CSE 521: Design and Analysis of Algorithms I **Fall 2020**

Lecture 6: Streaming / Locally Sensitive Hashing

Lecturer: Shayan Oveis Gharan *10/19/2020*

6.1 Introduction to Streaming Algorithms

As an application of hashing and the unbiased estimator, we are going to discuss streaming algorithms. Streaming algorithms has become a hot topic in computer science nowadays because of the massive amount of data that we have to process. Typically, we do not have enough space to store the entire data. Instead, we process the data in a streaming fashion, and sketch the information we want from the data by a few passes.

We will talk about algorithms for F_0 and F_2 estimation. Those are classic results appeared in the first paper of streaming algorithms [AMS96]. The problem is as follows.

Let $\mathcal{U} = \{1, ..., |\mathcal{U}|\}$ be a large universe of numbers, and let $X_1, ..., X_n$ be a sequence of numbers in $\mathcal{U}$. Let $f_i = \sum_{j=1}^{n} \mathbb{I}[X_j = i]$ be the number of times i appears in the sequence. For $0 \leq k \leq \infty$, we let $F_k = \sum_{i=1}^{|\mathcal{U}|} f_i^k$, where we define $0^0 = 0$. The interesting values of k for us are

- When $k = 0$, F_0 counts the number of distinct elements in the sequence.
- When $k = 2$, F_2 is the second moment of the vector $(f_1, ..., f_{|\mathcal{U}|})$.
- When $k = \infty$, F_∞ corresponds to the number of times the most frequent number shows up in the sequence.

The following theorem is proven in [AMS96]

Theorem 6.1. *There is a streaming algorithm that for any sequence $x_1, \ldots, x_n$ of the universe $\{1, 2, \ldots, |U|\}$ gives a $(1 - \epsilon)$ approximation of F_0 and F_2 using $O(\frac{\log |U| + \log n}{\epsilon^2} \cdot \log \frac{1}{\delta})$ space with probability $1 - \delta$.*

Here, we only give an algorithm for F_2 and we leave the algorithm for F_0 as a homework exercise.

We remark that allowing randomness and approximated solution is crucial. There is no hope to use a deterministic or exact algorithm to achieve logarithmic amount of space. Please see Tim Roughgarden's Lecture notes for more details.

6.2 F_2 moment

Before discussing ideas to prove Theorem 6.1, let us first discuss a natural idea for streaming problems. Since we are dealing with a "big data" problem, we may first down sample the input into a smaller length, then we calculate the second moment of the down sample and we use it to estimate the second moment of the original input. Consider the following set of two inputs.

$$1, 2, 3, 4, \dots, n$$
$$\underbrace{1, 1, \dots, 1}_{m \text{ times}}, \underbrace{2, 2, \dots, 2}_{m \text{ times}}, \dots, \underbrace{n/m, n/m, \dots, n/m}_{m \text{ times}},$$

where $m = \Omega(\sqrt{n})$. Observe that any down samples of the first sequence gives completely distinct numbers, and any down sample of the second sequence of size $O(\sqrt{n})$ also gives (almost) completely distinct numbers with a constant probability. So, any streaming algorithm that is based on down sampling sees almost the same thing, i.e., distinct elements, in both cases. However, the second moment of the first sequence is n and the second moment of the second one is $O(n^{3/2})$, so we don't expect a streaming algorithm based on down sampling to size at most $O(\sqrt{n})$ obtain an estimate better than $\sqrt{n}$ of the true second moment.

Our plan for the proof is that we design an unbiased estimator for F_2 that uses $O(\log |U| + \log n)$ amount of memory and has a relative variance of $O(1)$. Then, by **??** we only need $O(\frac{1}{\epsilon^2} \log \frac{1}{\delta})$ independent samples of our unbiased estimator; so it is enough to run $O(\frac{1}{\epsilon^2} \log \frac{1}{\delta})$ independent copies of our algorithm on the input sequence and run the algorithm of **??** on the output of these independent copies.

