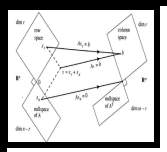
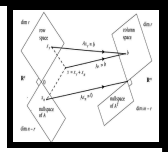



NOTES ON
PRINCIPAL COMPONENT ANALYSIS

THEORY AND CODE

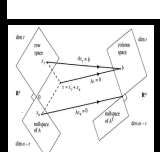
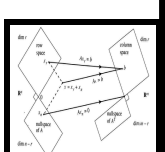
BY

0xLeo (github.com/0xleo)

AUGUST 3, 2020

DRAFT X.YY

MISSING: . . .

Contents

1 Principal Component Analysis

1.1 What is Principal Component Analysis useful for?

Principal Component Analysis (PCA) is a statistical way to identify patterns in data. Patterns are found based on how correlated the data are. Having found such patterns, it can reduce their dimensionality.

Suppose we have n samples of m-dimensional data, arranged in a matrix $\mathbf{X}_{n\times m}$. For example, suppose we have $n = 30$ individuals and for the i-th individual we have recorded their height (m), weight (kg) and IQ in a vector $\mathbf{x}_i \in \mathbb{R}^3$ ($m = 3$). A measurement $\mathbf{x}_i$ might look like:

$$\mathbf{x}_i = \begin{bmatrix} 1.8 & 70.3 & 105 \end{bmatrix}$$

These data could be visualised as a plot of 30 points in $\mathbb{R}^3$. PCA aims to answer the following questions:

1. Is there a simpler way to cluster the data? For example, if the measurements consist of 3D data, can they be clustered around a plane?
2. Which variables are correlated? In the example, we'd expect to see correlation between weight and height but not IQ with height.
3. Which variables are most significant when describing the dataset?

In general, when we have n measurements, each one containing m variables, or "features" (such as height, weight, etc), we stack in what's called a data matrix.

Definition 1.1 (data matrix). *Given n measurements of D variables, i.e. the measurement i is $\boldsymbol{x}_i = \begin{bmatrix} x_{i1} & x_{i2} & \dots & x_{im} \end{bmatrix}$, the data matrix contains the measurements as row vectors and the variables (features) as columns:*

$$\boldsymbol{X}_{n\times m} = \begin{bmatrix} \boldsymbol{x}_1 \\ \boldsymbol{x}_2 \\ \vdots \\ \boldsymbol{x}_n \end{bmatrix} = \begin{bmatrix} x_{11} & x_{12} & \dots & x_{1D} \\ x_{21} & x_{22} & \dots & x_{2D} \\ \vdots & \vdots & \ddots & \vdots \\ x_{n1} & x_{n2} & \dots & x_{nD} \end{bmatrix} \tag{1.1}$$

For instance, for our (height, weight, IQ) data, a toy data matrix would contain the measurements as rows and the *variables* ("features") as *columns*:

$$\mathbf{X} = \begin{bmatrix} 1.88 & 96 & 91 \\ 1.90 & 78 & 125 \\ 1.72 & 84 & 114 \\ 1.79 & 89 & 100 \\ 1.77 & 77 & 93 \end{bmatrix} \begin{matrix} \leftarrow \text{measurement } 1 \\ \leftarrow \dots \\ \leftarrow \dots \\ \leftarrow \dots \\ \leftarrow \text{measurement } 5 \end{matrix}$$

1.2 Required background

Before we study PCA, we need some linear algebra and statistics background. Proofs are left to the reader. Some data science and statistics terminology is also used.

Lemma 1.1. *If $\boldsymbol{A} \in \mathbb{R}^{n\times D}$, then matrices $AA^T \in \mathbb{R}^{n\times n}$ and $A^TA \in \mathbb{R}^{D\times D}$. Furthermore, AA^T and A^TA share the same, non-zero, eigenvalues.*

Lemma 1.2. *The eigenvalues of $A^\top A$ and $AA^\top$ are non-negative.*

In PCA, which is studied later, we often have a "fat" matrix $\mathbf{A}$ of size $M \times N$, $N \gg M$ and want to find the eigenvalues of the product $\mathbf{A}^\top\mathbf{A}$. Note that $\mathbf{A}^\top\mathbf{A}$ is of size $N \times N$, so e.g. for $N = 1000$, it is impractical to directly perform eigen-analysis on $\mathbf{A}^\top\mathbf{A}$ – a 1000×1000 matrix! Instead, we perform it on $\mathbf{A}\mathbf{A}^\top \in \mathbb{R}^{M\times M}$ and leverage a relationship between the eigendata of $\mathbf{A}^\top\mathbf{A}$ and $\mathbf{A}\mathbf{A}^\top$.

