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

Jan 2, 2020

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.

Principal component analysis and its various variants deal with the problem of fitting a low-dimensional affine subspace to a set of of data points in a high-dimensional space. With its family of methods it is one of the most used tools in data modeling, compression and visualization.

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.

In the above example we note two things. In the first plot we display the overlap of benign and malignant tumors as functions of the various features in the Wisconsing breast cancer data set. We see that for some of the features we can distinguish clearly the benign and malignant cases while for other features we cannot. This can point to us which features may be of greater interest when we wish to classify a benign or not benign tumour.

In the second figure we have computed the so-called correlation matrix, which in our case with thirty features becomes a 30×30 matrix.

We constructed this matrix using **pandas** via the statements and then

Diagonalizing this matrix we can in turn say something about which features are of relevance and which are not. But before we proceed we need to define covariance and correlation matrices. This leads us to the classical Principal Component Analysis (PCA) theorem with applications.

Basic ideas of the Principal Component Analysis (PCA)

The principal component analysis deals with the problem of fitting a low-dimensional affine subspace S of dimension d much smaller than the totaldimension D of the problem at hand (our data set). Mathematically it can be formulated as a statistical problem or a geometric problem. In our discussion of the theorem for the classical PCA, we will stay with a statistical approach. This is also what set the scene historically which for the PCA.

We have a data set defined by a design/feature matrix $\boldsymbol{X}$ (see below for its definition)

- Each data point is determined by p extrinsic (measurement) variables
- We may want to ask the following question: Are there fewer intrinsic variables (say $d << p$) that still approximately describe the data?
- If so, these intrinsic variables may tell us something important and finding these intrinsic variables is what dimension reduction methods do.

Introducing the Covariance and Correlation functions

Before we discuss the PCA theorem, we need to remind ourselves about the definition of the covariance and the correlation function. These are quantities

Suppose we have defined two vectors $\hat{x}$ and $\hat{y}$ with n elements each. The covariance matrix $\boldsymbol{C}$ is defined as

$$\boldsymbol{C}[\boldsymbol{x},\boldsymbol{y}] = \begin{bmatrix} \mathrm{cov}[\boldsymbol{x},\boldsymbol{x}] & \mathrm{cov}[\boldsymbol{x},\boldsymbol{y}] \\ \mathrm{cov}[\boldsymbol{y},\boldsymbol{x}] & \mathrm{cov}[\boldsymbol{y},\boldsymbol{y}] \end{bmatrix},$$

where for example

$$\mathrm{cov}[\boldsymbol{x},\boldsymbol{y}] = \frac{1}{n}\sum_{i=0}^{n-1}(x_i-\overline{x})(y_i-\overline{y}).$$

With this definition and recalling that the variance is defined as

$$\mathrm{var}[\boldsymbol{x}] = \frac{1}{n}\sum_{i=0}^{n-1}(x_i-\overline{x})^2,$$

we can rewrite the covariance matrix as

$$\boldsymbol{C}[\boldsymbol{x},\boldsymbol{y}] = \begin{bmatrix} \mathrm{var}[\boldsymbol{x}] & \mathrm{cov}[\boldsymbol{x},\boldsymbol{y}] \\ \mathrm{cov}[\boldsymbol{x},\boldsymbol{y}] & \mathrm{var}[\boldsymbol{y}] \end{bmatrix}.$$

The covariance takes values between zero and infinity and may thus lead to problems with loss of numerical precision for particularly large values. It is common to scale the covariance matrix by introducing instead the correlation matrix defined via the so-called correlation function

$$\mathrm{corr}[\boldsymbol{x},\boldsymbol{y}] = \frac{\mathrm{cov}[\boldsymbol{x},\boldsymbol{y}]}{\sqrt{\mathrm{var}[\boldsymbol{x}]\mathrm{var}[\boldsymbol{y}]}}.$$

The correlation function is then given by values $\mathrm{corr}[\boldsymbol{x},\boldsymbol{y}] \in [-1,1]$. This avoids eventual problems with too large values. We can then define the correlation matrix for the two vectors $\boldsymbol{x}$ and $\boldsymbol{y}$ as

$$\boldsymbol{K}[\boldsymbol{x},\boldsymbol{y}] = \begin{bmatrix} 1 & \mathrm{corr}[\boldsymbol{x},\boldsymbol{y}] \\ \mathrm{corr}[\boldsymbol{y},\boldsymbol{x}] & 1 \end{bmatrix},$$

In the above example this is the function we constructed using **pandas**.