Before designing algorithm for estimating F_2, let's revisit the random walk example that we talked about few lectures ago. Denote random variable X_i

$$X_i = \begin{cases} +1, & \text{w.p. } 1/2 \\ -1, & \text{w.p. } 1/2 \end{cases}$$

Let $X = \sum_i X_i$. Using the Hoeffding bound, we have showed that for any constant,

$$\mathbb{P}[|X| \geq c\sqrt{n}] \leq e^{-\Omega(c^2)}.$$

We will show that $X \geq \Omega(\sqrt{n})$ with a constant probability, that is $\mathbb{P}[X \geq \Omega(\sqrt{n})] \geq \Omega(1)$. This conclusion follows from the centreal limit theorem (CLT). Because $\lim_{n\to\infty} \frac{1}{\sqrt{n}} \sum_i X_i \to \mathcal{N}(0,1)$. So,

$$\sum_i X_i \to_{n\to\infty} \mathcal{N}(0, \sqrt{n}).$$

Therefore, we expect $\sum_i X_i$ to be (almost) uniform in the interval $[-\sqrt{n}, \sqrt{n}]$. So, the particle is almost uniformly distributed in the interval $[-\sqrt{n}, \sqrt{n}]$.

We show that $\mathbb{E}[X^2] \geq n$, which is enough to show $X \geq \Omega(\sqrt{n})$ with a constant probability, using the Hoeffding bound.

$$\begin{aligned}
\mathbb{E}[X^2] &= \mathbb{E}\left[\left(\sum_i X_i\right)^2\right] \\
&= \mathbb{E}\left[\sum_{i,j} X_i X_j\right] \\
&= \sum_{i,j} \mathbb{E}\left[X_i X_j\right] \\
&= \sum_i \mathbb{E}\left[X_i^2\right] + \sum_{i\neq j} \mathbb{E}\left[X_i X_j\right] \\
&= \sum_i 1 + \sum_{i \neq j} \mathbb{E}\left[X_i\right] \mathbb{E}\left[X_j\right] = n \qquad (6.1)
\end{aligned}$$

where we used the pairwise independence between X_i and X_j.

Back to estimating F_2 problem, we use a similar idea. Choose a pairwise independent hash function $h : U \to \{-1, 1\}$. Eventually we will see that we need to make h 4-wise independent. We start the algorithm letting $Y = 0$; when we read X_i, we'll update Y,

$$Y \leftarrow Y + h(X_i).$$

Note that at the end of the algorithm $Y = \sum_{i=1}^{n} h(X_i)$. We claim that Y^2 is the unbiased estimator of F_2.

Claim 6.2. *Y^2 is an unbiased estimator of F_2, i.e.,*

$$\mathbb{E}\left[Y^2\right] = F_2 = \sum_{i=1}^{|U|} f_i^2.$$

Before proving the claim let us consider two special cases. First suppose $X_1 = 1, X_2 = 2, \ldots, X_n = n$. Then, $\sum_i h(X_i)$ can be seen as a (pairwise independent) random walk of length n started from 0. So, by (6.1), $\mathbb{E}\left[Y^2\right] = n$ which is the same as F_2. Note that in (6.1) we only use pairwise independence of X_i, X_j (for all i, j).

For the second example, suppose $X_1 = X_2 = \cdots = X_n = 1$. Then, $f_2 = n^2$ and $Y = nh(1)$, So,

$$Y^2 = n^2 h(1)^2 = n^2 = f_2.$$

Now, we are ready to prove the claim.

Proof. First, observe that we can write $Y = \sum_i f_i h(i)$. Therefore,

$$\begin{aligned}
\mathbb{E}[Y^2] &= \mathbb{E}\left[(\sum_i f_i h(i))^2\right] && (6.2)\\
&= \mathbb{E}\left[\sum_{i,j} f_i f_j h(i)h(j)\right] && (6.3)\\
&= \sum_{i,j} f_i f_j \mathbb{E}\left[h(i)h(j)\right] && (6.4)\\
&= \sum_i f_i^2 \mathbb{E}\left[h(i)^2\right] + \sum_{i,j} f_i f_j \mathbb{E}\left[h(i)\right]\mathbb{E}\left[h(j)\right] && (6.5)\\
&= \sum_i f_i^2 + 0 = F_2, && (6.6)
\end{aligned}$$

where the second to last equality follows by pairwise independent of h. Note that the expectations are over random choices for our pairwise independent hash function h. Therefore, Y^2 is the unbiased estimator of F_2. □

Now, by Theorem 5.2, we upper bound the relative variance of Y^2. First, we show that $\sigma^2(Y^2) \leq 2(F_2)^2$. Then, we show that the relative variance of Y^2 is $O(1)$. In the proof of the next claim we use that h is 4-wise independence.

