Chapter 8: Dimensionality Reduction

Warren Alphonso

March 2019

ML problems can involve millions of features for each training instance. This makes training extremely slow and also makes it significantly harder to find a good generalizable solution. This is known as the *curse of dimensionality*.

1 The Curse of Dimensionality

Most points in a high-dimensional hypercube are very close to the border. For example, f we pick two points randomly in a unit square, the distance between them will be 0.52 on average. If jwe pick two random points in a 1,000,000-dimensional hypercube, the average distance will be about 408.25. This implies that high-dimensional datasets are likely to be very sparse and also that a new instance will likely be far away from any other training instance, which means our predictions will be much less reliable.

2 Main Approaches for Dimensionality Reduction

2.1 Projection

In most real-world problems, training instances are *not* spread out uniformly across all dimensions. Training instances actually lie within a much lower-dimensional space. One strategy to reduce dimensionality is to project every training instance perpendicularly onto this subspace. However, this is not always reliable because in many cases the subspace may twist or turn, such as in the famous *Swiss roll* dataset. Simply projecting onto a plane would squash different layers of the Swiss roll together.

2.2 Manifold Learning

The Swiss roll is an example of a 2D *manifold*. A 2D manifold is a 2D shape that can be bent or twisted in a higher-dimensional space. Dimensionality redution algorithms work by modeling the *manifold* on which the training instances lie. This is called *Manifold Learning*. If relies on the *manifold assumption*: that most

real-world high-dimensional datasets lie close to a much lower-dimensional manifold. This is empirically proven to usually be true. The manifold assumption is also followed by another assumption: that the task we are doing (classification or regression) is simpler if expressed in lower-dimensional space of the manifold. This assumption is not always true.

3 PCA

Principal Component Analysis (PCA) is the most popular dimensionality reduction algorithm. First, it identifies the hyperplane that lies closest to the data, and then it projects the data onto it.

3.1 Preserving the Variance

We want to select the axis that preserves the maximum amount of variance because it will most likely lose less information than other projections. Another justification for this is that we want to choose the axis that minimizes the mean squared distance between the original dataset and its projection onto that axis. This is the main idea behind PCA.

3.2 Principal Components

PCA identifies the axis that accounts for the largest amount of variance in the training set. It also finds a second axis, *orthogonal to the first one*, that accounts for the largest amount of remaining variance. It continues to find axes, orthogonal to previous axes, until it has as many axes as the number of dimensions in the dataset. The unit vector that defines the i^{th} axis is called the i^{th} *principal component*.
In order to find the principal components of a training set, we use a standard matrix factorization technique called *Singular Value Decomposition* (SVD) that can decompose the training set matrix X into the matrix multiplication of three matrices U, Σ, V^T, where V contains all the principal components that we are looking for:

$$V = \begin{pmatrix} \vec{c_1} & \vec{c_2} & ... & \vec{c_n} \\ & & & \end{pmatrix}$$

Figure 1: Principal components matrix

The following code uses NumPy's `svd()` function to get all the principal components and then extracts the first two PCs:

```
X_centered = X - X.mean(axis=0)
U, s, Vt = np.linalg.svd(X_centered)
c1 = Vt.T[:, 0]
c2 = Vt.T[:, 1]
```

PCA assumes the dataset is centered around the origin. Scikit-Learn's PCA classes take care of centering the data but if we implement PCA ourselves, we must center the data first manually.

3.3 Projecting Down to d Dimensions

After identifying all the principal components, we reduce the dimensionality of the datset down to d dimensions by projecting it onto the hyperplane defined by the first d principal components. To project the training set onto the hyperplane, we compute the matrix multiplication of the training set matrix X by the matrix W_d, defined as the matrix containing the first d principal components.

$$X_{d-proj} = XW_d$$

The following code projects theh training set onto the plane defined by the first two principal components:

```
W2 = Vt.T[:, :2]
X2D = X_centered.dot(W2)
```

3.4 Explained Variance Ratio

Another very useful piece of information is the *explained variance ratio*. It indicates the proportion of the dataset's variance that lies along the axis of each principal component.

```
from sklearn.decomposition import PCA

pca = PCA(n_components = 2)
X2D = pca.fit_transform(X)

pca.explained_variance_ratio_

->  array([0.84248607, 0.14631839])
```

This tells us that 84.2% of the dataset's variance lies along the first axis and 14.6% lies along the second axis.

