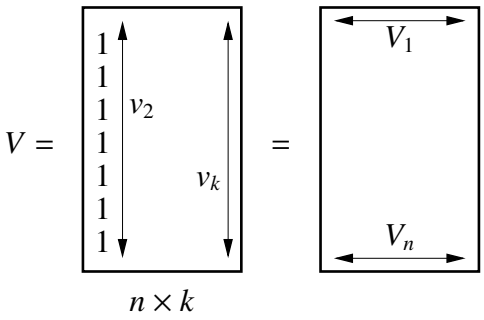
24 Multiple Eigenvectors; Latent Factor Analysis; Nearest Neighbors

Clustering w/Multiple Eigenvectors

[When we use the Fiedler vector for spectral graph clustering, it tells us how to divide a graph into two graphs. If we want more than two clusters, we can use divisive clustering: we repeatedly cut the subgraphs into smaller subgraphs by computing their Fiedler vectors. However, there are several other methods to subdivide a graph into k clusters in one shot that use multiple eigenvectors rather than just the Fiedler vector v_2. These methods are usually faster and sometimes give better results. They use k eigenvectors in a natural way to cluster a graph into k subgraphs.]

For k clusters, compute first k eigenvectors $v_1 = \mathbf{1}, v_2, \ldots, v_k$ of generalized eigensystem $Lv = \lambda Mv$.

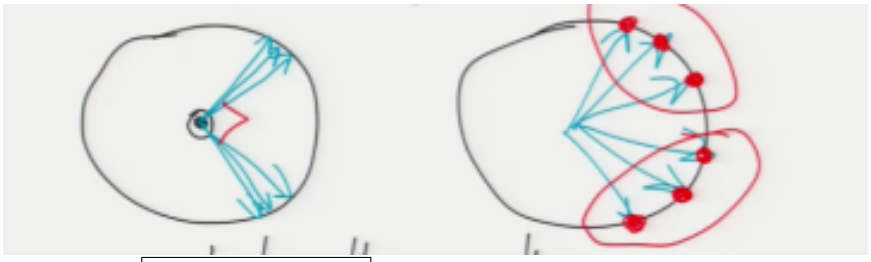


[V's columns are the eigenvectors with the k smallest eigenvalues.]
[Yes, we do include the all-1's vector v_1 as one of the columns of V.]

[Draw this by hand. eigenvectors.pdf]

Row V_i is spectral vector [my name] for vertex i. [The rows are vectors in a k-dimensional space I'll call the "spectral space." When we were using just one eigenvector, it made sense to cluster vertices together if their components were close together. When we use more than one eigenvector, it turns out that it makes sense to cluster vertices together if their spectral vectors point in similar directions.]

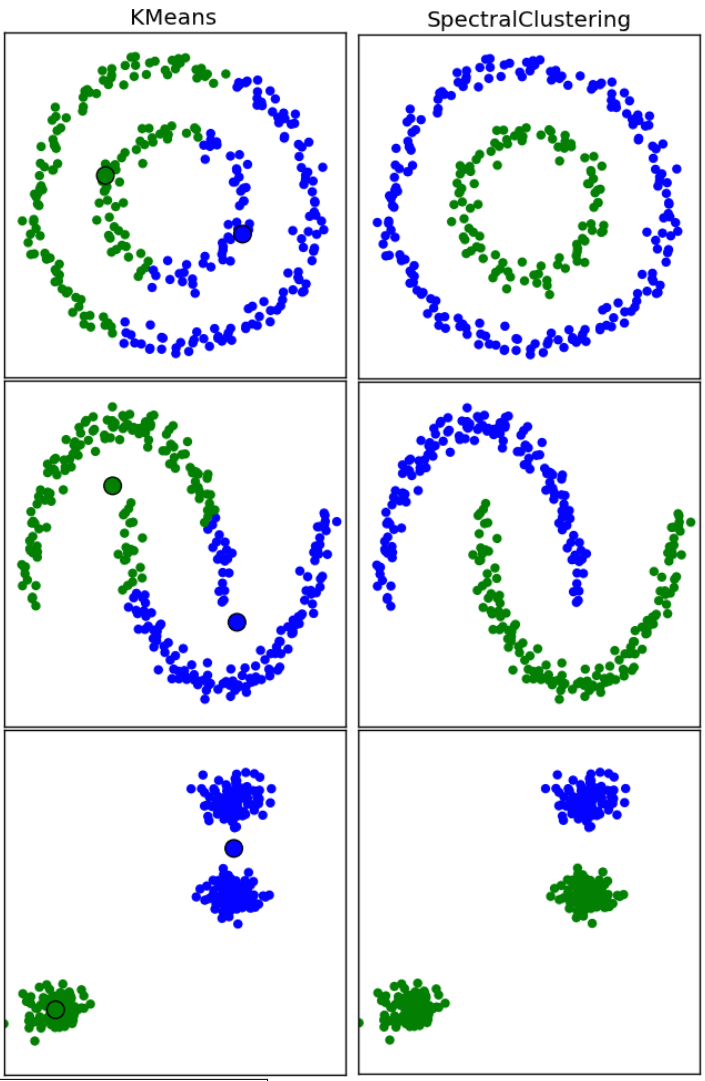
Normalize each row V_i to unit length.
[Now you can think of the spectral vectors as points on a unit sphere centered at the origin.]