Claim 6.3. $\sigma^2(Y^2) \leq 2(F_2)^2$.

Proof. Recall that $\sigma^2(Y^2) = \mathbb{E}Y^4 - (\mathbb{E}Y^2)^2$. So, first we upper bound $\mathbb{E}Y^4$.

$$\begin{aligned}
\mathbb{E}Y^4 &= \mathbb{E}\left(\sum_i f_i h(i)\right)^4 && (6.7)\\
&= \mathbb{E}\sum_{i,j,k,l} f_i f_j f_k f_l h(i)h(j)h(k)h(l) && (6.8)\\
&= \sum_{i,j,k,l} f_i f_j f_k f_l \mathbb{E}\left[h(i)h(j)h(k)h(l)\right] && (6.9)\\
&= \sum_{i\neq j} f_i^2 f_j^2 \binom{4}{2} + \sum_i f_i^4 && (6.10)
\end{aligned}$$

Note that in the last equality we used 4-wise independence of h. in particular if any of i, j, k, l appear an odd number of times then $\mathbb{E}[h(i)h(j)h(k)h(l)]$ is zero. So, the only case that it is nonzero is if i appears 4 times or i, j each appear 2 times. Therefore,

$$\begin{aligned}
\sigma^2(Y^2) &= 6\sum_{i\neq j} f_i^2 f_j^2 + \sum_i f_i^4 - \left(\sum_i f_i^2\right)^2 && (6.11)\\
&= 4\sum_{i\neq j} f_i^2 f_j^2 && (6.12)\\
&\leq 2\left(\sum_i f_i^2\right)^2 && (6.13)\\
&= 2(F_2)^2 && (6.14)
\end{aligned}$$

as desired. □

It follows from the above claim that the relative variance of Y^2 is

$$\frac{\sigma^2(Y^2)}{(F_2)^2} \leq \frac{2F_2^2}{F_2^2} = 2.$$

So, by Theorem 5.2 we only need $O(\frac{1}{\epsilon^2}\log\frac{1}{\delta})$ independent samples of Y to give a $1 \pm \epsilon$ approximation of F_2.

Now, let us discuss the final algorithm. We choose $\frac{20}{\epsilon^2}\log\frac{1}{\delta}$ independent 4-wise independent hash functions; For $k = 20/\epsilon^2$, let $h_1, \ldots, h_k$ be k of those functions. Start with $Y_1 = Y_2 = \cdots = Y_k = 0$, after reading X_i, let $Y_j = Y_j + h_j(X_i)$ for all $1 \leq j \leq k$. Then, $Z = \frac{1}{k}\sum_j Y_j$ gives a $1 \pm \epsilon$ approximation of F_2 with probability 9/10. To get $1 - \delta$ probability of success we just need to run the above idea on $\log\frac{1}{\delta}$ independent copies of Z and return the median of all values that we get.

The total amount of memory that we use is as follows. We need to use $O(\log n)$ amount of memory to store each Y; we need to use $O(\log|U|)$ amount of memory to store a 4-wise independent hash function. So, for each copy we need $O(\log n + \log|U|)$ amount of memory. Since we are using $O(\frac{1}{\epsilon}^2\log\frac{1}{\delta})$ independent copies we use $O(\frac{\log n + \log|U|}{\epsilon^2}\log\frac{1}{\delta})$ bits of memory as desired. This completes the proof of Theorem 6.1. More details can be referred to [AMS96].

6.3 Introduction to the Nearest Neighbor Search Problem

In this lecture we will discuss a reduction of the nearest neighbor search (NNS) problem to that of finding a locally sensitive hashing function as invented in [IM98].

The NNS problem is as follows: Suppose $P \subset \mathbb{R}^d$ is a set of n points. Given any $q \in \mathbb{R}^d$ find

$$\min_{p \in P} \operatorname{dist}(p, q).$$

The distance here could be any arbitrary distance function; in this lecture we will talk more about ℓ_1 or ℓ_2 distances even though the machinery that we describe can be generalized to a variety of distance functions. Some applications include: web search, document search, or clustering - these are all situations in which knowing how "far" an object is from other objects tells us important information.

A naive solution would be to store all of the points and simply loop over all $p \in P$ to find the minimum distance. This takes $O(n \cdot d)$ time and space, which is not good. Ideally we would like to have a query time that is sublinear in n; we may allow for a super-linear amount of memory to store the data structure.

