CHAPTER

11 Latent space models

Another useful unsupervised learning framework is the concept of *latent spaces*. In clustering, we identified structure in the data by grouping the data into discrete groups, and describing the characteristics of these groups. In contrast, latent space methods represent the data in a vector space (typically of lower dimension than the original feature space) in such a way that similar data points remain nearby. This new representation can then be used to organize and display the data, or as features for additional learning tasks.

The fundamental idea of latent space approaches is to identify a lower-dimensional representation or transform of x that is sufficient to (approximately) recover the original features:

$$z = f(x) \qquad\qquad x \approx \hat{x} = g(z)$$

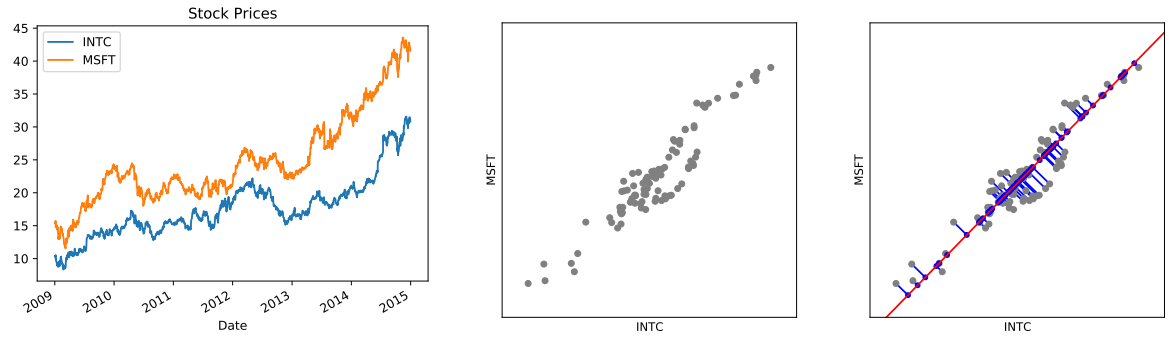
If the original features x are n-dimensional, but z is only k-dimensional, with $k \ll n$, the new representation can be viewed as compressing the data. Then, $f(\cdot)$ plays the role of the compression function, and $g(\cdot)$ decompresses the reduced-size representation z to recover x, or a close approximation of x.

11.1 A simple example

Consider a dataset of financial values, for example a vector of prices for the 500 individual stocks that make up the Standard & Poor 500 index.[1] If we wanted to describe, say, the change in values from the previous day, we could simply list all 500 values – or, we could communicate it in a compressed but approximate way, for example saying that the market overall was up by 5%, followed by increasingly refined details, e.g., that manufacturing companies were up more, but technology companies slightly down, etc. This style of information allows the listener to guess at the full vector of prices even though we have only transmitted a few numbers.

Let us make this example even more concrete. Consider representing a two-dimensional data set (the "S&P2") consisting of prices for Intel (INTC) and Microsoft (MSFT), shown in Figure 11.1(a). To account for differences in their base prices, we transform both to evaluate the percent gain (or loss) since January 1, 2009. A scatter plot of 100 random days is shown in Figure 11.1(b).

[1] Technically, the companies that make up the index are not stable over time; but we will ignore this inconvenient detail.



(a) (b) (c)

Figure 11.1: "S&P2" data (stocks INTC & MSFT). (a) Prices over time. (b) Scatterplot of data X, each day's percent gain relative to Jan. 1, 2009. (c) We can approximate each $x^{(i)}$ with a single scalar $z^{(i)}$, indicating the best approximation that lies in the vector v (red line). The resulting approximation errors are shown in blue.

Since our data x consist of two real values, our compressed representation z will be a single scalar number. For simplicity, let us choose $g(z)$ to be a linear function,

$$\hat{x} = [v_1 z + \mu_1 \ , \ v_2 z + \mu_2]$$

where $\mu = [\mu_1, \mu_2]$ are the mean of the data X. The reconstructed approximate values $\hat{x}$ will all lie on a one dimensional linear subspace (a line) passing through the mean of the data in x's 2D feature space, where the value of z corresponds to a position along this line. Figure 11.1(c) shows the subspace (red line) along with the error between each data point $x^{(i)}$ and its reconstructed approximation $\hat{x}^{(i)}$ (blue lines).

Notice that this is *not* a regression task – we are not planning to send Intel's price and then use it to predict Microsoft's. Instead, we are sending a single value z which is used to reconstruct both values in x (a 2D location), introducing a small amount of error into each. Thus the minimum reconstruction error is the distance from each point $x^{(i)}$ to the subspace defined by $g(z)$, rather than a measurement of the difference in a single target variable y.

Returning to our original S&P500 example, we would like to perform something similar, but (a) now with many more features (high dimensional x), and (b) using more than a single scalar value z to capture the variations in the data. Let us see how to extend this technique to such problems.

11.2 Principal Components Analysis (PCA)

Let X be a data matrix whose rows $x^{(i)}$ are our data observations. We wish to reconstruct each $x^{(i)}$ using k real numbers, formed as a vector $z^{(i)} \in \mathbb{R}^k$. Since $g(z)$ is linear, we can write $g(z) = z \odot V$ for some matrix $V \in \mathbb{R}^{k \times n}$. We wish to choose V so as to minimize our reconstruction error, which we will measure in terms of its (squared) *Frobenius norm*, i.e., the element-wise squared error between the matrix X and its reconstruction,

$$\|X - Z \odot V\|_F^2 = \sum_{ij} \left(X_{ij} - (Z \odot V)_{ij}\right)^2.$$

The vectors in V form a subspace in which our reconstructions $\hat{x}^{(i)}$ will lie; by selecting z appropriately we can reconstruct any $\hat{x}$ in the span of the rows of V. So, without loss of generality let us also make the rows of V orthonormal, so that $V \odot V^T = I$ (i.e., V is unitary).

