2021-03-23

1 Introduction

Dimensionality reduction involves taking a data set $\{x^i\}_{i=1}^n$ and finding corresponding vectors $\{y^i\}_{i=1}^n$ in some lower-dimensional space so as to preserve "important" properties of the data set. Natural desiderata for such a mapping might include:

- Preserving pairwise distances (isometry)
- Preserving angles locally (conformality)
- Preserving neighborhood structure
- Clustering points with similar class labels

A "nice to have" property is that there is a mapping $y = f(x)$ that accomplishes this goal and is learned in a somewhat explicit form.

So far, we have implicitly studied several *linear* dimensionality reduction methods in which the mapping from $x \mapsto y$ is a linear (or possibly affine) map. Examples include PCA and Fisher LDA embeddings, and we can also see some other matrix factorizations in a similar light. These methods are quite powerful, and are backed by standard linear algebra types of tools, at least for the standard loss functions we have used. There is much more to say; for instance, we have not discussed interesting topics like robust PCA, though we have seen many of the numerical methods that are ingredients for solving these problems

However, sometimes the restriction to only using linear maps is quite severe. For this reason, in this lecture, we consider *nonlinear* dimensionality reduction techniques. We will return to the topic later in specific contexts (kernel PCA when we talk about kernel methods, Laplacian eigenmaps when we talk about graph-based dimensionality reduction and clustering). But for now, we introduce a handful of influential dimensionality reduction methods that we will not have as much time to discuss later. As in the rest of the class, our goal is partly to highlight the idea of the schemes, but just as much to highlight interesting numerical methods that go into their efficient implementation: Nystrom approximation, connections to the SVD, clever uses of sparsity, fast multipole approximations, and so forth.

2 PCA and multi-dimensional scaling

We start with the idea of *multi-dimensional scaling* (MDS), which attempts to construct coordinates y^i such that the distances $\|y^i - y^j\|$ approximate some known "dissimilarity measure" between x^i and x^j. We observe that for

$$d_{ij}^2 = \|x^i - x^j\|^2 = \|x^i\|^2 - 2\langle x^i, x^j \rangle + \|x^j\|^2,$$

we can write the squared distance matrix $D^{(2)}$ as

$$D^{(2)} = r^{(2)} e^T - 2X^T X + e(r^{(2)})^T$$

where e is a vector of all ones and $(2)_i = \|x^i\|^2$. Let P denote a *centering transformation*

$$P = I - \frac{1}{n} ee^T,$$

i.e. P subtracts the mean of a vector from each component. Note that $Pe = 0$, and $XP = \bar{X}$ is the matrix with columns $x^j - \bar{x}$ where $\bar{x}$ is the mean of the columns. Therefore, applying P to $D^{(2)}$ on the left and right eliminates the rank-one terms; and scaling by $-1/2$ gives us

$$B = -\frac{1}{2} P D^{(2)} P = \bar{X}^T \bar{X}.$$

Thus, B is a Gram matrix for the centered data matrix, abd we can recover $\bar{X}$ (up to an orthogonal transformation) via the truncated eigendecomposition of B:

$$\bar{X} = Q \Lambda_d^{-1/2} V_d^T$$

for some orthogonal matrix $Q \in O(d)$. If we want a lower-dimensional embedding that optimally approximates the Gram matrix, we need only take fewer eigenvectors. Note that the eigenvalues and vectors of $\bar{X}^T \bar{X}$ are the singular vectors and (squared) singular values of $\bar{X}$. Thus, MDS when Euclidean distance is used for the dissimilarity is equivalent to PCA.

There are other forms of multi-dimensional scaling that use other objectives in fitting to a dissimiliarity matrix. For example, metric MDS minimizes the "stress"

$$\sum_{i \neq j} (d_{ij} - \|y^i - y^j\|)^2,$$

and other forms of MDS (Sammon mapping, Sammon with Bregman divergences, ordinal MDS) use yet other measures.

