us work up an intuition of how locality sensitive hashing works.

To my mind the three key ideas are:

1. The notion of a locality sensitive hash itself.

   Let $h() : \{0,1\}^n \to \{0,1\}^k$ be a function with $k < n$. We will pick $h()$ such that $h(x)$ looks somewhat random (and functions thus chosen are called "hashes" or "hash functions"). Very roughly (and this is far cruder than the definition actually used in the literature) $h()$ is a locally sensitive hash if for x, y chosen uniformly at random from S then the probability that x is near y is increased if we observe $h(x) = h(y)$. Throughout let us define the comparable classes under h as: $C_h(v) = \{z \in S | h(z) = v\}$.

   The brilliance of the idea is that the concept is deliberately weaker than more natural concepts like low-distortion mappings[1]. The weaker requirements allow simpler implementations.

2. The fact that coordinate selection is actually a locally sensitive hash function.

   By coordinate selection we are going to mean picking a set of indices $i(1), i(2), \cdots i(k)$ where all $i(j)$ have $1 \leq i(j) \leq n$ (repetition allowed).[2] We define the function $h() : \{0,1\}^n \to \{0,1\}^k$ to be such that $h(x)_j = x_{i(j)}$ (that is $h()$ just copies the k positions named by the $i()$). For our hash function $h()$ we will pick the $i()$ uniformly at random from $1 \cdots n$ (with repetition).

   This sort of function is much simpler than the kind of functions previously investigated.

3. The straightforward use of the above two facts to find near neighbors.

   The core idea (though there are a lot of variations and mechanisms refining the idea) is we want to pick an $h()$ such that for $x, y \in S$ such that x and y are near neighbors we have both:

   (a) $h(x) = h(y)$ with non-negligible probability when $x \triangle y$ is small.

   (b) $|C_h(v)|$ tends to be small.

   If both of these things are true then we can find many close x, y by repeating the following procedure a reasonable number of times:

   - Pick a hash function $h()$ from our described distribution.
   - Scan through all $x \in S$ and put each x in a list labeled v where $h(x) = v$.
   - For each list labeled v that is small enough inspect (and remember across all repetitions) all pairs of vectors in the list x, y that are near each other.

   The procedure is probabilistic (we use randomness in our construction of $h()$) and any pass of it could fail to find near pairs x, y (near items may fail to meet in the same cluster or they may hide in a large cluster we fail to inspect). We will mitigate these failures by repetition. But the runtime is excellent: each repetition takes no more than m evaluations of $h()$ plus the time to inspect all of the pairs in the small lists. Remember: we only inspect pairs from the same small list (not between different small lists. So the number of inspections is easily bounded, if we don't inspect lists larger than c then the total number of pair inspections can be no more than cm (which can be much smaller than the $\binom{m}{2}$ and $\binom{n}{d/2}$ pair inspections discussed earlier).

[1] For example applications of the JohnsonLindenstrauss lemma [DG03, AC06, Ach03]

[2] Repeated coordinates are of no benefit, but they do no harm and not having to account for avoiding repetition makes some of the exposition easier.

The original papers are much more complicated than this- because they are solving a hard problem in all its generality. As we said we are only going to demonstrate the method on easy data, so we don't need nearly as much machinery as the original references.

4 The Demonstration

Suppose we are in the extremely simple case that our set S is composed of m vectors in $\{0,1\}^n$ each of which was chosen uniformly at random from the set of all vectors in $\{0,1\}^n$ that have exactly w ones (with w small compared to n). This is a trivial example- but it is nevertheless interesting to see how the method works in this case.[3] We are going to pick our hash functions $h()$ as described above by picking $i(1), \cdots, i(k)$ uniformly at random (with repetition allowed)[4] from $1 \cdots n$. We will show for a range of k the "hash and then inspect only items with identical hash" method works.

4.1 Property 3a

We name conditions which ensure that it is not unlikely that $h(x) = h(y)$ when $|x \triangle y|$ is small.

Given distinct $x, y \in S$ there are exactly $x \triangle y$ coordinates where x and y differ. To have $h(x) = h(y)$ it must be that $h()$ picked all of its coordinates from the set $\{1 \cdots n\} - (x \triangle y)$. By our construction of $h()$ this happens with odds exactly $\left(\frac{n-|x \triangle y|}{n}\right)^k$.

A little algebra[5] tells us that this quantity is at least $e^{-k|x \triangle y|/(n-|x \triangle y|)}$.

So, to ensure $\text{Prob}[h(x) = h(y)] \geq \frac{1}{2}$ it is sufficient to insist that $k \leq (n/|x \triangle y| - 1)\log(2)$.

To restate: for $k \leq (n/d - 1)\log(2)$. we have $\text{Prob}[h(x) = h(y)] \geq \frac{1}{2}$ for x, y with $|x \triangle y| \leq d$ (where the probability is over the choice of $h()$). Or, if k isn't too big we expect close x, y to end up in the same cluster with good odds.

4.2 Property 3b

We name conditions which ensure that it is not unlikely that $|C_h(v)|$ is small.

