Machine Learning in Python
Week 2: Dimension Reduction

Dr. Sara Wade
sara.wade@ed.ac.uk

Contents

Figure 1: Examples from the 'Extended Yale Face Database', adapted from *Data-Driven Science and Engineering* by Brunton & Kutz.

1 Dimension Reduction

In unsupervised learning, we only have access to a set of features $\mathbf{x}_n \in \mathbb{R}^D$ measured on $n = 1, \ldots, N$ observations. The **feature matrix** (also called the **design** or **input** matrix) $\mathbf{X}$ is an $N \times D$ real-valued matrix collecting the observed data:

$$\mathbf{X} := \begin{bmatrix} \mathbf{x}_1^T \\ \mathbf{x}_2^T \\ \vdots \\ \mathbf{x}_N^T \end{bmatrix} = \begin{bmatrix} x_{1,1} & x_{1,2} & \ldots & x_{1,D} \\ x_{2,1} & x_{2,2} & \ldots & x_{2,D} \\ & & \vdots & \\ x_{N,1} & x_{N,2} & \ldots & x_{N,D} \end{bmatrix}.$$

Many real-world problems yield design matrices $\mathbf{X}$ that consist of a very large number of features $D \gg 1$. Is there any informative way to visualize and/or summarize such data? For example, one could examine two-dimensional scatterplots of each pair features, but this would result in $D(D-1)/2$ such plots and even for $D = 10$, there are 45 plots! Moreover, most of them are likely to be uninformative since they each contain just a small fraction of the total information present in the data. Clearly, a better method is required to visualize the data.

In practice, many of the features in high-dimensional problems are highly correlated and the dataset consists of many recurring "patterns". For example, this could be high resolution images, gene expression datasets, videos, etc. We might argue that the data live on a much lower-dimensional manifold embedded in the larger space $\mathbb{R}^D$. For example, consider the greyscale faces in Figure 1. The snapshots all share common features, orientations, etc. We can imagine that there might be a better representation than the high-dimensional raw pixel intensities that we start with.

In this direction, **dimension reduction** is a canonical form of unsupervised learning, which aims to learn a mapping from the high-dimensional space of observed features, $\mathbf{x} \in \mathbb{R}^D$, to a low-dimensional latent space, $\mathbf{z} \in \mathbb{R}^L$, that captures most of the information in the data. This low-dimensional representation is quite useful for data visualization. Additionally, it can be used in a pre-processing step before applying supervised learning techniques, in order to obtain a small set of useful features and help combat the curse of dimensionality.

2 Principal Component Analysis

In these notes, we focus on **principal component analysis** (PCA), the simplest and most widely-used form of dimensionality reduction. The basic idea in PCA is to find a linear and orthogonal projection of the high-dimensional feature space into a low-dimensional subspace, such that the low-dimensional representation provides a *good approximation* to the original data by explaining most of the variability present in the data. More formally, if we linearly project or **encode** $\mathbf{x}$ to get $\mathbf{z} = \mathbf{W}^T\mathbf{x}$, and then unproject or **decode** $\mathbf{z}$ to get $\widehat{\mathbf{x}} = \mathbf{W}\mathbf{z}$, then we want $\widehat{\mathbf{x}}$ to be close to $\mathbf{x}$ in ℓ_2 distance (or squared Euclidean distance). In this case, we assume that $\mathbf{W}$ is a $D \times L$ orthonormal matrix, i.e. its columns satisfy $\mathbf{w}_l^T\mathbf{w}_l = 1$ and $\mathbf{w}_l^T\mathbf{w}_k = 0$ for $l \neq k$, and of course that $L < D$ (as we are projecting to a lower-dimensional space). Thus, we can define the following **reconstruction error**:

$$\mathcal{L}(\mathbf{W}) = \frac{1}{N}\sum_{n=1}^{N}\|\mathbf{x}_n - \widehat{\mathbf{x}}_n\|_2^2,$$

where $\widehat{\mathbf{x}}_n$ is obtained by first encoding $\mathbf{x}_n$ to get $\mathbf{z}_n$ and then decoding $\mathbf{z}_n$ to get $\widehat{\mathbf{x}}_n$.

In PCA, the optimal solution that minimizes the reconstruction error subject to the constraint $\mathbf{W}$ is orthonormal is obtained by setting $\mathbf{W}$ to the $D \times L$ matrix containing the first L eigenvectors with the largest eigenvalues of the empirical covariance matrix:

$$\mathbf{\Sigma} = \frac{1}{N}\sum_{n=1}^{N}(\mathbf{x}_n - \overline{\mathbf{x}})(\mathbf{x}_n - \overline{\mathbf{x}})^T, \tag{1}$$

where $\overline{\mathbf{x}}$ denotes the empirical mean.

2.1 Derivation of PCA

We assume that the features of $\mathbf{X}$ have been **centred** to have zero mean (that is, the column means of $\mathbf{X}$ are zero) and also focus on the case when $N > D$ (but this is not essential). We would like to approximate each $\mathbf{x}_n \in \mathbb{R}^D$ by a low-dimensional representation $\mathbf{z}_n \in \mathbb{R}^L$. Our approximation (decoder) assumes that each $\mathbf{x}_n$ can be *explained* in terms of a weighted combination of basis vectors, $\mathbf{w}_1, \ldots, \mathbf{w}_L$, where $\mathbf{w}_l \in \mathbb{R}^D$, with weights given by $\mathbf{z}_n \in \mathbb{R}^L$; that is,

$$\mathbf{x}_n \approx \widehat{\mathbf{x}}_n = \sum_{l=1}^{L} z_{n,l}\mathbf{w}_l.$$

Let $\mathbf{Z}$ be the $N \times L$ matrix with rows $\mathbf{z}_n^T$; the rows of $\mathbf{Z}$ correspond to the low-dimensional representation of each data point, and the columns are referred to as the **scores** of the **principal components** or more generally, the **latent factors**. Each column $\mathbf{w}_l$ of the matrix $\mathbf{W}$ is referred to as the **loadings** of the lth principal component. We aim to minimize the (average) reconstruction error:

$$\mathcal{L}(\mathbf{W}, \mathbf{Z}) = \frac{1}{N}\sum_{n=1}^{N}\|\mathbf{x}_n - \mathbf{W}\mathbf{z}_n\|_2^2,$$

subject to the constraint that $\mathbf{W}$ is an orthonormal matrix.

