Data Analysis and Machine Learning: Preprocessing and Dimensionality Reduction

Morten Hjorth-Jensen[1,2]

[1]Department of Physics, University of Oslo
[2]Department of Physics and Astronomy and National Superconducting Cyclotron Laboratory, Michigan State University

Oct 19, 2019

Reducing the number of degrees of freedom, overarching view

Many Machine Learning problems involve thousands or even millions of features for each training instance. Not only does this make training extremely slow, it can also make it much harder to find a good solution, as we will see. This problem is often referred to as the curse of dimensionality. Fortunately, in real-world problems, it is often possible to reduce the number of features considerably, turning an intractable problem into a tractable one.

Here we will discuss some of the most popular dimensionality reduction techniques: the principal component analysis PCA, Kernel PCA, and Locally Linear Embedding (LLE). Furthermore, we will start by looking at some simple preprocessing of the data which allow us to rescale the data.

Preprocessing our data

Before we proceed however, we will discuss how to preprocess our data. Till now and in connection with our previous examples we have not met so many cases where we are too sensitive to the scaling of our data. Normally the data may need a rescaling and/or may be sensitive to extreme values. Scaling the data renders our inputs much more suitable for the algorithms we want to employ.

Scikit-Learn has several functions which allow us to rescale the data, normally resulting in much better results in terms of various accuracy scores. The **StandardScaler** function in **Scikit-Learn** ensures that for each feature/predictor we study the mean value is zero and the variance is one (every column in the design/feature matrix). This scaling has the drawback that it

does not ensure that we have a particular maximum or minimum in our data set. Another function included in **Scikit-Learn** is the **MinMaxScaler** which ensures that all features are exactly between 0 and 1. The

More preprocessing

The **Normalizer** scales each data point such that the feature vector has a euclidean length of one. In other words, it projects a data point on the circle (or sphere in the case of higher dimensions) with a radius of 1. This means every data point is scaled by a different number (by the inverse of it's length). This normalization is often used when only the direction (or angle) of the data matters, not the length of the feature vector.

The **RobustScaler** works similarly to the StandardScaler in that it ensures statistical properties for each feature that guarantee that they are on the same scale. However, the RobustScaler uses the median and quartiles, instead of mean and variance. This makes the RobustScaler ignore data points that are very different from the rest (like measurement errors). These odd data points are also called outliers, and might often lead to trouble for other scaling techniques.

Simple preprocessing examples, Franke function and regression

Simple preprocessing examples, breast cancer data and classification, Support Vector Machines

We show here how we can use a simple regression case on the breast cancer data using support vector machines (SVM) as algorithm for classification.

More on Cancer Data, now with Logistic Regression

Why should we think of reducing the dimensionality

In addition to the plot of the features, we study now also the covariance (or rather the correlation matrix). We use also **Pandas** to compute the correlation matrix.

Basic ideas of the Principal Component Analysis (PCA)

Introducing the Covariance and Correlation functions

Classical PCA Theorem

Prof of the PCA Theorem

Getting started with PCA

Principal Component Analysis

Principal Component Analysis (PCA) is by far 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.

The following Python code uses NumPy's **svd()** function to obtain all the principal components of the training set, then extracts the first two principal components

PCA assumes that the dataset is centered around the origin. Scikit-Learn's PCA classes take care of centering the data for you. However, if you implement PCA yourself (as in the preceding example), or if you use other libraries, don't forget to center the data first.

Once you have identified all the principal components, you can reduce the dimensionality of the dataset down to d dimensions by projecting it onto the hyperplane defined by the first d principal components. Selecting this hyperplane ensures that the projection will preserve as much variance as possible.

PCA and scikit-learn

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 centering the data): After fitting the PCA transformer to the dataset, you can access the principal components using the components variable (note that it contains the PCs as horizontal vectors, so, for example, the first principal component is equal to 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. More material to come here.

More on the PCA

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: 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:

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 minibatch at a time. This is useful for large training sets, and also to apply PCA online (i.e., on the fly, as new instances arrive).

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 $O(m \times d^2) + O(d^3)$, instead of $O(m \times n^2) + O(n^3)$, so it is dramatically faster than the previous algorithms when d is much smaller than n.

Kernel PCA

The kernel trick is 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. Recall that a linear decision boundary in the high-dimensional feature space corresponds to a complex nonlinear decision boundary in the original space. It turns out that the same trick can be applied to PCA, making it possible to perform complex nonlinear projections for dimensionality reduction. This is called Kernel PCA (kPCA). It is often 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

LLE

Locally Linear Embedding (LLE) is another very powerful nonlinear dimensionality reduction (NLDR) technique. It 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 neighbors (c.n.), and then looking for a low-dimensional representation of the training set where these local relationships are best preserved (more details shortly).

Other techniques

There are many other dimensionality reduction techniques, several of which are available in Scikit-Learn.

Here are some of the most popular:

- **Multidimensional Scaling (MDS)** reduces dimensionality while trying to preserve the distances between the instances.
- **Isomap** creates a graph by connecting each instance to its nearest neighbors, then reduces dimensionality while trying to preserve the geodesic distances 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 during 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 a Support Vector Machine (SVM) classifier discussed in the SVM lectures.