# Hands-on ML

Dimensionality reduction

The main motivations for dimensionality reduction are:

* To speed up a subsequent training algorithm (in some cases it may even remove noise and redundant features, making the training algorithm per‐ form better).
* To visualize the data and gain insights on the most important features.
* Simply to save space (compression).

The main drawbacks are:

* Some information is lost, possibly degrading the performance of subse‐ quent training algorithms.
* It can be computationally intensive.
* It adds some complexity to your Machine Learning pipelines.
* Transformed features are often hard to interpret

The curse of dimensionality refers to the fact that many problems that do not exist in low-dimensional space arise in high-dimensional space. randomly sampled high- dimensional vectors are generally very sparse, increasing the risk of overfitting and making it very difficult to identify patterns in the data without having plenty of training data.

In theory, one solution to the curse of dimensionality could be to increase the size of the training set to reach a sufficient density of training instances. Unfortunately, in practice, the number of training instances required to reach a given density grows exponentially with the number of dimensions

With just 100 features (much less than in the MNIST problem), you would need more training instances than atoms in the observable universe in order for training instances to be within 0.1 of each other on average, assuming they were spread out uniformly across all dimensions. s-on ML

* Projection

In most real-world problems, training instances are *not* spread out uniformly across all dimensions. Many features are almost constant, while others are highly correlated (as discussed earlier for MNIST). As a result, all training instances actually lie within (or close to) a much lower-dimensional *subspace* of the high-dimensional space

* Manifold Learning

Put simply, a 2D manifold is a 2D shape that can be bent and twisted in a higher-dimensional space. More generally, a *d*-dimensional manifold is a part of an *n*-dimensional space (where *d* < *n*) that locally resembles a *d*-dimensional hyperplane

It relies on the *manifold assumption*, also called the *manifold hypothesis*, which holds that most real-world high-dimensional datasets lie close to a much lower-dimensional manifold. This assumption is very often empirically observed.

If we reduce the dimensionality of your training set before training a model, it will definitely speed up training, and may also lead to a better or simpler solution; it all depends on the dataset.

PCA

*Principal Component Analysis* identifies the hyperplane that lies closest to the data, and then it projects the data onto it. Before you can project the training set onto a lower-dimensional hyperplane, you first need to choose the right hyperplane.

It seems reasonable to select the axis that preserves the maximum amount of variance, as it will most likely lose less information than the other projections. Another way to justify this choice is that it is the axis that minimizes the mean squared distance between the original dataset and its projection onto that axis

1. Identifying the 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. If it were a higher-dimensional data‐ set, PCA would also find a third axis, orthogonal to both previous axes, and a fourth, a fifth, and so on—as many axes as the number of dimensions in the dataset

*Singular Value Decomposition* (SVD) that can decompose the training set matrix **X** into the dot product of three matrices **U** · Σ · **V***T*, where **V***T* contains all the principal components that we are looking for

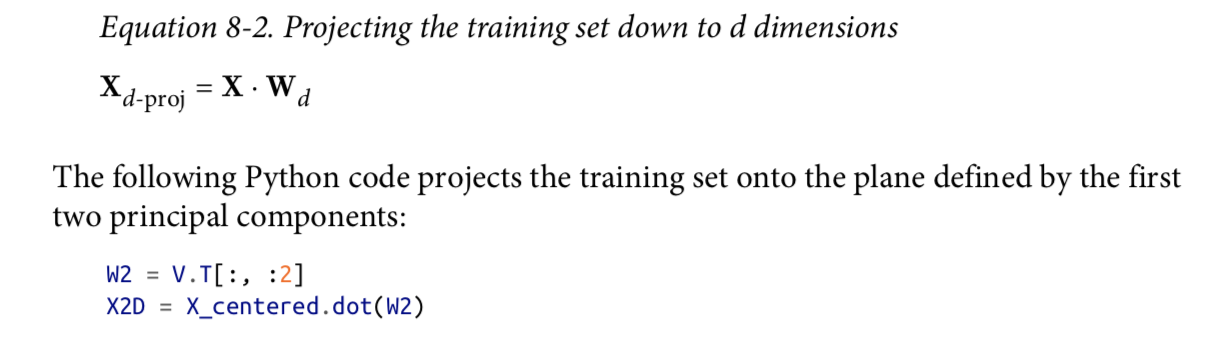
A screenshot of a cell phone

Description automatically generated

PCA assumes that the dataset is centred around the origin. As we will see, Scikit-Learn’s PCA classes take care of centring the data for you. However, if you implement PCA yourself (as in the pre‐ ceding example), or if you use other libraries, don’t forget to centre the data first.

1. Projecting Down to d Dimensions

To project the training set onto the hyperplane, you can simply compute the dot product of the training set matrix X by the matrix Wd, defined as the matrix containing the first *d* principal components.



