# Approximate Nearest Neighbors with Las Vegas Guarantees

Anonymous

December 15, 2016

**Abstract**

Most of the past work on approximate nearest neighbor problem is focused around the Monte Carlo algorithms that return the correct result only with high probability. This report (written as a reading project for the MIT's Advanced Algorithms class) is a survey of the Las Vegas algorithms for this problem as described in [Pag16]. We perform a detailed analysis on the efficiency of these algorithms and prove comparable expected time bounds to the classical methods, namely the seminal LSH construction of Indyk and Motwani [G+99]. We conclude that these algorithms can avoid the problem of false negatives at little or no cost in efficiency.

## 1 Introduction

Following problem is of significant importance to several areas of computer science including data mining and pattern recognition: given a set $S$ of $n$ points in $d$-dimensional space, construct a data structure that finds the point in $S$ that is closest to a query point $y$. This problem is called *nearest neighbor search* and it is known that exact solutions to this problem would disprove the strong exponential time hypothesis. Hence, instead of the exact version we will focus on the *approximate* nearest neighbor search.

As we have seen in class, there are many efficient data structures for the nearest neighbor search in low dimensions, especially in computational geometry. However, these data structures do not generalize well to the high dimensional cases since their query and preprocessing times increase exponentially with dimensionality $d$. Arguably, data structures that come closest to overcoming the high-dimensionality problem are based on locality-sensitive hashing (LSH).

In the following sections, we first explain the classical LSH methods [G+99] and their performance guarantees. Then we consider a novel construction of locality-sensitive hash families that is guaranteed to have hash collisions for every pair of vectors within a given radius $r$. Later, we generalize this construction to different cases to prove comparable expected time bounds for our algorithms.

## 2 Background

**Hamming Space.** In this paper we will work on nearest neighbor search in Hamming space, where data set and queries consist of binary vectors of the form $x, y \in \{0,1\}^d$, where $d$ is the dimensionality of the data. Length of a vector in this metric will simply be the number of non-zero bits: $||x|| = |\{i \in \{1,...,d\} \mid x_i = 1\}|$, and distance between two vectors is defined as the number of locations they differ: $||x - y|| = |\{i \in \{1,...,d\} \mid x_i \neq y_i\}|$. We also define bit-wise conjuction and disjunction operators which will be taken component-wise: $x \wedge y$ and $x \vee y$. In time bound analysis, we will assume that these unary and binary operations can be done in constant time, which is actually the case if vectors fit machine words.

**Definition.** A hashing family $\mathcal{H}$ is *r-covering* if for all possible pairs of vectors with distance at most $r$ there exists a hash in the family that maps these vectors to the same bucket. Formally:

$$||x - y|| \leq r \Rightarrow \exists h \in \mathcal{H} : h(x) = h(y)$$

Hash functions we will use throughout the paper are simply bit masking operations. In other words, given a vector $a$ from the Hamming Space, hashing function corresponding to that vector will be:

$$h(x) = x \wedge a$$

This can also be seen as the projection of vector $x$ on $a$.

**Important remark.** For simplicity, we will use $h$ to refer both the hash function and the projection vector $a$. Therefore a hashing function $h$ can also be written as:

$$h(x) = x \wedge h$$

**Problem Statement.** We want to perform nearest neighbor search in $d$-dimensional Hamming Space. Given:

- $S$: $n$ $d$-dimensional vectors from Hamming Space
- $r$: search radius
- $c$: approximation factor
- $y$: $d$-dimensional queries from Hamming Space

we want to find $c$-approximate nearest neighbor of query $y$ within radius $r$.

**Definition.** $c$-approximate nearest neighbor search within radius $r$ is defined as follows: Given a query $y$, if there exists an $x \in S$ with $||x - y|| \leq r$, algorithm must return an $x' \in S$ with $||x' - y|| \leq cr$.

Note that the problem we are tackling is actually a variant of the approximate nearest neighbor search where search radius $r$ is known before query time. However, the algorithms that we are going to present can be generalized, i.e. to the case of finding an $x'$ such that $||x' - y|| \leq c||x^* - y||$ where $x^*$ is the nearest neighbor of query $y$. This can be done by spanning radii from 1 to $d$ in $\mathcal{O}(\log(n)\log(d))$ steps (i.e. $r_i = \lceil (1 + 1/\log(n))^i \rceil$) and doing binary search to find distance of nearest neighbor. This adds a logarithmic factor to the query time and decreases the approximation factor but this can be compensated by adjusting $c$. In short, solving this version of the problem gives comparably efficient algorithms for the general case.