First let us get an upper bound on the expected size of a cluster. Since $w \ll n$ we have the cluster with largest expectation is $C_h(0)$) (or $\text{E}[|C_h(v)|] \leq \text{E}[|C_h(0)|]$ for all v). So it is enough to bound the expected size of $C_h(0)$. For each $y \in S$ the odds that $h(y) = 0$ are $\left(\frac{n-w}{n}\right)^k$. Or no more than $e^{-kw/n}$. So the expected size of any cluster is no more than $me^{-kw/n}$.

Now let us bound the odds a cluster is large. Markov's inequality tells us that if s random variable then for any $a > 0$ we have $\text{Prob}[s >= c] \leq \text{E}[s]/c$.[6] So we use our upper bound on expected cluster size and get:

$$\text{Prob}[|C_h(v)| \geq c] \leq me^{-kw/n}/c.$$

Some algebra tells us this quantity is at least $\frac{1}{4}$ for $k \geq n(\log(c) + \log(4))/w$. So if k is large enough we expect small clusters.

4.3 Combining Conditions

For k such that:

$$n(\log(c) + \log(4))/w \leq k \leq (n/d - 1)\log(2)$$

[3] It is important to note that the full methods from the references work on decidedly non-random data. The random choices they make in constructing their hash functions are enough to get the performance they want, they do not depend on the kindness having of random inputs.

[4] This is called "sampling with replacement."

[5] Plus the fact that for $n >= 1$ we have $(1 - 1/n)^n \leq e^{-1} \leq (1 - 1/n)^{n-1}$.

[6] Just some algebra from the obvious statement $\text{Prob}[s >= c]c \leq \text{E}[s]$ where E[] expected value operator.

We expect to be able to find x, y such that $|x \triangle y| \leq d$ (if there are such) in our clusters and we expect our clusters to be small (no more than size c, allowing us to inspect any small cluster in time $\binom{c}{2}$ and inspect all small clusters in time $O(mc)$). If there are $x, y \in S$ with $|x \triangle y| \leq d$ we will find some in one pass with probability at least $\frac{1}{4}$ (the idea being with probability at least $\frac{1}{2}$ some such x, y fall into the same cluster and that cluster is too large to inspect with probability no more that $\frac{1}{4}$). So even after 4 repetitions of the technique we have a good chance of success.

For example if our easy data is such that $\log(m) \ll w \ll n$ and $d(\log(\log(m)) + \log(4)) \ll w \log(2)$ then we could pick $c = \log(m)$ and expect to be able to detect x, y as far apart as $d = w/\log(m)$ in $m \log(m)$ pair inspections per repetition.

Of course if $w \ll n$ we expect the typical size of $|x \triangle y|$ to be almost $2w$ (much larger than the d we can expect to see up to). So the theory is mostly saying that the bulk of the large-spaced data stays out of the way when inspecting for near items. This is in fact useful, a set S constructed as we specified and then "salted" with a few near-variations of of vectors already in S (so there are some near neighbors to find) would be inspect-able by locality sensitive hashing.

Also note we have glossed over essential difficulties centered on the conditional expected size of clusters (like the size of $C_h(h(x))$ for a given x) as this adds complications.[7]

5 Conclusion

You should now be able to see that the essential tension in locality sensitive hashing is to build a hash function that is coarse enough that nearby vectors map to identical clusters yet fine enough that not too many points vectors map to any cluster (by tending to split far apart vectors into different clusters). The wonder is that fairly simple functions (for example coordinate selection functions) meet these conditions for a fairly wide range of parameters and data types. Understanding locality sensitive hashing does require some math but the method stands out as having a very high reward to effort ratio.

References

[AC06] Nir Ailon and Bernard Chazelle, *Approximate Nearest Neighbors and the Fast Johnson-Lindenstrauss Transform*, STOC (2006).

[Ach03] Dimitris Achlioptas, *Database-friendly random projections: Johnson-Lindenstrauss with binary coins*, Journal of Computer and System Sciences **66** (2003), no. 4, 671–687.

[AI08] Alexandr Andoni and Piotr Indyk, *Near-Optimal Hashing Algorithms for Approximate Nearest Neighbor in High Dimensions*, Communicatons of the ACM, Springer-Verlag, January 2008, pp. 117–122.

[BCFM98] A. Z. Broder, M. Charikar, A. M. Frieze, and M. Mitzenmacher, *Min-Wise Independent Permutations*, STOC (1998), 1–36.

[DG03] Sanjoy Dasgupta and Anupam Gupta, *An elementary proof of a theorem of Johnson and Lindenstrauss*, Random Structures and Algorithms **22** (2003), no. 1, 60–65.

[IG99] P Indyk and A Gionis, *Similarity search in high dimensions via hashing*, Proceedings of the 25th VLDB, 1999.

[IM98] Piotr Indyk and Rajeev Motwani, *Approximate Nearest Neighbors: Towards Removing the Curse of Dimensionality*, STOC (1998).

[Lub96] Michael Luby, *Pseudorandomness and cryptographic applications*, Princeton, 1996.

[MNP06] Rajeev Motwani, Assaf Naor, and Rina Panigrahi, *Lower bounds on Locality Sensitive Hashing.*

[SC08] M Slaney and M Casey, *Locality-sensitive hashing for finding nearest neighbors*, IEEE Signal Processing Magazine **25** (2008), no. 2, 128.

[7] These complications are overcome in the original literature by a more precise definition of locality preserving hash that involves separate notions of near and far.