CS265/CME309: Randomized Algorithms and Probabilistic Analysis
Lecture #8: Dimension Reduction and Nearest Neighbor Search

Gregory Valiant[*], updated by Mary Wootters

February 15, 2022

1 Dimension Reduction

Previously, we saw that any metric (X, d) with $|X| = n$ can be embedded into $R^{O(\log^2 n)}$ under any the ℓ_1 metric (actually, the same embedding works for any ℓ_p metic), with distortion $O(\log n)$. Here, we describe an extremely useful approach for reducing the dimensionality of a Euclidean (ℓ_2) metric, while incurring very little distortion. Such dimension reduction is useful for a number of reasons: on the practical side, many geometric algorithms have runtimes that scale poorly with the dimension of the space in which they operate, so being able to reduce the dimension while preserving geometry can be useful. Moreover, these dimension-reduction procedures have been used numerous times as components within other algorithms (e.g. Locality Sensitive Hashing). From a theoretical side, the JL lemma has plenty of applications throughout theoretical computer science and mathematics, in particular to understanding high-dimensional geometry.

2 Johnson-Lindenstrauss Transformation

The randomized dimension reduction approach is essentially due to Johnson and Lindenstrauss in 1984 [7], and many variants (and de-randomizations) have been explored in the past 30 years.

Theorem 1. *Given any $\epsilon \in (0, 1)$, and a set $X \subset \mathbb{R}^d$ with $|X| = n$, there exists a linear map $f : \mathbb{R}^d \to \mathbb{R}^m$ with $m = O(\frac{\log n}{\epsilon^2})$ that embeds (X, ℓ_2) into $(\mathbb{R}^m, \ell_2)$ with distortion at most $(1 + \epsilon)$.*

Proof. We will define a *randomized* map, and show that it has the desired property with high probability. More precisely, we'll first show that for any fixed $x, y \in X$, A approximately the preserves the distance between x and y with probability $o(1/n^2)$, and then we will take a union bound over all pairs of x and y in X.

[*]

Let A be an $m \times d$ matrix with entries chosen independently from $N(0, 1/m)$, and define the map $f : \mathbb{R}^d \to \mathbb{R}^m$ by $f(x) = Ax$. Hence each of the m coordinates of $f(x)$ are given by the projection of x onto a d-dimensional Gaussian.

One useful trick in analyzing such Gaussian projections is the spherical symmetry of the Gaussian. Hence, to analyze the distortion, for a given pair of vectors x, y, we could imagine rotating the coordinate system so that $x - y$ is a standard basis vector, e.g., $x - y = (1, 0, 0, 0, \ldots, 0)$. Such a rotation does not change the distribution, and simplifies the analysis. (Exercise: prove this by leveraging the fact that sums of Gaussians are Gaussian...)

Given that $x - y$ is a standard basis vector, $A(x - y)$ only depends on one row of A, and hence $||f(x) - f(y)||_2^2 = ||x - y||_2^2 \sqrt{\sum_{i=1}^m X_i^2}$, where the X_i's are independent Gaussians of variance $1/m$. Since the expected square of a zero-mean Gaussian is its variance, $\mathbf{E}[||f(x) - f(y)||_2^2] = ||x - y||_2^2$. Hence the theorem will follow provided we show that $\sum X_i^2$ is sufficiently tightly concentrated about its expectation. Specifically, if this is within a $(1 + \epsilon)$ factor of its expectation, namely $\|f(x) - f(y)\|_2^2 = (1 \pm \epsilon)\|x - y\|_2^2$, then that will imply that $\|f(x) - f(y)\|_2 = (1 \pm \epsilon)\|x - y\|_2$. [The reason we are analyzing this squared norm instead of directly analyzing the actual distance is that the square-root would complicate the analysis.]