Scikit-Learn’s PCA class implements PCA using SVD decomposition just like we did before. The following code applies PCA to reduce the dimensionality of the dataset down to two dimensions (note that it automatically takes care of centring the data):

A picture containing knife

Description automatically generated

Another very useful piece of information is the *explained variance ratio* of each principal component, available via the explained\_variance\_ratio\_ variable. It indicates the proportion of the dataset’s variance that lies along the axis of each principal component.

1. Choosing the Right Number of Dimensions

Instead of arbitrarily choosing the number of dimensions to reduce down to, it is generally preferable to choose the number of dimensions that add up to a sufficiently large portion of the variance (e.g., 95%). Unless, of course, you are reducing dimensionality for data visualization—in that case you will generally want to reduce the dimensionality down to 2 or 3.

The following code computes PCA without reducing dimensionality, then computes the minimum number of dimensions required to preserve 95% of the training set’s variance:

A picture containing knife

Description automatically generated

You could then set n\_components=d and run PCA again. However, there is a much better option: instead of specifying the number of principal components you want to preserve, you can set n\_components to be a float between 0.0 and 1.0, indicating the ratio of variance you wish to preserve:

A picture containing orange, red, player, man

Description automatically generated

PCA for Compression

Obviously after dimensionality reduction, the training set takes up much less space. For example, try applying PCA to the MNIST dataset while preserving 95% of its var‐ iance. You should find that each instance will have just over 150 features, instead of the original 784 features. So while most of the variance is preserved, the dataset is now less than 20% of its original size!

It is also possible to decompress the reduced dataset back to 784 dimensions by applying the inverse transformation of the PCA projection. Of course this won’t give you back the original data, since the projection lost a bit of information (within the 5% variance that was dropped), but it will likely be quite close to the original data.

The mean squared distance between the original data and the reconstructed data (compressed and then decompressed) is called the *reconstruction error*.

For example, the following code compresses the MNIST dataset down to 154 dimensions, then uses the inverse\_transform() method to decompress it back to 784 dimensions.

A picture containing knife

Description automatically generated

The equation of the inverse transformation is

A close up of a logo

Description automatically generated

Incremental PCA

One problem with the preceding implementation of PCA is that it requires the whole training set to fit in memory in order for the SVD algorithm to run. Fortunately, *Incremental PCA* (IPCA) algorithms have been developed: you can split the training set into mini-batches and feed an IPCA algorithm one mini-batch at a time. This is useful for large training sets, and also to apply PCA online

The following code splits the MNIST dataset into 100 mini-batches (using NumPy’s array\_split() function) and feeds them to Scikit-Learn’s IncrementalPCA class5 to reduce the dimensionality of the MNIST dataset down to 154 dimensions (just like before). Note that you must call the partial\_fit() method with each mini-batch rather than the fit() method with the whole training set:

A screenshot of a cell phone

Description automatically generated

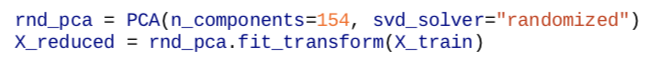
Alternatively, you can use NumPy’s memmap class, which allows you to manipulate a large array stored in a binary file on disk as if it were entirely in memory; the class loads only the data it needs in memory, when it needs it. Since the IncrementalPCA class uses only a small part of the array at any given time, the memory usage remains under control. This makes it possible to call the usual fit() method, as you can see in the following code:

A picture containing knife

Description automatically generated

Randomized PCA

Scikit-Learn offers yet another option to perform PCA, called *Randomized PCA*. This is a stochastic algorithm that quickly finds an approximation of the first *d* principal components. Its computational complexity is *O*(*m* × *d*2) + *O*(*d*3), instead of *O*(*m* × *n*2) + *O*(*n*3), so it is dramatically faster than the previous algorithms when *d* is much smaller than *n*.



Kernel PCA

The kernel trick, a mathematical technique that implicitly maps instances into a very high-dimensional space (called the *feature space*), enabling nonlinear classification and regression with Support Vector Machines. *Kernel PCA* (kPCA) is good at preserving clusters of instances after projection, or sometimes even unrolling datasets that lie close to a twisted manifold.

For example, the following code uses Scikit-Learn’s KernelPCA class to perform kPCA with an RBF kernel

A picture containing knife

Description automatically generated

Figure 8-10 shows the Swiss roll, reduced to two dimensions using a linear kernel (equivalent to simply using the PCA class), an RBF kernel, and a sigmoid kernel (Logistic).

A close up of a piece of paper

Description automatically generated

Selecting a kernel and Tuning Hyperparameters

Incremental PCA is useful for large datasets that don’t fit in memory, but it is slower than regular PCA, so if the dataset fits in memory you should prefer regular PCA. Incremental PCA is also useful for online tasks, when you need to apply PCA on the fly, every time a new instance arrives. Randomized PCA is useful when you want to considerably reduce dimensionality and the dataset fits in memory; in this case, it is much faster than regular PCA. Finally, Kernel PCA is useful for nonlinear datasets.