Lemma 1.3 (Eigenvalues of $\mathbf{A}^\top\mathbf{A}$ and $\mathbf{A}\mathbf{A}^\top$). *Let $\boldsymbol{A} \in \mathbb{R}^{M\times N}$ and λ_i, $\boldsymbol{u}_i$ be the eigenvalues and eigenvectors of $\boldsymbol{A}^\top\boldsymbol{A}$. If μ_i and $\boldsymbol{v}_i$ are the eigenvalues and eigenvectors of $\boldsymbol{A}\boldsymbol{A}^\top$, then the eigendata of $\boldsymbol{A}^\top\boldsymbol{A}$ are given by:*

$$\lambda_i = \mu_i \qquad \boldsymbol{u}_i = \boldsymbol{A}^\top\boldsymbol{v}_i \tag{1.2}$$

Don't get confused with the notation; $[x_{i1} \dots x_{in}]^\top = \begin{bmatrix} x_{1i} \\ \vdots \\ x_{ni} \end{bmatrix}$ refers to the i-th column (feature) of the data matrix.

Proof. Since $\mathbf{v}_i$ and μ_i are the eigendata of $\mathbf{A}\mathbf{A}^\top$:

$$\mathbf{A}\mathbf{A}^\top\mathbf{v}_i = \mu_i\mathbf{v}_i \Rightarrow$$

$$\mathbf{A}^\top\mathbf{A}\underbrace{\mathbf{A}^\top\mathbf{v}_i}_{\mathbf{u}_i} = \underbrace{\mu_i}_{\lambda_i}\underbrace{\mathbf{A}^\top\mathbf{v}_i}_{\mathbf{u}_i}$$

The last relationship precisely tells us that the eigenvalues of $\mathbf{A}^\top\mathbf{A}$ are $\lambda_i = \mu_i$ and the its eigenvectors are $\mathbf{u}_i = \mathbf{A}^\top\mathbf{v}_i$. □

The following definitions from statistics will be required.

DEFINITION 1.2 (mean of feature). *Let $\boldsymbol{X} \in \mathbb{R}^{n\times D}$ be the data matrix. Each column contains a feature $\boldsymbol{x}_i$ of n entries with $\boldsymbol{x}_i = \begin{bmatrix} x_{i1} & x_{i2} & \dots & x_{in}\end{bmatrix}^\top$. Then we define the mean of feature j as its average:*

$$\bar{x}_j = \frac{1}{n}\sum_{i=1}^{n} x_{ij} \tag{1.3}$$

DEFINITION 1.3 (variance). *Given a feature $\boldsymbol{x}_j \in \mathbb{R}^{n\times 1}$ in the data matrix with entries $x_{1j},\dots,x_{nj}$ and mean $\bar{x}_j$, its sample variance $\sigma^2_{\boldsymbol{x}_i}$ is defined as:*

$$\sigma^2_{\boldsymbol{x}_j} = \frac{1}{n-1}\sum_{i=1}^{n}(x_{ij}-\bar{x_j})^2 \tag{1.4}$$

Variance of a feature measures its spread.Variance of a feature measures its spread.

DEFINITION 1.4 (covariance). *Given two features (column vectors in data matrix) $\boldsymbol{x}_j,\boldsymbol{x}_k$ with entries $\begin{bmatrix} x_{j1} & x_{j2} & \dots & x_{jn}\end{bmatrix}^\top$, $\begin{bmatrix} x_{k1} & x_{k2} & \dots & x_{kn}\end{bmatrix}^\top$ respectively and means $\bar{x_j}$, $\bar{x_k}$ respectively, their covariance is defined as:*

$$\sigma(x_j,x_k) = \frac{1}{n-1}\sum_{i=1}^{n}(x_{ij}-\bar{x_j})(x_{ik}-\bar{x_k}) \tag{1.5}$$

Notice how the covariance of x_j with itself is its variance. Covariance provides intuition regarding the dispersion of $\mathbf{x}_j$ and $\mathbf{x}_k$ – i.e. how they change with respect to one another;

- Covariance $\sigma(\mathbf{x}_j,\mathbf{x}_k)$ is positive when variables $\mathbf{x}_j,\mathbf{x}_k$ "move together".
- Covariance is negative when $\mathbf{x}_j,\mathbf{x}_k$ "move inversely".
- Covariance is zero when $\mathbf{x}_j,\mathbf{x}_k$ move randomly w.r.t. each other, i.e. they are "uncorrelated" (fancy word for unrelated).