3.5 Choosing the Right Number of Dimensions

Instead of arbitrarily choosing the number of dimensions to reduce down to, it is preferable to choose the number of dimensions that add up to a sufficiently large portion of the variance (usually 95%). This is a good idea unless we are reducing dimensionality for data visualization, in which case we want to reduce to 2 or 3 dimensions.
The following code computes PCA without reducing dimensionality, then computes the minimum number of dimensions required to preserve 95% of variance:

```
pca = PCA()
pca.fit(X_train)
cumsum = np.cumsum(pca.explained_variance_ratio_)
d = np.argmax(cumsum >= 0.95) + 1
```

Now we could set `n_components = d` and run PCA again. A better way to do this in one step is to run the following code:

```
pca = PCA(n_components=0.95)
X_reduced = pca.fit_transform(X_train)
```

3.6 PCA for Compression

If we try applying PCA to MNIST while preserving 95% of its variance, we will have just over 150 features instead of the original 784 features. It is possible to decompress the reduced dataset back to 784 dimensions by applying the inverse transformation of the PCA projection. This will not give us the original data since the projection lost some information, but it will likely be very close to the original data. The mean squared distance between the original data and the reconstructed data is called the *reconstruction error*.
The following code reconstructs a reduced dataset:

```
pca = PCA(n_components = 154)
X_reduced = pca.fit_transform(X_train)
X_recovered = pca.inverse_transform(X_reduced)
```

The equation of the PCA inverse transformation, back to the original number of dimensions:

$$X_{recovered} = X_{d-proj} W_d^T$$

4 Kernel PCA

In Chapter 5, we discussed the kernel trick, a mathematical technique that implicitly maps instances into a very high-dimensional space (called the feature space). We can use the same trick on PCA, making it possible to perform complex nonlinear projections for dimensionality reduction. This is called *Kernel PCA*. It is usually good at preserving clusters of instances after projection.

4.1 Selecting a Kernel and Tuning Hyperparameters

As kPCA is an unsupervised learning algorithm, there is no clear performance measure to help you select the best kernel and hyperparameter values. However, dimensionality reduction is often a preparation step for a supervised learning task so we can use grid search to select the kernel and hyperparameters that lead to the best performance.
Another approach is to select the kernel and hyperparameters that yield the lowest reconstruction error. However, reconstruction is not as easy with linear PCA because the kernel trick is mathematically equivalent ot mapping the training set to an infinite-dimensional feature space, then projecting the transformed training set down using linear PCA. We cannot do the same for linear because inverting the linear PCA would project the point in infinite-dimensional space.

5 LLE

Locally Linear Embedding (LLE) is another very powerful nonlinear dimensionality reduction technique. It is a Manifold Learning technique that does not rely on projections like the previous algorithms. In short, LLE works by measuring how each training instance linearly relates to its closest neighbors and then looking for a low-dimensional representation of the training set where these local relationships are most preserved. It is particularly good at unrolling twisted manifolds when there is not much noise.
Here's how LLE works: for each training instance $x^{(i)}$, the algorithm identifies its k closest neighbors, then tries to reconstruct $x^{(i)}$ as a linear function of these neighbors. Specifically, it finds the weights $w_{i,j}$ such that the squared distance between $x^{(i)}$ and $\sum_{j=1}^{m} w_{i,j}x^{(j)}$ is as small as possible assuming $w_{i,j} = 0$ if $x^{(j)}$ is not one of the k closest neighbors of $x^{(i)}$. After this the weight matrix encodes the local linear relationships between the training instances. Next, the algorithm maps the training instances into a d-dimensional space while preserving these local relationships as much as possible.