Before proceeding to solve this, we note that since the data have been centred (i.e. $\overline{\mathbf{x}} = \mathbf{0}$ in (1)), we can write the empirical covariance matrix as

$$\mathbf{\Sigma} = \frac{1}{N}\sum_{n=1}^{N}\mathbf{x}_n\,\mathbf{x}_n^T = \frac{1}{N}\mathbf{X}^T\mathbf{X}.$$

Step 1: What is the optimal encoding? First let's start with given $\mathbf{W}$. To find the optimal encoding $\mathbf{Z}$, we want to minimize (with respect to $\mathbf{Z}$) the reconstruction

error:

$$\begin{aligned}
\mathcal{L}(\mathbf{W}, \mathbf{Z}) &= \frac{1}{N} \sum_{n=1}^{N} \|\mathbf{x}_n - \mathbf{W}\mathbf{z}_n\|_2^2 \\
&= \frac{1}{N} \sum_{n=1}^{N} (\mathbf{x}_n - \mathbf{W}\mathbf{z}_n)^T (\mathbf{x}_n - \mathbf{W}\mathbf{z}_n) \\
&= \frac{1}{N} \sum_{n=1}^{N} \mathbf{x}_n^T \mathbf{x}_n - 2\mathbf{x}_n^T \mathbf{W}\mathbf{z}_n + \mathbf{z}_n^T \underbrace{\mathbf{W}^T \mathbf{W}}_{=\mathbf{I}} \mathbf{z}_n \\
&= \frac{1}{N} \sum_{n=1}^{N} \mathbf{x}_n^T \mathbf{x}_n - 2\mathbf{x}_n^T \mathbf{W}\mathbf{z}_n + \mathbf{z}_n^T \mathbf{z}_n
\end{aligned}$$

Taking the derivative with respect to $\mathbf{z}_n$ and equating to zero gives:

$$\begin{aligned}
&\frac{\partial \mathcal{L}(\mathbf{W}, \mathbf{Z})}{\partial \mathbf{z}_n} = \frac{1}{N} \left(-2\mathbf{W}^T \mathbf{x}_n + 2\mathbf{z}_n\right) = 0, \\
&\Rightarrow \mathbf{z}_n = \mathbf{W}^T \mathbf{x}_n.
\end{aligned}$$

Thus, each component of the latent low-dimensional vector $\mathbf{z}_n$ is a normalized linear combination of the observed features with weights given by the loadings of the component:

$$z_{n,l} = w_{1,l} x_{n,1} + w_{2,l} x_{n,2} + \ldots + w_{D,l} x_{n,D}.$$

Step 2: What is the optimal loadings matrix W? Plugging in the optimal encoding $\mathbf{Z} = \mathbf{X}\mathbf{W}$, the loss for $\mathbf{W}$ is:

$$\begin{aligned}
\mathcal{L}(\mathbf{W}) &= \frac{1}{N} \sum_{n=1}^{N} \|\mathbf{x}_n - \mathbf{W}\mathbf{W}^T \mathbf{x}_n\|_2^2 \\
&= \frac{1}{N} \sum_{n=1}^{N} \mathbf{x}_n^T \mathbf{x}_n - \mathbf{x}_n^T \mathbf{W}\mathbf{W}^T \mathbf{x}_n \\
&= \text{const} - \frac{1}{N} \sum_{n=1}^{N} \sum_{l=1}^{L} \mathbf{x}_n^T \mathbf{w}_l \mathbf{w}_l^T \mathbf{x}_n \\
&= \text{const} - \sum_{l=1}^{L} \mathbf{w}_l^T \left(\frac{1}{N} \sum_{n=1}^{N} \mathbf{x}_n \mathbf{x}_n^T \right) \mathbf{w}_l \\
&= \text{const} - \sum_{l=1}^{L} \mathbf{w}_l^T \mathbf{\Sigma} \mathbf{w}_l.
\end{aligned}$$

Thus, the objective function factorizes across the L components; however, we have the constraint that the $\mathbf{w}_l$ must be orthogonal. So, we will proceed sequentially, by finding the loadings of the first component $\mathbf{w}_1$, and then we will find $\mathbf{w}_2$ given $\mathbf{w}_1$, and so on.

