15-451/651: Design & Analysis of Algorithms April 18, 2019
Lecture #22: Nearest Neighbor Search last changed: April 30, 2019

In this lecture we will study the Nearest Neighbor Search problem. The input to the problem is a set of points $\mathcal{P}$. Given a query point q, the goal is to find the point in the set $\mathcal{P}$ that is closest to this query point. Naively, this requires $O(n)$ comparisons, since we have to compare q to each point in $\mathcal{P}$. In the setting where the number of points is really large, it is not computationally feasible to use the brute-force algorithm. Therefore, our goal is to preprocess the input and create a data structure that can quickly answer queries.

The Nearest Neighbor Search problem has a rich history and numerous applications[1]. It is an algorithmic primitive for finding all similar pairs, solving clustering problems on large datasets, closest pair problems in computational geometry, and is used in recommendation systems, spell-checkers, and more.

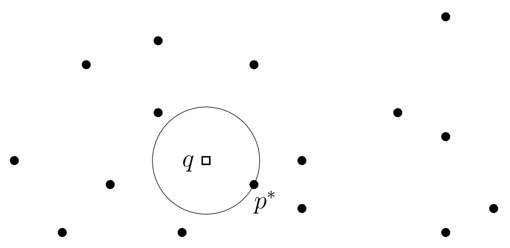
1 Nearest Neighbor Search

Formally, the nearest neighbor problem is defined as follows:

Definition 1 (Nearest Neighbor Search Problem) *Given a set of input points $\mathcal{P} = \{p_1, p_2, \dots p_n\}$ such that each $p_i \in \mathbb{R}^d$, and a query point $q \in \mathbb{R}^d$, find point*

$$p^* = \arg\min_{p_i \in \mathcal{P}} d(p_i, q)$$

i.e. the closest point to q in the set $\mathcal{P}$.



Here $d(x, y)$ is the distance between $x, y \in \mathbb{R}^d$; you can think of the Euclidean distance $\|x - y\|_2 = \sqrt{\sum_{i=1}^d (x_i - y_i)^2}$, or the Manhattan "taxicab" distance $\|x - y\|_1 = \sum_{i=1}^d |x_i - y_i|$, but you can define other distances too. E.g., we will talk about the Hamming distance later in the lecture.

However, trying to solve the Nearest Neighbor Search problem exactly for high-dimensional data is a challenging task, and known algorithms have an exponential dependence on the dimension! Such methods are not scalable in terms of dimension, a negative result which is sometimes called the "curse of dimensionality". Therefore, we relax our goal to finding *approximate* nearest neighbors. In this approximate version of the problem we instead want to find a point $p_i \in \mathcal{P}$ such that

$$d(p_i, q) \le c \cdot d(p^*, q),$$

where $c > 1$ is the "approximation" constant. Here is one formalization:

[1]See more at `https://en.wikipedia.org/wiki/Nearest_neighbor_search`

Definition 2 (c-Approximate r-Near Neighbor) *Given a point set $\mathcal{P} = \{p_1, p_2, \dots p_n\}$ and a query point $q \in \mathbb{R}^d$ such that*

$$\min_{p_i \in \mathcal{P}} d(p_i, q) \leq r,$$

output any point $p_j \in \mathcal{P}$ such that

$$d(p_j, q) \leq cr.$$

In other words, if there exists a point p_j within distance r of the query point q, output any point p_i that is within distance cr of q. (If there is no such point within r of the query q, we are allowed to return anything we want.)

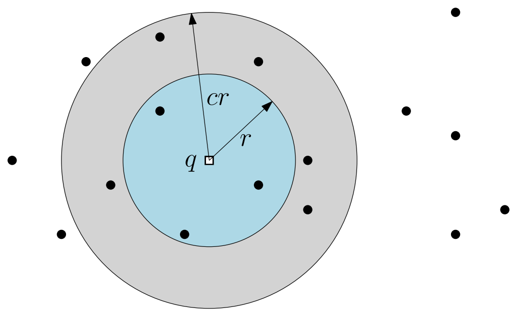


Figure 1: In c-Approximate r-Near Neighbor, if the smaller ball of radius r is non-empty, we must output some point within the larger ball of radius cr.

In fact, near neighbor searching is really a *two-stage* problem. In the first stage, we get the set $\mathcal{P}$ and process it to build a data structure. We worry about how much space this data structure uses. (Since we imagine building the data structure once and then doing tons of queries, we won't worry too much about the time to do this processing, at least for today.) In the second stage, we get query q, and we want the query time to be small. Please keep these two goals

(a) small space for the data structure, and
(b) small query time

in mind for the rest of the lecture.