[Draw this by hand vectorclusters.png] [A 2D example showing two clusters on a circle. If the graph has k components, the points in each cluster will have identical spectral vectors that are exactly orthogonal to all the other components' spectral vectors (left). If we modify the graph by connecting these components with small-weight edges, we get vectors more like those at right—not exactly orthogonal, but still tending toward distinct clusters.]

k-means cluster these vectors.

[Because all the spectral vectors lie on the sphere, k-means clustering will cluster together vectors that are separated by small angles.]



compkmeans.png, compspectral.png [Comparison of point sets clustered by k-means—just k-means by itself, that is—vs. a spectral method. To create a graph for the spectral method, we use an exponentially decaying function to assign weights to pairs of points, like we used for image segmentation but without the brightnesses.]

Invented by [our own] Prof. Michael Jordan, Andrew Ng [when he was still a student at Berkeley], Yair Weiss.

[This wasn't the first algorithm to use multiple eigenvectors for spectral clustering, but it has become one of the most popular.]

LATENT FACTOR ANALYSIS [aka Latent Semantic Indexing]

[You can think of this as dimensionality reduction for matrices.]

Suppose X is a term-document matrix: [aka bag-of-words model]
row i represents document i; column j represents term j. [Term = word.]
[Term-document matrices are usually sparse, meaning most entries are zero.]
X_{ij} = occurrences of term j in doc i
better: log (1+ occurrences) [So frequent words don't dominate.]
[Better still is to weight the entries so rare words give big entries and common words like "the" give small entries. To do that, you need to know how frequently each word occurs in general. I'll omit the details, but this is the common practice.]

Recall SVD $X = UDV^\top = \sum_{i=1}^{d} \delta_i u_i v_i^\top$. Suppose $\delta_i \le \delta_j$ for $i \ge j$.

Unlike PCA, we usually don't center X.
For greatest δ_i, each v_i lists terms in a genre/cluster of documents
each u_i " docus in a genre using similar/related terms
E.g. u_1 might have large components for the romance novels,
v_1 " " " " for terms "passion," "ravish," "bodice" ...
[... and δ_1 would give us an idea how much bigger the romance novel market is than the markets for every other genre of books.]

[v_1 and u_1 tell us that there is a large subset of books that tend to use the same large subset of words. We can read off the words by looking at the larger components of v_1, and we can read off the books by looking at the larger components of u_1.]

[The property of being a romance novel is an example of a latent factor. So is the property of being the sort of word used in romance novels. There's nothing in X that tells you explicitly that romance novels exist, but the genre is a hidden connection between them that gives them a large singular value. The vector u_1 reveals which books have that genre, and v_1 reveals which words are emphasized in that genre.]

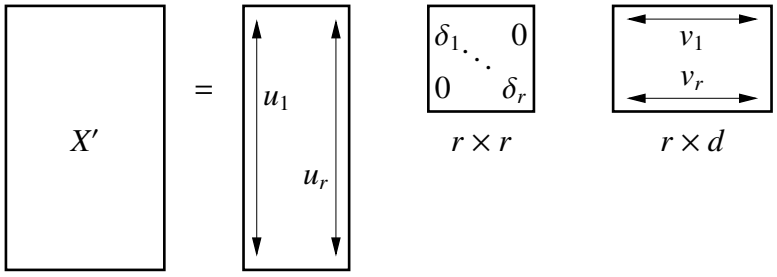
Like clustering, but clusters overlap: if u_1 picks out romances &
u_2 picks out histories, they both pick out historical romances.