Correlation Function and Design/Feature Matrix

In our derivation of the various regression algorithms like **Ordinary Least Squares** or **Ridge regression** we defined the design/feature matrix $\boldsymbol{X}$ as

$$\boldsymbol{X} = \begin{bmatrix} x_{0,0} & x_{0,1} & x_{0,2} & \dots & \dots x_{0,p-1} \\ x_{1,0} & x_{1,1} & x_{1,2} & \dots & \dots x_{1,p-1} \\ x_{2,0} & x_{2,1} & x_{2,2} & \dots & \dots x_{2,p-1} \\ \dots & \dots & \dots & \dots\dots & \dots \\ x_{n-2,0} & x_{n-2,1} & x_{n-2,2} & \dots & \dots x_{n-2,p-1} \\ x_{n-1,0} & x_{n-1,1} & x_{n-1,2} & \dots & \dots x_{n-1,p-1} \end{bmatrix},$$

with $\boldsymbol{X} \in \mathbb{R}^{n\times p}$, with the predictors/features p refering to the column numbers and the entries n being the row elements. We can rewrite the design/feature matrix in terms of its column vectors as

$$\boldsymbol{X} = \begin{bmatrix} \boldsymbol{x}_0 & \boldsymbol{x}_1 & \boldsymbol{x}_2 & \dots & \dots & \boldsymbol{x}_{p-1} \end{bmatrix},$$

with a given vector

$$\boldsymbol{x}_i^T = \begin{bmatrix} x_{0,i} & x_{1,i} & x_{2,i} & \dots & \dots x_{n-1,i} \end{bmatrix}.$$

With these definitions, we can now rewrite our 2×2 correaltion/covariance matrix in terms of a moe general design/feature matrix $\boldsymbol{X} \in \mathbb{R}^{n\times p}$. This leads to a $p\times p$ covariance matrix for the vectors $\boldsymbol{x}_i$ with $i = 0,1,\dots,p-1$

$$\boldsymbol{C}[\boldsymbol{x}] = \begin{bmatrix} \mathrm{var}[\boldsymbol{x}_0] & \mathrm{cov}[\boldsymbol{x}_0,\boldsymbol{x}_1] & \mathrm{cov}[\boldsymbol{x}_0,\boldsymbol{x}_2] & \dots & \dots & \mathrm{cov}[\boldsymbol{x}_0,\boldsymbol{x}_{p-1}] \\ \mathrm{cov}[\boldsymbol{x}_1,\boldsymbol{x}_0] & \mathrm{var}[\boldsymbol{x}_1] & \mathrm{cov}[\boldsymbol{x}_1,\boldsymbol{x}_2] & \dots & \dots & \mathrm{cov}[\boldsymbol{x}_1,\boldsymbol{x}_{p-1}] \\ \mathrm{cov}[\boldsymbol{x}_2,\boldsymbol{x}_0] & \mathrm{cov}[\boldsymbol{x}_2,\boldsymbol{x}_1] & \mathrm{var}[\boldsymbol{x}_2] & \dots & \dots & \mathrm{cov}[\boldsymbol{x}_2,\boldsymbol{x}_{p-1}] \\ \dots & \dots & \dots & \dots & \dots & \dots \\ \dots & \dots & \dots & \dots & \dots & \dots \\ \mathrm{cov}[\boldsymbol{x}_{p-1},\boldsymbol{x}_0] & \mathrm{cov}[\boldsymbol{x}_{p-1},\boldsymbol{x}_1] & \mathrm{cov}[\boldsymbol{x}_{p-1},\boldsymbol{x}_2] & \dots & \dots & \mathrm{var}[\boldsymbol{x}_{p-1}] \end{bmatrix},$$