The probability that $\sum_{i=1}^m X_i^2$ is concentrated within a factor of $(1 \pm \epsilon)$ of its expectation is identical to the probability that $\sum_{i=1}^m Z_i$ is within $(1 \pm \epsilon)$ of its expectation, where Z_i is drawn from $N(0, 1)$, so we will analyze this slightly simpler expression. To do this, we will prove a Chernoff-style bound, leveraging the easily verified fact that for $t < 1/2$, $\mathbf{E}[e^{tZ^2}] = 1/\sqrt{1 - 2t}$ for $Z \sim N(0, 1)$. We begin by bounding the probability that $\sum_i Z_i^2 > (1 + \epsilon)m$; a similar argument will show an analogous bound for $\Pr[\sum Z_i^2 < (1 - \epsilon)m]$.

$$\begin{aligned}
\Pr[\sum Z_i^2 > (1+\epsilon)m] &= \Pr[e^{t \sum Z_i^2} > e^{t(1+\epsilon)m}], \text{ [for } t > 0] \\
&\leq \frac{1/(1-2t)^{m/2}}{e^{t(1+\epsilon)m}} \text{ [by Markov's inequality, for } t \in (0, 1/2)] \\
&= e^{-m(t(1+\epsilon)+(1/2)\log(1-2t))}.
\end{aligned}$$

Using the fact that $\log(1 - 2t) > -2t - 4t^2$ (for $0 < t < \frac{1}{3}$), the above probability is at most $e^{-m\left(t\epsilon - 2t^2\right)}$, for every $t \in (0, 1/3)$. Optimizing this quadratic function $t\epsilon - 2t^2$ for t yields $t = \epsilon/4$. Plugging this in and simplifying yields that the above probability is at most $e^{-m\frac{\epsilon^2}{8}}$. Hence by choosing $m > \frac{17 \log n}{\epsilon^2}$, this probability is $o(1/n^2)$, and hence we may perform a union bound over all $< n^2/2$ possible pairs of points x, y to argue that with constant probability, the embedding does not significantly distort any of the $O(n^2)$ distances. □

3 Fancier Johnson-Lindenstrauss Transformations

The original paper of Johnson and Lindenstrauss actually considered a uniformly random orthogonal projection: that is, choose a uniformly random m-dimensional subspace, and let the rows of A be an orthogonal basis for that subspace. The version presented above, with i.i.d. Gaussians, is a bit faster to compute. In fact, an even easier version with random ±1 random variables instead of the Gaussians will work too (see, e.g., [1]).

However, there are even more nicely structured Johnson-Lindenstrauss Transformations out there. In this section, we'll give a quick summary of some of the cool work that's been done recently in this area.

The goal of this section is just to give you an idea of what's out there and some pointers to the literature—don't worry about the technical details of these results unless you are interested in them.

3.1 Recent Advances: Fast Johnson-Lindenstrauss Transformations

To achieve distortion ϵ for a set of n points in $\mathbb{R}^d$, using the above scheme would require time $O(dm) = O(\frac{d\log n}{\epsilon^2})$ to compute the embedding of each datapoint. Can we hope to speed this up?

The most naive hope would be that there is a variance of the Johnson-Lindenstrauss scheme in which the random projection matrix, A, can be chosen to be sparse—if most of the entries are 0, then we can multiply by A in time proportional to the number of nonzero entries, instead of time (which is less than dm if A is sparse). Unfortunately, the guarantees of the theorem erode for sparse A, and this won't work.

However, in 2006 Ailon and Chazelle introduced the *Fast Johnson-Lindenstrauss* which is a clever twist on this. Roughly, instead of multiplying by a sparse A, we transform our data into the Fourier basis (recall that the Fourier transform is just multiplication by a $d \times d$ matrix, which has the property that this can be computed in time $O(d\log d)$), and then multiply by our sparse A. This corresponds to multiplying by a dense A in the standard basis, and the analysis works out (intuitively, as long as A is a bit random, and not sparse, then things are okay). This ends up with runtime $O(d\log d + \frac{polylogn}{\epsilon^2})$, which is much better than $O(d\log n/\epsilon^2)$ in the many cases when $\frac{\log n}{\epsilon^2} \gg \log d$. Feel free to check out the original paper [2]—as well as some follow-up papers [3, 9]—for more details; the state-of-the-art[1] gives a matrix with $O(d\log d)$ embedding time with a target dimension of $O(\log n \text{ polylog } (d)/\epsilon^2)$.

