CS168: The Modern Algorithmic Toolbox
Lecture #4: Dimensionality Reduction

Tim Roughgarden & Gregory Valiant*

April 7, 2021

1 The Curse of Dimensionality in the Nearest Neighbor Problem

Lectures #1 and #2 discussed "unstructured data", where the only information we used about two objects was whether or not they were equal. Last lecture, we started talking about "structured data". For now, we consider structure expressed as a (dis)similarity measure between pairs of objects. There are many such measures; last lecture we mentioned Jaccard similarity (for sets), L_1 and L_2 distance (for points in $\mathbb{R}^k$, when coordinates do or do not have meaning, respectively), edit distance (for strings), etc.

How can such structure be leveraged to understand the data? We're currently focusing on the canonical *nearest neighbor* problem, where the goal is to find the closest point of a point set to a given point (either another point of the point set or a user-supplied query). Last lecture also covered a solution to the problem, the k-d tree. This is a good "first cut" solution to the problem when the number of dimensions is not too big — less than logarithmic in the size of the point set.[1] When the number k of dimensions is at most 20 or 25, a k-d tree is likely work well.

Why do we want the number of dimensions to be small? Because of the *curse of dimensionality*. Recall that to compute the nearest neighbor of a query q using a k-d tree, one first does a downward traversal through the tree to identify the smallest region of space that contains q. (Nodes of the k-d tree correspond to regions of space, with the regions corresponding to the children y, z of a node x.) Then one does an upward traversal of the tree, checking other cells that could conceivably contain q's nearest neighbor. The number of cells that have to be checked can scale exponentially with the dimension k.

[1] The trivial baseline for the nearest neighbor problem is brute-force search, where you just compute the distance between the query point and every point in the point set. Brute-force search scales linearly with the size of the point set. For large point sets one would prefer to maintain a data structure so that nearest-neighbor queries can be answered in sub-linear time.

The curse of dimensionality appears to be fundamental to the nearest neighbor problem (and many other geometric problems), and is not an artifact of the specific solution of the k-d tree. For further intuition, consider the nearest neighbor problem in one dimension, where the point set P and query q are simply points on the line. The natural way to preprocess P is to sort the points by value; searching for the nearest neighbor is just binary search to find the interval in which q lies, and then computing q's distance to the points immediately to the left and right of q. What about two dimensions? It's no longer clear what "sorting the point set" means, but let's assume that we figure that out. (The k-d tree offers one approach.) Intuitively, there are now *four* directions that we have to check for a possible nearest neighbor. In three dimensions, there are eight directions to check, and in general the number of relevant directions is scaling exponentially with k. There is no known way to overcome the curse of dimensionality in the nearest neighbor problem, without resorting to approximation (as we do below): every known solution that uses a reasonable amount of space uses time that scales exponentially in k or linearly in the number n of points.

2 Point of Lecture

Why is the curse of dimensionality a problem? The issue is that the natural representation of data is often high-dimensional. Recall our motivating examples for representing data points in real space. With documents and the bag-of-words model, the number of coordinates equals the number of words in the dictionary — often in the tens of thousands. Images are often represented as real vectors, with at least one dimension per pixel (recording pixel intensities) — here again, the number of dimensions is typically in the tens of thousands. The same holds for points representing the purchase or page view history of an Amazon customer, or of the movies watched by a Netflix subscriber.

The friction between the large number of dimensions we want to use to represent data and the small number of dimensions required for computational tractability motivates *dimensionality reduction*. The goal is to re-represent points in high-dimensional space as points in low-dimensional space, preserving interpoint distances as much as possible. We can think of dimensionality reduction as a form of lossy compression, tailored to approximately preserve distances. For an analogy, the count-min sketch (Lecture #2) is a form a lossy compression tailored to the approximate preservation of frequency counts.

Dimensionality reduction enables the following high-level approach to the nearest neighbor problem:

1. Represent the data and queries using a large number k of dimensions (tens of thousands, say).

2. Use dimensionality reduction to map the data and queries down to a smaller number d of dimensions (in the hundreds, say).