and the correlation matrix

$$\boldsymbol{K}[\boldsymbol{x}] = \begin{bmatrix} 1 & \mathrm{corr}[\boldsymbol{x}_0,\boldsymbol{x}_1] & \mathrm{corr}[\boldsymbol{x}_0,\boldsymbol{x}_2] & \dots & \dots & \mathrm{corr}[\boldsymbol{x}_0,\boldsymbol{x}_{p-1}] \\ \mathrm{corr}[\boldsymbol{x}_1,\boldsymbol{x}_0] & 1 & \mathrm{corr}[\boldsymbol{x}_1,\boldsymbol{x}_2] & \dots & \dots & \mathrm{corr}[\boldsymbol{x}_1,\boldsymbol{x}_{p-1}] \\ \mathrm{corr}[\boldsymbol{x}_2,\boldsymbol{x}_0] & \mathrm{corr}[\boldsymbol{x}_2,\boldsymbol{x}_1] & 1 & \dots & \dots & \mathrm{corr}[\boldsymbol{x}_2,\boldsymbol{x}_{p-1}] \\ \dots & \dots & \dots & \dots & \dots & \dots \\ \dots & \dots & \dots & \dots & \dots & \dots \\ \mathrm{corr}[\boldsymbol{x}_{p-1},\boldsymbol{x}_0] & \mathrm{corr}[\boldsymbol{x}_{p-1},\boldsymbol{x}_1] & \mathrm{corr}[\boldsymbol{x}_{p-1},\boldsymbol{x}_2] & \dots & \dots & 1 \end{bmatrix},$$

Covariance Matrix Examples

The Numpy function **np.cov** calculates the covariance elements using the factor $1/(n-1)$ instead of $1/n$ since it assumes we do not have the exact mean values. The following simple function uses the **np.vstack** function which takes each vector of dimension $1 \times n$ and produces a $2 \times n$ matrix $\boldsymbol{W}$

$$\boldsymbol{W} = \begin{bmatrix} x_0 & y_0 \\ x_1 & y_1 \\ x_2 & y_2 \\ \dots & \dots \\ x_{n-2} & y_{n-2} \\ x_{n-1} & y_{n-1} \end{bmatrix},$$

which in turn is converted into into the 2×2 covariance matrix $\boldsymbol{C}$ via the Numpy function **np.cov()**. We note that we can also calculate the mean value of each set of samples $\boldsymbol{x}$ etc using the Numpy function **np.mean(x)**. We can also extract the eigenvalues of the covariance matrix through the **np.linalg.eig()** function.

Correlation Matrix

The previous example can be converted into the correlation matrix by simply scaling the matrix elements with the variances. We should also subtract the mean values for each column. This leads to the following code which sets up the correlations matrix for the previous example in a more brute force way. Here we scale the mean values for each column of the design matrix, calculate the relevant mean values and variances and then finally set up the 2×2 correlation matrix (since we have only two vectors).

We see that the matrix elements along the diagonal are one as they should be and that the matrix is symmetric. Furthermore, diagonalizing this matrix we easily see that it is a positive definite matrix.

The above procedure with **numpy** can be made more compact if we use **pandas**.

Correlation Matrix with Pandas

We whow here how we can set up the correlation matrix using **pandas**, as done in this simple code

We expand this model to the Franke function discussed above.

Correlation Matrix with Pandas and the Franke function

We note here that the covariance is zero for the first rows and columns since all matrix elements in the design matrix were set to one (we are fitting the function in terms of a polynomial of degree n).

This means that the variance for these elements will be zero and will cause problems when we set up the correlation matrix. We can simply drop these elements and construct a correlation matrix without these elements.

Rewriting the Covariance and/or Correlation Matrix

We can rewrite the covariance matrix in a more compact form in terms of the design/feature matrix $\boldsymbol{X}$ as

$$\boldsymbol{C}[\boldsymbol{x}] = \frac{1}{n}\boldsymbol{X}\boldsymbol{X}^T = \mathbb{E}[\boldsymbol{X}\boldsymbol{X}^T].$$

To see this let us simply look at a design matrix $\boldsymbol{X} \in \mathbb{R}^{2\times 2}$

$$\boldsymbol{X} = \begin{bmatrix} x_{00} & x_{01} \\ x_{10} & x_{11} \end{bmatrix} = \begin{bmatrix} \boldsymbol{x}_0 & \boldsymbol{x}_1 \end{bmatrix}.$$