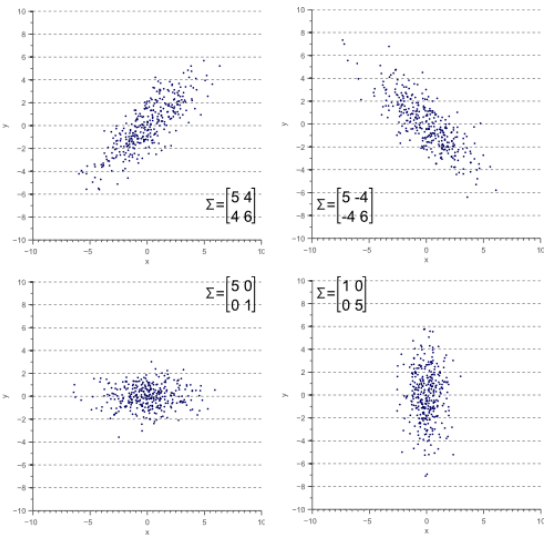


Fig. 1. Some covariance matrices Σ for various 2D datasets. Observe how the matrix values are related to the shape (https://pathmind.com/wiki/eigenvector).

From the definition, it is straightforward that the covariance function is symmetric.

COROLLARY 1.1 (covariance is symmetric). *For the covariance of any two feature vectors $\boldsymbol{x}_k$, $\boldsymbol{x}_k \in \mathbb{R}^{n\times 1}$:*

$$\sigma(\boldsymbol{x}_j,\boldsymbol{x}_k) = \sigma(\boldsymbol{x}_j,\boldsymbol{x}_k) \tag{1.6}$$

If we have multi-dimensional data, let's say n measurements of m-dimensional data, we compute the "covariance matrix" as follows.

DEFINITION 1.5 (covariance matrix). *Given n observations of D-dimensional variables, their covariance matrix $\boldsymbol{\Sigma} \in \mathbb{R}^{D\times D}$ refers to the symmetric array of numbers:*

$$\boldsymbol{\Sigma} = \frac{1}{n-1}\begin{bmatrix} \sigma_1^2 & \sigma_{12} & \dots & \sigma_{1D} \\ \sigma_{21} & \sigma_2^2 & \dots & \sigma_{2D} \\ \vdots & \vdots & \ddots & \vdots \\ \sigma_{D1} & \sigma_{D2} & \dots & \sigma_D^2 \end{bmatrix} \tag{1.7}$$

, where

- $\sigma_j^2 = \frac{1}{n-1}\sum_{i=1}^{n}(x_{ij}-\bar{x}_j)^2$ *is the variance of the j-th variable.*
- $\sigma_{jk}^2 = \frac{1}{n-1}\sum_{i=1}^{n}(x_{ik}-\bar{x}_j)(x_{ik}-\bar{x}_k)$ *is the covariance between variables j and k.*

Each entry jk of the covariance matrix contains the covariance between two feature vectors $\boldsymbol{x}_j$, $\boldsymbol{x}_k$, $1 \le j,k \le m$

We will now show how the covariance matrix $\boldsymbol{\sigma}$ can be computed from the data matrix $\mathbf{X}$ using matrix operations. We defined in Eq. (1.3) the mean vector of all features as:

$$\bar{\mathbf{x}} = \begin{bmatrix} \bar{x}_1 \\ \vdots \\ \bar{x}_D \end{bmatrix}, \quad \bar{x}_j = \frac{1}{n}\sum_{i=1}^{n} x_{ij}$$

