E^2LSH 0.1
User Manual

Alexandr Andoni
Piotr Indyk

January 11, 2006

Contents

Chapter 1

What is E^{2}LSH?

Short answer:

E^{2}LSH (*Exact Euclidean LSH*) is a package that provides a randomized solution for the high-dimensional near neighbor problem in the Euclidean space l_2. After preprocessing the data set, E^{2}LSH answers queries, typically in sublinear time, with each near neighbor being reported with a certain probability. E^{2}LSH is based on the Locality Sensitive Hashing (LSH) scheme described in [2].

Long answer:

The *R-near neighbor* problem is defined as follows. Given a set of points $\mathcal{P} \subset \mathbb{R}^d$ and a radius $R > 0$, construct a data structure that answers the following queries: for a query point q, find all points $p \in \mathcal{P}$ such that $||q - p||_2 \leq R$, where $||q - p||_2$ is the Euclidean distance between q and p. E^{2}LSH solves a randomized version of this problem, which we call a $(R, 1 - \delta)$-*near neighbor* problem. In this case, each point p satisfying $||q - p||_2 \leq R$ has to be reported with a probability at least $1 - \delta$ (thus, δ is the probability that a near neighbor p is not reported).

E^{2}LSH can be also used to solve the *nearest neighbor* problem, where, given the query q, the data structure is required the report the point in $\mathcal{P}$ that is closest to q. This can be done by creating several R-near neighbor data structures, for $R = R_1, R_2, \ldots R_t$, where R_t should be greater than the maximum distance from any query point to its nearest neighbor. The nearest neighbor can be then recovered by querying the data structures in the increasing order of the radiae, stopping whenever the first point is found.

E^{2}LSH is based on locality-sensitive hashing (LSH) scheme, as described in [2]. The original locality-sensitive hashing scheme solves the *approximate* version of the R-near neighbor problem, called a (R, c)-near neighbor problem. In that formulation, it is sufficient to report *any* point within the distance of at most cR from the query q, if there is a point in $\mathcal{P}$ at distance at most R from q (with a constant probability). For the approximate formulation, the LSH scheme achieves a time of $O(n^\rho)$, where $\rho < 1/c$.

To solve the $(R, 1 - \delta)$ formulation, E^{2}LSH uses the basic LSH scheme to get all near neighbors (including the approximate ones), and then drops the approximate near neighbors. Thus, the running time of E^{2}LSH depends on the data set $\mathcal{P}$. In particular, E^{2}LSH is slower for "bad" data sets, e.g., when for a query q, there are many points from $\mathcal{P}$ clustered right outside the ball of radius R centered at q (i.e., when there are many approximate near neighbors).

E^{2}LSH is also different from the original LSH scheme in that E^{2}LSH empirically estimates the optimal parameters for the data structure, as opposed to using theoretical formulas. This is because theoretical formulas are geared towards the worst case point sets, and therefore they are less adequate for the real data

sets. E^{2}LSH computes the parameters as a function of the data set $\mathcal{P}$ and optimizes them to minimize the actual running time of query on the host system.

The outline of the remaining part of the manual is as follows. Chapter 2 describes the package and how to use it to solve the near neighbor problem. In Chapter 3, we describe the LSH algorithm used to solve the $(R, 1-\delta)$ problem formulation, as well as optimizations for decreasing running time and memory usage. Chapter 4 discusses the structure of the code of the E^{2}LSH: the main data types and modules, as well as the main functions for constructing, parametrizing and querying the data structure. Finally, Chapter 5 contains FAQ.

Chapter 2

E^{2}LSH Usage

In this chapter, we describe how to use our E^{2}LSH package. First, we show how to compile and use the main script of the package; and then we describe two additional scripts to use when one wants to modify or set manually the parameters of the R-NN data structure. Next, we elaborate on memory usage of E^{2}LSH and how to control it. Finally, we present some additional useful utilities, as well as the formats of the data files of the package.

All the scripts and programs should be located in the `bin` directory relative to the E^{2}LSH package root directory.

2.1 Compilation

To compile the E^{2}LSH package, it is sufficient to run `make` from E^{2}LSH's root directory. It is also possible to compile by running the script `bin/compile` from E^{2}LSH's root directory.

2.2 Main Usage

The main script of E^{2}LSH is `bin/lsh`. It is invoked as follows:

```
bin/lsh R data_set_file query_set_file [successProbability]
```

The script takes, as its parameters, the name `data_set_file` of the file with the data set points and the file `query_set_file` with the query points (the format of the files is described in Section 2.6). Given these files, E^{2}LSH constructs the optimized R-NN data structure, and then runs the queries on the constructed data structure. The values `R` and `successProbability` specify the parameters R and $1-\delta$ of the $(R, 1-\delta)$-near neighbor problem that E^{2}LSH solves. Note that `successProbability` is an optional parameter; if not supplied, E^{2}LSH uses a default value of 0.9 (90% success probability).

2.3 Manual setting of the parameters of the R-NN data structure

As described in Chapter 3, the LSH data structure needs three parameters, denoted by k, L and m (where $m \approx \sqrt{L}$). However, the script `bin/lsh` computes those parameters automatically in the first stage of data structure construction. The parameters are chosen so that to optimize the estimated query time. However,

since these parameters are only estimates, there are no guarantees that these parameters are optimal for particular query points. Therefore, manual setting of these parameters may occasionally provide better query times.

There are two additional scripts that give the possibility of manual setting of the parameters: `bin/lsh_computeParams` and `bin/lsh_fromParams`. The first script, `bin/lsh_computeParams`, computes the optimal parameters for the R-NN data structure from the given data set points and outputs the parameters to the standard output. The usage of `bin/lsh_computeParams` is as follows:

```
bin/lsh_computeParams R data_set_file {query_set_file | .} [successProbability]
```

The script outputs an estimation of the optimal parameters of the R-NN data structure for the data set points in `data_set_file`. If one specifies the query set file as the third parameter, then we use several of the points from the query set for optimizing data structure parameters; if a dot (`.`) is specified, then we use instead several points from the data set for the same purpose. The output is written to standard output and may be redirected to a file (for a later use) as follows:

```
bin/lsh_computeParams R data_set_file query_set_file > data_set_parameters_file
```

See section 2.6 for description of the format of the parameter file.