Now, it is relatively easy to see that the optimal value of Z is given by solving,

$$\forall\, i,k \qquad 0 = \frac{\partial}{\partial Z_{ik}} \sum_{i'j'} \left(X_{i'j'} - \sum_{k'} Z_{i'k'} V_{k'j'}\right)^2 = 2 \sum_{j'} (X_{ij'} - \sum_{k'} Z_{ik'} V_{k'j'})\,(-V_{kj'})$$

$$\Rightarrow \qquad X \odot V^T = Z \odot V \odot V^T \;=\; Z$$

i.e., we should simply project X onto the subspace defined by V.

The Frobenius norm can be expressed as a matrix trace (sum of diagonal elements),

$$\|E\|_F^2 = \text{tr}(E^T \odot E)$$

so that

$$\begin{aligned}\|X - Z \odot V\|_F^2 &= \text{tr}(X^T X) - \text{tr}(V^T V X^T X) - \text{tr}(X^T X V^T V) - \text{tr}(V^T V X^T X V^T V) \\ &= \text{tr}(X^T X) - \text{tr}\big((XV^T)^T \odot (XV^T)\big) \\ &= \text{tr}(X^T X) - \text{tr}(Z^T Z)\end{aligned}$$

Then, since the trace of a matrix M is simply the sum of its eigenvalues, and V is unitary, this means we want to *maximize* the eigenvalues of $Z^T Z$, and so should select the span of V to include as many of the top eigenvectors of $X^T X$ as possible. If V is rank k, this can be done by taking the rows of V to be the top k eigenvectors of $X^T X$, i.e., compute the eigendecomposition,

$$X^T X = V^T \Lambda V \tag{11.1}$$

and retain the rows of V corresponding to the k largest eigenvalues Λ_{ii}. Then, the Frobenius norm of the residual errors is simply equal to the sum of the remaining $n - k$ eigenvalues of $X^T X$.

In computer science, this linear transformation goes by the name *principal component analysis* (PCA; [Pearson, 1901]). However, it is so fundamental that it has been independently re-discovered and named in many different settings: the Hotelling transform [Hotelling, 1933], proper orthogonal decomposition (POD) [Lumley, 1967], empirical orthogonal functions (EOF) [Lorenz, 1956], and the (discrete) Karhunen-Loève transform [Karhunen, 1946].

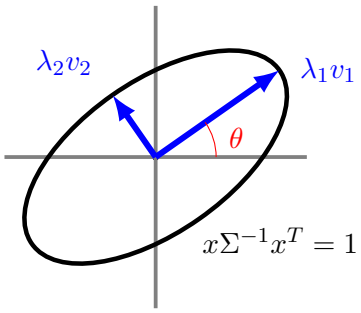
Gaussian Geometric Perspective

Notice that, for zero-mean data X, the matrix $\frac{1}{m} X^T \odot X$ is the empirical covariance matrix of the data, i.e., the maximum likelihood estimate of Σ when fitting a multivariate Gaussian distribution to the $x^{(i)}$. This illustrates that the principal components can be interpreted in terms of a Gaussian model fit to the data.

Consider a zero-mean Gaussian distribution with covariance Σ. In order to plot the probability density function, we might draw the iso-contour lines defined by $\log \mathcal{N}(x\,;\,0,\Sigma) = c$ for some constant c. Since Σ is real and symmetric, we can write $\Sigma = V \Lambda V^T$ where V is the unitary matrix of eigenvectors of Σ, and Λ is a diagonal matrix of their eigenvalues. The iso-contour is then defined by

$$x \Sigma^{-1} x^T = x(V^T \Lambda^{-1} V) x^T = (xV^T)\Lambda^{-1}(Vx^T) = \tilde{x} \Lambda^{-1} \tilde{x}^T$$

Since Λ is diagonal, we have $[\Lambda^{-1}]_{ii} = (\Lambda_{ii})^{-1}$, and V is unitary and thus equivalent to a change of basis (a rotation or mirroring of axes). So the iso-contour is like a diagonal Gaussian with variances Λ_{ii} in each direction, but rotated by the transformation $\tilde{x} = xV^T$, so that the major and minor axes of the ellipse are the rows of V.



$$\Lambda = \begin{bmatrix} 2 & 0 \\ 0 & 1 \end{bmatrix} = \begin{bmatrix} \lambda_1 & 0 \\ 0 & \lambda_2 \end{bmatrix}$$

$$V = \begin{bmatrix} \cos(\theta) & \sin(\theta) \\ -\sin(\theta) & \cos(\theta) \end{bmatrix} = \begin{bmatrix} v_1 \\ v_2 \end{bmatrix}$$

$$\Sigma = V^T \Lambda V$$

Figure 11.2: Gaussian iso-contour diagram.