It can be easily verified that $\bar{\mathbf{x}}$ can be computed in terms of $\mathbf{X}$ as:

$$\bar{\mathbf{x}} = n^{-1}\mathbf{X}^\top \mathbf{1}_{n\times 1} \tag{♣}$$

, where $\mathbf{1}_{n\times 1}$ is a column vector of ones. The covariance matrix can also be written in terms of the "centred" (zero mean) matrix $\mathbf{X}_C$

$$\boldsymbol{\sigma}(\mathbf{X}) = n^{-1}\mathbf{X}_c^\top \mathbf{X}_c, \quad \mathbf{X}_c = \begin{bmatrix} x_{11}-\bar{x_1} & x_{12}-\bar{x_2} & \dots & x_{1p}-\bar{x_p} \\ x_{21}-\bar{x_1} & x_{22}-\bar{x_2} & \dots & x_{2p}-\bar{x_p} \\ \vdots & \vdots & \ddots & \vdots \\ x_{n1}-\bar{x_1} & x_{n2}-\bar{x_2} & \dots & x_{np}-\bar{x_p} \end{bmatrix} = \mathbf{X} - \begin{bmatrix} \bar{x_1} & \bar{x_2} & \dots & \bar{x_p} \\ \vdots & \vdots & \ddots & \vdots \\ \bar{x_1} & \bar{x_2} & \dots & \bar{x_p} \end{bmatrix} = \mathbf{X} - \mathbf{1}_n\bar{\mathbf{x}}^\top$$

$$\therefore \mathbf{X}_C = \mathbf{X} - \mathbf{1}_{n\times 1}\bar{\mathbf{x}}^\top \tag{♣♣}$$

, where $\bar{\mathbf{x}} = \begin{bmatrix}\bar{x}_1 & \dots & \bar{x}_p\end{bmatrix}^\top$ is the vector of the feature means and $\mathbf{1}_n$ is a $n \times 1$ vector of ones. For instance, if the means vector was $\bar{\mathbf{x}} = \begin{bmatrix}1 & 2 & 3\end{bmatrix}^\top$, then the result $\mathbf{1}_{4\times 1}\bar{\mathbf{x}}^\top$ would be:

$$\begin{bmatrix}1\\1\\1\\1\end{bmatrix}_{4\times 1} \begin{bmatrix}1 & 2 & 3\end{bmatrix}_{1\times 3} = \begin{bmatrix}1 & 2 & 3\\1 & 2 & 3\\1 & 2 & 3\\1 & 2 & 3\end{bmatrix}_{4\times 3}$$

Remember, the goal is to express $\boldsymbol{\Sigma}$ in terms of $\mathbf{X}$. What's left is to express $\bar{\mathbf{x}}$ in terms of $\mathbf{X}$. Substituting Eq. (♣) in Eq. (♣♣):

$$\begin{aligned}\mathbf{X}_C &= \mathbf{X} - n^{-1}\mathbf{1}_{n\times 1}\mathbf{1}_{n\times 1}^\top\mathbf{X} = \\ &= (\mathbf{I}_n - n^{-1}\mathbf{1}_{n\times 1}\mathbf{1}_{n\times 1}^\top)\mathbf{X} = C\mathbf{X}, \quad \mathbf{C} := \mathbf{I}_n - n^{-1}\mathbf{1}_{n\times 1}\mathbf{1}_{n\times 1}^\top\end{aligned}$$

To recap, we found how the "centred" version of the data matrix $\mathbf{X} \in \mathbb{R}^{n \times D}$ can be obtained as a product. The centred $n \times D$matrix $\mathbf{X}_C$ contains the centred features (columns). Centring implies that the mean of each feature is zero and as we'll see later this simplifies the PCA calculations.

$$\mathbf{X}_C = C\mathbf{X}, \quad \mathbf{C} := \mathbf{I}_n - n^{-1}\mathbf{1}_{n\times 1} \tag{1.8}$$

Since $\mathbf{\Sigma}$ is expressed w.r.t $\mathbf{X}_C$, it's straightforwardd to express it w.r.t the original data matrix $\mathbf{X}$.

$$\therefore \mathbf{\Sigma}(\mathbf{X}) = \mathbf{X}_C^\top \mathbf{X}_C = (n-1)^{-1}(\mathbf{CX})^\top \mathbf{CX} = (n-1)^{-1}\mathbf{X}^\top \mathbf{C}^\top \mathbf{CX}$$

To reiterate, we established that the covariance matrix can be efficiently computed as a product containing the data matrix and a constant one.