The second script, `bin/lsh_fromParams`, takes as an input a file containing the parameters for the R-NN data structure (besides the files with the data set points and the query points). The script constructs the data structure given these parameters and runs queries on the constructed data structure. The usage of `bin/lsh_fromParams` is the following:

```
bin/lsh_fromParams data_set_file query_set_file data_set_params_file
```

The file `data_set_params_file` must be of the same format as the output of `bin/lsh_computeParams`. Note that one does not need to specify the success probability and R since these values are embedded in the file `data_set_params_file`.

Thus, running the following two lines

```
bin/lsh_computeParams R data_set_file query_set_file > data_set_parameters_file
bin/lsh_fromParams data_set_file query_set_file data_set_params_file
```

is equivalent to running

```
bin/lsh R data_set_file query_set_file
```

To modify manually the parameters for the R-NN data structure, one should modify the file `data_set_params_file` before running the script `bin/lsh_fromParams`.

For ease of use, the script `bin/lsh` also outputs the parameters it used for the constructed R-near neighbor data structure. These parameters are written to the file `data_set_file.params`, where `data_set_file` is the name of the supplied data set file.

2.4 Memory

E^{2}LSH uses a considerable amount of memory: for bigger data sets, the optimal parameters for the R-NN data structure might require an amount of memory which is greater than the available physical memory. Therefore, when choosing the optimal parameters, E^{2}LSH takes into consideration the upper bound on memory it can use. Note that if E^{2}LSH starts to swap, the performance decreases by a few orders of magnitude.

The user thus can specify the maximal amount of memory that E^{2}LSH can use (which should be at most the amount of physical memory available on the system before executing E^{2}LSH). This upper bound is specified in the file `bin/mem` in bytes. If this file does not exist, the main scripts will create one with an estimation of the available physical memory.

2.5 Additional utilities

`bin/exact` is an utility that computes the exact R-near neighbors (using the simple linear scan algorithm). Its usage is the same as that of `bin/lsh`:

```
bin/exact R data_set_file query_set_file
```

`bin/compareOutputs` is an utility for checking the correctness of the output generated by the E^{2}LSH package (by `bin/lsh` or `bin/lsh_fromParams`). The usage is the following:

```
bin/compareOutputs correct_output LSH_output
```

`correct_output` is the output from `bin/exact` and `LSH_output` is the output from `bin/lsh` (or `bin/lsh_fromParams`) for the same `R`, `data_set_file`, and `query_set_file`.

For each query point from `query_set_file`, `bin/compareOutputs` outputs whether E^{2}LSH's output is a subset of the output of `bin/exact`: in this case `OK=1`; if E^{2}LSH outputs a point that is not a R-near neighbor or outputs some points more that once, then `OK=0`. `bin/compareOutputs` also outputs for each query point the fraction of the R-near neighbors that E^{2}LSH manages to find. Finally, `query_set_file` outputs the overall statistics: the `and` of the `OK`s for all queries, as well as the ratio of the number of R-near neighbors found by E^{2}LSHto their actual number (as determined by `bin/exact`).

2.6 File formats

2.6.1 Data set file and query set file

Both the file for data set and for the query set (`data_set_file` and `query_set_file`) are text files with the following format:

```
coordinate_1_of_point_1 coordinate_2_of_point_1 ... coordinate_D_of_point_1
coordinate_1_of_point_2 coordinate_2_of_point_2 ... coordinate_D_of_point_2
...
coordinate_1_of_point_N coordinate_2_of_point_N ... coordinate_D_of_point_N
```

Each entry `coordinate_j_of_point_i` is a real number.

2.6.2 Output file format

The output of E^{2}LSH is twofold. The main results are directed to standard output (`cout`). The output stream has the following format:

```
Query point i : found x NNs. They are:
......
......
Total time for R-NN query: y
```

Additional information is reported to standard error (`cerr`).

2.6.3 File with the parameters for the R-NN data structure

The file with the parameters for the R-NN data structure is the output of the `bin/lsh_computeParams` and the command-line parameter `data_set_params_file` for the script `bin/lsh_fromParams`. It specifies the estimation of the optimal parameters for a spefic data set and for a specific machine. Below is an example of such a file:

```
1
R
0.53
Success probability
0.9
Dimension
784
R^2
0.280899972
Use <u> functions
1
k
20
m [# independent tuples of LSH functions]
35
L
595
W
4.000000000
T
9991
typeHT
3
```

All lines except the first one define the parameters in the following way (the first line is reserved for future use). Each odd line defines the value of a parameter (the preceding even line simply describes the name of the corresponding parameter). The parameters `R, Success Probability, Dimension, k, m, L, W` are the parameters that appear in the algorithm description (note that `Success Probability,`

`k, m, L` are interrelated values as described in 3.5.1). The parameter `R^2` is equal to R^2. The parameter `T` is reserved for specifying how many points to look through before the query algorithm stops, but this parameter is not implemented yet (and therefore is set to n).

2.6.4 The remainder of the parameter file

Note: understanding of the description below requires familarity with the algorithm of Chapter 3.

The parameter "`Use <u> functions`" signals whether to use the original g functions (each of the L functions g_i is a k-tuple of LSH functions; all kL LSH functions are independent) or whether to use g's that are not totally independent (as described in the section 3.4.1). If the value of the parameter is 0, original g's are used and $L = m$; if the value is 1, the modified g's are used and $L = m \cdot (m-1)/2$.