3 Nystrom and landmarks

We have seen before that there are other ways to represent a low-rank matrix than with an SVD. In particular, we can construct a representation with a subset of rows and columns of the matrix. We have referred to this as the CUR decomposition in general; in the symmetric case, it is often known as the Nystrom approximation. That is, if A is a square symmetric matrix with rank r, then there is a symmetric permutation such that $A_{11} \in \mathbb{R}^{r \times r}$ is invertible and

$$A = \begin{bmatrix} A_{11} & A_{21}^T \\ A_{21} & A_{22} \end{bmatrix} = \begin{bmatrix} A_{11} \\ A_{21} \end{bmatrix} A_{11}^{-1} \begin{bmatrix} A_{11} & A_{21}^T \end{bmatrix}.$$

Moreover, if we take the economy QR decomposition

$$\begin{bmatrix} A_{11} \\ A_{21} \end{bmatrix} = QR,$$

then we can decompose $RA_{11}^{-1}R^T = U\Lambda U^T$ and get the (truncated) eigendecomposition

$$A = (QU)\Lambda(QU)^T.$$

What does this have to do with MDS? Consider the case now where we use the Nystrom approximation of the squared distance matrix D:

$$D = \begin{bmatrix} D_{11} \\ D_{21} \end{bmatrix} D_{11}^{-1} \begin{bmatrix} D_{11} \\ D_{21}^T \end{bmatrix}$$

Centering gives

$$B = -\frac{1}{2} Z D_{11}^{-1} Z^T$$

where

$$Z = P \begin{bmatrix} D_{11} \\ D_{21} \end{bmatrix}.$$

Taking the QR decomposition allows us to approximate the largest eigenvectors as before.

The selected columns in the Nystrom scheme correspond to "landmarks" in the multi-dimensional scaling setting. The idea is that we never need to know the distance between arbitrary points; we only need the distances between those points and a set of landmarks. If we choose q landmarks, the

time required for this method is $O(nq^2)$ rather than $O(n^3)$, and we only need to evaluate (and store) nq distances.

The process also gives us an affine map that we can apply to embed new points in the lower-dimensional space after training; that is

1. Compute the vector $d \in \mathbb{R}^q$ of distances from the new point to each landmark, and let $z = d - \bar{d}$ be the "centered" version that comes from subtracting the mean distances to the landmarks among the original training data.

2. The k-dimensional embedding vector y is now given by $y = \Lambda_k^{-1/2} U_k^T R^{-T} z$.

4 Isomap

Preserving geometry does not always mean preserving distances in Euclidean space. The *Isomap* algorithm is intended for the case when the x^i data points lie close to some lower-dimensional manifold embedded in a high-dimensional Euclidean space. Rather than looking at Euclidean distances between points, Isomap looks at a matrix of (approximate) geodesic distances for the manifold. Unfortunately, because the manifold is implicit, the geodesic distances cannot be computed directly, but are approximated by shortest paths through a network connecting nearest neighbors, either taking a fixed number of nearest neighbors per node (k-Isomap) or taking all neighbors within some radius of each node (ϵ-Isomap).

There are many clever ideas for finding nearest neighbors, which we will not go into here. The brute force approach requires $O(n^2)$ time. Once the nearest neighbor graph is computed, however, we must compute all shortest paths through the graph, a task that generally takes $O(n^3)$ time using the Floyd-Warshall algorithm. Computing a full eigendecomposition of the resulting (centered and squared) distance matrix takes another $O(n^3)$ time. Fortunately, the landmark approach for MDS applies equally well here, and requires only that we find the distances from q landmarks to all other nodes ($O(qn \log n)$ time via Dijkstra's algorithm), after which our linear algebra costs are $O(nq^2)$.

The Isomap algorithm is a beautiful and useful idea, but with some important limitations. Inherent in the setup is the idea that there is a good embedding of the manifold in a low-dimensional Euclidean space; but the existence of such an embedding depends on the topology of the manifold. The

sphere surface is a two-dimensional manifold that is easily represented in $\mathbb{R}^3$, for example, but cannot be continuously embedded into $\mathbb{R}^2$. Isomap may also do poorly on manifolds with high Gaussian curvature, or on manifolds that are not very densely sampled.

5 LLE

A popular alternative to Isomap is the locally linear embedding (LLE). Similar to Isomap, the LLE algorithm starts by building a graph between points in the data set and nearby neighbors. For each node j, one seeks to find weights w_{ij} to approximately reconstruct x^j from the neighbors by minimizing

$$\| \sum_{i \in \mathcal{N}_j} x_i w_{ij} - x_j \|^2.$$

subject to the constraint $\sum_{i \in \mathcal{N}_j} w_{ij} = 1$ (necessary for translational invariance). Computationally, this step involves solving a large number of independent small equality-constrained least squares problems, one for each node. But it can equivalently be seen as finding a weight matrix W with a given sparsity that minimizes

$$\|(I - W)X^T\|_F^2 = X(I - W)^T(I - W)X.$$

Once the weights have been computed, one seeks a matrix Y to minimize

$$\|(I - W)Y^T\|_F^2$$

subject to the constraints that $YY^T = \frac{1}{n}I$ and $Ye = 0$ (i.e. the Y coordinate system is centered). This gives that Y is $n^{-1/2}$ times the singular vectors $v_{n-k}, \ldots, v_{n-1}$ associated with the smallest singular values of $I - W$ apart from the null vector e. Because the weight matrix is sparse, these smallest singular values can be computed using standard sparse eigensolver iterations such as the Lanczos method.