If we then compute the expectation value

$$\mathbb{E}[\boldsymbol{X}\boldsymbol{X}^T] = \frac{1}{n}\boldsymbol{X}\boldsymbol{X}^T = \begin{bmatrix} x_{00}^2 + x_{01}^2 & x_{00}x_{10} + x_{01}x_{11} \\ x_{10}x_{00} + x_{11}x_{01} & x_{10}^2 + x_{11}^2 \end{bmatrix},$$

which is just

$$\boldsymbol{C}[\boldsymbol{x}_0, \boldsymbol{x}_1] = \boldsymbol{C}[\boldsymbol{x}] = \begin{bmatrix} \mathrm{var}[\boldsymbol{x}_0] & \mathrm{cov}[\boldsymbol{x}_0, \boldsymbol{x}_1] \\ \mathrm{cov}[\boldsymbol{x}_1, \boldsymbol{x}_0] & \mathrm{var}[\boldsymbol{x}_1] \end{bmatrix},$$

where we wrote

$$\boldsymbol{C}[\boldsymbol{x}_0, \boldsymbol{x}_1] = \boldsymbol{C}[\boldsymbol{x}]$$

to indicate that this the covariance of the vectors $\boldsymbol{x}$ of the design/feature matrix $\boldsymbol{X}$.

It is easy to generalize this to a matrix $\boldsymbol{X} \in \mathbb{R}^{n\times p}$.

Towards the PCA theorem

We have that the covariance matrix (the correlation matrix involves a simple rescaling) is given as

$$\boldsymbol{C}[\boldsymbol{x}] = \frac{1}{n}\boldsymbol{X}\boldsymbol{X}^T = \mathbb{E}[\boldsymbol{X}\boldsymbol{X}^T].$$

Let us now assume that we can perform a series of orthogonal transformations where we employ some orthogonal matrices $\boldsymbol{S}$. These matrices are defined as $\boldsymbol{S} \in$

$\mathbb{R}^{p\times p}$ and obey the orthogonality requirements $\boldsymbol{S}\boldsymbol{S}^T = \boldsymbol{S}^T\boldsymbol{S} = \boldsymbol{I}$. The matrix can be written out in terms of the column vectors $\boldsymbol{s}_i$ as $\boldsymbol{S} = [\boldsymbol{s}_0, \boldsymbol{s}_1, \dots, \boldsymbol{s}_{p-1}]$ and $\boldsymbol{s}_i \in \mathbb{R}^p$.

Assume also that there is a transformation $\boldsymbol{S}\boldsymbol{C}[\boldsymbol{x}]\boldsymbol{S}^T = \boldsymbol{C}[\boldsymbol{y}]$ such that the new matrix $\boldsymbol{C}[\boldsymbol{y}]$ is diagonal with elements $[\lambda_0, \lambda_1, \lambda_2, \dots, \lambda_{p-1}]$.

That is we have

$$\boldsymbol{C}[\boldsymbol{y}] = \mathbb{E}[\boldsymbol{S}\boldsymbol{X}\boldsymbol{X}^T\boldsymbol{S}^T] = \boldsymbol{S}\boldsymbol{C}[\boldsymbol{x}]\boldsymbol{S}^T,$$

since the matrix $\boldsymbol{S}$ is not a data dependent matrix. Multiplying with $\boldsymbol{S}^T$ from the left we have

$$\boldsymbol{S}^T\boldsymbol{C}[\boldsymbol{y}] = \boldsymbol{C}[\boldsymbol{x}]\boldsymbol{S}^T,$$

and since $\boldsymbol{C}[\boldsymbol{y}]$ is diagonal we have for a given eigenvalue i of the covariance matrix that

$$\boldsymbol{S}_i^T\lambda_i = \boldsymbol{C}[\boldsymbol{x}]\boldsymbol{S}_i^T.$$

In the derivation of the PCA theorem we will assume that the eigenvalues are ordered in descending order, that is $\lambda_0 > \lambda_1 > \cdots > \lambda_{p-1}$.

The eigenvalues tell us then how much we need to stretch the corresponding eigenvectors. Dimensions with large eigenvalues have thus large variations (large variance) and define therefore useful dimensions. The data points are more spread out in the direction of these eigenvectors. Smaller eigenvalues mean on the other hand that the corresponding eigenvectors are shrunk accordingly and the data points are tightly bunched together and there is not much variation in these specific directions. Hopefully then we could leave it out dimensions where the eigenvalues are very small. If p is very large, we could then aim at reducing p to $l << p$ and handle only l features/predictors.

The Algorithm before the Theorem

Here's how we would proceed in setting up the algorithm for the PCA, see also discussion below here.