The parameter `typeHT` defines the type of the hash table used for storing the buckets containing data set points. Currently, values of 0 and 3 are supported, but we suggest to use the value 3. (Refering to the hash table types described in the section 3.6, the value 0 corresponds to the linked-list version of the hash tables, and the value 3 – to the hash tables with hybrid storage array Y.)

Chapter 3

Algorithm description

In this chapter, we describe first the general locality-sensitive hashing algorithm (as in [2] but with slight modifications). Next, we gradually add more details of the algorithm as well as the optimizations in our implementation.

3.1 Notations

We use l_p^d to denote the d-dimensional real space $\mathbb{R}^d$ under the l_p norm. For any point $v \in \mathbb{R}^d$, the notation $||v||_p$ represents the l_p norm of the vector v, that is

$$||v||_p = (\sum_{i=1}^{d} v_i^p)^{1/p}$$

In particular $||v|| = ||v||_2$ is the Euclidean norm.

Let the data set $\mathcal{P}$ be a finite subset of $\mathbb{R}^d$, and let $n = |\mathcal{P}|$. A point q will usually stand for the query point; the query point is any point from $\mathbb{R}^d$. Points v, u will usually stand for some points in the data set $\mathcal{P}$.

The *ball* of radius r centered at v is denoted by $B(v, r)$. For a query point q, we call v an R-near neighbor (or simply a near neighbor) if $v \in B(q, R)$.

3.2 Generic locality-sensitive hashing scheme

To solve the R-NN problem, we use the technique of Locality Sensitive Hashing or LSH [4, 3]. For a domain S of points, the LSH family is defined as:

Definition 1 *A family $\mathcal{H} = \{h : S \to U\}$ is called* locality-sensitive*, if for any q, the function $p(t) = \Pr_{\mathcal{H}}[h(q) = h(v) : ||q - v|| = t]$ is strictly decreasing in t. That is, the probability of collision of points q and v is decreasing with the distance between them.*

Thus, if we consider any points q, v, u, with $v \in B(q, R)$ and $u \notin B(q, R)$, then we have that $p(||q - v||) > p(||q - u||)$. Intuitively we could hash the points from $\mathcal{P}$ into some domain U, and then at the query time compute the hash of q and consider only the points with which q collides.

However, to achieve the desired running time, we need to amplify the gap between the collision probabilities for the range $[0, R]$ (where the R-near neighbors lie) and the range (R, ∞). For this purpose we concatenate several functions $h \in \mathcal{H}$. In particular, for k specified later, define a function family $\mathcal{G} = \{g : S \to U^k\}$

such that $g(v) = (h_1(v), \ldots, h_k(v))$, where $h_i \in \mathcal{H}$. For an integer L, the algorithm chooses L functions $g_1, \ldots, g_L$ from $\mathcal{G}$, independently and uniformly at random. During preprocessing, the algorithm stores each $v \in \mathcal{P}$ (input point set) in buckets $g_j(v)$, for all $j = 1, \ldots, L$. Since the total number of buckets may be large, the algorithm retains only the non-empty buckets by resorting to hashing (explained later).

To process a query q, the algorithm searches all buckets $g_1(q), \ldots, g_L(q)$. For each point v found in a bucket, the algorithm computes the distance from q to v, and reports the point v iff $||q - v|| \leq R$ (v is a R-near neighbor).

We will describe later how we choose the parameters k and L, and what time/memory bounds they give. Next, we present our choice for the LSH family $\mathcal{H}$.

3.3 LSH scheme for l_p norm

In this section, we will present the LSH family $\mathcal{H}$ that we use in our implementation. This LSH family is based on p-stable distributions, that works for all $p \in (0, 2]$. We use exactly the same LSH family as suggested by [2].

It should be noted that the implementation as described in Chapter 2 works only for the l_2 (Euclidean) norm.

Since we consider points in l_p^d, without loss of generality we can assume that $R = 1$, since otherwise, we can scale down all the points by a factor of R.

3.3.1 p-stable distributions

Stable distributions [5] are defined as limits of normalized sums of independent identically distributed variables (an alternate definition follows). The most well-known example of a stable distribution is Gaussian (or normal) distribution. However, the class is much wider; for example, it includes heavy-tailed distributions.
Stable Distribution: A distribution $\mathcal{D}$ over $\mathbb{R}$ is called *p-stable*, if there exists $p \geq 0$ such that for any n real numbers $v_1 \ldots v_n$ and i.i.d. variables $X_1 \ldots X_n$ with distribution $\mathcal{D}$, the random variable $\sum_i v_i X_i$ has the same distribution as the variable $(\sum_i |v_i|^p)^{1/p} X$, where X is a random variable with distribution $\mathcal{D}$.

It is known [5] that stable distributions exist for any $p \in (0, 2]$. In particular:

- a *Cauchy distribution* $\mathcal{D}_C$, defined by the density function $c(x) = \frac{1}{\pi} \frac{1}{1+x^2}$, is 1-stable
- a *Gaussian (normal) distribution* $\mathcal{D}_G$, defined by the density function $g(x) = \frac{1}{\sqrt{2\pi}} e^{-x^2/2}$, is 2-stable

From a practical point of view, note that despite the lack of closed form density and distribution functions, it is known [1] that one can generate p-stable random variables essentially from two independent variables distributed uniformly over $[0, 1]$.

3.3.2 Hash family

The LSH scheme proposed in [2] uses p-stable distributions as follows: compute the dot products $(a.v)$ to assign a hash value to each vector v. Formally, each hash function $h_{a,b}(v) : \mathbb{R}^d \to \mathbb{Z}$ maps a d dimensional vector v onto the set of integers. Each hash function in the family is indexed by a choice of random a and b where a is a d dimensional vector with entries chosen independently from a p-stable distribution and b is a real number chosen uniformly from the range $[0, w]$. For a fixed a, b the hash function $h_{a,b}$ is given by $h_{a,b}(v) = \lfloor \frac{a \cdot v + b}{w} \rfloor$

The intuition behind the hash functions is as follows. The dot product $a.v$ projects each vector to the real line. It follows from p-stability that for two vectors (v_1, v_2) the distance between their projections $(a.v_1 - a.v_2)$ is distributed as $||v_1 - v_2||_p X$ where X is a p-stable distribution. If one "chops" the real line into equi-width segments of appropriate size w and assign hash values to vectors based on which segment they project onto, then it is intuitively clear that this hash function will be locality preserving in the sense described above.