3.2 Recent Advances: Sparse Johnson-Lindenstrauss Transformations

The Johnson-Lindenstrauss transform we saw in Section 2, as well as the fast ones in Section 3.1, are dense: lots of the entries are non-zero. In many applications, it can be helpful if the transform is *sparse.* This both speeds up the time it takes to apply the transformation, and also enables extremely efficient sparse updates. For example, suppose you have already computed Ax, but now you want to update the i'th coordinate of x, say $x' \leftarrow x + c \cdot e_i$, where $c \in \mathbb{R}$ and where e_i is the i'th standard basis vector. How quickly can you compute the updated Ax'? If A is sparse, you just have to add c times the i'th column of A—which is a sparse vector—to the vector Ax you have already computed.

A *sparse* Johnson-Lindenstrauss transform was given by Dagupta, Kumar and Sarlós in 2010 [5]. The approach is based on random hash matrices: the first idea is to consider a matrix $H \in \{0, \pm 1\}^{m\times d}$ that is a random hash matrix: each row corresponds to a bucket, and each column column corresponds to an item. Each column of the matrix has exactly one nonzero in a random location, which is ± 1 randomly. It turns out that this construction won't work as a JL transform unless the target dimension is pretty big. (Why?) So instead, the idea is to start with $\tilde{O}(1/\epsilon)$ such hash matrices, and add them together. This turns out to do the trick. This construction was later derandomized by Kane and Nelson [8].

[1] The last time I checked...

3.3 Recent Advances: Stronger Johnson-Lindenstrauss Transformations

Recall the theorem we proved above, slightly restated:

Theorem 2. *Given any $\epsilon \in (0,1)$, and a set $X \subset \mathbb{R}^d$ with $|X| = n$, there exists a map $f : \mathbb{R}^d \to \mathbb{R}^m$ with $m = O(\frac{\log n}{\epsilon^2})$ such that for all $x, y \in X$,*

$$\|x - y\|_2 \leq \|f(x) - f(y)\|_2 \leq (1+\epsilon)\|x - y\|_2.$$

A sequence of recent works, culminating in the 2019 paper [10] established that an analog of the above theorem holds, where we replace "for all $x, y \in X$", with "for all $x \in X$ and for all $y \in \mathbb{R}^d$. Hence there is a mapping that preserves the distances from *any* point in $\mathbb{R}^d$ to any point in X. The construction of this mapping is much more complicated than the random projection that we analyzed above, and, in particular, the mapping is non-linear!

4 Introduction to Nearest Neighbor Search

A useful primitive in many data analysis and machine learning algorithms is the ability to efficiently find similar data points to a given point of interest. For example, given a the text from a webpage, or an image, the ability to quickly figure out similar documents or images seems extremely useful. (Of course, one must make sure that the comparison metric and the feature space we are looking in is appropriate.) This problem is known as "nearest neighbor search".

More formally, the set-up is the following. We have a set S of n vectors $S = \{x_1, \ldots, x_n\} \subset \mathbb{R}^d$, and we want to make some data structure, which ideally doesn't take space much more than $O(nd)$, the amount of space needed to represent the n input points, and which can efficiently handle queries of the form "which point x_i is closest to y?" for $y \in \mathbb{R}^d$.

Naively, we can have a data structure that takes space $O(nd)$ and query time $O(nd)$, just by storing all the points and doing a linear scan through them to find the closest point to y. However, for small d, we can do much better. For example, if $d = 1$, we can have space $O(n)$ and query time $O(\log n)$, by storing the x_i in sorted order (this makes sense for one-dimensional data) and just doing binary search. However, this idea turns out not to scale so well when d grows. For example, you can get algorithms that run in time $\mathrm{poly}(d) \cdot \log n$, which is great, but the space scales like $n^{O(d)}$, which is not great.