3. Compute answers to nearest-neighbor queries in the low-dimensional space.

Provided the dimensionality reduction subroutine approximately preserves all interpoint distances, the answer to the nearest-neighbor query in low dimensions is an approximately correct answer to the original high-dimensional query. Even if the reduced number d of dimensions is still too big to use k-d trees, reducing the dimension still speeds up algorithms significantly. For example, on Mini-Project #2, you will use dimensionality reduction to improve the running time of a brute-force search algorithm, where the running time has linear dependence on the dimension.

The three-step paradigm above is relevant for any computation that only cares about interpoint distances between the points, not just the nearest-neighbor problem. Distance-based clustering is another example.

It should now be clear that we would love to have subroutines for dimensionality reduction in our algorithmic toolbox. The rest of this lecture gives a few examples of such subroutines, and a unified way to think about them.

3 Role Model: Fingerprints

The best way to understand the new concepts in this lecture is via analogy to a hashing-based trick that you already know cold. We review the key idea here; our other examples of dimensionality reduction follow the same template.

Let's return to a world of unstructured abstract data, with no notion of distance between objects other than "equal" vs. "non-equal." The analog of dimensionality reduction is then: how can we re-represent all objects, using fewer bits than before, so that the "distinctness" relation is approximately preserved? This goal is, of course, right in the wheelhouse of hashing.

To maximize overlap with our later dimensionality reduction subroutines, we solve the problem in two steps. Suppose we have n objects from a universe U. Each object requires $\log_2 U$ bits to describe. In the first step, we choose a function h — for example, uniformly at random from a universal family, with range equal to all 32-bit values — and map each object x to

$$f(x) = h(x) \bmod 2. \tag{1}$$

This associates each object with a single bit, 0 or 1 — certainly this is much compressed compared to the original $\log_2 U$-bit representation! The properties of this mapping are:

1. If $x = y$, then $f(x) = f(y)$. That is, the property of equality is preserved.

2. If $x \neq y$ and h is a good hash function, then $\mathbf{Pr}[f(x) = f(y)] \leq \frac{1}{2}$. That is, the property of distinctness is preserved with probability at least 50%.

Achieving error 50% doesn't sound too impressive, but it's easy to reduce it in a second step, via the "magic of independent trials." Repeating the experiment above ℓ times — choosing ℓ different hash functions $h_1, \ldots, h_\ell$ and labeling each object x with ℓ bits $f_1(x), \ldots, f_\ell(x)$ — the properties become:

1. If $x = y$, then $f_i(x) = f_i(y)$ for all $i = 1, 2, \ldots, \ell$.
2. If $x \neq y$ and the h_i's are good and independent hash functions, then $\mathbf{Pr}[f(x) = f(y)] \leq 2^{-\ell}$.

For example, to achieve a user-specified error of $\delta > 0$, we only need to use $\lceil \log_2 \frac{1}{\delta} \rceil$ bits to represent each object. For all but the tiniest values of δ, this representation is much smaller than the original $\log_2 U$-bit one.

4 L_2 Distance and Random Projections

The "fingerprinting" subroutine of Section 3 approximately preserves a 0-1 function on object pairs ("not equal vs. equal"). What if we want to preserve approximately the distances between object pairs? For example, if we want to preserve the L_2 (a.k.a. Euclidean) distance

$$\sqrt{\sum_{i=1}^{k} (x_i - y_i)^2}$$

between points in $\mathbb{R}^k$, what's the analog of a hash function? This section proposes *random projection* as the answer. This idea results in a very neat primitive, the *Johnson-Lindenstrauss (JL) transform*, which says that if all we care about are the Euclidean distances between points, then we can assume (conceptually and computationally) that the number of dimensions is not overly huge (in the hundreds, at most).

4.1 The High-Level Idea

Assume that the n objects of interest are points $\mathbf{x}_1, \ldots, \mathbf{x}_n$ in k-dimensional Euclidean space $\mathbb{R}^k$ (where k can be very large). Suppose we choose a "random vector" $\mathbf{r} = (r_1, \ldots, r_k) \in \mathbb{R}^k$. (See Section 4.3 for details on how the r_i's are chosen.) Define a corresponding real-valued function $f_{\mathbf{r}} : \mathbb{R}^k \mapsto \mathbb{R}$ by taking the inner product of its argument with the randomly chosen coefficients $\mathbf{r}$:

$$f_{\mathbf{r}}(\mathbf{x}) = \langle \mathbf{x}, \mathbf{r} \rangle = \sum_{j=1}^{k} x_j r_j. \tag{2}$$

Thus, $f_{\mathbf{r}}(\mathbf{x})$ is a random linear combination of the components of $\mathbf{x}$. This function will play a role analogous to the single-bit function defined in (1) in the previous section. The function in (1) compresses a $\log_2 U$-bit object description to a single bit; the random projection in (2) replaces a vector of k real numbers with a single real number.