One can compute the probability that two vectors v_1, v_2 collide under a hash function drawn uniformly at random from this family. Let $f_p(t)$ denote the probability density function of the **absolute value** of the p-stable distribution. We will drop the subscript p whenever it is clear from the context. For the two vectors v_1, v_2, let $c = ||v_1 - v_2||_p$. For a random vector a whose entries are drawn from a p-stable distribution, $a.v_1 - a.v_2$ is distributed as cX where X is a random variable drawn from a p-stable distribution. Since b is drawn uniformly from $[0, w]$ it is easy to see that

$$p(c) = Pr_{a,b}[h_{a,b}(v_1) = h_{a,b}(v_2)] = \int_0^w \frac{1}{c} f_p(\frac{t}{c})(1 - \frac{t}{w})dt$$

For a fixed parameter w the probability of collision $p(c)$ decreases monotonically with $c = ||v_1 - v_2||_p$, satisfying the definition 1.

The optimal value for w depends on the data set and the query point, but it was suggested in [2] that $w = 4$ provides good results, and, therefore, we currently use the value $w = 4$ in our implementation.

3.4 Parameters for the LSH scheme

In our case, Hash Budge is given, where (Ki * Li) is a fixed number

To use LSH, we need to specify the parameters k and L. From the problem formulation, specifically from the requirement that a near neighbor is reported with a probability at least $1 - \delta$, we can derive a necessary condition on k and L. Consider a query point q and a near neighbor $v \in B(q, R)$. Let $p_1 = p(1) = p(R)$. Then, $Pr_{g \in \mathcal{G}}[g(q) = g(v)] \geq p_1^k$. Thus, q and v fail to collide for all L functions g_i with probability at most $(1 - p_1^k)^L$. Requiring that the point q collides with v on some function g_i is equivalent to inequation $1 - (1 - p_1^k)^L \geq 1 - \delta$, which implies that $L \geq \frac{\log \delta}{\log(1 - p_1^k)}$. Since there are no more conditions on k and L (other than minimizing the running time), we choose $L = \lceil \frac{\log \delta}{\log(1 - p_1^k)} \rceil$ (the running time is increasing with L).

We need to consider constraints on k and l

The value k is chosen as a function of the data set to minimize the running time of a query. Note that this is different from the LSH scheme in [2], where k is chosen as a function of the approximation factor.

For a fixed value of k and $L = L(k)$, we can decompose the query time into two terms. The first term is $T_g = O(dkL)$ for computing the L functions g_i for the query point q as well as retrieving the buckets $g_i(q)$ from hash tables. The second term is $T_c = O(d \cdot \#collisions)$ for computing the distance to all points encountered in the retrieved buckets. $\#collisions$ is the number of points encountered in the buckets $g_1(q), \ldots g_L(q)$; the expected value of $\#collisions$ is $E[\#collisions] = L \cdot \sum_{v \in \mathcal{P}} p^k(\|q - v\|)$.

Intuitively, T_g increases as a function of k, while T_c decreases as a function of k. The latter is due to the fact that higher values of k magnify the gap between the collision probabilities of "close" and "far" points, which (for proper values of L) decreases the probability of collision of far points. Thus, typically there exists an optimal value of k that minimizes the sum $T_g + T_c$ (for a given query point q). Note that there might be different optimal k's for different query points, therefore the goal would be optimize the mean query time for all query points. We discuss more on the optimization procedure in the section 3.5.1

3.4.1 Faster computation of hash functions

In this section, we describe a slight modification to the LSH scheme that enables a considerable reduction of the time T_g, the time necessary for computing the functions g_i.

In the original LSH scheme, we choose L functions $g_i = (h_1^{(i)}, \ldots, h_k^{(i)})$, where each function $h_j^{(i)}$ is chosen uniformly at random from the LSH family $\mathcal{H}$. For a given point q, we need $O(d)$ time to compute a function $h_j^{(i)}(q)$, and $O(dkL)$ time to compute all functions $g_1(q), \ldots, g_L(q)$.

To reduce the time for computing functions g_i for the query q, we reuse some of the functions $h_j^{(i)}$ (in this case, g_i are not totally independent). Specifically, in addition to functions g_i, define functions u_i in the following manner. Suppose k is even and m is a fixed constant. Then, for $i = 1 \ldots m$, let $u_i = (h_1^{(i)}, \ldots, h_{k/2}^{(i)})$, where each $h_j^{(i)}$ is drawn uniformly at random from the family $\mathcal{H}$. Thus u_i are vectors each of $k/2$ functions drawn uniformly at random from the LSH family $\mathcal{H}$.

Now, define functions g_i as $g_i = (u_a, u_b)$, where $1 \leq a < b \leq m$. Note that we obtain $L = m(m-1)/2$ functions g_i.

Since the functions g_i are interdependent, we need to derive a different expression for the probability that the algorithm reports a point that is within the distance R from the query point (a R-near neighbor). With new functions g_i, this probability is greater than or equal to $1 - \left(1 - p_1^{k/2}\right)^m - m \cdot p_1^{k/2} \cdot \left(1 - p_1^{k/2}\right)^{m-1}$. To require a success probability of at least $1 - \delta$, we restrict m to be such that $\left(1 - p_1^{k/2}\right)^m + m \cdot p_1^{k/2} \cdot \left(1 - p_1^{k/2}\right)^{m-1} \leq \delta$. This inequation yields a slightly higher value for $L = m(m-1)/2$ than in the case when functions g_i are independent, but L is still $O\left(\frac{\log 1/\delta}{p_1^k}\right)$. The time for computing the g_i functions for a query point q is reduced to $T_g = O(dkm) = O(dk\sqrt{L})$ since we need only to compute m fuctions u_i. The expression for the time T_c, the time to compute the distance to points in buckets $g_1(q), \ldots, g_L(q)$, remains unchanged due to the linearity of expectation.