[So you can think of latent factor analysis as a sort of clustering that permits clusters to overlap. Another way in which it differs from traditional clustering is that the u-vectors contain real numbers, and so some points have stronger cluster membership than others. One book might be just a bit romance, another a lot.]

Application in market research:
identifying consumer types (hipster, soccer mom) & items bought together.
[For applications like this, the first few singular vectors are the most useful. Most of the singular vectors are mostly noise, and they have small singular values to tell you so. This motivates approximating a matrix by using only some of its singular vectors.]

Truncated sum $X' = \sum_{i=1}^{r} \delta_i u_i v_i^\top$ is a low-rank approximation of X, of rank r.

[We choose the singular vectors with the largest singular values, because they carry the most information.]



[Draw this by hand. truncate.pdf]

X' is the rank-r matrix that minimizes the [squared] Frobenius norm

$$\|X - X'\|_F^2 = \sum_{i,j} \left(X_{ij} - X'_{ij}\right)^2$$

Applications:

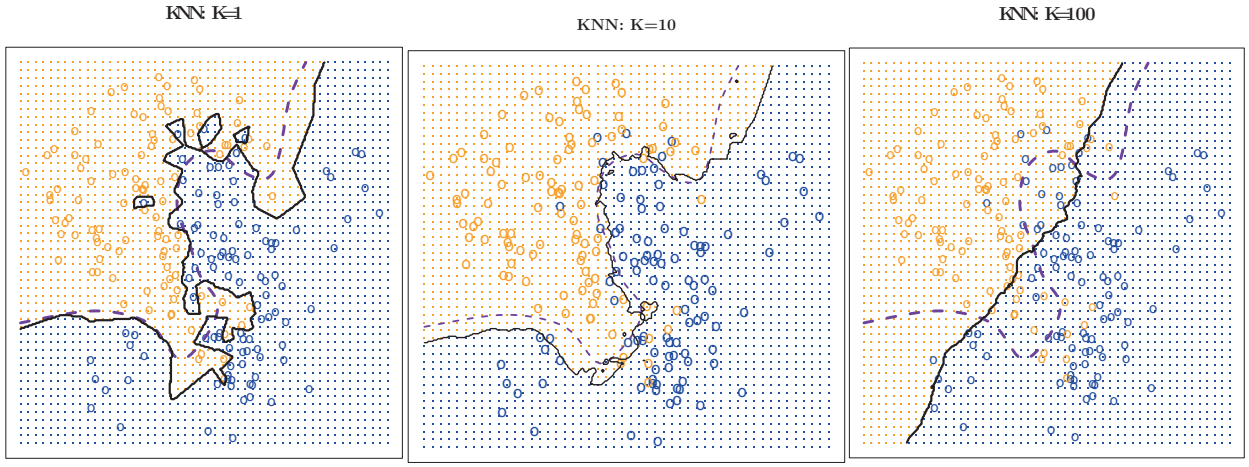
- Fuzzy search. [Suppose you want to find a document about gasoline prices, but the document you want doesn't have the word "gasoline"; it has the word "petrol." One cool thing about the reduced-rank matrix X' is that it will probably associate that document with "gasoline," because the SVD tends to group synonyms together.]
- Denoising. [The idea is to assume that X is a noisy measurement of some unknown matrix that probably has low rank. If that assumption is partly true, then the reduced-rank matrix X' might be better than the input X.]
- Matrix compression. [As you can see above, if we use a low-rank approximation with a small rank r, we can express the approximate matrix as an SVD that takes up much less space than the original matrix. Often this row-rank approximation supports faster matrix computations.]
- Collaborative filtering: fills in unknown values, e.g. user ratings.
  [Suppose the rows of X represents Netflix users and the columns represent movies. The entry X_{ij} is the review score that user i gave to movie j. But most users haven't reviewed most movies. We want to fill in the missing values. Just as the rank reduction will associate "petrol" with "gasoline," it will tend to associate users with similar tastes in movies, so the reduced-rank matrix X' can predict ratings for users who didn't supply any. You'll try this out in the last homework.]