# 3 Classical LSH

In this section, we succinctly describe the classical LSH [G+99] and its performance guarantees. The main algorithm can be formulated as follows:

- We pick $m$ uniformly random hash functions $h^{(i)}$ with norm $k$, i.e. we randomly select $m$ vectors from the set $\{h \in \{0,1\}^d : ||h|| = k\}$
- In preprocessing time, we construct $m$ hash tables: For each $h^{(i)}$, we construct the corresponding hash table $T^{(i)}$ by computing the $h^{(i)}(x)$ for each $x$ in our set $S$.
- In query time, given a query $y$: For each hash function $h^{(i)}$ compute $h^{(i)}(y)$ and retrieve all points from colliding bucket in $T^{(i)}$ until:
  - Either all the collisions have been retrieved.
  - Or total number of points that are retrieved exceeds $m$.

  Then we return a point from the retrieved set that is at most $cr$ far from the query, if there is any.

Clearly, this algorithm answers any query in time $\mathcal{O}(m)$. There are two reasons that this algorithm might not return the correct answer: $(i)$ It uses only a subset of collisions to answer a query $(ii)$ It uses uniformly random hash functions and in some sense there is always a chance that all of the hash functions are bad. However, it turns out that by setting $m = \mathcal{O}(n^{1/c})$ and picking an appropriate $k$, one can minimize the expected number of bad collisions and guarantee that the returned result is correct with high probability. Overall query runtime is $\mathcal{O}(n^{1/c})$.

Classical LSH algorithm is a *Monte Carlo* algorithm, since it returns the correct result only with high probability. Also it picks the sequence of hash functions independently, which inherently implies that we can only hope for high probability bounds. In the following sections, we will present a different algorithm and a different construction for the hash family that will allow us to eliminate the false negatives with little or no efficiency loss. Resulting algorithm will be a *Las Vegas* algorithm that always returns the correct answer, but will only have expected time bounds.

# 4 Algorithm

We will tackle the approximate nearest neighbor search problem by finding a hashing family $\mathcal{H}$ that satisfies the following conditions: First, we want to hash close vectors to the same bucket. In other words, if distance between two vectors is at most $r$, there must exist an $h \in \mathcal{H}$ such that $h$ maps these two vectors to the same value. Secondly, we want the probability that distant vectors is mapped to the same bucket to be low. Particularly, for each hash function $h$ in our family and for any pairs of vectors $x$ and $y$, we want the probability that $h(x) = h(y)$ to be low when $x$ and $y$ are distant vectors. One can formulate these two conditions as follows:

- If $\|x - y\| \leq r$ then there exists an $h \in \mathcal{H}$ for which $h(x) = h(y)$ ($r$-covering).
- For each $h \in \mathcal{H}$, $\Pr[h(x) = h(y)] \leq 2^{-\|x-y\|}$.

**Algorithm.** Assuming that we found an $\mathcal{H}$ that satisfies these, our algorithm will be as follows: First, we will create a data structure to store hashes of vectors in our database in pre-computation time. We will go through all vectors in our database and for each hash function we calculate the hash and put the vector to corresponding hash bucket. Hash functions will have separate sets of buckets so that we don't get collision between different hash functions.

After constructing the data structure, we will process a query by going through each hash function, apply hash to query to find its corresponding bucket, and calculate the distance between the query and each vector in that bucket. Algorithm terminates as soon as it finds an approximate neighbor. Figure 1 gives a simple pseudocode for this algorithm.

```
function BuildDataStructure(S, r)
  D = ∅
  for x ∈ S do
    for h ∈ H(r) do
      D[h, h(x)] := D[h, h(x)] ∪ {x}
  return D
```

```
function NearestNeighbor(D, y, r)
  for h ∈ H do
    for x ∈ D[h, h(y)] do
      if ||x − y|| ≤ cr then
        return x
  return null
```

Figure 1: Pseudocode for constructing (left) and querying (right) our data structure.