1.1 Two Strawman Solutions

At a high level, we want to design a function $W : \mathbb{R}^d \to \{0,1\}^k$ such that given $W(p_1)$ and $W(p_2)$ we can distinguish between p_1 and p_2 being close, i.e. $d(p_1, p_2) \leq r$, or p_1 and p_2 being far, i.e. $d(p_1, p_2) > cr$. Additionally, we want k to be small in order to store the data structure in small space. For concreteness, let $c = 1 + \epsilon$ and $k = O\left(\frac{\log(n)}{\epsilon^2}\right)$ for some $\epsilon > 0$.

We note that we do not yet explicitly describe what W is. We defer the construction of W to the next section. Assuming we have access to a W that satisfies the above properties, we can construct a data structure for the c-Approximate r-Near Neighbor problem. Note, we can think of W as a $n \times k$ matrix. The first strategy we consider is a linear scan.

Linear Scan.
1. For all $p_i \in \mathcal{P}$, precompute $W(p_i)$.
2. Given a query point q, compute $W(q)$.
3. For all $p_i \in \mathcal{P}$, compare $W(p_i)$ and $W(q)$.

Observe, we picked W such that using $W(p_1)$ and $W(p_2)$, we can distinguish if $d(p_1, p_2) \leq r$ or $d(p_1, p_2) > cr$. Therefore, we can solve c-Approximate r-Near Neighbor using the Linear Scan algorithm. We observe that the space complexity and query time of the above algorithm is $O\left(\frac{n\log(n)}{\epsilon^2}\right)$, which is nearly linear in the input size. While we are okay with nearly linear space, we would want faster query time. This brings us to our next strategy called exhaustive storage.

Exhaustive Storage.

1. For each bit string $\sigma \in \{0,1\}^k$, construct a table A such that

$$A[\sigma] = \{p_i \in \mathcal{P} \mid d(W(p_i), \sigma) \leq (1+\epsilon)r\}$$

   i.e. $A[\sigma]$ stores the set of point p_i that are close to σ after applying W to p_i.
2. On query q, output any point in the set $A[W(q)]$.

First, we observe that the query time for the above algorithm is $O\left(\frac{d\log(n)}{\epsilon^2}\right)$, which is the time to compute $W(q)$. We now have a query time that is independent of n! Next, we observe that our space complexity is $O(2^k) = n^{O(1/\epsilon^2)}$ which is significantly larger than the Linear Scan approach. We show how to obtain the best of both worlds, i.e. near-linear space complexity and sub-linear query time.

2 Locality Sensitive Hashing

Next, we introduce an important algorithmic primitive in the design of Nearest Neighbor algorithms called *Locality Sensitive Hashing* or LSH.

Definition 3 (Locality Sensitive Hashing (informal)) *A* locality sensitive hash function *is a random hash function $h : \mathbb{R}^d \to \{0,1\}^k$ (i.e., h drawn from a family $\mathcal{H}$) such that*

1. *If $d(q,p) \leq r$, then* $\Pr[h(q) = h(p)] = P_1$ *is "not-so-small", i.e. if p close to q, $h(q)$ and $h(p)$ collide with higher probability.*
2. *If $d(q,p) > cr$ then* $\Pr[h(q) = h(p)] = P_2$ *is "small", i.e. if q is far from p, $h(q)$ and $h(p)$ collide with lower probability.*

We will specify later what P_1 being "small" and P_2 being "not-so-small" actually mean. For now, $P_1 > P_2$ and we associate the following parameter with $\mathcal{H}$ to characterize this gap:

$$\rho = \frac{\log(1/P_1)}{\log(1/P_2)}. \tag{1}$$

Observe that since $P_1 > P_2$, we have $\rho < 1$.

If we could construct a locality sensitive hash function h such that, simultaneously, P_1 was "large" and P_2 was "small", then we could simply compute the hash table of $\mathcal{P}$, A. Then on query q, simply compute $A[h(q)]$ and we would be done. Unfortunately, it is known that it is impossible to have P_1 high and P_2 low simultaneously. Therefore, instead of using just one hash function for the entire input, we use $L = n^\rho$ hash functions for independent $h_1, \ldots, h_L \in \mathcal{H}$. (We will justify this choice of L later.) Note, since $\rho = \frac{\log 1/P_1}{\log 1/P_2} < 1$, the number of hash functions used is $n^\rho < n$.