If $d = 1$ we could pre-process the points by sorting them and then finding the distance minimizing point would simply reduce to binary searching for p in a list, and returning the closest of the two adjacent elements in the list. This takes $O(\log n)$ query time and $O(n)$ bits of memory.

Extending the pre-processing idea to higher dimensions d leads to what are known as k-d trees: here the idea is to partition the space by using coordinate-aligned planes chosen appropriately for the data at hand. Unfortunately k-d trees generally fail to beat the naive approach when $d = \Omega(\log n)$. It turns out that in all known approaches the size of the data structure (or the query-time) grows exponentially in d.

The main underlying difficulty is the well-known facts in high dimensions, which is usually referred to as the "curse of dimensionality". Suppose we partition the space by a grid where each cell is a cube of side length a. Then, a cube of side length a randomly positioned in the space intersects 2^d many cells of the grid. This phenomenon essentially implies that a NNS algorithm based on kd-trees takes time $O(2^d)$ in expectation to look into all of the nearby cells of a query point to find the closes point.

6.4 Reducing to Approximate Nearest Neighbors Search

We now describe the idea of [IM98]. Firstly, instead of solving the exact problem we will look for approximate solutions. That is instead of finding the closest point p to a query point q, we are happy to find a point p such that

$$\operatorname{dist}(p, q) \le c \cdot \min_{s \in P} \operatorname{dist}(s, q),$$

where $c > 1$ is the approximation factor of in our algorithm. As we will see the memory and the query time of our algorithm will be a function of c.

So, let us define the approximate NNS problem. For $c > 1, r > 0$, the ANNS(c, r) is defined as follows: Given a set point of points P, construct a data structure such that for any query point q, if there is a point p such that $\operatorname{dist}(p, q) \le r$, it returns a point p' such that

$$\operatorname{dist}(p', q) \le c \cdot r.$$

If there is no such p, then we return nothing.

It is not hard to see that we can give a c approximation to the nearest neighbor search problem using the solution to ANNS(c, r). In fact, all we need to do is to guess $\min_{p \in P} \operatorname{dist}(p, q)$ up to a multiplicative factor of $1 \pm \epsilon$. By an approporiate scaling assume

$$\operatorname{diam}(P) = \max_{p, p' \in P} \operatorname{dist}(p, p') \le 1$$

Also, suppose $\delta > 0$ is the minimum possible distance for all pairs of points in our dataset. Roughly speaking, $1/\delta$ can represent the bit precision of the data points stored in our system. We solve ANNS$(c(1-\epsilon), r)$ for the following values of r,

$$\delta, (1+\epsilon)\delta), (1+\epsilon)^2\delta, \ldots, 1.$$

We report the minimal value of r for which we find a point at distance $c(1-\epsilon)$ of q. This reduction imposes an additional $O(\log \frac{1}{\delta})$ overhead to the query time and the memory of our algorithm. This is because we need to maintain a separate data structure for each possible value of r in the above sequence.

6.5 Locally Sensitive Hashing functions

From now on we only focus on the ANNS(c, r). The main interesting idea of [IM98] is a reduction from this problem to the design of a locally sensitive hash (LSH) function. Roughly speaking, an LSH is a hash function which is sensitive to distance. Ideally, we would like to have a hash function that maps "close points" to the same value with a high probability and maps "far points" to different values. To be more precise, if $\text{dist}(p, q) \leq r$ we want them to map to the same value, with a high probability, and if $\text{dist}(p, q) > c \cdot r$ we want them to map to different values with a high probability. Let us give a formal definition

Suppose we have a family a functions $\mathcal{H} = \{h : P \to \mathbb{Z}\}$ of maps from our points P to the set of integers $\mathbb{Z}$; we say $\mathcal{H}$ is $(c, c \cdot r, p_1, p_2)$-LSH if: for all $p, q \in P$:

$$\begin{aligned} \text{dist}(p,q) \leqslant r &\implies \mathbb{P}[h(p) = h(q)] \geqslant p_1 \\ \text{dist}(p,q) \geqslant c \cdot r &\implies \mathbb{P}[h(p) = h(q)] \leqslant p_2 \end{aligned}$$

where the probabilities are over $h \sim \mathcal{H}$. Ideally, we want to have $p_1 \gg p_2$, but as we see this highly depends on the magnitude of c. The main idea in the reduction of [IM98] is that even if p_1 is slightly larger than p_2 it is possible to use many independently chosen functions from $\mathcal{H}$ to *boost* p_1 to a number close to 1 and p_2 to $1/n$.