3.5 Implementation details

3.5.1 R-NN data structure construction

For constructing the R-NN data structure, the algorithm first computes the parameters k, m, L for the data structure. The parameters k, m, L are computed as a function of the data set $\mathcal{P}$, the radius R, and the probability $1 - \delta$ as outlined in the section 3.4 and 3.4.1. For a value of k, the parameter m is chosen to be the smallest natural number satisfying $\left(1 - p_1^{k/2}\right)^m + m \cdot p_1^{k/2} \cdot \left(1 - p_1^{k/2}\right)^{m-1} \leq \delta$; L is set to $m(m-1)/2$. Thus, in what follows, we consider m and L as functions of k, and the question remains only of how to choose k.

For choosing the value k, the algorithm experimentally estimates the times T_g and T_c as a function of k. Remember that the time T_c is dependent on the query point q, and, therefore, for estimating T_c we need to use a set S of sample query points (the estimation of T_c is then the mean of the times T_c for points from S). The sample set S is chosen to be a set of several points chosen at random from the query set. (The package also provides the option of choosing S to be a subset of the data set $\mathcal{P}$.)

Note that to estimate T_g and T_c precisely, we need to know the constants hidded by the $O(\cdot)$ notation in the expressions for T_g and T_c. To compute these constants, the implementation constructs a sample data structure and runs several queries on that sample data structure, measuring the actual times T_g and T_c.

Concluding, k is chosen such that $\tilde{T}_c + T_g$ is minimal (while the data structure space requirement is within the memory bounds), where $\tilde{T}_c$ is the mean of the times T_c for all points in the sample query set S: $\tilde{T}_c = \frac{\sum_{q \in S} T_c(q)}{|S|}$.

Once the parameters k, m, L are computed, the algorithm constructs the R-NN data structure containing the points from $\mathcal{P}$.

3.5.2 Bucket hashing

Recall that the domain of each function g_i is too large to store all possible buckets explicitly, and only non-empty buckets are stored. To this end, for each point v, the buckets $g_1(v), \dots g_L(v)$ are hashed using the universal hash functions. For each function g_i, $i = 1 \dots L$, there is a hash table H_i containing the buckets $\{g_i(v) \mid v \in \mathcal{P}\}$. For this purpose, there are 2 associated hash functions $h_1 : \mathbb{Z}^k \to \{0, \dots, tableSize - 1\}$ and $h_2 : \mathbb{Z}^k \to \{0, \dots, C\}$ (each g_i maps to $\mathbb{Z}^k$). The function h_1 has the role of the usual hash function in an universal hashing scheme. The second hash function identifies the buckets in chains.

The collisions within each bucket are resolved by chaining. When storing a bucket $g(v) = (x_1, \dots x_k)$ in its chain, instead of storing the entire vector $(x_1, \dots x_k)$ for bucket identification, we store only $h_2(x_1, \dots x_k)$. Thus, a bucket $g_i(v) = (x_1, \dots x_k)$ has only the following associated information stored in its chain: the identifier $h_2(x_1, \dots, x_k)$, and the points in the bucket, which are $g_i^{-1}(x_1, \dots x_k) \cap \mathcal{P}$.

The reasons for using the second hash function h_2 instead of storing the value $g_i(v) = (x_1, \dots x_k)$ are twofold. Firstly, by using a fingerprint $h_2(x_1, \dots x_k)$, we decrease the amount of memory for bucket identification from $O(k)$ to $O(1)$. Secondly, with the fingerprint it is faster to look up a bucket in the hash table. The domain of the function h_2 is chosen big enough to ensure with a high probability that any two different buckets in the same chain have different h_2 values.

All L hash tables use the same primary hash function h_1 (used to dermine the index in the hash table) and the same secondary hash function h_2. These two hash functions have the form

$$h_1(a_1, a_2, \dots, a_k) = \left(\left(\sum_{i=1}^{k} r_i' a_i \right) \bmod prime \right) \bmod tableSize,$$

and

$$h_2(a_1, a_2, \dots, a_k) = \left(\sum_{i=1}^{k} r_i'' a_i \right) \bmod prime,$$

where r_i' and r_i'' are random integers, $tableSize$ is the size of the hash tables, and $prime$ is a prime number.

In the current implementation, $tableSize = |\mathcal{P}|$, a_i are represented by 32-bit integers, and $prime$ is equal to $2^{32} - 5$. This value of prime allows fast hash function computation without using modulo operations. Specifically, consider computing $h_2(a_1)$ for $k = 1$. We have that:

$$h_2(a_1) = (r_1'' a_1) \bmod \left(2^{32} - 5\right) = \left(low\left[r_1'' a_1\right] + 5 \cdot high\left[r_1'' a_1\right]\right) \bmod \left(2^{32} - 5\right)$$

where $low[r_1'' a_1]$ are the low-order 32 bits of $r_1'' a_1$ (a 64-bit number), and $high[r_1'' a_1]$ are the high-order 32 bits of $r_1'' a_1$. If we choose r_i'' from the range $[1, \dots 2^{29}]$, we will always have that $\alpha = low\left[r_1'' a_1\right] + 5 \cdot high\left[r_1'' a_1\right] < 2 \cdot \left(2^{32} - 5\right)$. This means that

$$h_2(a_1) = \begin{cases} \alpha & \text{, if } \alpha < 2^{32} - 5 \\ \alpha - \left(2^{32} - 5\right) & \text{, if } \alpha \geq 2^{32} - 5 \end{cases}$$

For $k > 1$, we compute progressively the sum $\left(\sum_{i=1}^{k} r_i'' a_i\right)$ mod $prime$ keeping always the partial sum modulo $(2^{32} - 5)$ using the same principle as the one above. Note that the range of the function h_2 thus becomes $\{1, \ldots 2^{32} - 6\}$.

