On the Problem of $p_1^{-1}$ in Locality-Sensitive Hashing

Thomas Dybdahl Ahle

IT University and BARC, Copenhagen

Abstract. A Locality-Sensitive Hash (LSH) function is called $(r,cr,p_1,p_2)$-sensitive, if two data-points with a distance less than $r$ collide with probability at least $p_1$ while data points with a distance greater than $cr$ collide with probability at most $p_2$. These functions form the basis of the successful Indyk-Motwani algorithm (STOC 1998) for nearest neighbour problems. In particular one may build a $c$-approximate nearest neighbour data structure with query time $\tilde{O}(n^{\rho}/p_1)$ where $\rho = \frac{\log 1/p_2}{\log 1/p_1} \in (0,1)$. This is sub-linear as long as $p_1$ is not too small. Such an algorithm is significant, since most high dimensional nearest neighbour problems suffer from the curse of dimensionality, and can’t be solved exact, faster than a brute force linear-time scan of the database.

Unfortunately many of the best LSH functions tend to have very low collision probabilities, including the best functions for Cosine and Jaccard Similarity. This means that the $n^{\rho}/p_1$ query time of LSH is often not sub-linear after all, even for approximate nearest neighbours!

In this paper, we improve the general Indyk-Motwani algorithm to reduce the query time of LSH to $\tilde{O}(n^{\rho}/p_1^{1-\rho})$ (and the space usage correspondingly.) Since $n^{\rho}/p_1^{1-\rho} < n \Leftrightarrow p_1 > n^{-1}$. our algorithm always obtains sublinear query time, for all collision probabilities at least $1/n$. For $p_1$ and $p_2$ small enough, our improvement over all previous methods can be up to a factor $n$ in both query time and space.

The improvement comes from a simple change to the Indyk-Motwani algorithm, which we call “LSH with High-Low Tables“. This technique can easily be implemented in existing software packages.

Keywords: locality-sensitive hashing · nearest neighbour · similarity search

1 Introduction

Locality Sensitive-Hashing (LSH) [16] is one of the most efficient approaches to the nearest neighbour search problem in high dimensional spaces. It comes with theoretical guarantees, and it has the advantage of easy adaption to nearly any metric or similarity function one might want to use for search.

The $(r_1,r_2)$-near neighbour problem is defined as follows: Given a set $X$ of points, we build a data-structure, such that given a query, $q$ we can quickly find
a point \( x \in X \) with distance \(< r_2 \) to \( q \), or determine that \( X \) has no points with distance \( \leq r_1 \) to \( q \). Given a solution to this “gap” problem, one can obtain a \( r_1/r_2 \)-approximate nearest neighbour data structure, or even an exact\footnote{In general we expect the exact problem to be impossible to solve in sub-linear time, given the hardness results of \cite{indyk2008improved}. However for practical datasets it is often possible.} solution using known reductions \cite{indyk2008improved}.

For any measure of similarity, the gap problem can be solved by LSH: we find a distribution of functions \( H \), such that \( p_1 \geq \text{Pr}_{h \sim H}[h(x) = h(y)] \) when \( x \) and \( y \) are similar (distance \( \leq r_1 \)), and \( p_2 \leq \text{Pr}_{h \sim H}[h(x) = h(y)] \) when \( x \) and \( y \) are dissimilar (distance \( \geq r_2 \)). Such a distribution is called \((r_1, r_2, p_1, p_2)\)-sensitive. If \( p_1 > p_2 \) the LSH framework gives a data-structure with query time \( \tilde{O}(n^\rho/p_1) \) for \( \rho = \frac{\log 1/p_1}{\log 1/p_2} \), which is usually significantly faster than the alternatives.

At least when \( p_1 \) is not too small.

The two most common families of LSH is Cross-Polytope (or Spherical) LSH \cite{crissman2015better} for Cosine similarity and MinHash \cite{Indyk98} for Jaccard Similarity.