An example is shown in Figure 11.2, in which

$$\Sigma = V\Lambda V^T \qquad \Lambda = \begin{bmatrix} 2 & 0 \\ 0 & 1 \end{bmatrix} \qquad V = \begin{bmatrix} \cos(\theta) & \sin(\theta) \\ -\sin(\theta) & \cos(\theta) \end{bmatrix}$$

From this viewpoint, it is intuitive that in order to minimize the squared error of the projection PCA should preserve the major axes of the covariance, as these are the directions in which the data have the most variation, and discard the directions in which the variance of the data is small.

Singular Value Decomposition

In practice, the eigendecomposition in Eq. 11.1 is often substituted for a different matrix factorization form, called the *singular value decomposition* (SVD). Again, assuming that the data matrix X is zero-mean, the SVD of X is

$$X = U \odot S \odot V^T$$

where S is a diagonal matrix, whose diagonal entries s_{ii} are called the singular values of X, and U and V are unitary matrices, so that $U^T \odot U = V \odot V^T = I$, the identity matrix. Thus, U and V's columns are orthogonal, unit-length vectors, called the left and right singular vectors of X, respectively.

To see how the SVD relates to the eigendecomposition in Eq. 11.1, notice that

$$m\,\Sigma = X^T X = (USV^T)^T (USV^T) = VS^T U^T USV^T = V(S^T S)V^T = V\Lambda V^T$$

i.e., V are also the eigenvectors of $X^T X$, and $s_{ii}^2 = \lambda_i$ are the eigenvalues. (Conversely, the vectors U are the eigenvectors of the $m \times m$ matrix XX^T.)

Then, retaining the top k eigenvectors is equivalent to retaining the k singular values with largest magnitude, and their associated columns of U and V. Let us denote the portion of U corresponding to the top k singular vectors as $U_{\leq k}$, and similarly for $S_{\leq k}$ and $V_{\leq k}$. We then have the rank-k approximation to X,

$$\hat{X} = U_{\leq k} \odot S_{\leq k} \odot V_{\leq k}^T.$$

Equivalently, the first k singular vectors define the rank-k approximation $\hat{X}$ which is closest to X in the Frobenius norm sense, i.e., $\|X - \hat{X}\|_F^2 = \sum_{i,j}(X_{ij} - \hat{X}_{ij})^2$.

Interestingly, although up to this point we have assumed that the rows correspond to data points (e.g., days) and columns to features (e.g., stocks), the singular value decomposition shows that (for a given X) we identify the same subspace even if we reverse these perspectives: if $X = U \odot S \odot V^T$, then $X^T = V \odot S \odot U^T$.

Pre-processing

So far we have assumed the data matrix X to be zero mean; if not, we can simply subtract the mean, an operation called "centering" the data. If we do not include the mean subtraction step, our approximation V will pass through the origin of the feature space x, which may distort the linear subspace that is selected. Thus centering the data is a common initial pre-processing step.

Another typical pre-processing step is to scale the features of x beforehand. Since PCA is selecting the direction(s) in which the data have largest variance, changing the scale (or equivalently, the units in which a feature is measured) can change the principal directions. Recall that in our stock example, we pre-processed the two features to have a meaningful, comparable scale (percent change since the initial date). If we had used the actual price values, our least-squares loss would have emphasized being more accurate in representing MSFT (the more expensive stock) than INTC. Conversely, if we had chosen to measure INTC's price in yen rather than dollars, its variance would be much higher than MSFT, and we would choose a direction very close to simply representing INTC's price.

In many settings, we do not control in what units each feature is measured, and often lack the domain knowledge to select a scale for each feature that has intrinsic meaning. In these cases, a default "rescaling" transformation may be used to place the features on comparable scales, for example, scaling each feature individually so that it has unit variance, or so that its values lie in a fixed range (say, $[0, 1]$ or $[-1, 1]$).

11.3 Examples

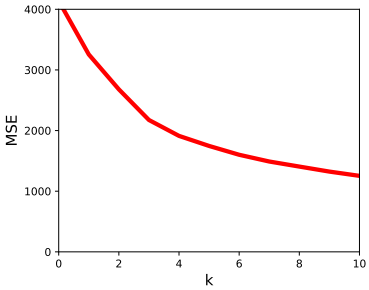
Principal component analysis is used all through statistics and signal processing as a form of compression or noise removal, to reduce the number of features for visualization or to prevent overfitting, or to construct a more "fundamental" basis in which to represent the data. In this section we show a few examples corresponding to representation of high dimensional data from images, text, and movie ratings.

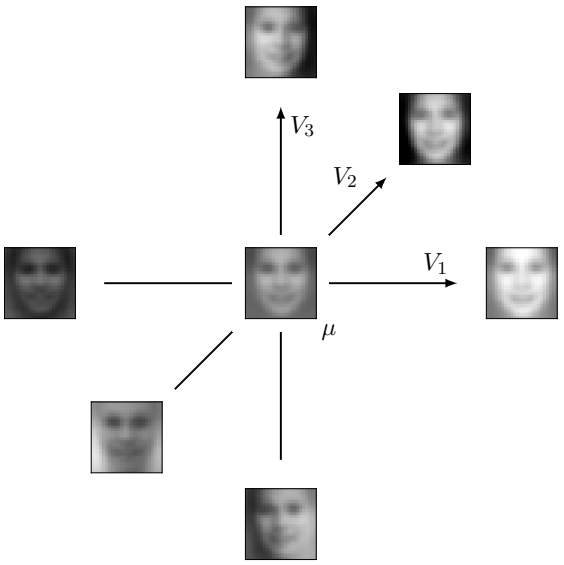
Images

Example 11-1 : Eigenfaces

Let us look at the principal components for a collection of face data. We use a collection of ≈ 5000 faces, in grayscale at 24×24 pixel resolution, from Viola and Jones [2001]; a small sample of these images is:

Treating each image as a $n = 576$ real-valued vector, we expect that there should be considerable structure in the data (the set of face images is much smaller than the set of all 24×24 images), although perhaps it will not be a linear subspace. If we use PCA to approximate the collection of images, we can evaluate the residual MSE as a function of the number of retained dimensions k. In the plot at right, we see that the first three components manage to explain about half of the variance in the pixels.