Focusing first on $\mathbf{w}_1$, we have that:

$$\mathcal{L}(\mathbf{w}_1) = \text{const} - \mathbf{w}_1^T \mathbf{\Sigma} \mathbf{w}_1. \tag{2}$$

Using Lagrange multipliers to impose the constraint that $\mathbf{w}_1^T \mathbf{w}_1 = 1$, we want to optimize:

$$\tilde{\mathcal{L}}(\mathbf{w}_1) = -\mathbf{w}_1^T \mathbf{\Sigma} \mathbf{w}_1 + \lambda_1 (\mathbf{w}_1^T \mathbf{w}_1 - 1).$$

Taking the derivative and equating to zero, we have:

$$\begin{aligned} &\frac{\partial \tilde{\mathcal{L}}(\mathbf{w}_1)}{\partial \mathbf{w}_1} = -2\mathbf{\Sigma}\mathbf{w}_1 + 2\lambda_1 \mathbf{w}_1 = 0, \\ &\Rightarrow \mathbf{\Sigma}\mathbf{w}_1 = \lambda_1 \mathbf{w}_1. \end{aligned}$$

Hence the optimal direction onto which we should project the data is an eigenvector of the covariance matrix. Left multiplying by $\mathbf{w}_1^T$ (and using that $\mathbf{w}_1^T \mathbf{w}_1 = 1$), we find that

$$\mathbf{w}_1^T \mathbf{\Sigma} \mathbf{w}_1 = \lambda_1.$$

We want to maximize this quantity (minimize the loss), so we pick the eigenvector corresponding to the largest eigenvalue. Notice that minimizing the reconstruction error is equivalent to maximizing the empirical variance of the projected data (see exercises). This is why we can also interpret PCA as finding the directions of maximal variance.

Next, let's find another direction $\mathbf{w}_2$ to further minimize the reconstruction error subject to the constraints $\mathbf{w}_1^T \mathbf{w}_2 = 0$ and $\mathbf{w}_2^T \mathbf{w}_2 = 1$ (equivalently, we want to find the second principal component which is a normalized linear combination of the observed features that has maximal variance and is uncorrelated with the first principal component). The error is:

$$\mathcal{L}(\mathbf{w}_2) = \text{const} - \mathbf{w}_2^T \mathbf{\Sigma} \mathbf{w}_2.$$

Using Lagrange multipliers to impose the constraints, we want to optimize:

$$\tilde{\mathcal{L}}(\mathbf{w}_2) = -\mathbf{w}_2^T \mathbf{\Sigma} \mathbf{w}_2 + \lambda_2 (\mathbf{w}_2^T \mathbf{w}_2 - 1) + \lambda_{1,2} (\mathbf{w}_2^T \mathbf{w}_1 - 0). \tag{3}$$

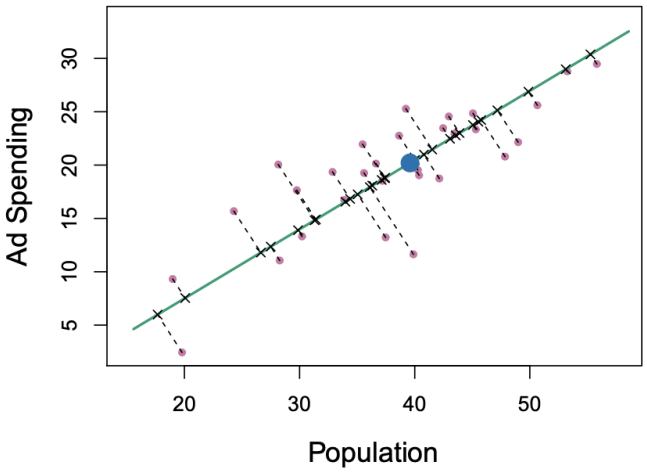


Figure 2: The population size and ad spending for different cities are shown as purple circles with the empirical mean indicated with a blue circle. The green solid line indicates the first principal component. The black crosses represent the projection of the cities onto the first principal component line (reconstructions) and the distances from each observation to the projections are represented using dashed black line segments. From *An Introduction to Statistical Learning* (2023).

The solution (see exercises) is given by the eigenvector with the second largest eigenvalue:

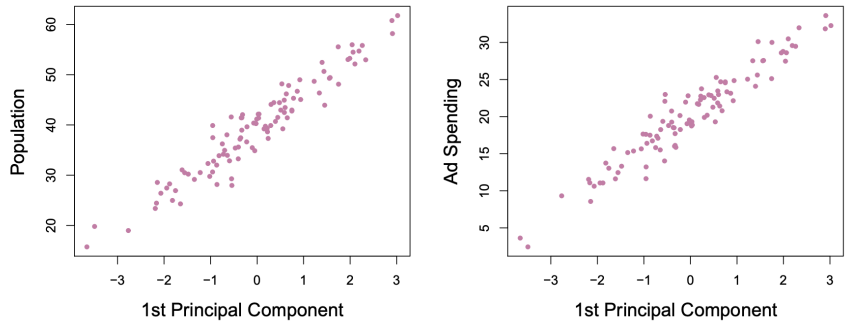
$$\Rightarrow \mathbf{\Sigma}\mathbf{w}_2 = \lambda_2 \mathbf{w}_2.$$

The proof continues in this way to show that $\mathbf{W}$ is comprised of the first L eigenvectors with the largest eigenvalues.

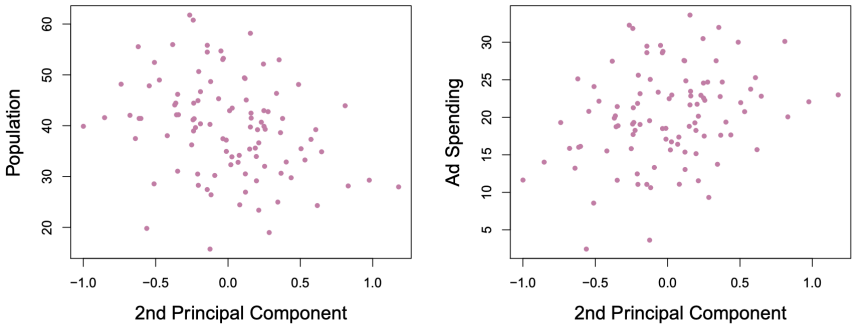
2.2 Geometric Interpretation

Figure 2 shows the population size and ad spending for different cities. The green solid line represents the direction of the first principal component. The black crosses represent the projection of the cities onto the first principal component line (the reconstructions $\widehat{\mathbf{x}}_n$ of the data) and the distances from each observation to the projections are represented using dashed black line segments. We can interpret PCA as finding the line which 1) makes the projected data as close as possible to the observed data (minimizes the sum of the squared length of the black dashed line segments) or 2) maximizes the variance of the first principal component scores (variance of the black crosses along the green line).