Since our hashing family $\mathcal{H}$ will be $r$-covering, if there is an $x \in S$ with $\|x-y\| \leq r$, $x$ will be mapped to the same value as $y$ under some $h \in \mathcal{H}$. Therefore in that case, our algorithm is guaranteed to return a vector $x'$ with $\|x' - y\| \leq cr$, making it a *Las Vegas*-type algorithm. We will bound the running time in expectation instead of in worst case, and there are two important values we need to consider when we analyze the running time:

**Size of our hashing family.** The algorithm goes through functions in the hash family until it finds a suitable answer, and we might need to go through all of them if there's no such answer. Therefore size of the hashing family introduces a lower bound to running time: $\mathcal{O}(|\mathcal{H}|)$

**Number of bad collisions.** These are the collisions of query $y$ with vectors in $S$ that are not $c$-approximate nearest neighbors. More precisely, we define:

$$\mathbb{E}[\#\text{bad collisions}] = \mathbb{E}[|\{(x,h) \in S \times \mathcal{H} \mid h(x) = h(y) \wedge \|x - y\| > cr\}|]$$

If we consider the case where we have many bad collisions but no approximate neighbor, our algorithm will need to go through all these collisions without terminating at any step. Therefore running time is also lower bounded by $\mathbb{E}[\#\text{bad collisions}]$.

Moreover, running time of our algorithm is upper bounded by $|\mathcal{H}| + \mathbb{E}[\#\text{bad collisions}]$. Therefore to get an efficient algorithm, we need to find a hash family that balance these two values $|\mathcal{H}|$ and $\mathbb{E}[\#\text{bad collisions}]$. Following sections of this paper will be devoted to finding efficient hash families for different search distances.

# 5 A Simple Construction

We need a way to construct the hashing family $\mathcal{H}$ so that it satisfies the two conditions of $c$-approximate nearest neighbor search we've stated above. In order to generate vectors $h$, we will use a binary matrix $M$ of size $d \times (r+1)$ where each element is randomly selected from $\{0,1\}$. Given a binary vector $v$ of dimension $(r+1)$, we will compute the matrix multiplication $Mv$ and take the modulo 2 of every component, which will give us a binary vector of dimension $d$. If we represent this mapping as $v \to h^v$, formally:

$$h^v = Mv \pmod 2$$

where modulo is taken component-wise.

We will apply this procedure to all possible vectors of dimension $r+1$, except zero vector, and create a set of $2^{r+1} - 1$ binary vectors of dimension $d$. We will call the hashing family constructed with this procedure $\mathcal{H}(r)$. Formally:

$$\mathcal{H}(r) = \{h^v \mid v \in \{0,1\}^{r+1} \backslash \{\mathbf{0}\}\}$$

**Claim.** Given a uniformly randomized binary matrix $M$, $\mathcal{H}(r)$ satisfies the following two conditions.

- If $\|x - y\| \leq r$ then there exists a $h \in \mathcal{H}(r)$ for which $h(x) = h(y)$.
- For each $h \in \mathcal{H}(r)$, $\Pr[h(x) = h(y)] \leq 2^{-\|x-y\|}$.

**Proof**. We consider each of the conditions separately:

**1.** To hash $x$ and $y$ to the same bucket, $h$ projection should ignore the bits that $x$ and $y$ differ, which means: $x_i \neq y_i \Rightarrow h_i = 0$. If $\|x - y\| \leq r$, there can be at most $r$ indices that they differ. Recalling $Mv = 0 \pmod 2$ for some $v$, we need $M_i v = 0 \pmod 2$ at these indices. Therefore we have a linear system with $r+1$ variables and at most $r$ equations, and there exists a non-zero $v$ that satisfies this system. Since we use all possible $v$'s in our hashing family construction, our family satisfies the constraint.

**2.** In order to get same hash from two different vectors, vector $h$ that we use for hashing should have value 0 at indices where vectors differ: $x_i \neq y_i \Rightarrow h_i = 0$. Since elements of $M$ are uniformly random and vector $v$ we use to generate $h$ is non-zero, components of $h$ will be randomly distributed: $\Pr[h_i = 0] = 1/2$. Therefore probability of getting $h_i = 0$ for all the indices where $x_i \neq y_i$ will be:

$$\Pr[h(x) = h(y)] = 2^{-\|x-y\|}$$

□

**Special Case.** We will first consider the special case $cr = \log(n)$, where this simple construction already gives a good running time. Previously we have shown that in order to get a good running time bound we need to upper bound $|\mathcal{H}| + \mathbb{E}[\#\text{bad collisions}]$.

Consider the following example problem. Suppose we have a set $S$ of $n = 2^{30}$ vectors from $\{0,1\}^{128}$ and wish to search for a vector at distance at most $r = 10$ from a query vector $y$, with

approximation factor $c = 3$. Note that $cr = 30 = \log(n)$ in this case. Number of hash functions in our 10-covering family is $2^{11} - 1 = 2047$. To compute the expected number of bad collisions i.e. collisions with distance at least 31, we will consider the second condition: second condition gives that expected number of bad collisions per hash function is at most $2^{-31} \times n = 1/2$. Hence, if we sum this up for each $h \in \mathcal{H}$, total number of bad collisions is at most 1024. Therefore any query can be answered in $\sim 3000$ operations.