If we look at these first three principal directions (visualized on the left), they turn out to be easy to interpret. Centering our plot at the mean face image, we can display the faces obtained by moving a fixed amount in the positive and negative directions for each of the basis vectors V_1, V_2, V_3. We see that the first axis, V_1, appears to correspond to general image brightness; V_2 corresponds to foreground/background lighting (brighter faces on a dark background versus darker faces on a white background); and V_3 corresponds to brighter lighting from the left versus from the right of the person's face. Since lighting effects change the magnitude of all the pixels in the image, it should not be too surprising that these are the most impactful in terms of capturing the variance in X.

Latent Semantic Analysis

Skip zero-mean step on text data

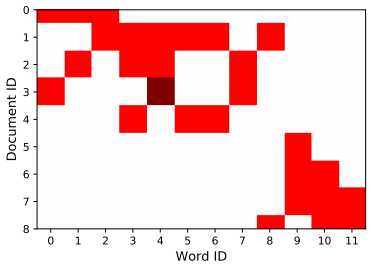
Example 11-2 : LSA

Consider a tiny data set, originally from Landauer et al. [1998], consisting of 9 "documents" (each a single paper title) and a vocabulary of 12 words. The titles are easily seen to group into two topics, one on human-computer interaction (HCI), and the other on mathematical graph theory. The data matrix X counts which words appear in which documents, and shows a rough block structure corresponding to the topics.

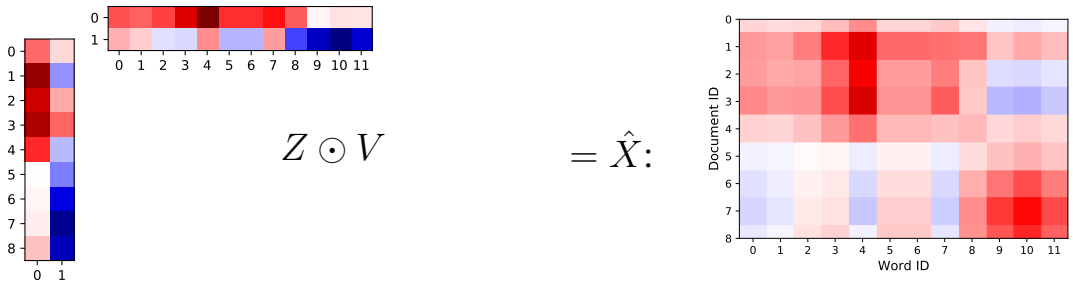
Documents:
0: "Human machine interface for ABC computer applications"
1: "A survey of user opinion of computer system response time"
2: "The EPS user interface management system"
3: "System and human system engineering testing of EPS"
4: "Relation of user perceived response time to error measurement"
5: "The generation of random, binary, ordered trees"
6: "The intersection graph of paths in trees"
7: "Graph minors IV: Widths of trees and well-quasi-ordering"
8: "Graph minors: A survey"

Words:
0: "human"
1: "interface"
2: "computer"
3: "user"
4: "system"
5: "response"
6: "time"
7: "EPS"
8: "survey"
9: "trees"
10: "graph"
11: "minors"

X:



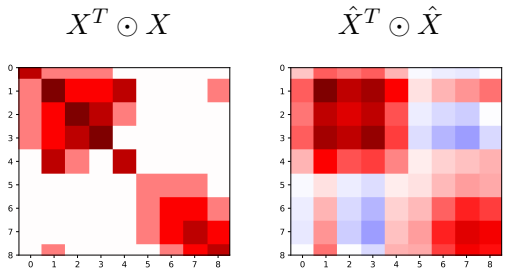
If we approximate the matrix X using a rank-2 matrix product, $X \approx Z \odot V$, we find that the "topics" (eigenvectors) found by SVD on X identifies one row, V_0, with positive (red) values for the HCI words, and less positive values for the graph theory words, and Z has $z_0^{(i)}$ positive for $i \in \{0, \ldots 4\}$, so that these words are likely to occur in the first five documents. The second row of V has strong negative values (blue) for the graph theory words, while $z_1^{(i)}$ is negative on documents $i \in \{5, 6, 7, 8\}$; thus their product is positive and indicates the graph theory words are likely to occur in these documents.



The rank-two approximation $\hat{X}$ looks like a smoother version of X, where the details of exactly which words were in which title has been lost, but the gist (which words "could have" been in each title) is more readily apparent.

In high-dimensional and sparse data, like text, using a latent space representation can also improve the quality of similarity or distance calculations compared to the original features.

For example, if we evaluate the inner product similarity for x by looking at the Gram matrix $X^T \odot X$, we see that within each block, most document pairs only share 1-2 words, so that (for example) document 1 is no more similar to document 0 (match on "computer") than it is to document 8 (match on "survey"). In contrast, the latent space version, $\hat{X}^T \odot \hat{X}$, reveals that document 1 is significantly more similar to all the HCI titles than to document 8.

