CPSC 536N: Randomized Algorithms 2014-15 Term 2

Lecture 9

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 ***frequency vectors***. The data is a sequence $(i_1, i_2, \ldots, i_n)$ of indices, where each $i_k \in \{1, \ldots, d\}$. At an abstract level, the goal is to maintain the frequency vector $x \in \mathbb{Z}^d$, where

$$x_j \;=\; |\{\; k \;:\; i_k = j \;\}|,$$

and then to output some properties of x, such as a norm $\|x\|_p$, or the number of non-zero entries, etc. If the algorithm were to maintain x explicitly, it could initialiaze $x \leftarrow 0$, then at each time step k, it receives the index i_k and increments x_{i_k} by 1. Given this explicit representation of x, one can easily compute the desired properties.

So far the problem is trivial. The algorithm can explicitly store the frequency vector x, or even the entire sequence $(i_1, i_2, \ldots)$, and compute any desired function of those objects. What makes the model interesting is: our goal is that

$$\text{the algorithm should use } O(\log(n)) \text{ words of space.} \tag{1}$$

This rules out the trivial solutions.

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 $((a_1, b_1), \ldots, (a_n, b_n))$. 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 frequency vector 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 $t \times d$ matrix whose entries are drawn independently from the distribution $N(0, 1/t)$. (This is the same as the linear map L defined in Lecture 7.) The algorithm will explicitly maintain the vector y, defined as $y := L \cdot x$. At time step k, the algorithm receives the index $j = i_k$ so (implicitly) the j^{th} 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 Lecture 7 imply that

$$\Pr\left[\,(1-\epsilon)\|x\| \;\le\; \|y\| \;\le\; (1+\epsilon)\|x\|\,\right] \;\ge\; 1 - \exp(-\Omega(\epsilon^2 t)).$$

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

How much space? The main object being updated by the algorithm is the vector y. This vector consumes $t = O(\log(n)/\epsilon^2)$ words of space, so have we achieved the goal (1)?

The issue is that updating the vector y requires the matrix L. How much space does it take to represent L?

- If one were not thinking carefully, one might think that, since every entry of L is an independent random variable, it needn't be stored at all: every time an entry of L is accessed, a new independent random variable could be generated. But this does not work: when mapping two different points to low-dimensional space, we must use the *same matrix* to map both points.
- The algorithm could store L explicitly, but would require td words of space, which is worse than the trivial solution of storing x explicitly! 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.
- Does it help to use the Fast Johnson-Lindenstrauss transform $L = SHD$ introduced in Lecture 8? Storing this L requires only $O(t)$ words to store the non-zero entries of S, which is good, but it requires $O(d)$ bits to store the diagonal entries of D, which is still too much.
- If the entries of L are generated by a pseudorandom generator then perhaps there is a way to save space? 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. This is likely to work well in practice, but may not have provable guarantees.
- Theorists would advocate the following 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*. That is precisely the goal of streaming algorithms, 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.
- Another approach advocated by theoreticians is to generate L using special randomized hash functions. We will discuss such hash functions in future lectures.

2 Nearest Neighbor

The nearest neighbor problem is a classic problem involving high-dimensional data. Given points $P = \{p_1, ..., p_n\} \in \mathbb{R}^d$, the goal is to build a (static) data structure so that, given a query point $q \in \mathbb{R}^d$, we can quickly find i minimizing $\|q - p_i\|$. We focus on the Euclidean norm $\|\cdot\| = \|\cdot\|_2$, but this problem is interesting for many norms.

Trivial solutions. 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 $O(nd)$ 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 $O(dn^{1-1/d})$ time for each query, which is only a substantial improvement over exhaustive search when the dimension d is a constant. This phenomenon, the failure of low dimensional methods when applied in high dimensions, is known as the "curse of dimensionality".

Overcoming the curse. In 1997-98, papers by Kleinberg, Indyk-Motwani and Kushilevitz-Ostrovsky-Rabani showed that the curse can be overcome through the use of randomization and approximation. The approximate form of the problem is defined as follows. Given a query point $q \in \mathbb{R}^d$, we must find a point $p \in P$ such that

$$\|p - q\| \;\leq\; (1+\epsilon) \cdot \min_{p' \in P} \left\|p' - q\right\|. \tag{2}$$

This is called the ***ϵ-approximate nearest neighbor*** problem (ϵ-NN).

Our goal is to preprocess P and and produce a data structure of size $\text{poly}(n)$. Given a query point q, we wish to spend time $\text{poly}(d, \log(n), 1/\epsilon)$ finding a point $p \in P$ satisfying (2).

The high-level idea of our solution is very simple. First, design a brute-force algorithm that requires space roughly $2^{O(d)}$ to solve ϵ-NN for d-dimensional data. Then, apply dimensionality reduction to reduce the data set to dimension $t \approx \log n$. Running the brute-force algorithm on the t-dimensional data only requires space $2^{O(d)} = n^{O(1)}$.

2.1 Point Location in Equal Balls

The first step is to reduce our problem to a simpler one, in which a query only needs to determine whether the closest point is at distance less than or greater than roughly r. This simpler problem is called the ***ϵ-Point Location in Equal Balls*** problem (ϵ-PLEB). It is defined as follows.

The input data is a collection of n balls of radius r, centered at points $P = \{p_1, \ldots, p_n\} \subset \mathbb{R}^d$. Let $B(p, r)$ denote the Euclidean ball of radius r around p. Given a query point $q \in \mathbb{R}^d$ we must answer the query as follows:

- If there is any $p_i \in P$ 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 $r < \|p_i - q\| \leq (1+\epsilon)r$), 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 ϵ-PLEB(r).

In other words, let us call a ball of radius r a "small ball" and a ball of radius $(1+\epsilon)r$ a "big ball". If q is contained in any small ball, we must say Yes and output the center of any big ball containing q. If q is not contained in any big 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 big ball containing q.

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

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

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

And so this gives a solution to the ϵ-NN problem, 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 ϵ-PLEB(r) proceeds as follows. We first partition the space into cuboids (d-dimensional cubes) of side length $\epsilon r/\sqrt{d}$. Note that the diameter of a cuboid is its side length times $\sqrt{d}$, 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 small 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 $\|p_j - q\| \le (1+\epsilon)r$, i.e., q is contained in the big 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 roughly $2^{O(d)} r^d / d^{d/2}$. On the other hand, the volume of a cuboid is $(\epsilon r/\sqrt{d})^d$. So the number of cuboids that intersect this ball is roughly

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

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

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

Unfortunately the preprocessing time and space is exponential in d, 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 $t \times d$ which maps our point set to $\mathbb{R}^t$ with $t = O(\log(n)/\epsilon^2)$, 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(n).

The analysis of PLEB changes as follows. The preprocessing step must apply the matrix to all points in P, which takes time $O(dnt)$. The time to set up the hash table improves to $O(1/\epsilon)^t = n^{O(\log(1/\epsilon)/\epsilon^2)}$. 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 $O(td) = O(d \log(n)/\epsilon^2)$,

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 and ϵ is a constant. Each query must perform binary search to find the minimum radius r for which PLEB(r) says Yes, so the total query time is

$$O\big(dt\big) + O\Big(\log\big(\log(R)/\epsilon\big) + \log(1/\epsilon)\Big) \cdot O(t).$$

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

2.4 Discussion

This nearest neighbor algorithm seems terribly inefficient, but it is not the last word on the subject. Even the original Indyk-Motwani paper suggested using a different approach known as "locality sensitive hashing". The state-of-the art is described in a recent survey of Andoni and Indyk.