Figure 1 recalls the geometry of the inner product, as the projection of one vector onto the line spanned by another, which should be familiar from your high school training.

If we want to use this idea to approximately preserve the Euclidean distances between points, how should we pick the r_j's? Inspired by our two-step approach in Section 3, we first try to preserve distances only in a weak sense. We then use independent trials to reduce the error.

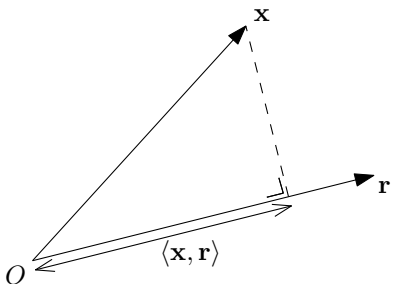

Figure 1: The inner product of two vectors is the projection of one onto the line spanned by the other.

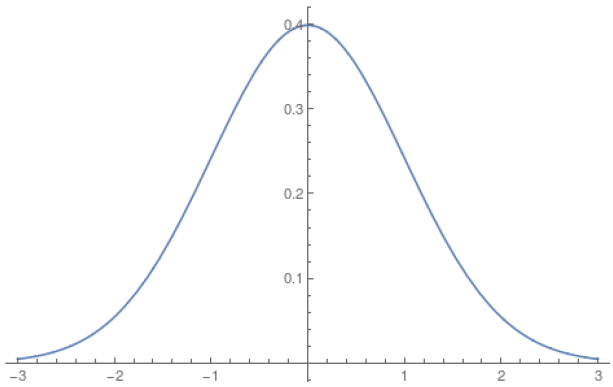

Figure 2: The probability density function for the standard Gaussian distribution (with mean 0 and variance 1).

4.2 Review: Gaussian Distributions

We briefly recall some properties of Gaussian (a.k.a. normal) distributions that are useful for us. Recall the basic shape of the distribution (Figure 2). The distribution is symmetric around its mean μ. Roughly 68% of its mass is assigned to points that are within one standard deviation σ of its mean. Recall that, for any distribution, the square σ^2 of the standard deviation is the variance. A Gaussian distribution is completely and uniquely defined by the values of its mean μ and variance σ^2. The *standard* Gaussian is the special case where $\mu = 0$ and $\sigma = \sigma^2 = 1$.[2]

We're designing our own dimensionality reduction subroutine, and can choose the random coefficients r_i however we want. Why use Gaussians? The nice property we'll exploit here is their closure under addition. Formally, suppose X_1 and X_2 are independent random variables with normal distributions $N(\mu_1, \sigma_1^2)$ and $N(\mu_2, \sigma_2^2)$, respectively. Then, the random variable $X_1 + X_2$ has the normal distribution $N(\mu_1 + \mu_2, \sigma_1^2 + \sigma_2^2)$.[3]

Note that you shouldn't be impressed that the mean of $X_1 + X_2$ equals the sum of the means of X_1 and X_2 — by linearity of expectation, this is true for *any* pair of random

[2]Perhaps you've previously been tortured by the density function $\frac{1}{\sqrt{2\pi}}e^{-x^2/2}$; we won't need this here.

[3]The proof is a few lines of crunching integrals, and can be found in any decent statistics textbook.

variables, even non-independent ones. Similarly, it's not interesting that the variance of $X_1 + X_2$ is the sum of the variances of X_1 and X_2 — this holds for any pair of independent random variables.[4] What's remarkable is that the distribution of $X_1 + X_2$ is a Gaussian (with the only mean and variance that it could possibly have). Adding two distributions from a family generally gives a distribution outside that family. For example, the sum of two random variables that are uniform on $[0, 1]$ certainly isn't uniformly distributed on $[0, 2]$ — there's more mass in the middle then on the ends.