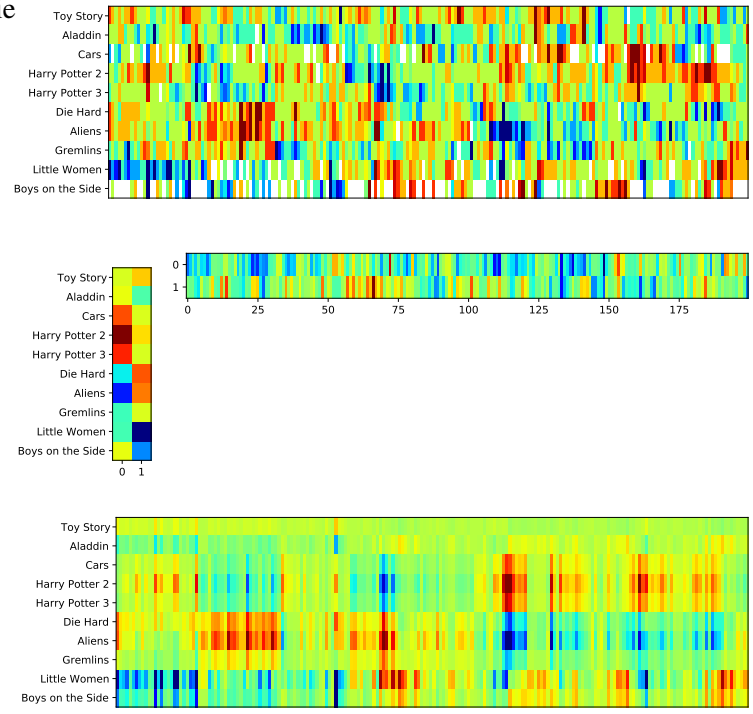


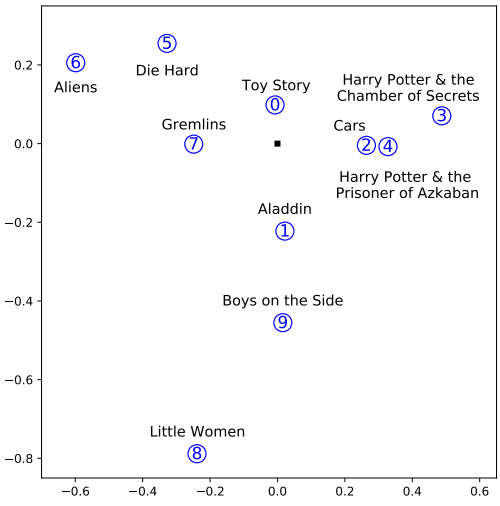
Collaborative Filtering

Example 11-3 : Movie ratings

Consider the median-centered movie rating data from Section 10.4, with users re-ordered so that similar sets of ratings are grouped visually.

If we apply PCA to these data (filling in zero for any missing ratings), and keep the first two principal directions, we obtain a two-dimensional representation of each movie (Z) and each user V, whose dot product approximates that user-movie rating. Looking at the reconstructed approximation $\hat{X}$, we can see that the ratings matrix looks "smoother", and that many of the patterns found by our clustering have become more obvious. (Note, however, that the clustering-based user order is only necessary for our visualization process – the quality of the approximation provided by PCA would be the same for any column ordering.)





We can also use the latent space representation to visualize the users or movies and their relative similarity to one another. Plotting the latent representation of each movie, $z^{(i)}$, allows us to interpret some meaning into the directions that have been identified. Each direction, for example z_0 (horizontal axis), indicates a preference by some users for one set of movies over another; so many users who liked the *Harry Potter* movies did not like *Aliens*, and vice versa. (A user j who liked both sets equally well will have $V_{0,j} \approx 0$.) Geometrically, each user is represented by a vector from the origin (black dot), with movies that are "far" in that vector direction predicted to have high ratings, and movies that are in the opposite direction predicted to have low ratings.

Example 11-4 : More movies

If we perform the same latent space process on a larger database of 200 movies, we see similar patterns emerge. Embedding the movies in $k = 4$ dimensions, we can get a sense of the meaning of each dimension by plotting the positions of some of the movies, and by listing the movies that would be most, and least, liked by a person whose latent representation was the unit vector e_i, e.g., $e_1 = [1, 0, 0, 0]$, etc. These visualizations make the first two dimensions of the embedding reasonably easy to interpret:

(a)

"Family oriented"
The Crow (1994)
Desperado (1995)
Ronin (1998)
From Dusk Till Dawn (1996)
Kill Bill: Vol. 1 (2003)
⋮ ⋮ ⋮
About a Boy (2002)
Beauty and the Beast (1991)
The Sound of Music (1965)
Witness (1985)
When Harry Met Sally (1989)

(Left to right axis)

(b)

"Blockbuster" vs. "Artistic"
Independence Day (1996)
Tomorrow Never Dies (1997)
Maverick (1994)
Twister (1996)
Mission: Impossible II (2000)
⋮ ⋮ ⋮
Fargo (1996)
Boogie Nights (1997)
Seven Samurai (1954)
Dr. Strangelove (1964)
Das Boot (1981)

(Bottom to top axis)

(c)

In panel (a), we illustrate the latent positions of 41 of the 200 movies in the first two dimensions. Panels (b) and (c) list the "most extreme" (positive and negative) movies in each of those directions. We see that the movies are automatically organized into reasonable positions, and that we can interpret the latent dimensions as having some intuitive meaning, with more family oriented movies (*Babe*, *Beauty and the Beast*) found on the right and more adult-audience movies (*From Dusk til Dawn*, *Big Lebowski*) found on the left. Meanwhile, big-budget blockbusters (*Independence Day*, *Titanic*) are located toward the bottom, while more artistic films (*Clockwork Orange*, *Citizen Kane*) are toward the top.