The first principal component provides a summary of both the population and ad



(a) First principal component



(b) Second principal component

Figure 3: Plots of the first (top row) and second (bottom row) principal component scores $z_{n,1}$ and $z_{n,2}$ against the observed features, population and ad spending. From *An Introduction to Statistical Learning* (2023).

spending for each city. In this case, the loadings are $w_{1,1} = 0.839$ and $w_{2,1} = 0.544$. So a city, with a negative score $z_{n,1} < 0$ indicates below average population size and ad spending, while a positive score suggests the opposite. Figure 3 indicates that both features have a strong relationship with the first principal component, thus it appears to capture most of the information contained in these two variables. The second principal component must be perpendicular, or orthogonal, to the first principal component. The second principal component scores are much closer to zero and there is little relationship with the features (see Figure 3), suggesting that in this case, one only needs the first principal component to accurately summarize population and ad spending.

The vectors $\mathbf{w}_1, \ldots, \mathbf{w}_L$ can be thought of as being the *dominant patterns* in the original data. In high-dimensions, in can be hard to interpret the latent dimensions. However, for images for example, we can plot each basis vector $\mathbf{w}_l$ as an image. For the Olivetti face database in Figure 4, we show some random images along with the mean

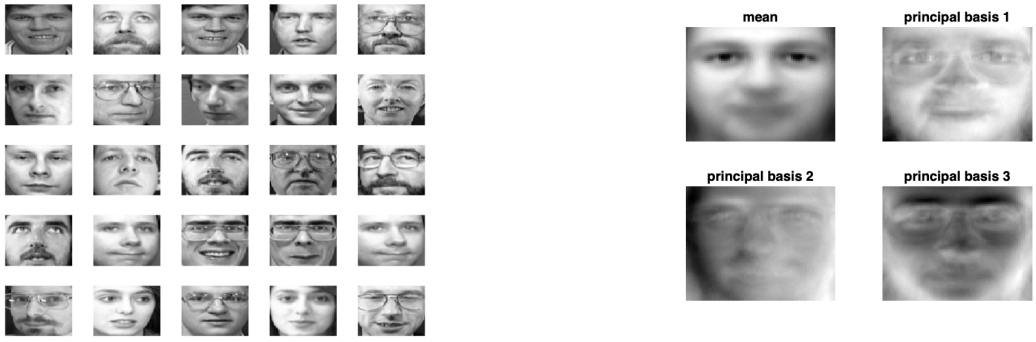


Figure 4: Left: randomly chosen 64×64 images from the Olivetti face database. Right: the *mean* face and the first three *eigenfaces*. From *An Introduction to Probabilistic Machine Learning* (2022).

and first three vectors $\mathbf{w}_l$ (referred to as *eigenfaces*). We can see that the main sources of variation in the data are related to the overall lighting, and then differences in the eyebrow region of the face.

2.3 Computational Issues

Covariance vs. correlation matrix. We have been working with the eigendecomposition of the covariance matrix. However, the results obtained from PCA will depend on the scales of the different features and will be dominated by the features with the highest variance (see exercises). Thus, in practice, it is better to first standardize the data, using the correlation matrix instead of the covariance matrix. Although, if the features are all measured on the same units (e.g. imaging data in Figure 4, gene expression data, etc.), then it is typically preferable to use the covariance matrix, as we want more highly variable pixels or genes to have more dominance.

High-dimensional data. We have focused on finding the eigenvectors of $\mathbf{\Sigma} = 1/N\, \mathbf{X}^T\mathbf{X}$, but if $D > N$, it is faster to work with the $N \times N$ **Gram matrix**, defined as:

$$\text{Gram matrix} = \mathbf{X}\mathbf{X}^T = \begin{bmatrix} \mathbf{x}_1^T\mathbf{x}_1 & \dots & \mathbf{x}_1^T\mathbf{x}_N \\ \vdots & & \vdots \\ \mathbf{x}_N^T\mathbf{x}_1 & \dots & \mathbf{x}_N^T\mathbf{x}_N \end{bmatrix}. \tag{4}$$

Suppose that $\mathbf{U}$ is an orthonormal matrix containing the eigenvectors of $\mathbf{X}\mathbf{X}^T$ with eigenvalues in the diagonal matrix $\mathbf{\Lambda}$. By definition $\mathbf{X}\mathbf{X}^T\mathbf{U} = \mathbf{U}\mathbf{\Lambda}$. Pre-multiplying by

$\mathbf{X}^T$ gives:

$$(\mathbf{X}^T\mathbf{X})\mathbf{X}^T\mathbf{U} = \mathbf{X}^T\mathbf{U}\mathbf{\Lambda}.$$

Thus, $\mathbf{X}^T\mathbf{U}$ are the eigenvectors of $\mathbf{X}^T\mathbf{X}$, with eigenvalues given in $\mathbf{\Lambda}$. However, they are not normalized and we can obtain the normalized eigenvectors $\mathbf{V} = \mathbf{X}^T\mathbf{U}\mathbf{\Lambda}^{-\frac{1}{2}}$ (see exercises). This provides an alternative way to compute the PCA basis when $D > N$. It is also used in **kernel PCA** for nonlinear dimension reduction (see Section 3).