COROLLARY 1.2 (covariance matrix as product)**.** *If the data matrix is $\boldsymbol{X}_{n\times p}$, whose its columns contain features, the covariance matrix can be computed as the product:*

$$\boldsymbol{\Sigma} = (n-1)^{-1}\boldsymbol{X}^\top \boldsymbol{C}^\top \boldsymbol{CX}, \quad \boldsymbol{C} = \boldsymbol{I}_n - n^{-1}\boldsymbol{1}_{n\times 1}\boldsymbol{1}_{n\times 1}^\top \tag{1.9}$$

We can quickly verify the validity of this by comparing with Matlab/ Octave's `cov` command.

```
>> X = [189 77 119;170 74 122;185 72 91;172 94 115]; % data matrix
>> n = size(X, 1);
>> C = eye(n) - 1/n*ones(n,1)*ones(n,1)';
>> cov_mat_form = 1/(n-1)*X'*C'*C*X;
>> cov_mat_form./cov(X) % divide element-wise
ans =
   1.00000   1.00000   1.00000
   1.00000   1.00000   1.00000
   1.00000   1.00000   1.00000
```

1.3 The idea behind PCA

PCA aims to reduce the dimensionality of the input data by projecting the D-dimensional input data to a d-dimensional space, $d \leq D$. First, the data are projected on a D-dimensional space spanned by some vectors $\mathbf{u}_1, \ldots, \mathbf{u}_D$. Basis vectors $\mathbf{u}_1, \ldots, \mathbf{u}_D$ fit along the most important directions of the projected data by mapping (projecting) the original data onto the vectors. Fitting a vector along the most important direction of the original data is expressed as maximising the variance of the points $\mathbf{x}_i \in \mathbb{R}^D$ projected along it. Having fit the data by projecting it, the next step PCA does it to keep only the d first most important projected vectors, projected along $\mathbf{u}_1, \ldots, \mathbf{u}_d$.

Fig. 2 shows how such a vector (PC1) would look like if we wanted to fit it so that the variance of the points projected to it is maximised. If we only use PC1 to represent the compressed data, we have reduced it from a 2D representation to 1D ($D = 2, d = 1$). So instead of reporting its 2D position, we can report its position (scalar) on the line, assuming the PC1 line is the new axis.

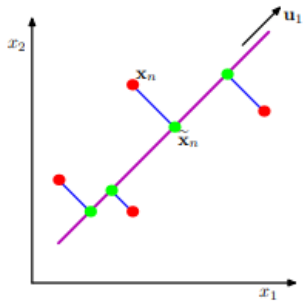


Fig. 2. Orthogonally projecting 2D data along an 1D line spanned by $\mathbf{u}_1$.

1.4 Problem formulation and objective

Before performing PCA, we always centre the data so that the basis they are represented by starts from the origin. Suppose our data consists of n D-dimensional observations $\mathbf{x}_1, \ldots, \mathbf{x}_n$, $\mathbf{x}_i \in \mathbb{R}^D$. We want to project them in an orthonormal basis $\mathbf{u}_1, \ldots, \mathbf{u}_D$ such that the projected points have certain properties.

DEFINITION 1.6 (PC loadings). *The orthonormal basis vectors $\boldsymbol{u}_1,\ldots,\boldsymbol{u}_D$ that form the basis of the projected data $\boldsymbol{x}_1,\ldots,\boldsymbol{x}_n, \boldsymbol{x}_i \in \mathbb{R}^D$ are called PC loadings.*

$$\boldsymbol{u}_i^\top \boldsymbol{u}_j = \begin{cases} 1 & i = j \\ 0 & i \neq j \end{cases} \tag{1.10}$$

Projected and spanned by the basis $\mathbf{u}_1,\ldots,\mathbf{u}_D$, a data point $\mathbf{x} \in \mathbb{R}^D$ is written as:

$$\mathbf{x} = \sum_{i=1}^{D} (\mathbf{x}^\top \mathbf{u}_i)\mathbf{u}_i$$

, where $(\mathbf{x}^\top \mathbf{u}_i)\mathbf{u}_i$ is its projection along axis $\mathbf{u}_i$. The objective of PC is to approximate each data vector $\mathbf{x}$ at a lower-dimensional subspace using as few subspace basis vectors as possible, i.e.