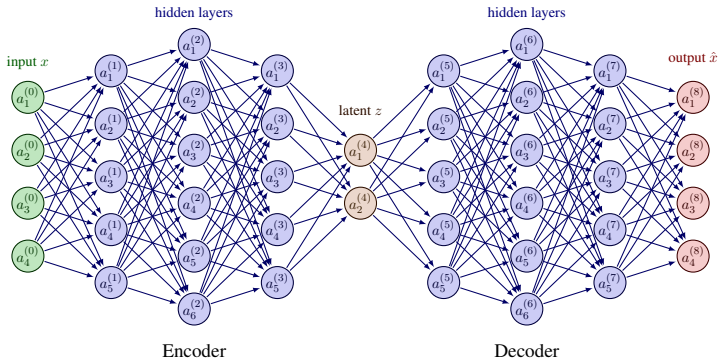
11.4 Autoencoder & predictive models

In order to extend the linear latent space analysis of PCA to more complex, nonlinear models, it is useful to adopt a "predictive" perspective that will allow us to bring to bear many of the techniques in previous chapters. In particular, we can view the dimensionality reduction problem as an encoding, or transformation from the original feature space (vector x) to a latent space (vector $z = f(x)$), in such a way that we can (approximately) recover or decode the original data ($\hat{x} = g(z) \approx x$). From this perspective, the models $f(\cdot)$ and $g(\cdot)$ form an **autoencoder**, a model that takes in x and predicts itself. Ideally, we have the property that $g(f(x)) \approx x$ for points x in or near the data distribution, while the function itself passes through a *bottleneck* (determined by the dimension of z).

When f and g are linear functions, $f(x) = V \odot x$ and $g(z) = W \odot z$, and we minimize the mean squared error $\mathbb{E}\Big[\|x = g(f(x))\|^2\Big]$, the resulting problem is simply that of principal component analysis. However, if we select a more general form for f and g, we can learn non-linear transformations of x. A popular framework is to define f and g using neural network-like models.

Example 11-5 : Neural network autoencoder

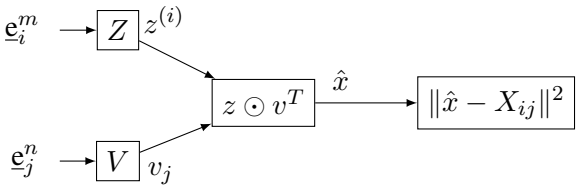
Discuss: multilayer neural network representation for f, g. The functions f and g can be trained simultaneously, with the values of z represented by the activation values at a "bottleneck" layer in the middle of the network.



Example on data? MNIST?

The predictive framework also enables other extensions of PCA-like latent space models. For example, suppose we are building a recommendation system, and wish to use collaborative filtering such as in Example 11-3. However, we *also* have access to a number of features about each user i and each movie j; when users join, we obtain basic demographic information (age, gender, country, etc.), and when movies are added, we have basic information about them as well (release date, genre, actors, director, budget). How can we create a system that incorporates both the basic features, which will be especially useful before a given user has rated many movies or before a movie has been rated by many users, and the user- and movie-specific representations found using dimensionality reduction?

We can start by writing the PCA model as a simple forward predictive model. Let $\underline{e}_i^m$ and $\underline{e}_j^n$ be the ith and jth Euclidean basis vectors in m and n dimensions, respectively – i.e., length n and m vectors of all zeros with a single "1" in the ith or jth position. Then, we can see that $z^{(i)} = (U \odot S)_i = \underline{e}_i^m \odot (U \odot S)$, and $v_j = \underline{e}_j^n \odot V$, leading to a predictive architecture that takes in i and j and estimates the rating $X_{ij} \approx z^{(i)} \odot v_j^T$ using the architecture shown at right.



It is then easy to extend this architecture in many ways. We can take in additional features and simply append them to $\underline{e}_i^n$ and $\underline{e}_j^m$, extending U and V as appropriate. We can change the simple dot product of each layer to something more complex, such as a neural network. We can also alter the prediction layer; for example, adding a nonlinearity to ensure that our ratings $\hat{x}$ are bounded in $\{1, \ldots, 10\}$. We can change the loss function, e.g., to be more robust to outliers and large errors, or to pay more attention to one type of error than another (perhaps we care more about errors in examples with high actual or predicted ratings, and less about movies that will be disliked). Finally, assuming our architecture is differentiable, we can train the entire predictive structure using generic optimization techniques such as stochastic gradient descent.

11.5 Extensions and Alternatives

- Factor analysis
- Alternative forms: direct minimization, tilde U * V
- Missing data etc.

- Optimize over entries of X subject to convex loss, trace norm etc?
- Nonlinear distance preserving embeddings? Isomap etc?
- Autoencoders & VAEs
- Autoencoder-like (word2vec context, etc)?