Before describing the reduction, let us give an example of LSH for binary vectors. We will see several examples in HW2. Suppose $P \subseteq \{0, 1\}^d$ with Manhattan distance function

$$\text{dist}(p, q) = \|p - q\|_1,$$

i.e. $\text{dist}(p, q)$ is the number of coordinates at which p and q have different bits. Consider the family $\mathcal{H} := \{h_i\}_{i=1}^d$ where

$$h_i(p) = p_i$$

is the ith bit of p. Then observe that for each $p, q \in \{0, 1\}^d$

$$\mathbb{P}[h(p) = h(q)] = \frac{\#\text{ bits in common}}{\text{total bits}} = \frac{d - \|p-q\|_1}{d} = 1 - \frac{\|p-q\|_1}{d}.$$

Therefore,

$$\mathbb{P}[h(p) = h(q)] = \begin{cases} \geq 1 - \frac{r}{d} \approx e^{-r/d} & \text{if } \text{dist}(p,q) \leq r \\ \leq 1 - \frac{c \cdot r}{d} \approx e^{-c \cdot r/d} & \text{if } \text{dist}(p,q) \geq c \cdot r \end{cases}.$$

So, $\mathcal{H}$ is $(c, c \cdot r, e^{-r/d}, e^{-c \cdot r/d})$-LSH.

6.6 Reduction to LSH

Now let us discuss the reduction from ANNS(c, r) to LSH? Well if we had a $(r, c \cdot r, p_1, p_2)$-LSH family such that $p_1 \approx 1$ and $p_2 \approx 0$ we could solve the problem as follows: We start by choosing a function $h \sim \mathcal{H}$ uniformly at random and we store $h(p)$ for all points in P. Given a query point q, we compute $h(q)$ and see if there is any point $p \in P$ where $h(p) = h(q)$. Note that we can do the lookup in $O(1)$ time using a hash table as we discussed in previous lectures. If there is no such point p, then with high probability there is no point at distance $c \cdot r$ of q. Thus we only need to show that if we are given an $(r, c \cdot r, p_1, p_2)$-LSH family with the assumption $p_1 > p_2$, then we can boost it to get $p_1 \approx 1$ and $p_2 \approx 0$.

We do this boosting in two steps. First, we just try to make p_2 small. To do this it suffices to take k independent hash functions from $\mathcal{H}$, and hash each point $p \in P$ to a k-dimensional vector,

$$h(p) = [h_1(p), \dots, h_k(p)].$$

Then, by the independence of $h_1, \dots, h_k$, for any two points p, q,

$$\mathrm{dist}(p, q) \geqslant c \cdot r \implies \mathbb{P}[h(p) = h(q)] \leqslant p_2^k.$$

But this doesn't help us increase p_1. In fact, the above hash function maps two close points to the same vector with probability at least p_1^k. How do we do this? We choose ℓ independent copies of the above k-dimensional hash function, $f_1, f_2, \dots, f_\ell$, for a sufficiently large ℓ, with high probability there is an i such that $f_i(p) = f_i(q)$. Assume,

$$\begin{aligned} f_1(p) &= [h_{1,1}(p), \dots, h_{1,k}(p)] \\ &\vdots \\ f_\ell(p) &= [h_{\ell,1}(p), \dots, h_{\ell,k}(p)] \end{aligned}$$

It follows that if $\mathrm{dist}(p, q) \leq r$, then

$$\begin{aligned} \mathbb{P}[\exists i \mid f_i(p) = f_i(q)] &= 1 - \mathbb{P}[\forall i, f_i(p) \neq f_i(q)] \\ &= 1 - \mathbb{P}[f_i(p) \neq f_i(q)]^\ell \\ &\geqslant 1 - (1 - p_1^k)^\ell \end{aligned}$$

The details of the algorithm is described in Equation 6.6.

Next, we describe how to tune the parameters k, ℓ. We choose k such that $p_2^k = 1/n$. Also, assume

$$p_1 = p_2^\rho, \tag{6.15}$$

for some $\rho < 1$. As we will see ρ is the main parameter that determines the running time/memory of our algorithm. We choose $\ell = \Theta(n^\rho \ln n)$.