Cross-Polytope is the basis of the Falcon software package \cite{Nandakumar14}, and solves the \((r, cr)\)-near neighbour problem on the sphere in time \( \tilde{O}(n^{1/c^2}/p_1) \). Here \( p_1 = \exp\left(-\frac{\varepsilon^2}{d} (1 - o(1)) \log d\right) \), where \( \tau = \|p - q\|_2 \in [0, 2] \) is the distance between two close points. We see that already at \( \tau \approx \sqrt{2} \) (which corresponds to near orthogonal vectors) the \( 1/p_1 \) factor results in a factor \( d \) slow-down. For larger \( \tau \in (\sqrt{2}, 2) \) the slow-down can grow arbitrarily large. Using dimensionality reduction techniques, like the Johnson Lindenstrauss transform, one may assume \( d = \varepsilon^{-2} \log n \) at the cost of a factor \( 1 + \varepsilon \) distortion of the distances. However if \( \varepsilon \) is just 1/100, the slow-down factor of \( d \) is still worse than, say, \( n^{1/2} \) for datasets of size up to \( 10^8 \), and so if \( c \leq \sqrt{2} \) we get that \( n^\rho/p_1 \) is larger than \( n \). So \textit{worse than a brute force scan of the database!}

The MinHash algorithm was introduced by Broder et al. for the Alta Vista search engine, but is used today for similarity search on sets in everything from natural language processing to gene sequencing. MinHash solves the \((j_1, j_2)\) gap similarity search problem, where \( j_1 \in (0, 1) \) is the Jaccard Similarity of similar sets, and \( j_2 \) is that of dissimilar sets, in time \( \tilde{O}(n^\rho/j_1) \) where \( \rho = \frac{\log 1/j_1}{\log 1/j_2} \). (In particular MinHash is \((j_1, j_2, j_1, j_2)\)-sensitive in the sense defined above.) Now consider the case \( j_1 = n^{-1/2} \) and \( j_2 = n^{-3/10} \). This is fairly common as illustrated in fig. 1a In this case \( \rho = \frac{\log 1/j_1}{\log 1/j_2} = 5/6 \), so we end up with \( n^\rho/j_1 = n^{13/12} \). Again \textit{worse than a brute force scan of the database!}

In this paper we reduce the query time of LSH to \( n^\rho/p_1^{1-\rho} \), which is less than \( n \) for all \( p_1 > 1/n \). In the MinHash example above, we get \( n^\rho/p_1^{1-\rho} = n^5/6 + (3/5)/6 = n^{7/8} \). More than a factor \( n^{0.208} \) improvement(!) In general the improvement of \( p_1^{\rho} \) may be as large as a factor of \( n \) when \( p_1 \) and \( p_2 \) are both close to \( 1/n \). This is illustrated in fig. 1b.

The improvements to LSH comes from a simple observation: During the algorithm of Indyk and a certain “amplification” procedure has to be applied
\[ \kappa = \frac{\log n}{\log \frac{1}{p_2}} \] times. When \( \log 1/p_2 \) does not divide \( n \), which is extremely likely, the amount of amplification has to be approximated by the next integer. We propose instead an ensemble of two kinds of LSH tables with respectively \( \lceil \kappa \rceil \) and \( \lfloor \kappa \rfloor \) concatenations of the hash function. We call those respectively “High” and “Low” tables. When analysed sufficiently precisely yields the improvements described above.

1.1 Related Work

We will review various directions in which LSH has been improved and generalized, and how those results related to what is presented in the present article.

In many cases, the time required to sample and evaluate the hash functions dominate the time required by LSH. Recent papers \[15,11\] have reduced the number of distinct calls to the hash functions which is needed. The most recent paper in the line of work is \[11\], which reduces the number of calls to \((\frac{\log n}{\log 1/p_2})^2/p_1\). On top of that, however, they still require \( n^\rho/p_1 \) work, so the issue with small \( p_1 \) isn’t touched upon. In fact, some of the algorithms in \[11\] increase the dependency from \( n^\rho/p_1 \) to \( n^\rho/(p_1 - p_2) \).

Other work has sought to generalize the concept of Locality Sensitive Hashing to so-called Locality Sensitive Filtering, LSF \[8\]. However, the best work for set similarity search based on LSF \[3,12\] still have factors similar in spirit to \( p_1^{-1} \). E.g., the Chosen Path algorithm in \[12\] uses query time \( \tilde{O}(n^\rho/b_1) \), where \( b_1 \) is the similarity between close sets.

A third line of work has sought to derandomize LSH. The result is so-called Las Vegas LSH \[2,21\]. Here the families \( H \) are built combinatorially, rather than at random, to guarantee the data structure always return a near neighbour, when one exists. While these methods don’t have probabilities, they still end up with similar factors for similar reasons.