- Set up the datapoints for the design/feature matrix $\boldsymbol{X}$ with $\boldsymbol{X} \in \mathbb{R}^{n\times p}$, with the predictors/features p referring to the column numbers and the entries n being the row elements.

$$\boldsymbol{X} = \begin{bmatrix} x_{0,0} & x_{0,1} & x_{0,2} & \dots & \dots x_{0,p-1} \\ x_{1,0} & x_{1,1} & x_{1,2} & \dots & \dots x_{1,p-1} \\ x_{2,0} & x_{2,1} & x_{2,2} & \dots & \dots x_{2,p-1} \\ \dots & \dots & \dots & \dots\dots & \dots \\ x_{n-2,0} & x_{n-2,1} & x_{n-2,2} & \dots & \dots x_{n-2,p-1} \\ x_{n-1,0} & x_{n-1,1} & x_{n-1,2} & \dots & \dots x_{n-1,p-1} \end{bmatrix},$$

- Center the data by subtracting the mean value for each column. This leads to a new matrix $\boldsymbol{X} \to \overline{\boldsymbol{X}}$.

- Compute then the covariance/correlation matrix $\mathbb{E}[\boldsymbol{X}\boldsymbol{X}^T]$.
- Find the eigenpairs of $\boldsymbol{C}$ with eigenvalues $[\lambda_0, \lambda_1, \ldots, \lambda_{p-1}]$ and eigenvectors $[\boldsymbol{s}_0, \boldsymbol{s}_1, \ldots, \boldsymbol{s}_{p-1}]$.
- Order the eigenvalue (and the eigenvectors accordingly) in order of decreasing eigenvalues.
- Keep only those l eigenvalues larger than a selected threshold value, discarding thus $p - l$ features since we expect small variations in the data here.

Writing our own PCA code

We will use a simple example first with two-dimensional data drawn from a multivariate normal distribution with the following mean and covariance matrix:

$$\mu = (-1, 2) \qquad \Sigma = \begin{bmatrix} 4 & 2 \\ 2 & 2 \end{bmatrix}$$

Note that the mean refers to each column of data. We will generate $n = 1000$ points $X = \{x_1, \ldots, x_N\}$ from this distribution, and store them in the 1000×2 matrix $\boldsymbol{X}$.

The following Python code aids in setting up the data and writing out the design matrix. Note that the function **multivariate** returns also the covariance discussed above and that it is defined by dividing by $n - 1$ instead of n.

Now we are going to implement the PCA algorithm. We will break it down into various substeps.

Compute the sample mean and center the data. The first step of PCA is to compute the sample mean of the data and use it to center the data. Recall that the sample mean is

$$\mu_n = \frac{1}{n} \sum_{i=1}^{n} x_i$$

and the mean-centered data $\bar{X} = \{\bar{x}_1, \ldots, \bar{x}_n\}$ takes the form

$$\bar{x}_i = x_i - \mu_n.$$

When you are done with these steps, print out μ_n to verify it is close to μ and plot your mean centered data to verify it is centered at the origin! Compare your code with the functionality from **Scikit-Learn** discussed above. The following code elements perform these operations using **pandas** or using our own functionality for doing so. The latter, using **numpy** is rather simple through the **mean()** function.

Alternatively, we could use the functions we discussed earlier for scaling the data set. That is, we could have used the **StandardScaler** function in **Scikit-Learn**, a function which 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). You would then not get the same results, since we divide by the variance. The diagonal covariance matrix elements will then be one, while the non-diagonal ones need to be divided by $2\sqrt{2}$ for our specific case.

Compute the sample covariance. Now we are going to use the mean centered data to compute the sample covariance of the data.

$$\Sigma_n = \frac{1}{n-1}\sum_{i=1}^{n} \bar{x}_i^T \bar{x}_i = \frac{1}{n-1}\sum_{i=1}^{n}(x_i - \mu_n)^T(x_i - \mu_n)$$

where the data points $x_i \in \mathbb{R}^p$ (here in this example $p = 2$) are column vectors and x^T is the transpose of x. We can write our own code or simply use either the functionaly of **numpy** or that of **pandas**, as follows Note that the way we define the covariance matrix here has a factor $n - 1$ instead of n. Our own code here is not very elegant and asks for improvements.

Depending on the number of points n, we will get results that are close to the covariance values defined above. The plot shows how the data are clustered around a line with slope close to one. Is this expected?