Example 11-6 : Word2Vec

- Associate representation z_w with each word w
- Dataset: pairs of words w and their context $\{w'\}$ (positive); w and "non-context" $\{\tilde{w}'\}$
- Now, train model; "CBOW" = use sum of $z_{w'}$ to predict w, e.g.,

$$\sigma\Big(\big(\sum_{w'} z_{w'}\big) \odot \theta_w \Big)$$

- or, skip-gram: use z_w to predict w'.

11.6 Implicit Embedding Methods

Redo

Another method to try to capture the structure in high-dimensional data comes from a geometric, **manifold learning** view. Consider the facial image data from Example 11-1. The data themselves consist of 24×24 greyscale values, i.e., they lie in a 576-dimensional hypercube. However, they clearly do not fill the hypercube; most of the locations in that 576-dimensional space do not look anything like faces. The set of images of faces must form a lower-dimensional manifold embedded within that space. If we can identify this manifold, we may be able to understand the "intrinsic dimensionality" of facial images (i.e., how many axes of variation there are). Relatedly, if we can find a low-dimensional description of the data, we may be able to sidestep the curse of dimensionality (see Section 2.4), since distances computed in the "true" low-dimensional space may be much more reliable than distances in the full, 576-dimensional space. On a practical note, we may want to find low-dimensional positions for our data in order to organize or plot our data for better visualization.

To do so, we will associate each data point $x^{(i)}$ with some lower-dimensional position $z^{(i)}$, and then directly optimize the values of the latent positions $\{z^{(i)}\}$ in order to preserve some property of the original data $\{x^{(i)}\}$ – for example, which other data are closest to (or most similar to) each point. In particular, we can contrast this section's "direct embedding" methods with the auto-encoder approach: in an auto-encoder, we searched for a parameterized pair of mappings $z = f(x)$ and $x = g(z)$, which we could use to "embed" each point x or reconstruct x given its embedding z. By representing the points $\{z^{(i)}\}$ explicitly, we do not need to know the form of these mappings f, g; we only work with the resulting positions $z^{(i)}$. This has the advantage of not requiring us to define f or g, and the ability to learn non-smooth mappings, but the disadvantage of being more difficult to apply to novel data points x that are not in our training set.

Multidimensional Scaling

Suppose that we are able to measure the dissimilarity between each pair of data points, D_{ij}. The framework of (metric) **multidimensional scaling** (MDS) attempts to find positions $z^{(i)}$ that preserve these dissimilarities. We define a loss function that measures how closely our new coordinates' Euclidean distances, $\|z^{(i)} - z^{(j)}\|$, match our desired dissimilarities D_{ij}, and then search for the positions $\{z^{(i)}\}$ that minimize this loss.

"Classical" MDS uses a particular loss in order to solve this optimization in closed form, as a low-rank matrix decomposition. In particular, we first compute the "centered" dissimilarity matrix,

$$B_{ij} = D^z_{ij} - \frac{1}{n}\sum_{j'} D_{ij'} - \frac{1}{n}\sum_{i'} D_{i'j} + \frac{1}{n^2}\sum_{i',j'} D_{i'j'},$$

i.e., we subtract the row- and column- averages and re-add the overall average. Then, we measure the accuracy of our new coordinates $z^{(i)}$ via the so-called *strain* loss:

$$J(\{z^{(i)}\}) = \sum_{i,j} \Big(B_{ij} - z^{(i)} \odot z^{(j)} \Big)^2$$

It turns out that the optimal k-dimensional locations $\{z^{(i)}\}$ are then given by an eigendecomposition of B, i.e., $B = U\Lambda U^T$ and $z = \Lambda^{\frac{1}{2}} U^T$.

Classical MDS is closely related to Principal Components Analysis, since if the original dissimilarities D_{ij} are themselves squared Euclidean distances between the original high-dimensional data $x^{(i)}$, and the $x^{(i)}$ are zero-mean, we have

$$D_{ij} = \|x^{(i)} - x^{(j)}\|^2 \qquad \Rightarrow \qquad B_{ij} = x^{(i)} \odot x^{(j)},$$

and, if we compute the singular value decomposition of the data matrix $X = U\,S\,V^T$, we find,

$$B = XX^T = U\,S\,V^T\,V\,S^T\,U^T = U\,(SS^T)\,U^T,$$

and thus $z = SU^T$, exactly the same positions as PCA selects. (For this reason, classical MDS is sometimes called principal coordinates analysis, or PCoA.)

More general forms of metric MDS which use different loss functions, such as the common "stress" loss,

$$J(\{z^{(i)}\}) = \sum_{i,j} \Big(D_{ij} - \|z^{(i)} - z^{(j)}\| \Big)^2,$$

cannot be solved easily in closed form and are typically solved to local optima through gradient descent. Even more generally, non-metric MDS methods attempt to match the *ordering* of the dissimilarities, rather than their precise value.

