Lecture 7

Prof. Nick Harvey *University of British Columbia*

In this lecture we will see two applications of the Johnson-Lindenstrauss lemma.

1 Streaming Algorithms

In 1996, Alon, Matias and Szegedy introduced the **streaming model** of computation. Given that they all were at AT&T Labs at the time, their work was presumably motivated by the problem of monitoring network traffic. Their paper was highly influential: it earned them the Gödel prize, and it motivated a huge amount of follow-up work, both on the theory of streaming algorithms, and on applications in networking, databases, etc.

The model aims to capture scenarios in which a computing device with a very limited amount of storage must process a huge amount of data, and must compute some aggregate statistics about that data. The motivating example is a network switch which must process dozens of gigabytes per second, and may only have have a few kilobytes or megabytes of fast memory. We might like to compute, for example, the number of distinct traffic flows (source/destination pairs) traversing the switch, the variance of the packet sizes, etc.

Let us formalize this model using **histograms**. The data is a sequence $(i_1, i_2, \ldots, i_m)$ of indices, where each $i_t \in \{1, \ldots, n\}$. Intuitively, we want to compute the histogram $x \in \mathbb{Z}^n$, where

$$x_j \;=\; |\{\, t \;:\; i_t = j\}|.$$

If the algorithm were to maintain x explicitly, it could start with $x = 0$, then at each time step t, it receives the index i_t and increments x_{i_t} by 1. After receiving all the data, the algorithm could compute various statistics of x, such as a norm $\|x\|_p$, or the number of non-zero entries, etc.

So far the problem is trivial. What makes the model interesting, is that we will only allow the algorithm space $O(\text{polylog}(m))$. (We will assume that $n = \text{poly}(m)$, so it takes $O(\log m)$ bits to represent any integer in $0, \ldots, \max\{m, n\}$.) So the algorithm cannot store the data $(i_1, \ldots, i_m)$ explicitly, nor can it store the histogram x explicitly!

Remarkably, numerous interesting statistics can still be computed in this model, if we allow randomized algorithms that output approximate answers. Nearly optimal bounds are known for the amount of space required to estimate many interesting statistics.

Today we will give a simple algorithm to estimate the L_2-norm, namely $\|x\| = (\sum_i x_i^2)^{1/2}$. As an example of a scenario where this would be useful, consider a database table $((a_1, b_1), \ldots, (a_m, b_m))$. A self-join with the predicate $a = a$ would output all triples (a, b, b') where (a, b) and (a, b') belong to the table. What is the size of this self-join? It is simply $\|x\|^2$, where x is the histogram for the a values in the table. So a streaming algorithm for estimating $\|x\|$ could be quite useful in database query optimization.

The Algorithm. The idea is very simple: instead of storing x explicitly, we will store a *dimensionality reduced* form of x. Let L be a $d \times n$ matrix whose entries are drawn independently from the distribution $N(0, 1/d)$. (This is the same as the linear map L defined in the previous lecture.) The algorithm will explicitly maintain the vector y, defined as $y := L \cdot x$. At time step t, the algorithm receives the index

$j = i_t$ so (implicitly) the jth coordinate of x increases by 1. The corresponding change in y is to add the jth column of L to y.

To analyze this algorithm, we use the Johnson-Lindenstrauss lemma. Our results from last time imply that

$$\Pr[\ (1-\epsilon)\|x\| \ \leq\ \|y\| \ \leq\ (1+\epsilon)\|x\|\ ]\ \geq\ 1-\exp(-\Omega(\epsilon^2 d)).$$

So if we set $d = \Theta(1/\epsilon^2)$, then $\|y\|$ gives a $(1+\epsilon)$ approximation of $\|x\|$ with constant probability. Or, if we want y to give an accurate estimate at each of the m time steps, we can take $d = \Theta(\log(m)/\epsilon^2)$.

How much space? At first glance, it seems that the space used by this algorithm is just the space needed to store the vector y, which is d words of space. (There is also the issue of how many bits of accuracy are needed when generating the Gaussian random variables, but we will ignore that issue. As discussed last time, the Johnson-Lindenstrauss lemma works equally well with $\{+1, -1\}$ random variables, so numerical accuracy is not a concern.)