Diagonalize the sample covariance matrix to obtain the principal components. Now we are ready to solve for the principal components! To do so we diagonalize the sample covariance matrix Σ. We can use the function **np.linalg.eig** to do so. It will return the eigenvalues and eigenvectors of Σ. Once we have these we can perform the following tasks:

- We compute the percentage of the total variance captured by the first principal component
- We plot the mean centered data and lines along the first and second principal components
- Then we project the mean centered data onto the first and second principal components, and plot the projected data.
- Finally, we approximate the data as

$$x_i \approx \tilde{x}_i = \mu_n + \langle x_i, v_0 \rangle v_0$$

where v_0 is the first principal component.

Collecting all these steps we can write our own PCA function and compare this with the functionality included in **Scikit-Learn**.

The code here outlines some of the elements we could include in the analysis. Feel free to extend upon this in order to address the above questions.

This code does not contain all the above elements, but it shows how we can use **Scikit-Learn** to extract the eigenvector which corresponds to the largest eigenvalue. Try to address the questions we pose before the above code. Try also to change the values of the covariance matrix by making one of the diagonal elements much larger than the other. What do you observe then?

Classical PCA Theorem

We assume now that we have a design matrix $\boldsymbol{X}$ which has been centered as discussed above. For the sake of simplicity we skip the overline symbol. The matrix is defined in terms of the various column vectors $[\boldsymbol{x}_0, \boldsymbol{x}_1, \dots, \boldsymbol{x}_{p-1}]$ each with dimension $\boldsymbol{x} \in \mathbb{R}^n$.

We assume also that we have an orthogonal transformation $\boldsymbol{W} \in \mathbb{R}^{p \times p}$. We define the reconstruction error (which is similar to the mean squared error we have seen before) as

$$J(\boldsymbol{W}, \boldsymbol{Z}) = \frac{1}{n} \sum_i (\boldsymbol{x}_i - \overline{\boldsymbol{x}}_i)^2,$$

with $\overline{\boldsymbol{x}}_i = \boldsymbol{W}\boldsymbol{z}_i$, where $\boldsymbol{z}_i$ is a row vector with dimension $\mathbb{R}^n$ of the matrix $\boldsymbol{Z} \in \mathbb{R}^{p \times n}$. When doing PCA we want to reduce this dimensionality.

The PCA theorem states that minimizing the above reconstruction error corresponds to setting $\boldsymbol{W} = \boldsymbol{S}$, the orthogonal matrix which diagonalizes the empirical covariance(correlation) matrix. The optimal low-dimensional encoding of the data is then given by a set of vectors $\boldsymbol{z}_i$ with at most l vectors, with $l << p$, defined by the orthogonal projection of the data onto the columns spanned by the eigenvectors of the covariance(correlations matrix).

The proof which follows will be updated by mid January 2020.

Proof of the PCA Theorem

To show the PCA theorem let us start with the assumption that there is one vector $\boldsymbol{w}_0$ which corresponds to a solution which minimized the reconstruction error J. This is an orthogonal vector. It means that we now approximate the reconstruction error in terms of $\boldsymbol{w}_0$ and $\boldsymbol{z}_0$ as

$$J(\boldsymbol{w}_0, \boldsymbol{z}_0) = \frac{1}{n} \sum_i (\boldsymbol{x}_i - z_{i0}\boldsymbol{w}_0)^2 = \frac{1}{n} \sum_i (\boldsymbol{x}_i^T\boldsymbol{x}_i - 2z_{i0}\boldsymbol{w}_0^T\boldsymbol{x}_i + z_{i0}^2\boldsymbol{w}_0^T\boldsymbol{w}_0),$$

which we can rewrite due to the orthogonality of $\boldsymbol{w}_i$ as

$$J(\boldsymbol{w}_0, \boldsymbol{z}_0) = \frac{1}{n} \sum_i (\boldsymbol{x}_i^T\boldsymbol{x}_i - 2z_{i0}\boldsymbol{w}_0^T\boldsymbol{x}_i + z_{i0}^2).$$

Minimizing J with respect to the unknown parameters z_{0i} we obtain that

$$z_{i0} = \boldsymbol{w}_0^T\boldsymbol{x}_i,$$

where the vectors on the rhs are known.