Computing PCA using SVD. The **singular value decomposition** (SVD) of the matrix $\mathbf{X}$ decomposes it as:

$$\mathbf{X} = \mathbf{U}\mathbf{D}\mathbf{V}^T. \tag{5}$$

This decomposition exists for any matrix, and for the $N \times D$ design matrix $\mathbf{X}$, the sizes of the matrices in its SVD decomposition are:

$$\begin{aligned}
\mathbf{U}: &\quad N \times N \quad \text{(number of data points} \times \text{number of data points)},\\
\mathbf{D}: &\quad N \times D \quad \text{(number of data points} \times \text{number of features)},\\
\mathbf{V}: &\quad D \times D \quad \text{(number of features} \times \text{number of features)},
\end{aligned}$$

where the matrices $\mathbf{U}$ and $\mathbf{V}$ are orthonormal, i.e. $\mathbf{U}^T\mathbf{U} = \mathbf{I}_N$ and $\mathbf{V}^T\mathbf{V} = \mathbf{I}_D$. For simplicity, we will focus on the case when $N > D$. The matrix $\mathbf{D}$ has the following structure:

$$\mathbf{D} = \begin{bmatrix} \sigma_1 & & & \\ & \sigma_2 & & \\ & & \ddots & \\ & & & \sigma_D \\ & \cdots & [\mathbf{0}] & \cdots \end{bmatrix},$$

that is, the first D rows are constructed from a diagonal matrix with entries $\sigma_1 \geq \sigma_2 \geq \cdots \geq \sigma_D \geq 0$, and the remaining entries are zero. The σ_d are the **singular values** of $\mathbf{X}$.

Note that the last $N - D$ columns of $\mathbf{U}$ are irrelevant (since they will be multiplied by zero). The **reduced SVD** avoids computing these unnecessary elements (and is a default option in most linear algebra packages) by factorizing $\mathbf{X}$ as

$$\mathbf{X} = \mathbf{U}_1\mathbf{D}_1\mathbf{V}^T,$$

where $\mathbf{U}_1$ is the $N \times D$ matrix containing the first D columns of $\mathbf{U}$ and $\mathbf{D}_1$ is a diagonal

$D \times D$ matrix containing the first D rows of $\mathbf{D}$. SVD also provides a method for inverting matrices, and you may come across it in other settings, e.g. linear regression. It is available in most linear algebra libraries (e.g. `scipy.linalg.svd`).

The SVD decomposition of $\mathbf{X}$ also provides a decomposition of $\mathbf{\Sigma}$ as:

$$\begin{aligned} N\mathbf{\Sigma} = \mathbf{X}^T\mathbf{X} &= \mathbf{V}\mathbf{D}_1\mathbf{U}_1^T\mathbf{U}_1\mathbf{D}_1\mathbf{V}^T \\ &= \mathbf{V}\mathbf{D}_1\mathbf{D}_1\mathbf{V}^T. \end{aligned}$$

Hence, the eigenvectors of $\mathbf{\Sigma}$ are equal to $\mathbf{V}$ (the right singular vectors of $\mathbf{X}$) and the eigenvalues of $\mathbf{\Sigma}$ are the squared singular values divided by N. Notice also that we can compute the encoding $\mathbf{Z}$ from the SVD decomposition:

$$\mathbf{Z} = \mathbf{X}\mathbf{W} = \mathbf{U}_1\mathbf{D}_1\mathbf{V}^T\mathbf{W} = \tilde{\mathbf{U}}\tilde{\mathbf{D}},$$

where $\tilde{\mathbf{U}}$ contains the first L columns of $\mathbf{U}$ and $\tilde{\mathbf{D}}$ is a diagonal matrix containing only the first L singular values. Then, the reconstructed data is the same as a truncated SVD approximation:

$$\widehat{\mathbf{X}} = \mathbf{Z}\mathbf{W}^T = \tilde{\mathbf{U}}\tilde{\mathbf{D}}\mathbf{W}^T.$$

Thus, we can perform PCA either using an eigen decomposition of $\mathbf{\Sigma}$ or an SVD decompostion of $\mathbf{X}$. The latter is often preferred for computational reasons.

2.4 Choosing the Number of Principal Components

In general, an $N \times D$ data matrix $\mathbf{X}$ has $\min(N, D)$ principal components. However, we are not interested in all of them; rather, we would like to use the smallest number required in order to get a good understanding and visualization of the patterns in the data. How many principal components are needed? Unfortunately, there is no single answer to this question.

Typically, we decide on the number of principal components by examining a **scree plot**, which plots the eigenvalues λ_l of $\mathbf{\Sigma}$ against l in order of decreasing magnitude. One can show (see exercises) that the reconstruction error when using L principal components is:

$$\mathcal{L}_L = \frac{1}{N}\sum_{n=1}^{N}\|\mathbf{x}_n - \widehat{\mathbf{x}}_n\|_2^2 = \sum_{l=L+1}^{D}\lambda_l, \tag{6}$$

and that the proportion of variance explained (PVE) by the lth principal component is:

$$\text{PVE}_l = \frac{\sum_{n=1}^{N} \mathbf{z}_{n,l}^2}{\sum_{n=1}^{N} \mathbf{x}_n^T \mathbf{x}_n} = \frac{\lambda_l}{\sum_{j=1}^{D} \lambda_j}.$$

Thus, we chose L by eyeballing this screeplot and selecting the value at which point the proportion of variance explained by each subsequent principal component drops off. This drop is typically referred to as an **elbow** in the scree plot. An example is shown in Figure 5, where one might argue that a good amount of variance is explained by the first two principal components and that there is an elbow after the second principal component.

While we don't have time to cover it in the course, another approach to select the number of principal components based on an **approximate Bayesian model selection** criterion was proposed by Minka (2000). This is implemented in the `PCA` transformer of sklearn by setting the parameter `n_components='mle'`. We have covered PCA from a purely loss-based perspective, but an interesting probabilistic construction is provided in **probabilistic PCA**, by Tipping and Bishop (1999). Note only does this provide a modelling framework underlying PCA, but it has also motivated other ideas and extensions, including the Bayesian model selection of Minka (2000).