There is one small problem: the matrix L must not change during the execution of the algorithm. So, every time the algorithm sees the same index j in the data stream, it must add *the same* jth column of L to y. We cannot generate a new random column each time. The naive way to accomplish this would be to generate L at the beginning of the algorithm and explicitly store it so that we can use its columns in each time step. However, L has $d \cdot n$ entries, so storing L is even worse than storing x!

The solution to this problem is to observe that L is a random object, so we may not need to store it explicitly. In a practical implementation, L will be generated by a pseudorandom generator initialized by some seed, so we can regenerate columns of L at will by resetting the seed.

Alternatively, there is another solution which has provable guarantees but is probably too complicated to use in practice. Long before the streaming model was introduced, Nisan designed a beautiful pseudorandom generator which produces provably good random bits, but only for algorithms *which use a small amount of space*. Streaming algorithms meet that requirement, so we can simply use Nisan's method to regenerate the matrix L as necessary. Unfortunately, we do not have time to discuss the details.

2 Nearest Neighbor

The nearest neighbor problem is a classic problem involving high-dimensional data. Given points $P = \{p_1, ..., p_m\} \in \mathbb{R}^n$, preprocess P so that, given a query point $q \in \mathbb{R}^n$, we can quickly find i minimizing $\|q - p_i\|$. As usual we focus on the Euclidean norm, but this problem is interesting for many norms.

This problem can trivially be solved in polynomial time. We could do no processing of P, then for each query find the closest point by exhaustive search. This requires time $O(nm)$ for each query. An alternative approach is to use a kd-tree, which is a well-known data structure for representing geometric points. Unfortunately this could take $O(nm^{1-1/n})$ time for each query, which is only a substantial improvement over exhaustive search when the dimension n is a constant. This phenomenon, the failure of low dimensional methods when applied in high dimensions, is known as the "curse of dimensionality".

We will present an improved solution, by allowing the use of randomization and approximation. We will instead solve the **ϵ-approximate nearest neighbor** problem. Given a query point $q \in \mathbb{R}^n$, we must find a point $p \in P$ such that

$$\|p - q\| \ \leq\ (1+\epsilon) \cdot \|p' - q\| \qquad \forall p' \in P.$$

Our solution is based on a reduction to a simpler problem, the **ϵ-Point Location in Equal Balls** problem. The input data is a collection of m balls of radius r, centered at points $P = \{p_1, \ldots, p_m\} \subset \mathbb{R}^n$.

Let $B(p, r)$ denote the ball of radius r around p. Given a query point $q \in \mathbb{R}^n$ we must answer the query as follows:

- If there is any p_i with $q \in B(p_i, r)$, we must say Yes, and we must output any point p_j with $q \in B(p_j, (1+\epsilon)r)$.
- If there is no point p_i with $q \in B(p_i, (1+\epsilon)r)$, we must say No.
- Otherwise (meaning that the closest p_i to q has $r \leq \|p_i - q\| \leq (1+\epsilon)r$), we can say either Yes or No. (As before, if we say Yes we must also output a point p_j with $q \in B(p_j, (1+\epsilon)r)$.)

Let us call this problem PLEB(r).

In other words, let us call a ball of radius r a "green ball" and a ball of radius $(1+\epsilon)r$ a "red ball". If q is contained in any green ball, we must say Yes and output the center of any red ball containing q. If q is not contained in any red ball, we must say No. Otherwise, we could say either Yes or No, but in the former case we must again output the center of any red ball containing q.

2.1 Reducing Approximate Nearest Neighbor to PLEB

We now explain how to solve the **ϵ-approximate nearest neighbor** problem using any solution to the **ϵ-Point Location in Equal Balls** problem. Scale the point set P so that the minimum interpoint distance is at least 1, then let R be the maximum interpoint distance. So $1 \leq \|p - p'\| \leq R$ for all $p, p' \in P$. For every radius $r = (1+\epsilon)^0, (1+\epsilon)^1, \ldots, R$, we initialize our solution to PLEB(r). Given any query point q, we use binary search to find the minimum r for which PLEB(r) says Yes. Let $p \in P$ be the point that it returns.

The requirements of PLEB(r) guarantee that $\|p-q\| \leq r(1+\epsilon)$. On the other hand, since PLEB($r/(1+\epsilon)$) said No, we know that there is no point $p' \in P$ with $\|p' - q\| \leq r/(1+\epsilon)$. Thus p satisfies

$$\|p - q\| \;\leq\; r(1+\epsilon) \;\leq\; (1+\epsilon)^2 \cdot \|p' - q\| \qquad \forall p' \in P.$$

And so this gives a solution to ϵ-approximate nearest neighbor, with a slightly different ϵ.