PCA Proof continued

We have now found the unknown parameters z_{i0}. These correspond to the projected coordinates and we can write

$$J(\boldsymbol{w}_0) = \frac{1}{p} \sum_i (\boldsymbol{x}_i^T\boldsymbol{x}_i - z_{i0}^2) = \text{const} - \frac{1}{n} \sum_i z_{i0}^2.$$

We can show that the variance of the projected coordinates defined by $\boldsymbol{w}_0^T\boldsymbol{x}_i$ are given by

$$\mathrm{var}[\boldsymbol{w}_0^T\boldsymbol{x}_i] = \frac{1}{n}\sum_i z_{i0}^2,$$

since the expectation value of

$$\mathbb{E}[\boldsymbol{w}_0^T\boldsymbol{x}_i] = \mathbb{E}[z_{i0}] = \boldsymbol{w}_0^T\mathbb{E}[\boldsymbol{x}_i] = 0,$$

where we have used the fact that our data are centered.

Recalling our definition of the covariance as

$$\boldsymbol{C}[\boldsymbol{x}] = \frac{1}{n}\boldsymbol{X}\boldsymbol{X}^T = \mathbb{E}[\boldsymbol{X}\boldsymbol{X}^T],$$

we have thus that

$$\mathrm{var}[\boldsymbol{w}_0^T\boldsymbol{x}_i] = \frac{1}{n}\sum_i z_{i0}^2 = \boldsymbol{w}_0^T\boldsymbol{C}[\boldsymbol{x}]\boldsymbol{w}_0.$$

We are almost there, we have obtained a relation between minimizing the reconstruction error and the variance and the covariance matrix. Minimizing the error is equivalent to maximizing the variance of the projected data.

The final step

We could trivially maximize the variance of the projection (and thereby minimize the error in the reconstruction function) by letting the norm-2 of $\boldsymbol{w}_0$ go to infinity. However, this norm since we want the matrix $\boldsymbol{W}$ to be an orthogonal matrix, is constrained by $||\boldsymbol{w}_0||_2^2 = 1$. Imposing this condition via a Lagrange multiplier we can then in turn maximize

$$J(\boldsymbol{w}_0) = \boldsymbol{w}_0^T\boldsymbol{C}[\boldsymbol{x}]\boldsymbol{w}_0 + \lambda_0(1 - \boldsymbol{w}_0^T\boldsymbol{w}_0).$$

Taking the derivative with respect to $\boldsymbol{w}_0$ we obtain

$$\frac{\partial J(\boldsymbol{w}_0)}{\partial \boldsymbol{w}_0} = 2\boldsymbol{C}[\boldsymbol{x}]\boldsymbol{w}_0 - 2\lambda_0\boldsymbol{w}_0 = 0,$$

meaning that

$$\boldsymbol{C}[\boldsymbol{x}]\boldsymbol{w}_0 = \lambda_0\boldsymbol{w}_0.$$

The direction that maximizes the variance (or minimizes the construction error) is an eigenvector of the covariance matrix! If we left multiply with $\boldsymbol{w}_0^T$ we have the variance of the projected data is

$$\boldsymbol{w}_0^T\boldsymbol{C}[\boldsymbol{x}]\boldsymbol{w}_0 = \lambda_0.$$

If we want to maximize the variance (minimize the construction error) we simply pick the eigenvector of the covariance matrix with the largest eigenvalue. This establishes the link between the minimization of the reconstruction function

J in terms of an orthogonal matrix and the maximization of the variance and thereby the covariance of our observations encoded in the design/feature matrix $\boldsymbol{X}$.

The proof for the other eigenvectors $\boldsymbol{w}_1, \boldsymbol{w}_2, \ldots$ can be established by applying the above arguments and using the fact that our basis of eigenvectors is orthogonal, see Murphy chapter 12.2. The discussion in chapter 12.2 of Murphy's text has also a nice link with the Singular Value Decomposition theorem. For categorical data, see chapter 12.4 and discussion therein.

Additional part of the proof for the other eigenvectors will be added by mid January 2020.

Geometric Interpretation and link with Singular Value Decomposition

This material will be added by mid January 2020.

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. First we center the data using either **pandas** or our own code

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.

Back to the Cancer Data

We can now repeat the above but applied to real data, in this case our breast cancer data. Here we compute performance scores on the training data using logistic regression.

We see that our training data after the PCA decomposition has a performance similar to the non-scaled data.

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.