As mentioned, the reason \( p_1^{-1} \) shows up in all these different approaches, is that they all rely on the same amplification procedure, which has to be applied an integer number of times. One might wonder if tree based methods, which do an adaptive amount of amplification, could get rid of the \( 1/p_1 \) dependency. However as evidenced by the classical and current work \[7,13,14\] these methods still have a factor \( 1/p_1 \). We leave it open whether this might be avoidable with better analysis, perhaps inspired by our results for “High-Low” tables.

Finally we note that in their seminal paper on tight LSH lower bounds \[19\], O’Donnell and Wu mention discuss exactly the problem with \( p_1 \) and \( p_2 \) polynomially small. They consider whether the lower bounds could be made tighter to capture the factor \( 1/p_1 \). We similarly now wonder if the factor \( 1/p_1^{-\rho} \) could be shown optimal.

2 Preliminaries

Before we give the new LSH algorithm, we will recap the traditional analysis. For a more comprehensive introduction to LSH, see the Mining of Massive Datasets
(a) Density plots of all pairwise Jaccard similarities in the datasets studied by Mann et al. [18], written in terms of the size of their corresponding datasets. Curiously the largest datasets, like Orkut and Spotify, have the smallest median Jaccard similarities, even when expressed in terms of the dataset size.

We see that reasonable values for $j_1 = p_1$ range between $n^{-1/3}$ and $n^{-1/6}$.

(b) $p_1^{-p_2}$: The possible improvements in query time and space, over classical LSH, as a function of $p_1$ and $p_2$. With $p_1 = n^{-1/4}$ and $p_2 = n^{-1/3}$ we save a factor of $n^{3/16} = n^{0.1875}$.

Fig. 1: Overview over available savings
book [17], Chapter 3. In the remainder of the article we will use the notation 
\[ n = \{1, \ldots, n\} \].
Assume we are given a \((r_1, r_2, p_1, p_2)\)-sensitive LSH family, \(H\), as defined in
the introduction. Let \(k\) and \(L\) be some integers defined later, and let \([m]\) be the
range of the hash functions, \(h \in H\). Let \(n\) be an upper bound on the number of
points to be inserted.\(^2\) The Indyk-Motwani data-structure consists of \(L\) hash
tables, each with \(m^k\) hash buckets.
To insert a point, \(x\), we draw \(L \cdot k\) functions from \(H\), denoted by \((h_{i,j})_{i \in [L], j \in [k]}\).
In each table \(i \in [L]\) we insert \(x\) into the bucket keyed by \((h_{i,1}(x), h_{i,2}(x), \ldots, h_{i,k}(x))\).
Given a query point \(q\), the algorithm iterates over the \(L\) tables and retrieves the
data points hashed into the same buckets as \(q\). The process stops as soon as a
point is found within distance \(r_1\) from \(q\).
The algorithm as described has the performance characteristics listed below.
Here we assume the hash functions can be sampled and evaluated in constant
time. If this is not the case, one can use the improvements discussed in the related
work.

- Query time: \(O(L(k + np_2^k)) = O(n^\rho p_1^{-1} \log n)\).
- Space: \(O(nL) = O(n^{1+\rho}p_1^{-1})\) plus the space to store the data points.
- Success probability 99%.

To get these bounds, we have defined \(k = \lceil \log n \log 1/p_2 \rceil \) and
\[ L = \lceil p_1^{-k} \rceil \leq \exp \left( \log 1/p_1 \cdot \lceil \log n / \log 1/p_2 \rceil \right) + 1 \leq n^\rho / p_1 + 1. \]

It’s clear from this analysis that the \(p_1^{-1}\) factor is only necessary when \( \log n / \log 1/p_2 \)
is not an integer. However in those cases it is clearly necessary, since there is
no obvious way to make a non-integer number of function evaluations. We also
cannot round \(k\) down instead of up, since the number of false positives would
explode: rounding down would result in a factor of \(p_2^{-1}\) instead of \(p_1^{-1}\) — much
worse.