However, in practice, the number of principal components required will depend on the data and problem at hand. If the first few principal components show interesting patterns, then we may choose to also examine subsequent principal components in EDA. If we are using PCA to obtain a lower-dimensional set of features to use in a supervised learning task, we may wish to use a larger number of principal components for more accurate reconstructions and in fact, this number can be treated as a hyperparameter and selected via model selection tools (that we will see in Week 4).

3 Beyond PCA

PCA is the simplest and most widely-used form of dimensionality reduction and the basis for numerous extensions. We provide a brief summary of some extensions for interested students. **Probabilistic PCA** (PPCA, Tipping & Bishop (1999)) is the probabilistic form of PCA that provides the PCA solution in the zero-noise limit. **Factor analysis** (FA) is a generalization of PPCA that allows for dimension-specific noise. **Independent component analysis** (ICA) generalizes factor analysis to allow the distribution of the latent factor to be any non-Gaussian distribution. Some nonlin-

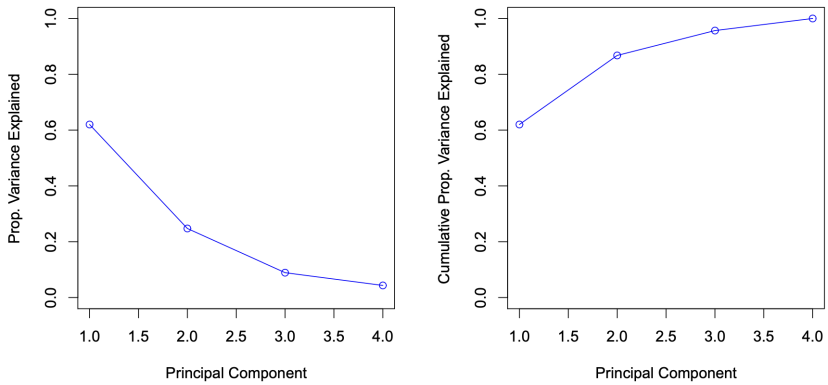


Figure 5: A scree plot depicting the proportion of variance explained (left) and the cumulative proportion of variance explained (right). From *An Introduction to Statistical Learning* (2023).

ear dimension reduction techniques include, e.g. **kernelized PCA** (Schölkopf et al (1998)), **Gaussian process latent variable models** (GPLVM, Lawrence (2003)), **t-distributed stochastic neighbor embeddings** (t-SNE, van der Maaten & Hinton (2008)), **uniform manifold approximation and projection** (UMAP, McInnes, Healy, & Melville (2018)) and **autoencoders**. Many of the techniques are available in `sklearn`, and for details and examples, see `sklearn`'s information on Matrix factorization and `sklearn.decomposition`, as well as Manifold learning and `sklearn.manifold`.

3.1 Kernel PCA

Lastly, we introduce kernel PCA, in which the **kernel trick** is used to extend PCA from linear to nonlinear reduction.

3.1.1 Kernels

A **kernel function**, denoted $k(\mathbf{x}, \mathbf{x}')$, is a real-valued function of two inputs $\mathbf{x}, \mathbf{x}' \in \mathcal{X}$. The general notation for the input space $\mathcal{X}$ is used to highlight that kernels can be defined over more abstract input spaces, such as strings. Typically, the kernel is assumed to be symmetric, i.e. $k(\mathbf{x}, \mathbf{x}') = k(\mathbf{x}', \mathbf{x})$, and non-negative, i.e. $k(\mathbf{x}, \mathbf{x}') \geq 0$.

One possible choice is the **radial basis function** (RBF) kernel, defined as:

$$k(\mathbf{x}, \mathbf{x}') = \exp\left(-\frac{1}{2\ell^2}(\mathbf{x} - \mathbf{x}')^T(\mathbf{x} - \mathbf{x}')\right),$$

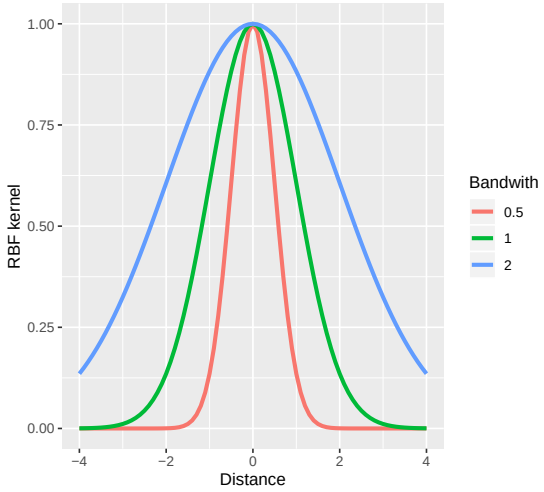


Figure 6: Radial basis function: as a function of the Euclidean distance $\sqrt{(\mathbf{x}-\mathbf{x}')^T(\mathbf{x}-\mathbf{x}')}$ for different bandwidth parameters.