4.3 Step 1: Unbiased Estimator of Squared L_2 Distance

We now return to the random projection function $f_{\mathbf{r}}$ defined in (2), with the random coefficients $r_1, \ldots, r_k$ chosen independently from a standard Gaussian distribution. We next derive the remarkable fact that, for every pair $\mathbf{x}, \mathbf{y}$ of points in $\mathbb{R}^k$, the square of $f_{\mathbf{r}}(\mathbf{x}) - f_{\mathbf{r}}(\mathbf{y})$ is an unbiased estimator of the squared Euclidean distance between $\mathbf{x}$ and $\mathbf{y}$.

Fix $\mathbf{x}, \mathbf{y} \in \mathbb{R}^k$. The L_2 distance between $\mathbf{x}$ and $\mathbf{y}$, denoted $\|\mathbf{x} - \mathbf{y}\|_2$, is $\sqrt{\sum_{j=1}^{k}(x_j - y_j)^2}$. By the definition of $f_{\mathbf{r}}$, we have

$$f_{\mathbf{r}}(\mathbf{x}) - f_{\mathbf{r}}(\mathbf{y}) = \sum_{j=1}^{k} x_j r_j - \sum_{j=1}^{k} y_j r_j = \sum_{j=1}^{k} (x_j - y_j) r_j. \tag{3}$$

Here's where the nice properties of Gaussians come in. Recall that the x_j's and y_j's are fixed (i.e., constants), while the r_j's are random. For each $j = 1, 2, \ldots, k$, since r_j is a Gaussian with mean zero and variance 1, $(x_j - y_j)r_j$ is a Gaussian with mean zero and variance $(x_j - y_j)^2$. (Multiplying a random variable by a scalar λ scales the standard deviation by λ and hence the variance by λ^2.) Since Gaussians add, the right-hand side of (3) is a Gaussian with mean 0 and variance

$$\sum_{j=1}^{k} (x_j - y_j)^2 = \|\mathbf{x} - \mathbf{y}\|_2^2.$$

Whoa — this is an unexpected connection between the output of random projection and the (square of the) quantity that we want to preserve. How can we exploit it? Recalling the definition $\mathrm{Var}(X) = \mathbf{E}[(X - \mathbf{E}[X])^2]$ of variance as the expected squared deviation of a random variable from its mean, we see that for a random variable X with mean 0, $\mathrm{Var}(X)$ is simply $\mathbf{E}[X^2]$. Taking X to be the random variable in (3), we have

$$\mathbf{E}\big[(f_{\mathbf{r}}(\mathbf{x}) - f_{\mathbf{r}}(\mathbf{y}))^2\big] = \|\mathbf{x} - \mathbf{y}\|_2^2. \tag{4}$$

That is, the random variable $(f_{\mathbf{r}}(\mathbf{x}) - f_{\mathbf{r}}(\mathbf{y}))^2$ is an unbiased estimator of the squared Euclidean distance between $\mathbf{x}$ and $\mathbf{y}$.

[4] But this doesn't generally hold for non-independent random variables, right?

4.4 Step 2: The Magic of Independent Trials

We've showed that random projection reduces the number of dimensions from k to just one (replacing each $\mathbf{x}$ by $f_{\mathbf{r}}(\mathbf{x})$), while preserving squared distances in expectation. Two issues are: we care about preserving distances, not squares of distances;[5] and we want to almost always preserve distances very closely (not just in expectation). We'll solve both these problems in one fell swoop, via the magic of independent trials.

Suppose instead of picking a single vector $\mathbf{r}$, we pick d vectors $\mathbf{r}_1, \ldots, \mathbf{r}_d$. Each component of each vector is drawn i.i.d. from a standard Gaussian. For a given pair $\mathbf{x}, \mathbf{y}$ of points, we get d independent unbiased estimates of $\|\mathbf{x} - \mathbf{y}\|_2^2$ (via (4)). Averaging independent unbiased estimates yields an unbiased estimate with less error.[6] Because our estimates in (4) are (squares of) Gaussians, which are very well-understood distributions, one can figure out exactly how large d needs to be to achieve a target approximation (for details, see [3]). The bottom line is: for a set of n points in k dimensions, to preserve all $\binom{n}{2}$ interpoint Euclidean distances up to a $1 \pm \epsilon$ factor, one should set $d = \Theta(\epsilon^{-2} \log n)$.[7]

4.5 The Johnson-Lindenstrauss Transform

Rephrasing and repackaging what we've already done yields the *Johnson-Lindenstrauss (JL) transform*. This was originally a result in pure functional analysis [4], and was ported over to the algorithmic toolbox in the mid-90s [5]. The JL transform is an influential result, and in the 21st century, many variations and improvements have been proposed (see the end of the section).