6 t-SNE

The t-distributed Stochastic Neighbor Embedding (t-SNE) is a popular method for mapping high-dimensional data to very low (usually 2) dimensional representations for visualization. Given a data point x^i, one defines a conditional

probability $p_{j|i}$ that x^j would appear as its "neighbor" proportional to

$$p_{j|i} \propto \exp\left(-\frac{\|x^i - x^j\|^2}{2\sigma_i^2}\right)$$

with the diagonal probabilities set to zero. To simplify things somewhat, we work with the symmetrized conditional probabilities $p_{ij} = (p_{i|j} + p_{j|i})/(2n)$. In the lower-dimensional y space, one makes a similar construction, but with a heavier-tailed t-distribution:

$$q_{ij} \propto (1 + \|y^i - y^j\|^2)^{-1}.$$

The embedding is then chosen so that the KL-divergence

$$C = \sum_{i,j} p_{ij} \log \frac{p_{ij}}{q_{ij}},$$

which measures the distance between the joint P distribution in the high-dimensional space and the joint Q distribution in the low-dimensional space, is as small as possible.

Though the t-SNE paper starts from the stochastic motivation associated with the earlier SNE method (like t-SNE, but using a Gaussian distribution in the low-dimensional space as well), the authors describe some of the motivation in very mechanical terms. The original SNE method was subject to the y points crowding too close together, and they note that t-SNE has a "repulsive force" term that pushes them apart. The force interpretation comes from treating the KL-divergence C as a potential energy, in which case the gradients

$$\nabla_{y^i} C = 4Z \sum_{j \neq i} (p_{ij} - q_{ij}) q_{ij} (y^i - y^j)$$

are naturally seen as forces (where Z here is the normalization term $Z = \sum_{k \neq l} (1 + \|y_k - y_l\|^2)^{-1}$). We can split this into a short-range attractive piece

$$\sum_{j \neq i} p_{ij} q_{ij} (y^i - y^j)$$

which only involves a local neighborhood because of the rapid decay of the Gaussian in p_{ij}; and a long-range repulsive piece

$$-\sum_{j \neq i} q_{ij}^2 (y_i - y_j).$$

Though we cannot take advantage of sparsity in this long-range repulsion term, we can take advantage of smoothness; that is, $q_{ij} \approx q_{ik}$ when $\|y^k - y^j\| \ll \|y^i - y^j\|$. Thus, we can "clump together" all the terms that are sufficiently far from a region in space. This is the same as the idea that we can treat the gravity of the sun as a point mass from the perspective of far-away bodies like the planets, and we can similarly approximate the gravitational pull of distant galaxies as a single pull from a (tremendously large but tremendously distant) center of mass. This is the idea underlying *tree codes* like the famous Barnes-Hut algorithm. With more careful control of the error, one gets the famous *fast multipole method*, which can also be used in this setting. The same idea appears in fast algorithms for kernel interpolation and Gaussian process regression in low-dimensional spaces, as we will discuss in a few weeks.

7 Autoencoders

We will say relatively little about neural networks in this class, but it is worth saying at least a few words about the idea of an autoencoder. As with many things involving neural networks, autoencoders seem to work better in practice than we know how to argue that they ought to.

The idea behind an autoencoder is that one seeks to train a neural network with a "bottleneck" to approximate the identity. That is, we seek weight vectors θ^1 and θ^2 for two halves of a feed-forward neural network so that

$$F(x^i; \theta^1, \theta^2) = f_{\text{decode}}(f_{\text{encode}}(x^i; \theta^1); \theta^2) \approx x^i,$$

where f_{encode} maps from the high-dimensional input space to the outputs of a low-dimensional "bottleneck" layer, and f_{decode} maps back up to the high-dimensional space. The weights are trained by stochastic gradient descent on a loss such as function $\sum_i (F(x^i) - x^i)^2$. *Variational* autoencoders output not just an intermediate feature vector, but the parameters for an intermediate feature distribution (e.g. means and variances on each parameter). The gain for this additional complexity is a tendency to favor smoother mappings.