IsoMap

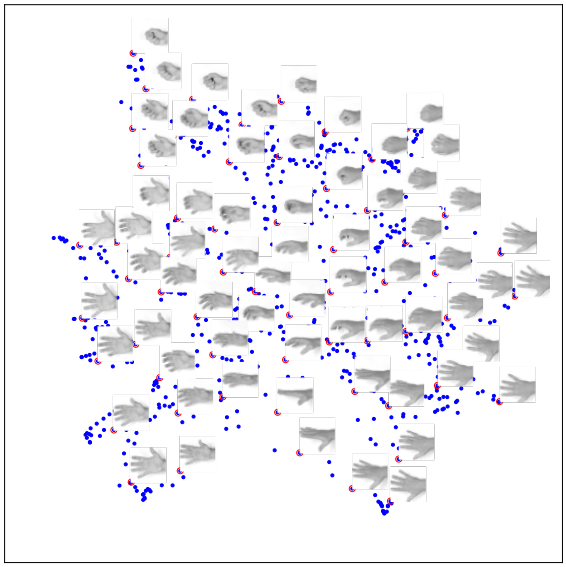
The IsoMap algorithm **?** extends classical MDS by creating a neighborhood-based embedding. Let us assume that the data $\{x^{(i)}\}$ live in a low-dimensional manifold embedded in $\mathbb{R}^n$. From this perspective, we may be able to identify a few nearby points as being similar to our target, but the usual, Euclidean notion of distance is not appropriate for measuring the similarity of distant points. Classical MDS requires the dissimilarity of each pair, D_{ij} – therefore, we will try to identify the local neighborhood of each point, and then use these "local geometry" relationshiops to infer the overall, global distances between each pair.

For each data point i, we select the k nearest data points as having reliable, "trusted" dissimilarities given by their Euclidean distances. This defines a graph over the data, which we make symmetric for convenience. We then define the dissimilarity of any untrusted pair of data as the shortest path within our graph, where path length is defined by the sum of dissimilarities along the edges. Performing any shortest-path algorithm (say, Djikstra), we can compute a full matrix of dissimilarities D and perform classical MDS, effectively extrapolating from our trustworthy distances to estimate more distant dissimilarities.

Example 11-7 : Hand images

In an example inspired by **?**, we can find the latent structure of a collection of images of a hand as it travels through two degrees of freedom: rotating around the wrist, and opening and closing the fingers. In this sense, the overall data should be representable using only two-dimensional coordinates. However, this relationship is hidden within the overall pixel-based observations $x^{(i)}$, which are 900 dimensional (corresponding to 30×30 images).

Select k neighbors; show image list of hands, "nearest" neighbor of each; overall 2D geometry of data



Locally linear embedding

Represent each point as a linear combination of neighbors; find new locations preserving that linear relationship

t-distributed stochastic neighbor embedding

A popular method for finding low-dimensional embeddings suitable for visualization is the t-distributed stochastic neighbor embedding, or t-SNE method. We can view t-SNE as a form of multidimensional scaling: we define a notion of (dis)similarity between each pair of data $x^{(i)}$ and $x^{(j)}$, and then search for positions in our low-dimensional space, $\{z^{(i)}\}$, that reproduce these similarities according to a particular loss function. For concreteness, let our dissimilarity among the $\{x^{(i)}\}$ be simply the squared Euclidean distance, $D_{ij} = \|x^{(i)} - x^{(j)}\|^2$. For t-SNE, we define a collection of probabilities that are proportional to this dissimilarity:

$$p_{j|i} = \frac{\exp(-\|x^{(i)} - x^{(j)}\|^2 / 2\sigma_i^2)}{\sum_{k \neq i} \exp(-\|x^{(i)} - x^{(k)}\|^2 / 2\sigma_i^2)},$$

with $p_{i|i} = 0$. Notice that $\sum_j p_{j|i} = 1$; we can interpret the value $p_{j|i}$ as the probability of data point i picking data point j as its neighbor, and this particular choice of $p_{j|i}$ as doing so proportionally to each $x^{(j)}$'s probability under a Gaussian distribution centered at $x^{(i)}$ with variance σ_i^2. (We examine how to

select σ_i^2 in the sequel.) These weights are also analagous to those used in weighted nearest neighbor estimation, for example (see Section **??**).

We next make these similarity scores symmetric and normalize them, by taking $p_{ij} = \frac{1}{2m}(p_{j|i} + p_{i|j})$, so that $\sum_{i,j} p_{ij} = 1$. For a given set of embedded locations $\{z^{(i)}\}$, we define

$$q_{ij} = \frac{(1 + \|z^{(i)} - z^{(j)}\|^2)^{-1}}{\sum_{l \neq k}((1 + \|z^{(k)} - z^{(l)}\|^2)^{-1}},$$

i.e., proportionally to a student t-distribution centered at $z^{(i)}$. Finally, we score the resulting q according to its KL-divergence from p: $J(\{z^{(i)}\}) = \sum_{i \neq j} p_{ij} \log(p_{ij}/q_{ij})$.

Much like other multidimensional scaling optimizations, this loss cannot be minimized in closed form; instead, we initialize the $z^{(i)}$ at random or using some simpler embedding technique, and then optimize their values using gradient descent.

The t-SNE method can be sensitive to the choices of bandwidth σ_i, which determines the effective neighborhood of data point i. As with weighted neighborhood methods, we may wish to set σ_i smaller in regions with higher density of data. A typical method for t-SNE is to set each σ_i such that the entropy of the distribution $p_{j|i}$ is equal to some constant, which is then taken to be a hyperparameter of the model; higher entropy values result in larger effective neighborhoods for each data point.

Example 11-8 : t-SNE for hand images

Reprising Example 11-7, we can perform a t-SNE embedding of the same 30×30 grayscale images of rotating and closing hands, to identify a two-dimensional space in which they can be easily visualized. Like IsoMap, the resulting t-SNE embedding discovers the general structure of the data, finding two-dimensional positions for each image that generally preserve similarity, and from which we can see the primary axes of rotation and opening/closing. (As with Example 11-7, the resulting data locations are rotated to better highlight the axes' intution.)