The JL transform, for domain and range dimensions k and d, is defined using a $d \times k$ matrix $\mathbf{A}$ in which each of the kd entries is chosen i.i.d. from a standard Gaussian distribution. See Figure 3. This matrix defines a mapping from k-vectors to d-vectors via

$$\mathbf{x} \mapsto \tfrac{1}{\sqrt{d}}\mathbf{A}\mathbf{x},$$

where the $1/\sqrt{d}$ scaling factor corresponds to the average over independent trials discussed in Section 4.4.

To see how this mapping $f_{\mathbf{A}}$ corresponds to our derivation in Section 4.4, note that for

[5] The fact that X^2 has expectation μ^2 does not imply that X has expectation μ. For example, suppose X is equally likely to be 0 or 2 and $\mu = \sqrt{2}$.

[6] We'll discuss this idea in more detail next week, but we've already reviewed the tools needed to make this precise. Suppose $X_1, \ldots, X_d$ are independent and all have mean μ and variance σ^2. Then $\sum_{i=1}^d X_i$ has mean $d\mu$ and variance $d\sigma^2$, and so the average $\frac{1}{d}\sum_{i=1}^d X_i$ has mean μ and variance σ^2/d. Thus, averaging d independent unbiased estimates yields yields an unbiased estimate and drops the variance by a factor of d.

[7] The constant suppressed by the Θ is reasonable, no more than 2. In practice, it's worth checking if you get away with d smaller than what is necessary for this theoretical guarantee. For typical applications, setting d in the low hundreds should be good enough for acceptable results. See also Mini-Project #2.

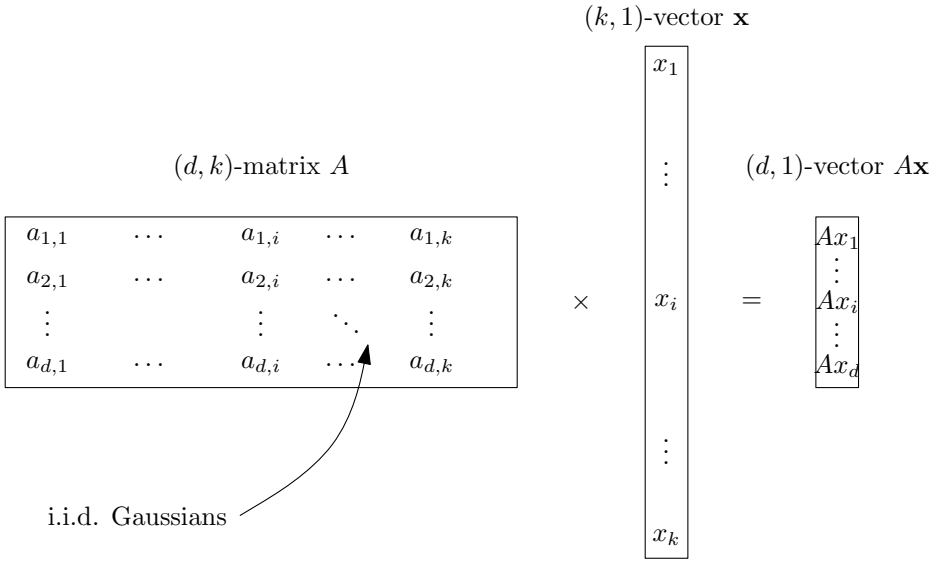

Figure 3: The Johnson-Lindenstrauss tranform A for dimension reduction.