Indeed, as there are $2^{r+1}-1$ functions in our hash family, time to calculate all hash tables will be $\mathcal{O}(2^r)$. Expected number of bad collisions we will find in these hashes can be calculated as follows. We found that probability of two vectors $x$ and $y$ to collide on a hash $h$ to be $\Pr[h(x) = h(y)] = 2^{-\|x-y\|}$. So given a query $y$, if a vector $x$ from our data set is not a $c$-approximate vector (i.e. $\|x - y\| > cr = \log(n)$) it will collide with $y$ on a hash function with probability less than $2^{-cr}$. Since there are $2^{r+1} - 1$ hash functions in our family and there are $n$ vectors in our data set, from linearity of expectations expected number of collisions that we need to deal with is at most: $(2^{r+1} - 1)2^{-cr}n = (2^{r+1} - 1)2^{-\log(n)}n = 2^{r+1} - 1 = \mathcal{O}(2^r)$. Hence both calculating hash buckets and processing collisions in these buckets are time bounded by $\mathcal{O}(2^r)$, expected query time of our algorithm is also $\mathcal{O}(2^r) = \mathcal{O}(n^{1/c})$. Note that this matches to the time bound of the classical LSH.

# 6 Generalizing to Large Distances

Above simple construction does not work for large distances. The reason is, when search radius $r$ is large, our simple construction leads to a large number of hash functions, i.e. $|\mathcal{H}|$ becomes very large. For instance, in above example, when search radius becomes 100 instead of 10, number of hash functions in our hash family becomes $2^{101} - 1$ instead of $2^{10} - 1$, which is certainly not feasible for our runtime. To solve this issue, we will construct a new hashing family $\mathcal{H}^L$ that is still $r$-covering, but has a lot less number of hash functions. Since we are reducing the number of hash functions, the number of bad collisions will increase, but hopefully our construction will balance these two values out.

**Intuition for our construction.** We will use the idea of partitioning [AGK06]. Consider partitioning of indices $\{1, 2, \ldots, d\}$ into $b$ equi-sized sets $P_1, P_2, \ldots, P_b$. For instance, a simple partitioning would be

$$\{1, 2, \ldots, u\} \cup \{(u+1), \ldots, 2u\} \cup \cdots \cup \{(d-u+1), \ldots, d\}$$

where $u$ is equal to $\lfloor d/b \rfloor$. For any two vectors $x$ and $y$ in $d$-dimensional Hamming Space with distance $r$, consider the set of *mismatching* indices i.e.

$$S_{\text{mismatch}} = \{i \in \{1, 2, \ldots, d\} \mid x_i \neq y_i\}$$

Since number of mismatching indices is $r$, from pigeonhole principle, there exists a partition set $P_i$ with at most $\lfloor r/b \rfloor$ mismatching indices. Here we aim to come up with an $r$-covering family with less number of hash functions, and therefore, a natural idea might be to use our simple hashing family $\mathcal{H}(r')$ with $r' = \lfloor r/b \rfloor$. We can project $\mathcal{H}(r')$ to each of these partitions to get an $r$-covering family $\mathcal{H}^L$. Here projection of a hash function to a partition is defined as follows: We keep the bits of hash at indices in partition set, while setting other bits that are not picked by partition to zero. Formally, on a hash function $h \in \{0, 1\}^d$ and on a partition set $P$,

$$\text{Projection}(h, P) = h \wedge u$$

where $u$ is defined as the $d$-dimensional Hamming vector with $u_i = 1 \Leftrightarrow i \in P$.

Moreover, we define projecting hash family $\mathcal{H}$ to a partition set $P$ simply as projecting each of the hash functions $h \in \mathcal{H}$ to $P$. That is,

$$\text{Projection}(\mathcal{H}, P) = \bigcup_{h \in \mathcal{H}} \text{Projection}(h, P)$$