3 LSH with High-Low Tables
The idea of the main algorithm is to create some LSH tables with \(k\) rounded
down, and some with \(k\) rounded up. We call those respectively “high probability”
tables and “low probability” tables. In short “LSH with High-Low Tables”.
The main theorem is the following:

**Theorem 1.** Let \(H\) be a \((r_1, r_2, p_1, p_2)\)-sensitive LSH family, and let \(\rho = \log 1/p_1 / \log 1/p_2\).
Assume \(p_1 > 1/n\) and \(p_2 > 1/n\). Then High-Low tables give a solution to the
\((r_1, r_2)\)-near neighbour problem with the following properties:

\(^2\) If we don’t know how many points will be inserted, several black box reductions
allow transforming LSH into a dynamic data structure.
- **Query time:** $O(n^\rho/p_1^{1-\rho} \log n)$.
- **Space:** $O(n^{1+\rho}/p_1^{1-\rho})$ plus the space to store the data points.
- **Success probability** 99%.

**Proof.** Assume $r_1, r_2, p_1, p_2$ are given. Define $\rho = \log \frac{1/p_1}{\log 1/p_2}$, $\kappa = \frac{\log n}{\log 1/p_2}$, and $\alpha = \lceil \kappa \rceil - \kappa \in [0, 1)$. We build $\lceil a \rceil + \lceil b \rceil$ tables (for $a, b \geq 0$ to be defined), where the $\lceil a \rceil$ tables are “Low” tables, which use the hash function concatenated $\lceil \kappa \rceil$ times as keys, and the remaining $\lceil b \rceil$ are “High” tables, which use it concatenated $\lceil \kappa \rceil$ times.

The total number of High and Low tables to query is then $\lceil b \rceil + \lceil a \rceil$. The expected total number of far points we have to retrieve is

$$n(\lceil a \rceil p_2^{\lceil \kappa \rceil} + \lceil b \rceil p_2^{\lceil \kappa \rceil}) = n([a]p_2^{\kappa-1+\alpha} + [b]p_2^{\kappa+\alpha})$$

$$= [a]p_2^{-1+\alpha} + [b]p_2^\alpha$$

$$\leq ap_2^{-1+\alpha} + (b+1)p_2^\alpha$$

$$\leq ap_2^{-1+\alpha} + bp_2^\alpha + 1.$$ 

For the second equality, we used the definition of $\kappa$: $p_2^\kappa = 1/n$. We only count the expected number of points seen that are at least $r_2$ away from the query. This is because the algorithm, like classical LSH, terminates as soon as it sees a point with distance less than $r_2$.

Given any point in the database within distance $r_1$ we must be able to find it with high enough probability. This requires that the query and the point shares a hash-bucket in one of the tables. The probability that this is doesn’t happen in any of the $\lceil a \rceil$ low tables, and not any of the $\lceil b \rceil$ high tables is

$$(1 - p_1^{\lceil \kappa \rceil})^{\lceil a \rceil}(1 - p_1^{\lceil \kappa \rceil})^{\lceil b \rceil} \leq (1 - p_1^{\lceil \kappa \rceil})^{a-1}(1 - p_1^{\lceil \kappa \rceil})^{b}$$

$$\leq \exp(-(a p_1^{\lceil \kappa \rceil} - b p_1^{\lceil \kappa \rceil})(1 - p_1^{\lceil \kappa \rceil})^{-1})$$

$$= \exp(-(a p_1^{1+\alpha} + b p_1^\alpha)n^{-\rho})(1 - p_1^{\lceil \kappa \rceil})^{-1}$$

$$\leq \exp(-(a p_1^{1+\alpha} + b p_1^\alpha)n^{-\rho}) \cdot 2.$$ 

For the first inequality we used that $1 - x \leq \exp(-x)$. For the equality, we used the definition of $\kappa$ and $\rho$: $p_1^\kappa = p_2^{\rho \kappa} = n^{-\rho}$. For the last inequality we have assumed $p_2 > 1/n$ so $\lceil \kappa \rceil \geq 1$, and that $p_1 < 1/2$, since otherwise we could just get the theorem from the classical LSH algorithm.