Can we do better? It turns out that we can if we allow ourselves a little bit of slack. That is, instead of returning the exact nearest neighbor to y, we can try to return an approximate nearest neighbor. Formally, given c, our goal will be to return x_i so that

$$\|x_i - y\|_2 \leq c \cdot \min_j \|x_j - y\|_2.$$

One way to do this is to use a Johnson-Lindenstrauss transform, to "beat" the curse of dimensionality. If we try to do this in the most straightforward way, this does improve things, although it doesn't quite get us to our goal. For example, if we first apply a JL transform to our data $S = \{x_1, \ldots, x_n\}$ (with, say, $\epsilon = 1/2$), then we'll get a data set $\{x'_1, \ldots, x'_n\} \subseteq \mathbb{R}^{d'}$, for $d' = O(\log n)$. If we run a nearest neighbors algorithm on the image (for example, the one mentioned above with query time $\mathrm{poly}(d) \log n$), the query time will be $\mathrm{poly}(d') \cdot \log n = \text{ polylog } (n)$ (great!), and space needed will be $n^{O(d')} = n^{O(\log n)}$. This might be an improvement if $d \gg \log n$, but it's still not the linear-in-n that we were hoping for.

Note: At this point we are done with the material covered in the before-class videos. The following section is for reference after class.

5 Locality Sensitive Hashing

In order to do better for approximate nearest neighbors, we will use something called *locality-sensitive hashing*. Locality sensitive hashing schemes, as their name suggests, are hashing schemes with the property that points that are close have a higher probability of hashing to the same bucket.

The idea is the following. Given a locality-sensitive hashing scheme, which hashes points to other points that are close to them, a nearest-neighbor search can be performed by simply hashing the query vector v, and then checking which vectors are hashed to the same bucket. Since their introduction in the late 1990's in a paper of Indyk and Motwani (from Stanford) [6], there has been a huge body of research describing locality sensitive hashing schemes for various metrics, with various tradeoffs between the various parameters (storage space, preprocessing time, etc.). Despite the volume of research, many of the most basic questions are still open—we currently do not know if the schemes we have are near optimal or not. See [4] for a (only slightly outdated) survey.

Before we describe and analyze such a scheme, we will further relax our problem, to "near" neighbors, rather than "nearest" neighbors. In this version of the problem, we are also given a radius r, and a promise that there exists some $x_i \in S$ (our data set) so that $\|y - x_i\|_2 \leq r$. The goal is to find some $x_j \in S$ so that $\|y - x_j\|_2 \leq c \cdot r$, for some approximation factor c.

Notice that if we can solve c-approximate-near-neighbors, then we can solve c-approximate nearest neighbors, up to some additive error. Indeed, suppose that we know that S has radius R (let's say that $R = 1$ after re-scaling things appropriately). Then we can solve the c-approximate-near-neighbor problem for $r = \epsilon$, $r = 2\epsilon$, $r = 4\epsilon$, $r = 8\epsilon$, and so on, up to $r = 1$. This is an extra $\log(1/\epsilon)$ steps, and we'll always find some $x_j \in S$ so that

$$\|y - x_j\|_2 \leq \max\{2c \cdot \min_i \|y - x_j\|_2, c\epsilon\}.$$

(Why?) Note that this doesn't quite solve the c-approximate nearest neighbors problem because of this $c\epsilon$ term, but by making ϵ sufficiently small we can effectively solve the problem.

Thus, we focus on the c-approximate near-neighbor problem, for $r = \epsilon$. (It's only going to geteasier as r gets larger, so let's look at the case where r is as small as it's going to get).

We now explore an extremely simple locality sensitive hashing scheme, based on the Johnson-Lindenstrauss transformation, that illustrates some of the properties and intuitions of more complex schemes:

Algorithm 1. RANDOM HYPERPLANE HASHING

Input: n points in d dimensions, integer s representing the number of hash tables we will construct, and an integer k representing the length of each hash.