NEAREST NEIGHBOR CLASSIFICATION

[We're done with unsupervised learning. Now I'm going back to classifiers, and I saved the simplest for the end of the semester.]

Idea: Given query point q, find the k sample pts nearest q.
Distance metric of your choice.
Regression: Return average label of the k pts.
Classification: Return class with the most votes from the k pts OR
return histogram of class probabilities.

[The histogram of class probabilities tries to estimate the posterior probabilities of the classes. Obviously, the histogram has limited precision. If $k = 3$, then the only probabilities you'll ever return are 0, 1/3, 2/3, or 1. You can improve the precision by making k larger, but you might underfit. The histogram works best when you have a huge amount of data.]



allnn.pdf (ISL, Figures 2.15, 2.16) [Examples of 1-NN, 10-NN, and 100-NN. A larger k smooths out the boundary. In this example, the 1-NN classifier is badly overfitting the data, and the 100-NN classifier is badly underfitting. The 10-NN classifier does well: it's reasonably close to the Bayes decision boundary. Generally, the ideal k depends on how dense your data is. As your data gets denser, the best k increases.]

[There are theorems showing that if you have a lot of data, nearest neighbors can work quite well.]

Theorem (Cover & Hart, 1967):
As $n \to \infty$, the 1-NN error rate is $< B(2 - B)$ where B = Bayes risk.
if only 2 classes, $\leq 2B(1 - B)$

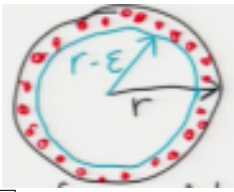
[There are a few technical requirements of this theorem. The most important is that the training points and the test points all have to be drawn independently from the same probability distribution—just like in our last lecture, on learning theory. The theorem applies to any separable metric space, so it's not just for the Euclidean metric.]

[By the way, this Cover is the same Thomas Cover of Cover's Theorem in the last lecture. He's a professor in Electrical Engineering and Statistics at Stanford, and these are the first and third journal articles he published.]

Theorem (Fix & Hodges, 1951):
As $n \to \infty$, $k \to \infty$, $k/n \to 0$, k-NN error rate converges to B. [Which means optimal.]

The Geometry of High-Dimensional Spaces

Consider shell between spheres of radii r & $r - \epsilon$.



[Draw this by hand concentric.png] [Concentric balls. In high dimensions, almost every point chosen uniformly at random in the outer ball lies outside the inner ball.]

Volume of outer ball $\propto r^d$
Volume of inner ball $\propto (r - \epsilon)^d$
Ratio of inner ball volume to outer =

$$\frac{(r-\epsilon)^d}{r^d} = \left(1 - \frac{\epsilon}{r}\right)^d \approx \exp\left(-\frac{\epsilon d}{r}\right) \qquad \text{which is small for large } d.$$

E.g. if $\frac{\epsilon}{r} = 0.1$ & $d = 100$, inner ball has 0.0027% of volume.

Random points from uniform distribution in ball: nearly all are in outer shell.
" " " Gaussian " : nearly all are in some shell.

[If the dimension is very high, the majority of the random points generated from an isotropic Gaussian distribution are approximately at the same distance from the center. So they lie in a thin shell. As the dimension grows, the standard deviation of a random point's distance to the center gets smaller and smaller compared to the distance itself. You can think of a point from a multivariate Gaussian distribution as a sample of d scalar values from a univariate Gaussian. As d gets bigger, the mean of the squares of the components converges to the true mean for the population.]

[This is one of the things that makes machine learning hard in high dimensions. Sometimes the nearest neighbor and the farthest neighbor aren't much different.]

Exhaustive k-NN Alg.

Given query point q:

- Scan through all n sample pts, computing (squared) distances to q.
- Maintain a max-heap with the k shortest distances seen so far.
  [Whenever you encounter a sample point closer to q than the point at the top of the heap, you remove the heap-top point and insert the better point. Obviously you don't need a heap if $k = 1$ or even 3, but if $k = 101$ a heap will substantially speed up keeping track of the distance to beat.]

Time to construct classifier: 0 [This is the only $O(0)$-time algorithm we'll learn this semester.]
Query time: $O(nd + n \log k)$
expected $O(nd + k \log^2 k)$ if random pt order
[It's a cute theoretical observation that you can slightly improve the expected running time by randomizing the point order so that only expected $O(k \log k)$ heap operations occur. But in practice I don't recommend it; you'll probably lose more from cache misses than you'll gain from fewer heap operations.]