We now define $a$ and $b$, both $\geq 0$, such that

$$ap_2^{-1+\alpha} + bp_2^\alpha = a + b \quad \text{and} \quad (1)$$

$$ap_1^{1+\alpha} + bp_1^\alpha = n^\rho. \quad (2)$$

By the previous calculations this will guarantee the number of false positives is not more than the number of tables, and a constant success probability.
We can achieve this by taking

\[
\begin{bmatrix} a \\ b \end{bmatrix} = \left[ \frac{p_2^{-1+\alpha} - 1}{p_1^{-1+\alpha}} \right]^{-1} \begin{bmatrix} 0 \\ n^\rho \end{bmatrix} = \frac{n^\rho}{(p_2^{-1+\alpha} - 1)p_1^\alpha + (1 - p_2^\alpha)p_1^{-1+\alpha}} \left[ \frac{1 - p_2^\alpha}{p_2^{-1+\alpha} - 1} \right].
\]

We can check that both values are non-negative, since \( \alpha \in [0, 1] \), so the definition is meaningful.

When actually implementing “LSH with High-Low Tables”, these are the values you should use for the number of respectively the high and low probability tables. That will ensure you take full advantage of when \( \alpha \) is not worst case, and you may do even better than the theorem assumes.

To complete the theorem we need to prove \( a + b \leq n^\rho p_1^{\rho - 1} \). For this we bound

\[
\frac{a + b}{n^\rho} = \frac{p_2^{-1+\alpha} - p_2^\alpha}{(p_2^{-1+\alpha} - 1)p_1^\alpha + (1 - p_2^\alpha)p_1^{-1+\alpha}} \leq \left( \frac{(p_1 - p_2) \log 1/p_1}{(1 - p_2) \log 1/p_2} \right)^\rho \left( \frac{(1 - p_2) \log p_1/p_2}{p_1 - p_2} \log 1/p_2 \right) = \exp \left( D \left( \frac{1}{p_1 - 1} \right) \right).
\]

Here \( D(r\|x) = r \log \frac{r}{x} + (1 - r) \log \frac{1 - x}{1 - r} \) is the Kullback-Leibler divergence. The two inequalities are proven in the Appendix as lemma 1 and lemma 3. The first bound comes from maximizing over \( \alpha \in [0, 1] \), so in principle we might be able to do better if \( \kappa = \log n/\log 1/p_2 \) is close to an integer. The second bound is harder, but the realization that the left hand side can be written on the form of a divergence helps a lot, since those have known properties we can exploit. The bound is tight up to a factor 2, so no significant improvement is possible.

Bringing it all together the expected query time is equal to the number of tables we have to query plus the number of far points to inspect. Using equation (??) we have:

\[
|a| + |b| + n(|a|p_2^{|\kappa|} + |b|p_2^{|\kappa|}) \leq (a + b + 1) + (ap_2^{1-\alpha} + bp_2^\alpha + 1) = 2(a + b) + 2 \leq 2n^\rho/p_1^{\rho - 1} + 2.
\]

Similarly, recall that to succeed the data structure must be able to find a near point when one exists. Recalling the previous computations and equation (??) we have:

\[
1 - (1 - p_1^{|\kappa|})^{|a|}(1 - p_1^{|\kappa|})^{|b|} \geq 1 - \exp(-ap_1^{|\kappa|} - bp_1^{|\kappa|})(1 - p_1^{|\kappa|})^{-1} = 1 - \exp(-1) \cdot 2 \geq 0.26.
\]

Finally we can boost the success probability from 26% to 99% by repeating the entire data-structure 16 times.
4 Conclusion

We have shown that using LSH tables with \(\left\lfloor \frac{\log n}{\log 1/p_2} \right\rfloor\) repetitions as well as with \(\left\lceil \frac{\log n}{\log 1/p_2} \right\rceil\) can, when done carefully, reduce the query time of LSH from \(n^\rho/p_1\) to \(n^\rho/p_1^{1-\rho}\), thereby guaranteeing sub-linear time.

From our experiments we have shown this to be an important practical as well as theoretical problem, and may be part of an explanation to why LSH sometimes underperforms its promises.

It remains an open problem whether it is possible to completely remove this multiplicative dependency on \(p_1^{-1}\) which is curiously not needed when \(\log 1/p_2\) divides \(\log n\).