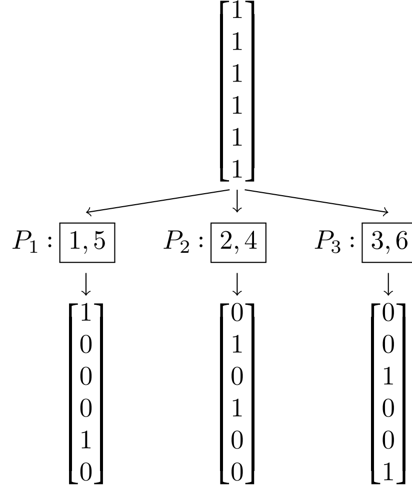


Figure 2: A sample projection for $d = 6$ and $b = 3$

The idea is to construct a hashing family $\mathcal{H}^L$ by projecting $\mathcal{H}(r')$ to each of the partitions. Hopefully, $\mathcal{H}^L$ will have a lot less number of hash functions, but will still remain $r$-covering. If we write this more formally,

$$\mathcal{H}^L = \bigcup_{i=1}^{b} \text{Projection}(\mathcal{H}(r'), P_i)$$

Clearly, number of hash functions in $\mathcal{H}^L$ is the number of hash functions in $\mathcal{H}(r')$ times the number of partitions. Therefore $|\mathcal{H}^L| = b|\mathcal{H}(r')| = b(2^{r'+1} - 1) = \mathcal{O}\left(b\, 2^{\lfloor r/b \rfloor}\right)$. Note that if $b$ is large, this number is a lot less than the size of our original $r$-covering family which was $2^{r+1} - 1$.

Now we can prove that this hashing family is $r$-covering, with the intuition that if two vectors mismatch at $r$ indices, there is a partition that has at most $r' = \lfloor r/b \rfloor$ mismatches. Therefore when we project $\mathcal{H}(r')$ to this partition, there exists a hash function in this projection that maps these two vectors to the same value.

**Claim.** For any $x, y \in \{0,1\}^d$ with $\|x - y\| \leq r$, there exists a hash function $h \in \mathcal{H}^L$ with $h(x) = h(y)$.

**Proof**. Consider the set of mismatching indices $S_{\text{mismatch}}$ for $x$ and $y$ as defined above. For each partition $P_i$, let $s_i$ denote the number of mismatching indices in $P_i$. That is,

$$s_i = |S_{\text{mismatch}} \cap P_i|$$

Clearly, $\sum s_i = \|x - y\| \leq r$, so from pigeonhole principle there exists a $j$ with $s_j \leq \lfloor r/b \rfloor = r'$. Now consider the corresponding hash family,

$$\mathcal{H}_j^L = \text{Projection}(\mathcal{H}(r'), P_j)$$

From our previous construction of $\mathcal{H}(r')$, we know that $\mathcal{H}(r')$ is $r'$-covering, i.e. for any $u, v$ with $\|u - v\| \leq r'$, there exists a hash function $h \in \mathcal{H}(r')$ with $h(u) = h(v)$. Observe that the projection operation does not change any bit from 0 to 1, and thus, any projection of $\mathcal{H}(r')$ remains $r'$-covering. Consequently, $\mathcal{H}_j^L$ is an $r'$-covering hash family.

For any hash function $h$ in $\mathcal{H}_j^L$, all the indices except the ones in the $P_j$ have value 0. Therefore, to prove that there exists an $h \in \mathcal{H}_j^L$ with $h(x) = h(y)$, it is enough to account only for the indices that are in $P_j$. Note that the number of mismatching indices in $P_j$ is at most $r'$. Therefore there exists a $h \in \mathcal{H}_j^L$ with $h(x) = h(y)$ since $\mathcal{H}_j^L$ is $r'$-covering,. Result follows.

□

Now, it remains to bound the number of bad collisions under hash family $\mathcal{H}^L$ to get a good expected runtime for our algorithm. However, it turns out that in this simplest form $\mathcal{H}^L$ leads to a large number of bad collisions even for the vectors that are very distant from each other. That is mainly because hash functions in $\mathcal{H}^L$ are very *sparse*, i.e. average number of nonzero indices per hash function is very low. To solve this problem, we will refine our idea of partitioning. Our refinement will aim to increase the average number of nonzero bits per hash function, while maintaining the $r$-covering property.

**Refinement.** In the above construction, each index appears only in a single partition set, and this leads to low number of indices per set. Since our hashing family was constructed by projecting $\mathcal{H}(r')$ to each of the partitions, projected hash functions had low number of nonzero indices. A natural idea to fix this is to put each index into multiple partition sets.

Let $q$ denote the number of partition sets each index appears in. For instance for $q = 2$, a possible partitioning would be,

$$\{1, 2, \ldots, 2u\} \cup \{(u+1), \ldots, 3u\} \cup \cdots \cup \{(d-u+1), \ldots, d, 1, \ldots, u\}$$

where $u$ is equal to $\lfloor d/b \rfloor$. Note that when compared to previous example partition, this one has 2 times more indices per partition set.