a fixed pair $\mathbf{x}, \mathbf{y}$ of k-vectors, we have

$$\begin{aligned} \|f_{\mathbf{A}}(\mathbf{x}) - f_{\mathbf{A}}(\mathbf{y})\|_2^2 &= \left\| \frac{1}{\sqrt{d}}\mathbf{A}\mathbf{x} - \frac{1}{\sqrt{d}}\mathbf{A}\mathbf{y} \right\|_2^2 && (5) \\ &= \frac{1}{d}\|\mathbf{A}(\mathbf{x}-\mathbf{y})\|_2^2, && (6) \\ &= \frac{1}{d}\sum_{i=1}^{d} \left(a_i^T(\mathbf{x}-\mathbf{y})\right)^2, && (7) \end{aligned}$$

with a_i^T denotes the ith row of $\mathbf{A}$. Since each row a_i^T is just a k-vector with entries chosen i.i.d. from a standard Gaussian, each term

$$\left(a_i^T(\mathbf{x}-\mathbf{y})\right)^2 = \left(\sum_{j=1}^{k} a_{ij}(x_j - y_j)\right)^2$$

is precisely the unbiased estimator of $\|\mathbf{x}-\mathbf{y}\|_2^2$ described in (3) and (4). Thus (5)–(7) is the average of d unbiased estimators. Provided d is sufficiently large, with probability close to 1, all of the low-dimensional interpoint squared distances $\|f_{\mathbf{A}}(\mathbf{x}) - f_{\mathbf{A}}(\mathbf{y})\|_2^2$ are very good approximations of the original squared distances $\|\mathbf{x}-\mathbf{y}\|_2^2$. This implies that, with equally large probability, all interpoint Euclidean distances are approximately preserved by the mapping $f_{\mathbf{A}}$ down to d dimensions.

Thus, for any point set $\mathbf{x}_1, \ldots, \mathbf{x}_n$ in k-dimensional space, and any computation that cares only about interpoint Euclidean distances, there is little loss in doing the computation on the d-dimensional $f_{\mathbf{A}}(\mathbf{x}_i)$'s rather than on the k-dimension $\mathbf{x}_i$'s.

The JL transform is not usually implemented exactly the way we described it. One simplification is to use ± 1 entries rather than Gaussian entries; this idea has been justified both empirically and theoretically (see [1]). Another line of improvement is add structure to the matrix so that the matrix-vector product $\mathbf{Ax}$ can be computed particularly quickly a la the Fast Fourier Transform — this is known as the "fast JL transform" [2]. Finally, since the JL transform can often only bring the dimension down into the hundreds without overly distorting interpoint distances, additional tricks are often needed. One of these is "locality sensitive hashing (LSH)," touched on briefly in Section 6.

5 Jaccard Similarity and MinHash

Section 4 makes the case for using random projections for preserving Euclidean distances. What about other notions of similarity? Several analogs of random projection for other measures are known; we next discuss a representative one.

5.1 The High-Level Idea

A long time ago (mid/late 90s, post-Web but pre-Google) in a galaxy far, far away, there was a Web search engine called Alta Vista. When designing a search engine, an immediate problem is to filter search results to remove near-duplicate documents — for example, two different Web pages that differ only in their timestamps. To turn this into a well-defined problem, one needs a similarity measure between documents. Alta Vista decided to use *Jaccard similarity*. Recall from last lecture that this is a similarity measure for sets: for two sets $A, B \subseteq U$, the Jaccard similarity is defined as

$$J(A,B) = \frac{|A \cap B|}{|A \cup B|}.$$

Jaccard similarity is easily defined for multi-sets (see last lecture); here, to keep things simple, we do not allow an element to appear in a set more than once.

Random projection replaces k real numbers with a single one. So an analog here would replace a set of elements with a single element. The plan is implement a random such mapping that preserves Jaccard similarity in expectation, and then to use independent trials as in Section 4.4 to boost the accuracy.

5.2 MinHash

For sets, the analog of random projection is the *MinHash* subroutine:

1. Choose a permutation π of the universe U uniformly at random.[8]

[8] Just as we use well-chosen hash functions in place of random functions, actual implementations of MinHash use hash functions instead of truly random permutations of U. Unlike many applications of hashing, in this context it makes sense to use a hash function h that maps U back to itself, or perhaps to an

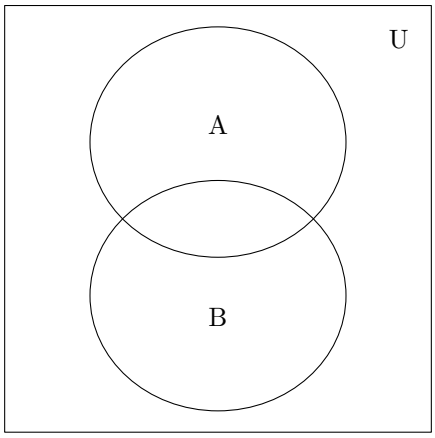

Figure 4: Two set $A, B \subseteq U$.