2.1 LSH for Hamming Space

Next, we construct a LSH for Hamming space, where the points lie in the Boolean cube $\{0,1\}^d$, and the distance between two d-bit vectors x and y is given by the number of coordinates on which they differ. Formally, the Hamming distance metric is given by $\mathrm{Ham}(x,y) := |\{x_i \neq y_i\}|$ where $x, y \in \{0,1\}^d$ and x_i denotes the i-th coordinate of x. The random hash function $g : \{0,1\}^d \to \{0,1\}^k$ is very simple:

> *Choose k random positions (independently and uniformly, with replacement) from the input vector, and output the bits at those positions.*

Formally, the hash family $\mathcal{H} := \{g : \{0,1\}^d \to \{0,1\}^k\}$, is defined as follows. For $p \in \{0,1\}^d$, let

$$g(p) := (h_1(p), h_2(p), \ldots, h_k(p)),$$

where

$$h_i(p) := p_j \text{ for uniformly random coordinate } j \in [d].$$

So $g(p)$ just keeps k random coordinates of the point p. Since we choose the coordinates independently, the probability of a collision is

$$\Pr[g(p) = g(q)] = \prod_{i=1}^{k} \Pr[h_i(p) = h_i(q)].$$

So if we can show that a single hash function h_i has collision with probability P, the collision probability for g is P^k, since we have to collide on all k copies.

Next, we show that the parameter ρ (which was defined in (1)) is identical for the hash functions g and h, and is approximately $1/c$. (Recall that c was the approximation factor.)

Fact 4 $\rho_g = \rho_h$

Proof: Some useful notation first. If $d(q,p) \leq r$, define $\Pr[h(q) = h(p)] = P_{1,h}$, and if $d(q,p) > cr$, $\Pr[h(q) = h(p)] = P_{2,h}$. Similarly, if $d(q,p) \leq r$, define $\Pr[g(q) = g(p)] = P_{1,g}$ else if $d(q,p) > cr$, $\Pr[g(q) = g(p)] = P_{2,g}$. Then observe,

$$\Pr[g(p) = g(q)] = \prod_{i=1}^{k} \Pr[h_i(p) = h_i(q)] \implies \begin{cases} P_{1,g} = P_{1,h}^k \\ P_{2,g} = P_{2,h}^k \end{cases}$$

since g consists of k independent copies of h. Then,

$$\rho_g = \frac{\log 1/P_{1,g}}{\log 1/P_{2,g}} = \frac{\log 1/P_{1,h}^k}{\log 1/P_{2,h}^k} = \frac{k \log 1/P_{1,h}}{k \log 1/P_{2,h}} = \rho_h$$

■

We then estimate the value of the parameter ρ_g.

Claim 5 $\rho \approx \frac{1}{c}$.

Proof: Observe that for $p, q \in \{0,1\}^d$, there are $\text{Ham}(p,q)$ locations where p and q differ, so

$$\forall i,\ \Pr[h_i(p) = h_i(q)] = 1 - \frac{\text{Ham}(p,q)}{d}.$$

For simplicity we assume $r \ll d$. This assumption is justified since we can always embed in a higher dimension, and the analysis goes through without the following approximation.

Consider the case where $\text{Ham}(p,q) \leq r$. Then,

$$\forall i,\ \Pr[h_i(p) = h_i(q)] = 1 - \frac{\text{Ham}(p,q)}{d} \geq 1 - \frac{r}{d} = P_{1,h}$$

Similarly, consider the case where $\text{Ham}(p,q) > cr$. We have,

$$\forall i,\ \Pr[h_i(p) = h_i(q)] = 1 - \frac{\text{Ham}(p,q)}{d} \leq 1 - \frac{cr}{d} = P_{2,h}$$

Using a Taylor series approximation, $e^x = 1 + x + \frac{x^2}{2!} + \cdots$. So, up to an additive error of $O((cr/d)^2)$:

$$P_{1,h} = 1 - \frac{r}{d} \approx e^{-r/d}$$
$$P_{2,h} = 1 - \frac{cr}{d} \approx e^{-cr/d}$$

This implies

$$\rho_g = \frac{\log 1/P_{1,h}}{\log 1/P_{2,h}} \approx \frac{r/d}{cr/d} = \frac{1}{c}.$$

■

2.2 LSH for Nearest Neighbor Search

We now present an algorithm for the c-approximate r-near neighbor problem in Hamming Space, via the above LSH construction. We will use the technique outlined earlier. In particular, we show how to use the Locality Sensitive Hash function to obtain near-linear space complexity and sub-linear query time. Observe, this beats the Linear Scan and Exhaustive Search algorithms presented in Subsection 1.1. Note, here we should think of W to be the LSH function.