We can apply the pigeonhole principle along very similar lines. That is, if we distribute $q$ copies of $r$ *mismatching* indices across $b$ partition sets, there will always be a set with at most $r^* = \lfloor rq/b \rfloor$ mismatches. Now to get an $r$-covering family, we can project $\mathcal{H}(r^*)$ to each of the partitions. Note that this time size of our hashing family, $b2^{r^*+1}$, is larger than the previous family when $q > 1$.

In order to avoid any adversary input sequence and get a good expected runtime, we need to randomize our partition. This randomization will be done as follows: Consider $b$ initially empty buckets. For each index, we pick $q$ random buckets and add this index to these buckets. Notice that after this process, number of indices per partition might not be the same, but it turns out that it is enough for us to have them close in *expected* sense.

To formulate all these mathematically, let $\text{Subsets}(b, q)$ denote the subsets of $\{1, \ldots, b\}$ of size $q$. Consider the following randomized function,

$$s : \{1, \ldots, d\} \to \text{Subsets}(b, q)$$

As we explained above, $s(i)$ represents the partitions which index $i$ belongs. Therefore $s$ will correspond the following partition: For any $i \in \{1, \ldots, b\}$ $P_i$ will consist of $j \in \{1, \ldots, d\}$ with $i \in s(j)$. We construct our hashing family $\mathcal{H}^L$ by projecting $\mathcal{H}(r^*)$ to each of the partitions. That is

$$\mathcal{H}^L = \bigcup_{i=1}^{b} \text{Projection}(\mathcal{H}(r^*), P_i)$$

**Claim.** For any $x, y \in \{0,1\}^d$ with $||x-y|| \le r$, there exists a hash function $h \in \mathcal{H}^L$ with $h(x) = h(y)$.

**Proof**. Very similar to the previous proof, except now we use pigeonhole principle for $q$ copies of $r$ mismatching indices across $b$ partitions, instead of just 1 copy. □

Now that we control the average number of 1s in our hash family with the parameter $q$, we can proceed to get a bound on number of bad collisions. We will first prove the following lemma:

**Lemma.** For any $x, y \in \{0,1\}^d$, and for any $h \in \mathcal{H}^L$:

$$\Pr[h(x) = h(y)] = (1 - q/2b)^{||x-y||}$$

**Proof**. In order for any hash function $h$ to map $x$ and $y$ to the same value, it must have zeros on the mismatching indices of $x$ and $y$. In other words, for any $i \in \{1, \dots, d\}$ with $x_i \neq y_i$, we need to have $h_i = 0$. There are two cases that might lead to $h_i = 0$.

In the first case, corresponding partition set $P$ for $h$ might not have index $i$. In that case index $i$ is basically set to 0 after projection onto $P$. Since our partition is randomized, probability that index $i$ was not in the partition is simply $1 - q/b$.

In the second case, index $i$ might belong to the corresponding partition $P$, but was already zero before projection. Put differently, let $h' \in \mathcal{H}(r^*)$ be the hash function that is used to generate $h$ via projection. It might be the case that $h'_i$ was already zero. In the previous section we proved that the probability that $h'_i = 0$ for any $h' \in \mathcal{H}(r^*)$ is $1/2$. Therefore, probability of the second case is simply $q/2b$. Combining these two cases, probability that $h_i = 0$ for any index $i$ is,

$$1 - q/b + q/2b = (1 - q/2b)$$

Notice that we constructed our hash family in such a way that for any $h \in \mathcal{H}^L$ and for any $i, j \in \{1, \dots, d\}$, $h_i$ and $h_j$ are independent. That is because, both the partition function $s$ and the hash family $\mathcal{H}(r^*)$ are randomized in an index independent fashion. Therefore,

$$\Pr[h(x) = h(y)] = \prod_{x_i \neq y_i} \Pr[h_i = 0] = (1 - q/2b)^{\|x-y\|}$$

□

**Corollary.** For any $x, y \in \{0,1\}^d$, expected number of hash functions that map them to same value is

$$\mathbb{E}\left[|\{h \in \mathcal{H}^L \mid h(x) = h(y)\}|\right] = (1 - q/2b)^{\|x-y\|}\, b\,(2^{r^*+1} - 1)$$

This follows from summing up the expectation of collision for every hash function in $\mathcal{H}^L$ using the linearity of expectation.