2. Map each set S to its minimum element $\text{argmin}_{x \in S}\, \pi(x)$ under π.

Thus MinHash "projects" a set to its minimum element under a random permutation.

The brilliance of MinHash is that it gives a remarkably simple unbiased estimator of Jaccard similarity. To see this, consider an arbitrary pair of sets $A, B \subseteq U$ (Figure 4). First, if the smallest (under π) element z of $A \cup B$ lies in the intersection $A \cap B$, then $\text{argmin}_{x \in A}\, \pi(x) = \text{argmin}_{x \in B}\, \pi(x) = z$, so A and B have the same MinHash. Second, if the smallest element z of $A \cup B$ does not lie in $A \cap B$ — say it's in A but not B — then the MinHash of A is z while the MinHash of B is some element strictly larger than z (under π). Thus, A and B have the same MinHash exactly when the smallest element of $A \cup B$ lies in $A \cap B$. Since all relative orderings of $A \cup B$ are equally likely under a random permutation, each element of $A \cup B$ is equally likely to be the smallest. Thus:

$$\mathbf{Pr}[\text{MinHash}(A) = \text{MinHash}(B)] = \frac{|A \cap B|}{|A \cup B|} = J(A, B).$$

As usual, we can boost accuracy through the magic of independent trials. If we want an accurate estimate (up to $\pm\epsilon$) of all $\binom{n}{2}$ Jaccard similarities of pairs of n objects, then crunching some probabilities again shows that averaging $\Theta(\epsilon^{-2} \log n)$ independent estimates is good enough. Accurately estimating all Jaccard similarities is overkill for many applications, which motivates "locality sensitive hashing (LSH)," discussed briefly in the next section.

6 A Glimpse of Locality Sensitive Hashing

Recall from Section 5.1 the motivating application of filtering near-duplicate objects. If we only wanted to filter *exact* duplicates, then there is an easy and effective hashing-based

even larger set to reduce the number of collisions. Collisions add error to the Jaccard similarity estimation procedure, but as long as there are few collisions — as with a good hash function with a large range — the effect is small.

solution:

1. Hash all n objects into b buckets using a good hash function. (b could be roughly n, for example).

2. In each bucket, use brute-force search (i.e., compare all pairs) on the objects in that bucket to identify and remove duplicate objects.[9]

Why is this a good solution? Duplicate objects hash to the same bucket, so all duplicate objects are identified. With a good hash function and a sufficiently large number b of buckets, different objects usually hash to different buckets. Thus, in a given bucket, we expect a small number of distinct objects, so brute-force search in a bucket does not waste much time comparing objects that are distinct and in the same bucket due to a hash function collision.

Naively extending this idea to filter *near*-duplicates fails utterly. The problem is that two objects x and x' that are almost the same (but still distinct) are generally mapped to unrelated buckets by a good hash function. To extend duplicate detection to near-duplicate detection, we want a function h such that, if x and x' are almost the same, then h is likely to map x and x' to the same bucket. This is the idea behind *locality sensitive hashing (LSH)*.

(Additional optional material, not covered in lecture, to be added.)

References

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

[2] N. Ailon and B. Chazelle. Faster dimension reduction. *Communications of the ACM*, 53(2):97–104, 2010.

[3] S. Dasgupta and A. Gupta. An elementary proof of a theorem of Johnson and Lindenstrauss. *Random Structures and Algorithms*, 22(1):60–65, 2003.

[4] W. B. Johnson and J. Lindenstrauss. Extensions of Lipschitz mappings into a Hilbert space. In *Conference in modern analysis and probability*, pages 189–206, 1984.

[5] Y. Rabinovich N. Linial, E. London. The geometry of graphs and some of its algorithmic applications. *Combinatorica*, 15(2):215–245, 1995.

[9] It's often possible to be smarter here, for example if it's possible to sort the objects.