- *Pick s matrices of dimension $k \times d$, denoted $A_1 \ldots, A_s$, by drawing each entry of A_i independently from $N(0,1)$.*
- *For every point x, hash it to each of the s hash tables as follows: for $i \in [1, \ldots, s]$, set x's ith hash to be the length k vector $sign(A_i x)$ whose jth index is ±1 according to the sign of the jth coordinate of the vector $A_i x$.*

First we argue that the points that get hashed to the same bucket as x will tend to have a small angle with x:

Claim 2. *For $x, y \in \mathbb{R}^d$, the probability that they are hashed to the same bucket in the ith hash table is $(1 - \frac{angle(x,y)}{\pi})^k$, where $angle(x, y)$ denotes the angle, in radians between the vectors x and y.*

Proof. Consider the jth index of the hash of x and y. The entries corresponding to x and y will agree if, and only if points x and y lie on the same side of the hyperplane defined by the the jth row of matrix A. Because of the spherical symmetry of the d dimensional Gaussian, the probability that this random hyperplane splits the points x and y is exactly equal to the probability that a random line in the 2-dimensional plane spanned by x, y, passing through the origin, splits points x and y. This probability is exactly $angle(x, y)/\pi$. The claim now follows from the independence of the k coordinates. □

Hence for $x, y \in \mathbb{R}^d$, the probability that they are not hashed to the same bucket in any of the s hash tables is approximately

$$(1 - (1 - \frac{angle(x, y)}{\pi})^k)^s \approx e^{-s \cdot e^{-k \cdot angle(x,y)/\pi}}.$$

To see the implications of this statement, consider setting $k = \frac{\pi \log n}{2\epsilon}$, and $s = \sqrt{n}$. For this setting of parameters, if $angle(x, y) \leq \epsilon$, then the probability that they will hash to the same bucket in at least one of the hash tables is roughly

$$1 - e^{-s \cdot e^{-k \cdot angle(x,y)/\pi}} \geq 1 - e^{-s/\sqrt{n}} = 1 - 1/e > 1/2.$$

On the other hand, if $angle(x, y) \geq 5\epsilon$, then the probability that they will hash to the same bucket in at least one of the hash tables is roughly

$$1 - e^{-s \cdot e^{-k \cdot angle(x,y)/\pi}} \leq 1 - e^{-sn^{5/2}} \approx 1/n^2.$$

Hence, if we have hashed n points, via a union bound, with constant probability, *no* pair x, y with $angle(x, y) > 5\epsilon$ will collide in any of the s hash tables.

The above reasoning shows that this hashing approach will allow us to construct a set of hash tables with the following properties. Suppose we have hashed a set of points $S = \{x_1, \ldots, x_n\}$.

- Given a point $y \in \mathbb{R}^d$, it takes time $O(\frac{d\sqrt{n}\log n}{\epsilon})$ to hash y.
- If there exists some $x \in S$ with $angle(x, y) < \epsilon$, then with constant probability there's some x' so that $angle(x', y) \leq 5\epsilon$, and x' and y are hashed to the same bucket in at least one of the $\sqrt{n}$ hashes.
- For any $x \in S$ with $angle(x, y) \geq 5\epsilon$, it's very unlikely that x will collide with y in any bucket.

If it were the case that $angle(x, y)$ were the distance we cared about, this would give us a 5-approximate near neighbors scheme! (And, as noted above, repeating this $\log(1/\epsilon)$ times for various different values of "r" will basically give us a 10-approximate nearest-neighbors scheme). The scheme is just to check each of the $\sqrt{n}$ hashes; if we every find an x' that lands in y's bucket, return it.