3.5.3 Additional optimizations

We use the following additional optimizations.

- **Precomputation of $g_i(q)$, $h_1(g_i(q))$, and $h_2(g_i(q))$.** To answer a query, we first need to compute $g_i(q)$. As mentioned in the section 3.4.1, since $g_i = (u_a, u_b)$, we need only to compute $k/2$-tuples $u_a(q)$, $a = 1, \ldots m$. Further, for searching for buckets $g_i(q)$ in hash tables, we need in fact only $h_1(g_i(q))$ and $h_2(g_i(q))$. Precomputing h_1 and h_2 in the straight-forward way would take $O(Lk)$ time. However, since g_i are of the form (u_a, u_b), we can reduce this time in the following way. Note that each function h_j, $j \in \{1, 2\}$, is of the form $h_j(x_1, \ldots x_k) = ((\sum_{t=1}^{k} r_i^j x_i) \text{mod } A_j) \text{mod } B_j$. Therefore, $h_j(x_1, \ldots x_k)$ may be computed as $((\sum_{t=1}^{k/2} r_t^j x_t + \sum_{t=k/2+1}^{k} r_t^j x_t) \text{mod } A_j) \text{mod } B_j$. If we denote $\mathrm{r}^{\mathrm{j}}_{\mathrm{left}} = (r_1^j, \ldots r_{k/2}^j)$, and $\mathrm{r}^{\mathrm{j}}_{\mathrm{right}} = (r_{k/2+1}^j, \ldots r_k^j)$, then we have that $h^j(g_i(q)) = ((\mathrm{r}^{\mathrm{j}}_{\mathrm{left}} \cdot u_a(v) + \mathrm{r}^{\mathrm{j}}_{\mathrm{right}} \cdot u_b(v)) \text{mod } A_j) \text{mod } B_j$. Thus, it suffices to precompute only $\mathrm{r}^{\mathrm{j}}_{\mathrm{side}} \cdot u_a(v)$, where $j \in \{1, 2\}$, $side \in \{left, right\}$, $a \in \{1, \ldots m\}$, which takes $O(km)$ time.

- **Skipping repeated points.** In the basic query algorithm, a point $v \in \mathcal{P}$ might be encountered more than once. Specifically, a point $v \in \mathcal{P}$ might appear in more than one of the buckets $g_1(q), \ldots g_L(q)$. Since there is no need to compute the distance to any point $v \in \mathcal{P}$ more than once, we keep track of the points for which we already computed the distance $||q - v||$, and not compute the distance a second time. For this purpose, for each particular query, we keep a vector e_i, such that $e_i = 1$ if we already encountered the point $v_i \in \mathcal{P}$ in an earlier bucket (and computed the distance $||q - v||$), and $e_i = 0$ otherwise. Thus, the first time we encounter a point v in the buckets, we compute the distance to it, which takes $O(d)$ time; for all subsequent times we encounter v we spend only $O(1)$ time (for checking the vector e_i).

  Note that with this optimization, the estimation of T_c is not accurate anymore. This is because T_c is the time for computing the distance to points encountered in the buckets $g_1(q), \ldots g_L(q)$. Taking into the consideration only the distance computations (i.e., assuming we spend no time on subsequent copies of a point), the new expression for T_c is

$$E[T_c] = d \cdot \sum_{v \in \mathcal{P}} \left(1 - \left(1 - p(||q - v||)^{k/2}\right)^m - m \cdot p(||q - v||)^{k/2} \cdot \left(1 - p(||q - v||)^{k/2}\right)^{m-1}\right)$$

3.6 Memory

The R-NN data structure described above requires $O(nL)$ memory (for each function g_i, we store the n points from $\mathcal{P}$). Since, L increases with k, the memory requirement is big for big data set or for moderate data set for which optimal time is achived with higher values of k. Therefore, an upper limit on memory imposes an upper limit on k.

Because the memory requirement is big, the constant in front of $O(nL)$ is very important. In our current implementation, with the best variant of the hash tables, this constant is 12 bytes. Note that it is the structure and layout of the L hash tables that plays the substantial role in the memory usage.

Below we show two variants of the layout of the hash tables that we deployed. We assume that 1) the number of points is $n \leq 2^{20}$; 2) each pointer is 4 bytes long; 3) $tableSize = n$ for each hash table.

One of the most straightforward layouts of a hash table H_i is the following. For each index l of the hash table, we store a pointer to a singly-linked list of buckets in the chain l. For each bucket, we store its value $h_2(\cdot)$, and a pointer to a singly-linked list of points in the bucket. The memory requirement per hash table is $4 \cdot tableSize + 8 \cdot \#buckets + 8 \cdot n \leq 20n$, yielding a constant of 20.

To reduce this constant to 12 bytes, we do the following. Firstly, we index all points in $\mathcal{P}$, such that we can refer to points by index (this index is constant across all hash tables). Refering to a point thus takes only 20 bits (and not 32 as in the case of a pointer). Consider now a hash table H_i. For this hash table, we deploy a table Y of 32-bit unsigned integers that store all buckets (with values $h_2(\cdot)$) and points in the buckets (thus, Y is a hybrid storage table since it stores both buckets' and points' description). The table has a length of $\#buckets + n$ and is used as follows. In the hash table H_i, at index l, we store the pointer to some index e_l of Y; e_l is the start of the description of the chain l. A chain is stored as follows: $h_2(\cdot)$ value of the first bucket in chain (at position e_l in Y) followed by the indeces of the points in this bucket (positions $e_l + 1, \ldots e_l + n_1$); $h_2(\cdot)$ value of the second bucket in the chain (position $e_l + n_1 + 1$) followed by the indeces of the points in this second bucket (positions $e_l + n_1 + 2, \ldots e_l + n_1 + 1 + n_2$); and so forth.