2.2 Solving PLEB

The main idea here is quite simple. We discretize the space, then use a hash table to identify locations belonging to a ball.

Preprocessing. In more detail, the preprocessing step for PLEB(r) proceeds as follows. We first partition the space into cuboids (n-dimensional cubes) of side length $\epsilon r/\sqrt{n}$. Note that the diameter of a cuboid is its side length times $\sqrt{n}$, which is ϵr. Each cuboid is identified by a canonical point, say the minimal point contained in the cuboid. We then create a hash table, initially empty. For each point p_i and each cuboid C that intersects $B(p_i, r)$, we insert the (key, value) pair (C, p_i) into the hash table.

Queries. Now consider how to perform a query for a point q. The first step is to determine the cuboid C that contains q, by simple arithmetic. Next, we look up C in the hash table. If there are no matches, that means that no ball $B(p_i, r)$ intersects C, and therefore q is not contained in any ball of radius r (a green ball). So, by the requirements of PLEB(r), we can say No.

Suppose that C is in the hash table. Then the hash table can return us an arbitrary pair (C, p_j), which tells us that $B(p_j, r)$ intersects C. By the triangle inequality, the distance from p_j to q is at most r plus

the diameter of the cuboid. So $\|p_j - q\| \leq (1+\epsilon)r$, i.e., q is contained in the red ball around p_j. By the requirements of PLEB(r), we can say Yes and we can return the point p_j.

Time and Space Analysis. To analyze this algorithm, we first need to determine the number of cuboids that intersect a ball of radius r. The volume of a ball of radius r is approximately $2^{O(n)}r^n/n^{n/2}$. On the other hand, the volume of a cuboid is $(\epsilon r/\sqrt{n})^n$. So the number of cuboids that intersect this ball is roughly

$$\frac{2^{O(n)}r^n/n^{n/2}}{(\epsilon r/\sqrt{n})^n} \;=\; O(1/\epsilon)^n.$$

Therefore the time and space used by the preprocessing step is $O(1/\epsilon)^n$.

To perform a query, we just need to compute the cuboid containing q then look up that cuboid in the hash table. This takes $O(n)$ time, which is optimal, since we must examine all coordinates of the vector q.

Unfortunately the preprocessing time and space is exponential in n, which is terrible. The curse of dimensionality has struck again! The next section gives an improved solution.

2.3 Approximate Nearest Neighbor by Johnson-Lindenstrauss

Our last key observation is, by applying the Johnson-Lindenstrauss lemma, we can assume that our points lie in a low-dimensional space. Specifically, we can randomly generate a matrix $d \times n$ which maps our point set to $\mathbb{R}^d$ with $d = O(\log(m)/\epsilon^2)$, while approximately preserving distances between all points in P with high probability. That same map will also approximately preserve distances between any query points and the points in P, as long as the number of queries performed is at most poly(m).

The analysis of PLEB changes as follows. The preprocessing step must apply the matrix to all points in P, which takes time $O(nmd)$. The time to set up the hash table improves to $O(1/\epsilon)^d = n^{O(\log(1/\epsilon)/\epsilon^2)}$. So assuming ϵ is a constant, the preprocessing step runs in polynomial time. Each query must also apply the Johnson-Lindenstrauss matrix to the query point, which takes time $O(dn) = O(n\log(n)/\epsilon^2)$,

Finally, we analyze the reduction which allowed us to solve Approximate Nearest Neighbor. The preprocessing step simply initializes PLEB(r) for all values of R, of which there are $\log_{1+\epsilon} R = O(\log(R)/\epsilon)$. So the total preprocessing time is

$$n^{O(\log(1/\epsilon)/\epsilon^2)} \;\cdot\; O(\log(R)/\epsilon),$$

which is horrible, but polynomial time assuming R is reasonable. Each query must perform binary search to find the minimum radius r for which PLEB(r) says Yes, so the total query time is

$$O\big(nd\big) + O\Big(\log\big(\log(R)/\epsilon\big) + \log(1/\epsilon)\Big) \cdot O(d).$$

Assuming R is reasonable, this is $O(n\log(n)/\epsilon^2)$.