Random Projection

Projections into low-d space like PCA speed up NN, but distances are approximate.

[Most fast nearest-neighbor algorithms in more than a few dimensions are *approximate* nearest neighbor algorithms; we don't necessarily expect to find the exact nearest neighbors. For classification, that's usually sufficient.]

Random projection is cheap alternative to PCA as preprocess for NN or clustering.
[Projects onto a random subspace instead of the "best" subspace, but takes a fraction of the time of PCA.]

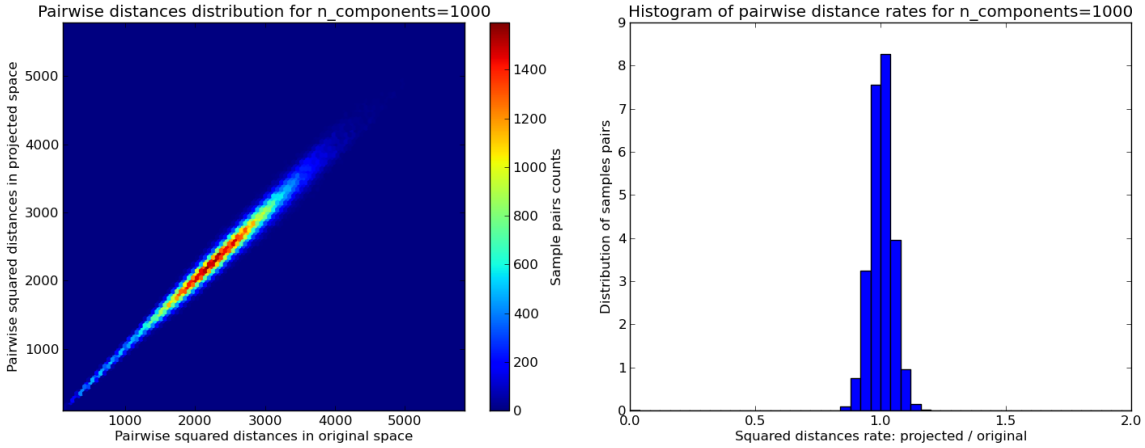
Pick a random subspace $S \subset \mathbb{R}^d$ of dimension k, where $k = \dfrac{2\ln(1/\delta)}{\epsilon^2/2 - \epsilon^3/3}$.

For any pt q, let $\hat{q}$ be $\sqrt{\frac{d}{k}}$ times [orthogonal] projection of q onto S.

For any two pts $q, w \in \mathbb{R}^d$, $(1-\epsilon)|qw|^2 \le |\hat{q}\hat{w}|^2 \le (1+\epsilon)|qw|^2$ with probability $\ge 1 - 2\delta$.

[So the distance between the two points after projecting is rarely much different than the distance before. For reasonably reliable clustering or approximate nearest neighbor search, it's customary to choose $\delta \le 1/n^2$. In practice, you can experiment with k to find the best speed-accuracy tradeoff. But the key observation is that you need a subspace of dimension $\Theta(\log n)$. The hidden constant is large, though. For example, you can bring 1,000,000 sample points down to a 10,362-dimensional space with a 10% error in the distances.]

[If you want a proof of this, look up references about the Johnson–Lindenstrauss Lemma.]



100000to1000.pdf [Comparison of inter-point distances before and after projecting points in 100,000-dimensional space down to 1,000 dimensions. This example suggests that the theoretical bounds are a bit pessimistic compared to practice.]

[Why does this work? A random projection of a vector is equivalent to taking a random vector and selecting k components. The mean of the squares of those k sampled components approximates the mean for the whole population.]

[How do you get a uniformly distributed random projection direction? You can choose each component from a univariate Gaussian distribution, then normalize the vector to unit length. How do you get a random subspace? You can choose k random vectors, then use Gram-Schmidt orthogonalization to make them mutually orthonormal. Interestingly, Indyk and Motwani show that if you skip the expensive normalization and Gram-Schmidt steps, random projection still works almost as well, because random vectors in a high-dimensional space are nearly equal in length and nearly orthogonal to each other with high probability.]