Note that we need also to store the number of buckets in each chain as well as the number of points in each bucket. Instead of storing the chain length, we store for each bucket a bit that says whether that bucket is the last one in the chain or not; this bit is one of the unused bits of the 4-byte integer storing the index of the first point in the corresponding bucket (i.e., if the $h_2(\cdot)$ value of the bucket is stored at position e in Y, then we use a high-order bit of the integer at position $e + 1$ in Y). For storing the length of the bucket, we use the remaining unused bits of the first point in the bucket. When the remaining bits are not enough (there are more than $2^{32-20-1} - 1 = 2^{11} - 1$ points in the bucket), we store a special value for the length (0), which means that there are more than $2^{11} - 1$ points in the bucket, and there are some additional points (that do not fit in the $2^{11} - 1$ integers alloted in Y after the $h_2(\cdot)$ value of the bucket). These additional points are also stored in Y but at a different position; their start index and number are stored in the unused bits of the remaining $2^{11} - 2$ points that follow the $h_2(\cdot)$ value of the bucket and the first point of the bucket (i.e., unused bits of the integers at positions $e + 2, \ldots e + 2^{11} - 1$).

3.7 Future possible optimizations

- **Parameter w of the LSH scheme.** In addition to optimizing k, the algorithm could optimize the parameter w to achieve the best query time. The function p depends on the parameter w, and, thus, both times T_c and T_g depend on w. Currently, we use a fixed value of 4, but the optimal value of w is a function of the data set $\mathcal{P}$ (and a sample query set).

- **Generalization of functions $g_i = (u_a, u_b)$.** In section 3.4.1 we presented a new approach to choosing functions g_i. Specifically, we choose functions $g_i = (u_a, u_b)$, $1 \leq a < b \leq m$, where each u_j, $j = 1 \ldots m$, is a $k/2$-tuple of random independent hash functions from the LSH family $\mathcal{H}$. In this way, we decreased the time to compute functions $g_i(q)$ for a query q from $O(dkL)$ to $O(dk\sqrt{L})$.

  This approach could be generalized to functions g that are t-tuples of functions u, where u are drawn indepently from $\mathcal{H}^{k/t}$, reducing in this way the asymptotic time for computing the functions g. We did not pursue this generalization since, even if L is still $O\left(\frac{\log 1/\delta}{p_1^k}\right)$, the constant hidden by $O(\cdot)$ notation for $t > 2$ would probably nullify the theoretical gain for the typical data sets.

Chapter 4

The E^{2}LSH Code

4.1 Code overview

The core of the E^{2}LSH code is divided into three main components:

- `LocalitySensitiveHashing.cpp` – contains the implementation of the main LSH-based R-near neighbor data strucure (except the hashing of the buckets). The main functionality is for constructing the data structure (given the parameters such as k, m, L), and for answering a query.
- `BucketHashing.cpp` – contains the implementation of the hash tables for the universal hashing of the buckets. The main functionality is for constructing hash tables, adding new buckets/points to it, and looking-up a bucket.
- `SelfTuning.cpp` – contains functions for computing the optimal parameters for the R-near neighbor data structure. Contains all the functions for estimating the times T_c, T_g (including the functions for estimating $\#collisions$).

Additional code making part of the core is contained in the following files:

- `Geometry.h` – contains the definition for a point (`PPointT` data type);
- `NearNeighbors.cpp`,`NearNeighbors.h` – contain the functions at the interface of the E^{2}LSH core (see a more detailed description below);
- `Random.cpp`, `Random.h` – contain the pseudo-random number generator;
- `BasicDefinitions.h` – contains the definitions of general-purpose types (such as, `IntT`, `RealT`) and macros (such as the macros for timing operations);
- `Utils.cpp`, `Utils.h` – contain some general purpose functions (e.g., copying vectors).

Another important part of the package is the file `LSHMain.cpp`, which is a sample code for using E^{2}LSH. `LSHMain.cpp` only reads the input files, parses the command line parameters, and calls the corresponding functions from the package.

The most important data structures are:

- `RNearNeighborStructureT` – the fundamental R-near neighbor data structure. This structure contains the parameters used to construct the structure, the description of the functions g, the index of the points in the structure, as well pointers to the L hash tables for storing the buckets. The structure is defined in `LocalitySensitiveHashing.h`.

- `UHashStructureT` – the structure defining a hash table used for hashing the buckets. Collisions are resolved using chaining as explained in sections 3.5.2 and 3.6. There are 2 main types of hash tables: `HT_LINKED_LIST` and `HT_HYBRID_CHAINS` (the field `typeHT` contains the type of the hash table). The type `HT_LINKED_LIST` corresponds to the linked-list version of the hash table, and `HT_HYBRID_CHAINS` – to the one with hybrid storage array Y (see section 3.6 for more details). Each hash table also contains pointers to the descriptions of the functions $h_1(\cdot)$ and $h_2(\cdot)$ used for the universal hashing. The structure is defined in `BucketHashing.h`.

- `RNNParametersT` – a struct containing the parameters necessary for constructing the `RNearNeighborStructureT` data structure. It is defined in `LocalitySensitiveHashing.h`.

- `PPoint` – a struct for storing a point from $\mathcal{P}$ or a query point. This structure contains the coordinates of the point, the square of the norm of the point, and an index, which can be used by the callee outside the E^{2}LSH code (such as `LSHMain.cpp`) for identifying the point (for example, it might by the index of the point in the set $\mathcal{P}$); this index is not used within the core of E^{2}LSH.

4.2 E^{2}LSH Interface

The following are the functions at the interface of E^{2}LSH core code. The first two functions are sufficient for constructing the R-NN data structure, and querying it afterwards; they are declared in the file `NearNeighbors.h`. The following two functions provide a separation of the estimation of the parameter (such as k, m, L) from the construction of the R-NN data structure itself.