Now for any query $y$, we are ready to count the number of bad collisions, i.e. $(x, h) \in S \times \mathcal{H}^L$ pairs with $h(x) = h(y)$ and $\|x - y\| > cr$. For any $x \in S$, expected number of hash functions that map $x$ and $y$ to the same value is

$$\begin{aligned} \mathbb{E}\left[|\{h \in \mathcal{H}^L \mid h(x) = h(y)\}|\right] &= (1 - q/2b)^{\|x-y\|} b\,(2^{r^*+1} - 1) \\ &\leq (1 - q/2b)^{cr+1} b\, 2^{(rq/b)+1} \end{aligned} \tag{1}$$

If we sum this up for all n vectors in $S$ our bound becomes:

$$\mathbb{E}[\#\text{bad collisions}] \leq n\,(1 - q/2b)^{cr+1}\, b\, 2^{(rq/b)+1}$$

Now we need to pick the parameters $b$ and $q$ in such a way that $\mathbb{E}[\#\text{bad collisions}]$ and $|\mathcal{H}^L|$ are balanced. Since $|\mathcal{H}^L| \cong b\, 2^{(rq/b)+1}$, it is natural to pick $b$ and $q$ such that the number $n\,(1 - q/2b)^{cr+1}$ is close to 1. One such possible choice would be,

$$\boxed{b = r \quad \text{and} \quad q = 2\lceil \ln(n)/c \rceil}$$

Then if we follow the math,

$$|\mathcal{H}^L| < b\, 2^{rq/b+1} \leq r\, 2^{2\ln(n)/c+3} = 8r\, n^{\ln(4)/c}$$

and,

$$\mathbb{E}[\#\text{bad collisions}] < n\,(1 - q/(2r))^{cr}\, r\, 2^{q+1} < n \exp(-qc/2))\, r\, 2^{q+1} < r\, 2^{q+1} < 8r\, n^{\ln(4)/c},$$

Therefore our runtime is bounded by $|\mathcal{H}^L| + \mathbb{E}[\#\text{bad collisions}] = \mathcal{O}(r\, n^{\ln(4)/c})$. Note that this differs from Classical LSH by a factor $\ln(4)$ in the exponent, while avoiding the problem of false negatives.

# 7 Generalizing to Small Distances

Our simple construction fails again for small distances since this time number of bad collisions turns out to be really large. For instance, in the above example, when search radius $r = 1$, expected number of bad collisions,

$$\mathbb{E}\left[\#\text{bad collisions}\right] = n2^{r+1-\|x-y\|} = n2^{r-cr} = n/4 = 2^{28}$$

which does not look promising.

**Intuition for our construction.** To decrease the number of bad collisions, we will try to increase the average number of 1's in our hash functions. We need to achieve an $r$-covering family with dense hash functions, so we might need to increase the size of our hash family. Again, it comes down to finding a *sweet spot* that balances the size of hashing family and the number of bad collisions. Our main idea is as follows:

In our simple construction, we used to take a vector $v \in \{0,1\}^{r+1}$, multiply it with the generator matrix $M \in \{0,1\}^{d\times(r+1)}$ to get a $d$-dimensional hash function $h$. Therefore for any $i \in \{1,\dots,d\}$, $h_i = M_i\, v \pmod 2$. To increase the probability that $h[i] = 1$, a natural idea would be to increase the number of generator matrices. Let $M^1,\dots,M^t$ be the new generator matrices. We will set $h_i$ to 1 if at least one of the generator matrices $M^j$ leads to $M_i^j\, v = 1 \pmod 2$. Mathematically this can be formulated as,

$$h_i = \bigvee_j M_i^j\, v \pmod 2$$

For any vector $x \in \{0,1\}^d$ with $\|x\| = r$, we should be able to find a hash function in our family with $h(x) = 0$. Hence, we need to be able to find a hash function $h$ that has zeros at each of the $r$ nonzero indices of $x$. Then this creates $t$ equations per nonzero index, leading to $rt$ equations in total. So we need to have $rt + 1$ free variables to guarantee a non-trivial solution to this equation system. Therefore, it is natural to pick our generator functions from $\{0,1\}^{d\times(rt+1)}$

To formalize this intuition, let $M^1,\dots,M^t$ be randomized matrices from $\{0,1\}^{d\times(rt+1)}$, and we define a function $h^v : \{0,1\}^{(rt+1)} \to \{0,1\}^d$ that satisfies

$$h_i^v = \bigvee_j M_i^j\, v \pmod 2$$

for each $i = \{1,\dots,d\}$. Then we construct our hash family $\mathcal{H}^S$ by generating a hash function for each $\{0,1\}^{(rt+1)}\backslash\mathbf{0}$ using function $h^v$. That is

$$\mathcal{H}^S = \{h^v \mid v \in \{0,1\}^{rt+1}\backslash\{\mathbf{0}\}\}$$

**Claim.** For any $x,y \in \{0,1\}^d$ with $\|x-y\| \le r$, there exists a hash function $h \in \mathcal{H}^S$ with $h(x) = h(y)$.