Data Structure: Given data points $\mathcal{P}$

1. Allocate L hash tables, $A_1, \ldots, A_L$ each with a fresh Hamming LSH g_i for $i \in [l]$.
2. Hash all points in $\mathcal{P}$ into tables $A_1, \ldots, A_L$.

Query: Given query q,

1. Compute $g_1(q), \ldots, g_L(q)$.
2. Check each table $A_1[g_1(q)], \ldots, A_L[g_L(q)]$ for collisions.
3. For each collision $p \in \mathcal{P}$ under g_i, check if $d(p,q) \leq cr$. If so, output p. If none found, FAIL.

Note that we have to show the following properties:

1. (Space bound) The size of the data structures is $O(nL\log n)$.
2. (Query time bound) The expected time to do query on some q is $O(Ld) \ll O(nd)$.
3. (Correctness) On any q, if there exists a point $p^* \in \mathcal{P}$ with $d(p,q) \leq r$, then we output some $p \in \mathcal{P}$ with $d(p,q) \leq cr$ with probability at least 1/2.

But first, let us choose the parameters L and k; these will be chosen to ensure the above properties.

2.2.1 Setting the Parameters L and k

Recall, for each hash table, we have a success probability of $P_{1,g} = P_{1,h}^k$. Since we have L tables, and we are hoping for a success probability of at least 0.5, we set $L = O(1/P_{1,h}^k)$. Also recall that the probability that we have a collision when p, q are far is $P_{2,g} = P_{2,h}^k$. We pick k such that $P_{2,h}^k = \frac{1}{n}$ (for reasons described in Subsection 2.2.2). Therefore, $k = \frac{\log(n)}{\log(1/P_{2,h})}$.
Observe, for the above choices of L and k, we have

$$\begin{aligned} P_{1,g} &= \Pr[g(p) = g(q) \mid d(p,q) \leq r] \\ &\geq P_{1,h}^k \\ &= P_{1,h}^{\frac{\log(n)}{\log(1/P_{2,h})}} \\ &= n^{\frac{-\log(1/P_{1,h})}{\log(1/P_{2,h})}} \\ &= n^{-\rho} \end{aligned} \tag{2}$$

and

$$\begin{aligned} P_{2,g} &= \Pr[g(p) = g(q) \mid d(p,q) \geq cr] \\ &\leq P_{2,h}^k \\ &= \frac{1}{n} \end{aligned} \tag{3}$$

2.2.2 Analysis

Space Usage and Expected Query Time. We store L tables and for each table we hash the entire input set $\mathcal{P}$. Now each hash table could have up to 2^k buckets. However, it can only have n non-empty buckets, and we don't want to spend a lot of memory storing empty buckets. To get around this, we can hash the non-empty buckets to locations in another hash table H^i of length n using a hash function h_i from a universal family. Then, when looking up a hash bucket $g_i(q)$ given a query q and the i-th original table, we can additionally compute $h_i(g_i(q))$ to look at the appropriate entry in H^i. This entry will contain all the points p for which $g_i(p) = g_i(q)$ but it may also contain additional points. However, the expected number of additional points is $O(1)$, since h is drawn from a universal family. Also, in each entry of H, we do not need to store the actual points $p \in \{0,1\}^d$ that hash there, but rather we can store the identity of the point, i.e., if our i-th input point hashes to a given entry in H^i, we just store the index i. We also store our original ordered list of n points in $\{0,1\}^d$. Then when we read the index i, we look up the i-th point in our original list. Thus, the total memory is $O(L \cdot n \cdot \log n + n \cdot d) = O(n^{1+\rho}\log(n) + nd)$ bits.