Of course, we care about $\|x-y\|_2$, not $angle(x,y)$. If x and y have norm 1, then it's always true that

$$angle(x,y) \leq \|x-y\|_2 \leq \frac{2}{\pi} angle(x,y).$$

Indeed, $angle(x,y)$, when measured in radians, is just the arc length between x and y. This is always greater than the Euclidean distance between x and y, and the discrepancy is greatest when $\|x-y\|_2 = 2$, and x and y are on the opposite sides of a sphere. In that case the ratio that they are off by is $2/\pi$.

Thus, if we additionally think that our data live on the surface of a sphere (which happens), this gives us an approximate nearest-neighbors scheme. If the data don't live on the surface of a sphere, but live in a bounded region, it turns out that we can project them onto the surface of a large sphere that's one dimension higher, without too much distortion.

The amount of storage needed for this scheme is $n^{O(1)}$. We need $O(nd)$ to store the points, and then for each of $s = \sqrt{n}$ hash functions we need to store the hash tables, each of which have 2^k buckets. That's a total of $O(\sqrt{n}2^{O(\log n/\epsilon)}) = n^{O(1/\epsilon)}$. Since we are assuming that ϵ is a constant, this is $n^{O(1)}$. We also need to store kds random Gaussians to store the matrices A_i, so that's another $O(\epsilon^{-1}d\sqrt{n})$, and assuming $d \leq n$, this is also $n^{O(1)}$. This isn't the linear-in-n that we were hoping for, but it's definitely better than the $O(n^{\log n})$ that we saw in the previous section.

The query time is $O(\epsilon^{-1}d\sqrt{n}\log n)$ to hash y, plus the amount of time it takes to go through the $\sqrt{n}$ buckets that y hashes to to look and see if there's anything there. This takes time $O(\sqrt{n})$ (notice that we don't need to do any more processing if we find something, we just return it without thinking). Thus, the running time is $O(\epsilon^{-1}d\sqrt{n}\log n)$. If d is small (say $d \approx \log n$ or something like that, or even d as large as $n^{0.49}$) then this is sublinear in n!

References

[1] Dimitris Achlioptas. Database-friendly random projections: Johnson-lindenstrauss with binary coins. *Journal of computer and System Sciences*, 66(4):671–687, 2003.

[2] Nir Ailon and Bernard Chazelle. Approximate nearest neighbors and the fast johnson-lindenstrauss transform. In *Proceedings of the thirty-eighth annual ACM symposium on Theory of computing*, pages 557–563, 2006.

[3] Nir Ailon and Edo Liberty. Fast dimension reduction using rademacher series on dual bch codes. *Discrete & Computational Geometry*, 42(4):615, 2009.

[4] Alexandr Andoni and Piotr Indyk. Near-optimal hashing algorithms for approximate nearest neighbor in high dimensions. *COMMUNICATIONS OF THE ACM*, 51(1):117, 2008.

[5] Anirban Dasgupta, Ravi Kumar, and Tamás Sarlós. A sparse johnson: Lindenstrauss transform. In *Proceedings of the forty-second ACM symposium on Theory of computing*, pages 341–350, 2010.

[6] Piotr Indyk and Rajeev Motwani. Approximate nearest neighbors: towards removing the curse of dimensionality. In *Proceedings of the thirtieth annual ACM symposium on Theory of computing*, pages 604–613, 1998.

[7] William B Johnson and Joram Lindenstrauss. Extensions of lipschitz mappings into a hilbert space. *Contemporary mathematics*, 26(189-206):1, 1984.

[8] Daniel M Kane and Jelani Nelson. A derandomized sparse johnson-lindenstrauss transform. *arXiv preprint arXiv:1006.3585*, 2010.

[9] Felix Krahmer and Rachel Ward. New and improved johnson–lindenstrauss embeddings via the restricted isometry property. *SIAM Journal on Mathematical Analysis*, 43(3):1269–1281, 2011.

[10] Shyam Narayanan and Jelani Nelson. Optimal terminal dimensionality reduction in euclidean space. In *Proceedings of the 51st Annual ACM SIGACT Symposium on Theory of Computing*, pages 1064–1069, 2019.