**Proof**. To map $x$ and $y$ to the same value, $h$ function must *cover* the mismatching indices of $x$ and $y$, i.e. $h_i$ must be zero whenever $x_i \neq y_i$. In order for $h_i$ to be 0, all of the corresponding $t$ generator matrices must give $M_i^j\, v \pmod 2 = 0$, since we are using OR operator. This leads to $t$ linear equations in modulo 2. If we do this for each mismatching index, we get a linear equation system in modulo 2 with at most $rt$ equations. Since $v$ is a $rt+1$ dimensional vector, there exists a non-zero $v$ that satisfies all these equations. Claim follows. □

Now we have an $r$-covering hash family, and we will try to balance the number of bad collisions and size of this hash family, using the introduced parameter $t$. To analyze the number of bad collisions, we prove the following lemma.

**Lemma.** For any $x,y \in \{0,1\}^d$, and for any $h \in \mathcal{H}^S$

$$\Pr\left[h(x) = h(y)\right] = 2^{-t\|x-y\|}$$

**Proof**. Again, to *cover* mismatching indices of $x$ and $y$, $h_i$ must be 0, whenever $x_i \neq y_i$. Also, In order to get $h_i = 0$, all of the corresponding $t$ generator matrices must give $M_i^j\, v \pmod 2 = 0$. Since we created each generator matrix independently, probability that $h_i$ becomes 0 is $2^{-t}$. Moreover, bits of $h$ are independent variables. Therefore we get:

$$\Pr\left[h(x) = h(y)\right] = \prod_{x_i \neq y_i} \Pr\left[h_i = 0\right] = 2^{-t\|x-y\|}$$

□

**Corollary.** For any $x, y \in \{0,1\}^d$, expected number of hash functions that map them to same value is

$$\mathbb{E}\left[\left|\{h \in \mathcal{H}^S \mid h(x) = h(y)\}\right|\right] = 2^{-t\|x-y\|}(2^{rt+1} - 1)$$

This follows from summing up the expectation of collision for every hash function in $\mathcal{H}^S$ using the linearity of expectation.

If we sum this up for all $x \in S$, our bound becomes,

$$\mathbb{E}\left[\#\text{bad collisions}\right] \leq n2^{-t\|x-y\|}(2^{rt+1} - 1)$$

Now we need to pick the parameter $t$ so that $\mathbb{E}\left[\#\text{bad collisions}\right]$ and $|\mathcal{H}^S|$ are balanced. Therefore we need to get $n2^{-t\|x-y\|}$ as close to 1 as possible. A natural choice for $t$ would be $\lceil \log n/cr \rceil$. For this $t$, if one follows through the math, we get

$$|\mathcal{H}^S| < 2^{tr+1} \leq 2^{(\log(n)/(cr)+1)r+1} = 2^{r+1}n^{1/c} \quad .$$

and

$$\mathbb{E}\left[\#\text{bad collisions}\right] < n2^{-tcr}2^{tr+1} \leq 2^{tr+1} \leq 2^{r+1}n^{1/c} \quad .$$

Therefore runtime for our algorithm becomes $|\mathcal{H}^S| + \mathbb{E}\left[\#\text{bad collisions}\right] = \mathcal{O}(2^r n^{1/c})$.

# 8 Conclusion

In this report, we surveyed elementary Las Vegas algorithms for the approximate nearest neighbor search problem. We started with a simple construction that worked for a special case and we generalized it for large and small search radii. Presented algorithms avoid the possibility of false negatives while achieving comparable runtime performance to the standard Monte Carlo methods.

# References


[G+99] Aristides Gionis, Piotr Indyk, Rajeev Motwani, et al. "Similarity search in high dimensions via hashing". In: *VLDB*. Vol. 99. 6. 1999, pp. 518–529.

[AGK06] Arvind Arasu, Venkatesh Ganti, and Raghav Kaushik. "Efficient exact set-similarity joins". In: *Proceedings of the 32nd international conference on Very large data bases*. VLDB Endowment. 2006, pp. 918–929.

[Pag16] Rasmus Pagh. "Locality-sensitive hashing without false negatives". In: *Proceedings of the Twenty-Seventh Annual ACM-SIAM Symposium on Discrete Algorithms*. SIAM. 2016, pp. 1–9.