1. To construct the R-NN data structure given as input $1-\delta$, R, d, and the data set $\mathcal{P}$, one can use the function

```
PRNearNeighborStructT
  initSelfTunedRNearNeighborWithDataSet(RealT thresholdR,
                                        RealT successProbability,
                                        Int32T nPoints,
                                        IntT dimension,
                                        PPointT *dataSet,
                                        IntT nSampleQueries,
                                        PPointT *sampleQueries)
```

   The function will estimate optimal parameters k, m, L, and will construct a R-NN data structure from this parameters.

   The parameters of the function are the input data of the algorithm: $R, 1-\delta, n, d$, and respectively $\mathcal{P}$. The parameter `sampleQueries` represents a set of sample query points – the function optimizes the parameters of the constructed data structure for the points specified in the set `sampleQueries`.

`sampleQueries` could be a sample of points from the actual query set or from the data set $\mathcal{P}$. `nSampleQueries` specifies the number of points in the set `sampleQueries`.

The return value of the function is the R-NN data structrure that is constructed. This function is declared in `NearNeighbors.h`.

2. For a query operation, one can use the function

```
Int32T getRNearNeighbors(PRNearNeighborStructT nnStruct,
                         PPointT queryPoint,
                         PPointT *(&result),
                         IntT &resultSize)
```

The parameters of the function have the following meaning:

- `nnStruct` – the R-NN data structure on which to perform the query;
- `queryPoint` – the query point;
- `result` – the array where the near neighbors are to be stored (if the size of this array is not sufficient for storing the near neighbors, the array is reallocated to fit all near neighhbors);
- `resultSize` – the size of the `result` array (if the `result` is resized, this value is changed accordingly);

The function `getRNearNeighbors` returns the number of near neighbors that were found. The function is declared in the file `NearNeighbors.h`.

3. For estimating the optimal parameters for a R-NN data structure, one can use the function

```
RNNParametersT computeOptimalParameters(RealT R,
                                        RealT successProbability,
                                        IntT nPoints,
                                        IntT dimension,
                                        PPointT *dataSet,
                                        IntT nSampleQueries,
                                        PPointT *sampleQueries)
```

The parameters of the function are the input data of the algorithm: R, $1-\delta$, n, d, and respectively $\mathcal{P}$. The parameter `sampleQueries` represents a set of sample query points – the function optimizes the parameters of the data structure for the points specified in the set `sampleQueries`. `sampleQueries` could be a sample of points from the actual query set or from the data set $\mathcal{P}$. `nSampleQueries` specifies the number of points in the set `sampleQueries`.

The return value is the structrure with optimal parameters. This function is declared in `SelfTuning.h`.

4. For constructing the R-NN data structure from the optimal parameters, one can use the function

```
PRNearNeighborStructT initLSH_WithDataSet(RNNParametersT algParameters,
                                          Int32T nPoints,
                                          PPointT *dataSet)
```

`algParameters` specify the parameters with which the R-NN data structure will be constructed.

The function returns the constructed data structure. The function is declared in `LocalitySensitiveHashing.h`.

Chapter 5

Frequent Anticipated Questions

In this section we give answers to some questions that the reader might ask when compiling and using the package.

1. **Q:** How to compile this thing ?

   A: Since you are reading this manual, you must have already gunzipped and untarred the original file. Now it suffices to type `make` in the main directory to compile the code.

2. **Q:** OK, it compiles. Now what ?

   A: You can run the program on a provided data set `mnist1k.dts` and query set `mnist1k.q`. They reside in the main directory. Simply type

   ```
   bin/lsh 0.6 mnist1k.dts mnist1k.q >o
   ```

   This will create an output file `o` containing the results of the search. To see if it worked, run the exact algorithm

   ```
   bin/exact 0.6 mnist1k.dts mnist1k.q >o.e
   ```

   and compare the outputs by running

   ```
   bin/compareOutputs o.e o
   ```

   You should receive an answer that looks like:

   ```
   Overall: OK = 1. NN_LSH/NN_Correct = 5/5=1.000
   ```

   This means that the run was "OK", and that the randomized LSH algorithm found 5 out of 5 (i.e., all) near neighbors within distance 0.6 from the query points. Note that, since the algorithm is randomized, your results could be different (e.g., 4 out of 5 near neighbors). However, if you are getting 0 out of 5, then probably something is wrong.

3. **Q:** I ran the code, but it is so slow!

   A: If the code is unusually slow, it might mean two things:

   - The code uses so much memory that the system starts swapping. This typically degrades the performance by 3 orders of magnitude, so should be definitely avoided. To fix it, specify the amount of available (not total) memory in the file `bin/mem`.
   - The search radius you specified is so large that many/most of the data points are reported as near neighbors. If this is what you want, then the code will not run much faster. In fact, you might be better off using linear scan instead, since it is simpler and has less overhead.

     Otherwise, try to adjust the search radius so that, on average, there are few near neighbors per query. You can use `bin/exact` for experiments, it is likely to be faster than LSH if you perform just a few queries.

Bibliography

[1] J. M. Chambers, C. L. Mallows, and B. W. Stuck. A method for simulating stable random variables. *J. Amer. Statist. Assoc.*, 71:340–344, 1976.

[2] M. Datar, N. Immorlica, P. Indyk, and V. Mirrokni. Locality-sensitive hashing scheme based on p-stable distributions. *DIMACS Workshop on Streaming Data Analysis and Mining*, 2003.

Original E2LSH paper

[3] A. Gionis, P. Indyk, and R. Motwani. Similarity search in high dimensions via hashing. *Proceedings of the 25th International Conference on Very Large Data Bases (VLDB)*, 1999.

[4] P. Indyk and R. Motwani. Approximate nearest neighbor: towards removing the curse of dimensionality. *Proceedings of the Symposium on Theory of Computing*, 1998.

[5] V.M. Zolotarev. *One-Dimensional Stable Distributions*. Vol. 65 of Translations of Mathematical Monographs, American Mathematical Society, 1986.