Fix a query point q; it follows by linearity of expectation that for any i,

$$\mathbb{P}[\exists p : \mathrm{dist}(p, q) > c \cdot r, f_i(p) = f_i(q)] = n \cdot p_2^k \leq 1.$$

Summing up over all i, in expectation there are $O(\ell)$ points in our data set which map to the same hash value as q for some i. This implies an overhead of $O(\ell)$ in the query time.

On the other hand, if $\mathrm{dist}(p, q) \leq r$ for some $p \in P$, then

$$\mathbb{P}[\exists i : f_i(p) = f_i(q)] \geq 1 - (1 - p_1^k)^\ell = 1 - (1 - p_2^{\rho k})^\ell = 1 - (1 - n^{-\rho})^\ell \approx 1 - e^{-\ell n^{-\rho}} = 1 - 1/n.$$

In summary, for any point p at distance at most r, our algorithm outputs p with probability at least $1 - 1/n$. The algorithm in expectation had $O(\ell \cdot d)$ overhead to examine $O(\ell)$ points at distance more than $c \cdot r$ form q.

Algorithm 1 LSH Algorithm

```
Preprocessing:
  Choose $k \cdot \ell$, $h_{1,1}, \dots, h_{\ell,k}$ functions uniformly at random from $\mathcal{H}$.
  Construct $\ell$ hash tables; for all $1 \le i \le \ell$ store $f_i(p) = (h_{i,1}(p), \dots, h_{i,k}(p))$ for all $p \in P$ in the $i$-th table.
  For all $i$, sort all values of $\{f_i(p) : p \in P\}$.
Query($q$):
  for $i = 1 \to \ell$ do
    Compute $f_i(q)$.
    Find all points $p$ where $f_i(p) = f_i(q)$ using a binary search on table $i$. For all such points if dist$(p, q) \le c \cdot r$, output $p$.
  end for
```

6.7 Space and Time Complexity of the Reduction

The algorithm needs to maintain $O(\ell)$ hash tables. In each hash table we need to store $n = |P|$ hash values where each value is a k dimensional vector. So, the space complexity of the algorithm is

$$O(\ell \cdot n \cdot k) = O(n^{1+\rho} \frac{\log n}{\log \frac{1}{p_2}}).$$

For any query point q the query time is $O(\ell \cdot k)$ time to compute $f_i(q)$ for all $1 \le i \le \ell$. For any candidate close point p we spend $O(d)$ time to calculate dist(p, q). Let $|O|$ be size of the output, i.e., the number of points at distance $c \cdot r$ from q. In expectation we examine $O(\ell)$ far points that we don't output. So, the query time is $O(d(\ell + |O|)$ in expectation. So, the query time is

$$O(d(\ell + |O|) + \ell \cdot k) = O\left(n^\rho \left(d + \frac{\log n}{\log \frac{1}{p_2}}\right) + |O|d\right).$$

Ignoring lower order terms, in particular the size of the output and the dimension, the algorithm runs with memory $O(n^{1+\rho})$ and querytime $O(n^\rho)$.

Let us calculate ρ for the binary vector example that we described at the beginning. Recall that ρ is chosen such that $p_1 = p_2^\rho$, so

$$\rho = \frac{\ln \frac{1}{p_1}}{\ln \frac{1}{p_2}} = \frac{r/d}{c \cdot r/d} = \frac{1}{c}.$$

For example, if $c = 2$, we need $O(n^{1.5})$ to store hash tables and we have $O(\sqrt{n})$ query time. As we see the query time (and memory) get significantly better as we increase c. In practice, we may tune the parameter c based on the amount of resources available to us.

It has been a very active area of research to design the best of LSH functions for many metrics. In PS3 we design LSH for ℓ_1, ℓ_2 distance where $\rho = 1/c$.

References

[AMS96] N. Alon, Y. Matias, and M. Szegedy. "The space complexity of approximating the frequency moments". In: *STOC*. ACM. 1996, pp. 20–29 (cit. on pp. 6-1, 6-4).

[IM98] P. Indyk and R. Motwani. "Approximate nearest neighbors: towards removing the curse of dimensionality". In: *STOC*. ACM. 1998, pp. 604–613 (cit. on pp. 6-4, 6-5, 6-6).