References


A Proofs of Inequalities

Lemma 1. Let \( p_1, p_2 \in (0, 1) \) and \( \rho = \log \frac{1/p_1}{\log 1/p_2} \). Then for all \( \alpha \in [0, 1] \) we have

\[
f(\alpha) = \frac{p_2^{-1+\alpha} - p_1^{-\alpha}}{p_2^{1+\alpha} - 1} \leq \frac{(p_1 - p_2) \log 1/p_1}{(1 - p_1) \log p_1/p_2} \left( \frac{1 - p_2}{p_2 \log 1/p_2} \right)^\alpha.
\]

Proof. We first show that \( f(\alpha) \) is log-concave, which implies it is maximized at the unique \( \alpha \) such that \( f'(\alpha) = 0 \). Log-concavity follows easily by noting

\[
\frac{d^2 \log f(\alpha)}{d\alpha^2} = -\frac{(1 - p_1)(p_1 - p_2)p_2^{1+\alpha}(\log \frac{1}{p_2})^2}{((1 - p_1)p_2 + p_2^2(p_1 - p_2))^2} \leq 0.
\]

Meanwhile

\[
\frac{df(\alpha)}{d\alpha} = \frac{p_1(1 - p_2)(p_2/p_1)^\alpha}{((1 - p_1)p_2 + p_2^2(p_1 - p_2))^2} \left[ (p_1 - p_2)p_2^2 \log \frac{1}{p_1} - (1 - p_1)p_2 \log \frac{p_2}{p_1} \right],
\]

which implies \( f(\alpha) \) is maximized in

\[
\alpha = \log \frac{(1 - p_1)p_2 \log \frac{p_2}{p_1}}{(p_1 - p_2) \log \frac{1}{p_1} / \log p_2}.
\]

Plugging into \( f \) yields the lemma. Note that \( f(\alpha) \) is not regularly concave as \( p_1 \) and \( p_2 \) gets small enough. Hence the use of log-concavity is necessary.
Next, we state a useful inequality, which is needed for the last proof.

**Lemma 2.** Let \( p, r \in (0, 1) \), then

\[
1 - \frac{1 - p}{r} \leq p^{1/r} \leq \frac{pr}{1 - p(1 - r)}.
\]

**Proof.** We have \( \frac{d}{dp} p^{1/r} = p^{1/r} (pr)^{-2} (1 - r) \), so \( p^{1/r} \) is convex as a function of \( p \). Since \( 1 - \frac{1 - p}{r} \) is its tangent (at \( p = 1 \)) we get the first inequality.

For the second inequality, define \( g(p) = p^{1/r} / \frac{pr}{1 - p(1 - r)} \). Then \( g(1) = 1 \) and \( g(p) \) is non-decreasing, since

\[
g'(p) = p^{1/r} (pr)^{-2} (1 - p)(1 - r) \geq 0.
\]

This shows that for \( p \leq 1 \) we have \( p^{1/r} / \frac{pr}{1 - p(1 - r)} \leq 1 \), which is what we wanted to prove.

Finally we show the following bound, which implies the inequality \( \exp(D(p \parallel 1/p-1 \parallel 1/p^2-1)) \leq p^{1/p-1} \) in the theorem:

**Lemma 3.** Let \( p, r \in (0, 1) \) and let \( x = \frac{1}{p} - \frac{1}{pr} - 1 \), then

\[
D(r \parallel x) \leq r \log \frac{r}{x} \leq (1 - r) \log \frac{1}{p},
\]

where \( D(r \parallel x) = r \log \frac{r}{x} + (1 - r) \log \frac{1 - r}{1 - x} \).

**Proof.** Using the upper bound of lemma 2 it follows directly that \( x \in (0, r) \). This suffices to show the first inequality of (2), since for \( x \leq r \) we have \( \frac{1 - r}{1 - x} \leq 1 \) and so the second term of \( D(r \parallel x) \) is non-positive.

For the second inequality, we note that it is equivalent after manipulations to \( x \geq r p^{1/r - 1} \). Plugging in \( x \) and after more simple manipulations, that’s equivalent in the range to \( p^{1/r} \geq 1 - \frac{1 - r}{r} \), which is lemma 2. This finishes the proof of lemma 3.

Note that it is somewhat surprising that the last argument in the proof of lemma 3 works, since if we had plugged the lower bound from lemma 2 directly into the problem we would have had

\[
r \log \frac{r}{x} \leq r \log \frac{pr}{p + r - 1},
\]

which is much weaker than what we prove, and is not even defined for \( p + r < 1 \).