1. Using GridSearchCV

Dimensionality reduction is often a preparation step for a supervised learning task (e.g., classification), so you can simply use grid search to select the kernel and hyper‐ parameters that lead to the best performance on that task. For example, the following code creates a two-step pipeline, first reducing dimensionality to two dimensions using kPCA, then applying Logistic Regression for classification. Then it uses Grid SearchCV to find the best kernel and gamma value for kPCA in order to get the best classification accuracy at the end of the pipeline:

A screenshot of a social media post

Description automatically generated

1. Lowest reconstruction error

Another approach, this time entirely unsupervised, is to select the kernel and hyper‐ parameters that yield the lowest reconstruction error. However, reconstruction is not as easy as with linear PCA

If we could invert the linear PCA step for a given instance in the reduced space, the reconstructed point would lie in feature space, not in the original space (e.g., like the one represented by an x in the diagram). Since the feature space is infinite-dimensional, we cannot compute the reconstructed point, and therefore we cannot compute the true reconstruction error.

Fortunately, it is possible to find a point in the original space that would map close to the reconstructed point. This is called the reconstruction *pre-image*. Once you have this pre-image, you can measure its squared distance to the original instance. You can then select the kernel and hyperparameters that minimize this reconstruction pre-image error.

Train a supervised regression model, with the projected instances as the training set and the original instances as the targets. Scikit-Learn will do this automatically if you set fit\_inverse\_transform=True, as shown in the following code:

Now you can use grid search with cross-validation to find the kernel and hyperpara‐ meters that minimize this pre-image reconstruction error.

A screenshot of a social media post

Description automatically generated

Locally Linear Embedding

LLE is a Manifold Learning technique that does not rely on projections like the previous algorithms. In a nutshell, LLE works by first measuring how each training instance linearly relates to its closest neighbours (c.n.), and then looking for a low-dimensional representation of the training set where these local relationships are best preserved. This makes it particularly good at unrolling twisted manifolds, especially when there is not too much noise.

Here’s how LLE works: first, for each training instance x(i), the algorithm identifies its k closest neighbors (in the preceding code k = 10), then tries to reconstruct x(i) as a linear function of these neighbors. More specifically, it finds the weights wi,j such that the squared distance between x(i) and ∑m w x j is as small as possible, assuming wi,j = 0 if x(j) is not one of the k closest neighbors of x(i). Thus the first step of LLE is the constrained optimization problem described in Equation 8-4, where W is the weight matrix containing all the weights wi,j. The second constraint simply normalizes the weights for each training instance x(i).

A screenshot of a cell phone

Description automatically generated

The second step is to map the training instances into a d-dimensional space (where d < n) while preserving these local relationships as much as possible. If z(i) is the image of x(i) in this d-dimensional space, then we want the squared distance between z(i) and ∑m w z j to be as small as possible. This idea leads to the unconstrained optimization problem described in Equation 8-5. It looks very similar to the first step, but instead of keeping the instan‐ces fixed and finding the optimal weights, we are doing the reverse: keeping the weights fixed and finding the optimal position of the instances’ images in the low- dimensional space. Note that Z is the matrix containing all z(i).\

A close up of a logo

Description automatically generated

Scikit-Learn’s LLE implementation has the following computational complexity: *O*(*m* log(*m*)*n* log(*k*)) for finding the *k* nearest neighbors, *O*(*mnk*3) for optimizing the weights, and *O*(*dm*2) for constructing the low-dimensional representations. Unfortu‐ nately, the *m*2 in the last term makes this algorithm scale poorly to very large datasets.

Other Dimensionality Reduction Techniques

There are many other dimensionality reduction techniques, several of which are available in Scikit-Learn. Here are some of the most popular:

* *Isomap* creates a graph by connecting each instance to its nearest neighbors, then reduces dimensionality while trying to preserve the *geodesic distances*9 between the instances.
* *t-Distributed Stochastic Neighbor Embedding* (t-SNE) reduces dimensionality while trying to keep similar instances close and dissimilar instances apart. It is mostly used for visualization, in particular to visualize clusters of instances in high-dimensional space (e.g., to visualize the MNIST images in 2D).
* *Linear Discriminant Analysis* (LDA) is actually a classification algorithm, but dur‐ ing training it learns the most discriminative axes between the classes, and these axes can then be used to define a hyperplane onto which to project the data. The benefit is that the projection will keep classes as far apart as possible, so LDA is a good technique to reduce dimensionality before running another classification algorithm such as an SVM classifier.
* *Multidimensional Scaling* (MDS) reduces dimensionality while trying to preserve the distances between the instances

A screenshot of a map

Description automatically generated