For the expected query time, observe that we compute $g_i(q) \in \{0,1\}^k$, which requires $O(k)$ time. Repeating this for all $i \in [L]$ contributes $O(Lk)$ time. Next, we need to check for false positives. Each false positive requires $O(d)$ time to compute. Recall, by Equation 3, the probability that two far points collide is $P_{2,g} = \frac{1}{n}$. In expectation, $n \cdot \frac{1}{n} = 1$ points collide in each of the L tables, and we have an additional $O(1)$ points which may collide under h_i, and so we have to check an expected $O(L)$ points for false positives, contributing a running time of $O(Ld)$. Therefore, we have a total expected query time of $O(Ld + Lk) = O(n^\rho(k+d))$. Note that we would never need to read more than d entries to determine $g_i(q)$ because there are at most d coordinates to sample (even if $k > d$, we can still determine which bucket to examine in $O(d)$ time), so this gives $O(n^\rho d)$ total expected query time.

Correctness. To argue correctness, we show that if there exists a point p^* such that $d(q,p^*) \leq r$, then the probability the above algorithm fails is at most $1/2$. Observe, the algorithm fails if p^* is not in $\{A_1[g_1(q)], A_2[g_2(q)], \ldots A_L[g_L(q)]\}$. Therefore,

$$\begin{aligned}
\Pr\left[p^* \notin \{A_1[g_1(q)], A_2[g_2(q)], \ldots A_L[g_L(q)]\}\right] &= \prod_{i=1}^{L} \Pr[g_i(p^*) \neq g_i(q)] \\
&\leq \left(1 - \frac{1}{n^\rho}\right)^L \\
&= \left(1 - \frac{1}{n^\rho}\right)^{n^\rho} \\
&\leq \frac{1}{e}
\end{aligned}$$

where the second inequality follows from Equation 2 and the third follows from our choice of L. Thus, we obtain the following theorem :

Theorem 6 *Given $\mathcal{P} = \{p_1, p_2, \ldots p_n\} \in \{0,1\}^n$, equipped with the Hamming metric, there exists an algorithm to preprocess the points and obtain the following guarantees:*

1. *(Space bound) The size of the data structures is $O(n^{1+\rho}\log(n) + nd)$ bits.*
2. *(Query time bound) The expected time to do query on some q is $O(n^\rho d)$.*
3. *(Correctness) On any q, if there exists a point $p^* \in \mathcal{P}$ with $d(p,q) \leq r$, then we output some $p \in \mathcal{P}$ with $d(p,q) \leq cr$ with probability at least $1/2$.*

3 Linear Sketching and CountSketch

Here are the main points about linear sketching:

1. We want to keep track of some vector $x^t \in \mathbb{R}^n$ that is changing over time. Each time some update Δ_t arrives, and then $x^{t+1} \leftarrow x^t + \Delta_t$.
2. A *(linear)* sketch is a short fat matrix $S : \mathbb{R}^n \to \mathbb{R}^k$ where think of $k \ll n$. So instead of maintaining $x^t \in \mathbb{R}^n$ explicitly, we maintain the sketch $Sx^t \in \mathbb{R}^k$.
3. The advantage of this *linear* sketch is that if we know $y^t := Sx^t$, then
$$y^{t+1} = Sx^{t+1} = S(x^t + \Delta_t) = Sx^t + S\Delta_t = y^t + S\Delta_t.$$
And if we can compute the sketch $S\Delta_t$ of the "update" vector Δ_t quickly and in little space, we're all set. That's what we will do.

3.1 Maintaining the Euclidean norm of a vector via CountSketch

1. Want to maintain some info so that we can answer question: what is $\|x^t\|^2$? I.e., want to output some estimate Z such that $Z \in (1 \pm \epsilon)\|x^t\|^2$

2. We maintain two hash functions:

   (a) One maps coordinates of x into k bins: i.e., $h : [n] \to [k]$, it is a 2-wise independent hash function.

   (b) The other maps each coordinate into random bits: $\sigma : [n] \to \{-1, 1\}$, which is 4-wise independent hash function.

3. This gives a sketch matrix $S \in \{-1, 1\}^{k \times n}$, where the i^{th} column has a single non-zero entry in the $h(i)^{th}$ row. This entry has value $\sigma(i)$.

$$\begin{bmatrix} 0 & 0 & 1 & 0 & 0 & 1 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & -1 & 1 & 0 & -1 & 0 \\ 0 & -1 & 0 & 0 & 0 & 0 & 0 & 1 \end{bmatrix}$$

4. We never write down this matrix S. Indeed, suppose $x^{t+1} \leftarrow x^t + (0, \ldots, 0, a, 0, \ldots, 0)^\intercal$, where the change a is in the i^{th} coordinate. Then to compute Sx^{t+1} from Sx^t, we just add in $S(0, \ldots, 0, a, 0, \ldots, 0)^\intercal$ to Sx^t. Which is the same as adding in $a \cdot \sigma(i)$ to the $h(i)^{th}$ coordinate of Sx^t. This can be done in constant time (two hash function evals, $O(1)$ arithmetic ops).