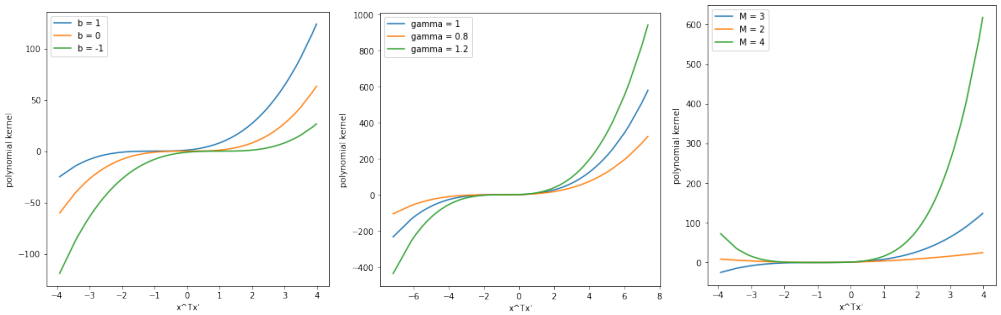


Figure 7: Polynomial kernel: as a function of the scalar product $\mathbf{x}^T\mathbf{x}'$. Left: γ is fixed to 1, M is fixed to 3, and location of the curve changes as b changes. Middle: b is fixed to 1, M is fixed to 3, the curve becomes steeper as γ increases. Right: γ is fixed to 1, b is fixed to 1, and the shape of the polynomial curve changes as M changes.

with **length-scale** or **bandwidth** parameter $\ell > 0$. This function has a bell-curve shape, with the spread controlled by the bandwidth. Figure 6 illustrates the shape of the RBF kernel for different bandwidth parameters.

Another possible option is the **polynomial** kernel:

$$k(\mathbf{x}, \mathbf{x}') = (-\gamma \mathbf{x}^T \mathbf{x}' + b)^M, \tag{7}$$

with parameters $\gamma > 0$, b, and degree $M \in \mathbb{N}$. This function has a polynomial shape, as illustrated in Figure 7, with γ controlling the steepness, b controlling the location, and

M controlling the shape.

Within `sklearn`, built-in functions are available for a number of kernels, such as the `rbf_kernel` and `polynomial_kernel`. For other kernels and further details, see

- `metrics`;
- `metrics.pairwise`;
- `metrics.pairwise.pairwise_kernels`.

3.1.2 Kernel Trick

The simply idea behind the kernel trick is to replace all inner products of the form $\mathbf{x}^T\mathbf{x}'$ in the Gram matrix (4) with a kernel function $k(\mathbf{x}, \mathbf{x}')$. This requires us to first rewrite the computations, so that they only depend on the inputs through their inner products, and then to replace all inner products with the kernel function. It turns out that many machine learning algorithms, in regression, classification, dimension reduction, and clustering, can be kernelized in this way. Note that for trick to work, we require the kernel to be a **Mercer kernel**, also called a **positive definite kernel**. This means that the **Gram matrix**, defined by

$$\mathbf{K} = \begin{bmatrix} k(\mathbf{x}_1, \mathbf{x}_1) & \dots & k(\mathbf{x}_1, \mathbf{x}_N) \\ \vdots & & \vdots \\ k(\mathbf{x}_N, \mathbf{x}_1) & \dots & k(\mathbf{x}_N, \mathbf{x}_N) \end{bmatrix},$$

must be positive definite for any set of inputs $\{\mathbf{x}_n\}_{n=1}^N$. For example, it can be shown that the polynomial and RBF kernels are examples of Mercer kernels.

Mercer's Theorem, which is out of scope for this course[1], states that for all Mercer kernels, we can write $k(\mathbf{x}, \mathbf{x}') = \phi(\mathbf{x})^T\phi(\mathbf{x}')$, where $\phi(\mathbf{x})$ represents an expansion of our inputs into a new feature space. The polynomial kernel is an example, in which we can explicitly write the feature expansion. For instance, consider $M = 2$, $\gamma = 1$, and $b = 1$ and $\mathbf{x} \in \mathbb{R}^2$, we have:

$$\begin{aligned} k(\mathbf{x}, \mathbf{x}') = (1 + \mathbf{x}^T\mathbf{x}')^2 &= (1 + x_1x_1' + x_2x_2')^2 \\ &= 1 + 2x_1x_1' + 2x_2x_2' + (x_1x_1')^2 + (x_2x_2')^2 + 2x_1x_1'x_2x_2'. \end{aligned}$$

[1] Section 17.1.1 of Murphy (2022)

This can be written as $k(\mathbf{x}, \mathbf{x}') = \phi(\mathbf{x})^T \phi(\mathbf{x}')$, where

$$\phi(\mathbf{x}) = \left(1, \sqrt{2}x_1, \sqrt{2}x_2, x_1^2, x_2^2, \sqrt{2}x_1 x_2\right)^T$$

contains all terms up to degree $M = 2$. So, using the kernel transforms the inputs into a six-dimensional feature space. Instead, the RBF kernel corresponds to an infinite-dimensional feature expansion $\phi(\mathbf{x})$. Thus, when using the kernel trick, we are implicitly replacing $\mathbf{x}$ with $\phi(\mathbf{x})$, yet, by working with the kernel function, we can avoid having to deal with high, and potentially infinite, dimensional vectors.

3.1.3 Applying the kernel trick to PCA

After applying the kernel trick to the Gram matrix in Section 2.3, $\mathbf{U}$ and $\mathbf{\Lambda}$ now represent the orthonormal matrix containing the eigenvectors and diagonal matrix of eigenvalues of the Gram matrix $\mathbf{K}$, with $\tilde{\mathbf{U}}$ and $\tilde{\mathbf{\Lambda}}$ corresponding to the first L eigenvectors and eigenvalues. The normalized eigenvectors are $\tilde{\mathbf{V}} = \mathbf{\Phi}^T \tilde{\mathbf{U}} \tilde{\mathbf{\Lambda}}^{-\frac{1}{2}}$, where $\mathbf{\Phi}$ represents the corresponding design matrix in the new feature space. However, we can't actually compute $\tilde{\mathbf{V}}$, since $\phi(\mathbf{x})$ is potentially infinite-dimensional, but we can compute the encoding of any point $\mathbf{x}_n$ as:

$$\begin{aligned}
\mathbf{z}_n &= \tilde{\mathbf{V}}^T \phi(\mathbf{x}_n) \\
&= \left(\mathbf{\Phi}^T \tilde{\mathbf{U}} \tilde{\mathbf{\Lambda}}^{-\frac{1}{2}}\right)^T \phi(\mathbf{x}_n) \\
&= \tilde{\mathbf{\Lambda}}^{-\frac{1}{2}} \tilde{\mathbf{U}}^T \mathbf{\Phi} \phi(\mathbf{x}_n) = \tilde{\mathbf{\Lambda}}^{-\frac{1}{2}} \tilde{\mathbf{U}}^T \begin{bmatrix} k(\mathbf{x}_1, \mathbf{x}_n) \\ \vdots \\ k(\mathbf{x}_N, \mathbf{x}_n) \end{bmatrix}.
\end{aligned}$$

Thus, kernel PCA provides **only the encodings**. Methods have been developed for reconstructing the data from the encodings (e.g. Bakir, Weston & Schölkopf (2004)), however this is out of scope for this course.

There is one final detail to worry about. The new features are not centered. It can be shown that Gram matrix of the centered features is $\tilde{\mathbf{K}} = \mathbf{CKC}$, where the centering matrix is $\mathbf{C} = \mathbf{I}_N - \frac{1}{N}\mathbf{1}_N \mathbf{1}_N^T$ (we will skip the details but for those interested, see 20.4.6 of Murphy 2022). We will briefly kernel PCA in extra material of this week's lab, available through `sklearn.decomposition.KernelPCA`. A simple comparison of PCA and kernel PCA is provided in Figure 8 on synthetic data from two nested cirlces. While PCA results in little change (only rotation), as the data is uncorrelated, with kernel PCA, we

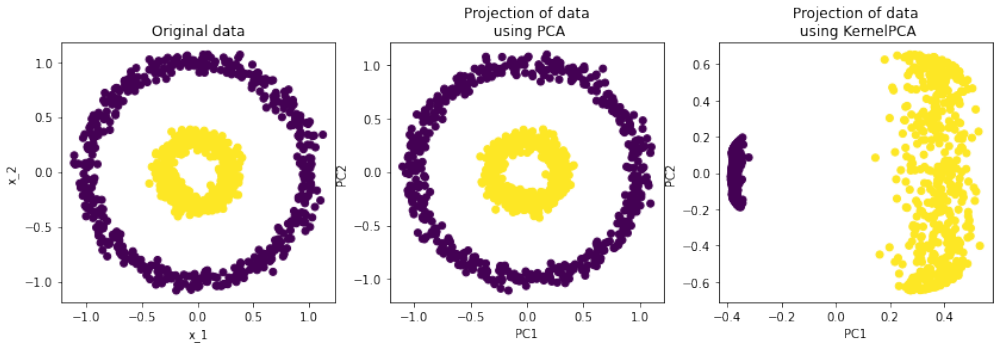


Figure 8: Kernel PCA on synthetic data. Left: the data is made of two nested circles. Middle: PCA projection of the data results in little change, as the data is uncorrelated. Right: kernel PCA projection with RBF separates the circles along the first dimension.

can linearly separate the two classes/circles along the first PCA.

Further Reading

The course notes provide complete material for the exam and assessments. For those interested in further reading:

- Chp. 12 on Unsupervised Learning of *An Introduction of Statistical Learning* by James et al. (2023), a https://www.statlearning.com/;
- Chp. 14 on Unsupervised Learning of *Elements of Statistical Learning* by Hastie, Tibshirani, & Friedman (2001), https://hastie.su.domains/Papers/ESLII.pdf;
- Chp. 20 on Dimension Reduction of *Probabilistic Machine Learning: An Introduction*, K. Murphy (2022), https://probml.github.io/pml-book/book1.html.
- Chp. 17 on Kernel Methods of *Probabilistic Machine Learning: An Introduction*, K. Murphy (2022), https://probml.github.io/pml-book/book1.html.
- Chp 1 *Data-Driven Science and Engineering*, Brunton & Kutz.

Exercises

1. Show that minimizing the loss for $\mathbf{w}_1$ in (2) is equivalent to finding the weights $\mathbf{w}_1$ that maximize the empirical variance of $z_{1,1}, \ldots, z_{N,1}$.
2. Show that the solution for $\mathbf{w}_2$ which minimizes the loss in (3) is given by the eigenvector with the second largest eigenvalue.

3. Let $\mathbf{U}$ be an $N \times N$ orthonormal matrix containing the eigenvectors of the Gram matrix. We showed that $\mathbf{V} = \mathbf{X}^T\mathbf{U}$ are the eigenvectors of the empirical covariance matrix. Show that a) the columns of $\mathbf{V}$ are not normalized, and b) how to obtain the normalized eigenvectors.

4. Derive the expression for the reconstruction loss in (6) by:

   (a) Show that the reconstruction loss with L principal components is

   $$\mathcal{L}_L = \frac{1}{N}\left(\sum_{n=1}^{N} \mathbf{x}_n^T\mathbf{x}_n\right) - \sum_{l=1}^{L} \lambda_l.$$

   (b) Next, use this to show that:

   $$\mathcal{L}_L = \sum_{l=L+1}^{D} \lambda_l.$$

   Hint: if $L = D$, the reconstruction loss is $\mathcal{L}_D = 0$.

5. Generate data from a bivariate Gaussian distribution,

   $$\mathbf{x}_n \sim \mathrm{N}\left(\begin{bmatrix}0\\0\end{bmatrix}, \begin{bmatrix}25 & 4\\4 & 1\end{bmatrix}\right).$$

   Perform PCA with and without standardizing the data. How do the results differ?