5. Now we claim that $(Sx)^2$ is a good estimate of $\|x\|^2$. Why? First, let's consider the expectation $\mathrm{E}[(Sx)^2]$.

$$\begin{aligned}
\mathrm{E}[(Sx)^2] &= \mathrm{E}\left[\sum_{j=1}^{k}\left(\sum_{i\in[n]} \delta(h(i)=j)\sigma(i)x_i\right)^2\right] \\
&= \mathrm{E}\left[\sum_{j=1}^{k}\sum_{i,i'\in[n]} \delta(h(i)=j)\cdot\delta(h(i')=j)\cdot\sigma(i)\,x_i\,\sigma(i')\,x_{i'}\right] \\
&= \sum_{j=1}^{k}\sum_{i,i'\in[n]} \mathrm{E}[\delta(h(i)=j)\cdot\delta(h(i')=j)]\cdot\mathrm{E}[\sigma(i)\sigma(i')]\;\;x_i\,x_{i'} \\
&= \sum_{j=1}^{k}\sum_{i,i'\in[n]} \Pr[h(i)=j)\wedge h(i')=j]\cdot\mathrm{E}[\sigma(i)\sigma(i')]\;\;x_i\,x_{i'}
\end{aligned}$$

But $\mathrm{E}[\sigma_i] = 0$, since it takes on value -1 and 1 with equal probability. And since the hash function σ is pairwise independent, the expectation will be zero *except when $i = i'$, when it is* 1! So we get

$$= \sum_{j=1}^{k}\sum_{i\in[n]} \Pr[h(i)=j)]\cdot x_i^2$$

And finally, since $\Pr[h(i) = j] = 1/k$, we get $\sum_{j=1}^{k} \|x\|^2/k = \|x\|^2$. So the expectation of our estimator $(Sx)^2$ is indeed correct!

6. The expectation being correct is fine, what about the variance? In the recitation we will show that the variance is also controlled, i.e.,

$$\mathrm{Var}((Sx)^2) = 2\|x\|^4/k. \tag{4}$$

   This is where we will use the 4-wise independence of the hash function σ, and the pairwise independence of the function h, because the above argument only used the pairwise independence of σ!

7. Now if we know the expectation (a.k.a. mean) is correct, and the variance is bounded as in (4), we can use Chebyshev's inequality.[2] This says that we are close to the mean with high probability. Recall it says that for a random variable Z with mean μ and variance σ^2,

$$\Pr[|Z - \mu| \geq \lambda] \leq \frac{\sigma^2}{\lambda^2}.$$

   (Please don't confuse this variance symbol σ with the hash function σ.) In our case the r.v. is $Z = (Sx)^2$, with mean $\mu = \|x\|^2$, and variance at most $2\|x\|^4/k$, then

$$\Pr[|Z - \mu| \geq \varepsilon\mu] \leq \frac{2\|x\|^4/k}{\varepsilon^2\mu^2} = \frac{2\|x\|^4/k}{\varepsilon^2\|x\|^4} = \frac{2}{k\varepsilon^2}.$$

   So if we want a failure probability of $1/10$, say, then we can set $k = \frac{20}{\varepsilon^2}$.

8. In summary, the sketch S we defined (using the hash functions h, σ), gives us a $(1 \pm \varepsilon)$ estimate of the squared Euclidean length of a vector with probability at least $9/10$, as long as $k \geq 20/\varepsilon^2$.

9. Note that instead of maintaining $x \in \mathbb{R}^n$, we just needed to store the sketch $Sx \in \mathbb{R}^k$, where $k \approx 20/\varepsilon^2$. That's a huge reduction in space!! Also, it's a linear sketch, so we can maintain this sketch as the vector x changes via updates.

[2] Recall how to prove this too: $\Pr[|Z - \mu| \geq \lambda] = \Pr[(Z - \mu)^2 \geq \lambda^2] \leq \frac{\mathrm{E}[(Z-\mu)^2]}{\lambda^2}$, where the inequality is Markov's inequality. Now the numerator is just the definition of variance σ^2.