$$\hat{\mathbf{x}} = \sum_{i=1}^{d} (\mathbf{x}^\top \mathbf{u}_i)\mathbf{u}_i, \quad d \leq D \tag{1.11}$$

COROLLARY 1.3 (objective of PCA – minimise the reconstruction error). *The orthogonal reconstruction error for one centred data vector $\boldsymbol{x}$ is defined as:*

$$E(\boldsymbol{x}) = \|\hat{\boldsymbol{x}} - \boldsymbol{x}\|^2 = \sum_{i=d+1}^{D} (\boldsymbol{x}^\top \boldsymbol{u}_i)^2 \tag{1.12}$$

The total reconstruction error $\mathscr{L}(\boldsymbol{u}_1,\ldots,\boldsymbol{u}_d) = \sum_{i=1}^{n} E(\boldsymbol{x}_i;\boldsymbol{u}_1,\ldots,\boldsymbol{u}_d) = \sum_{j=1}^{n}\sum_{i=d+1}^{D} (\boldsymbol{x}_j^\top \boldsymbol{u}_i)^2$ is what we strive to minimise.

We minimise it by choosing appropriate basis vectors $\mathbf{u}_1,\ldots,\mathbf{u}_d$. In the next section, we will see how these vectors should be chosen if we have a pre-defined value of d. Later we will also see how many PC loadings should be chosen to have a "good enough" representation of the original data, i.e. what the value of d should be.

As mentioned, PCA orthogonally projects the data from a space of dimensionality D to one of d, $d \leq D$. Projection is accomplished as a linear operation. Remember, if $\mathbf{X} = \begin{bmatrix}\mathbf{x}_1 & \ldots & \mathbf{x}_n\end{bmatrix}$ be a matrix that contains D-dimensional measurements $\mathbf{x} \in \mathbb{R}^D$, $i = 1,\ldots,n$. A projection $\mathbf{Y}$ of $\mathbf{X}$ through a linear map $\mathbf{A}: \mathbb{R}^D \to \mathbb{R}^d$ is given by:

$$\underset{(d\times n)}{\mathbf{Y}} = \underset{(d\times D)}{\mathbf{A}} \cdot \underset{(D\times n)}{\mathbf{X}} \tag{1.13}$$

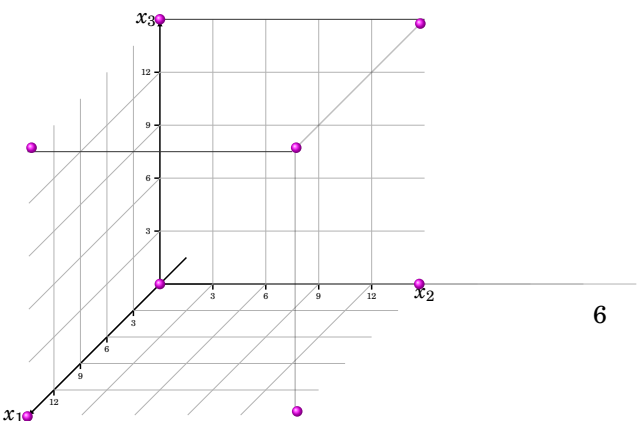
EXAMPLE 1.1. *For example, suppose we shave some 3D observations centred around 8 clusters:*

$$\boldsymbol{X} = \begin{bmatrix} 0 & 0 & 0 & 0 & 15 & 15 & 15 & 15 & 0 & 15 \\ 0 & 0 & 15 & 15 & 0 & 0 & 15 & 15 & 15 & 0 \\ 0 & 15 & 0 & 15 & 0 & 15 & 0 & 15 & 15 & 15 \end{bmatrix}$$

Let's the try multiplying by an arbitrary projection matrix $\mathbf{A}$, then the output is:

$$\boldsymbol{A} = \begin{bmatrix} 0 & 1 & 0 \\ 1 & 0 & 0 \end{bmatrix}, \quad \boldsymbol{Y} = \boldsymbol{A}\boldsymbol{X} = \begin{bmatrix} 0 & 0 & 15 & 15 & 0 & 0 & 15 & 15 & 15 & 0 \\ 0 & 0 & 0 & 0 & 15 & 15 & 15 & 15 & 0 & 15 \end{bmatrix}$$

Visually, the projection matrix reduces the data from a cube to a square:



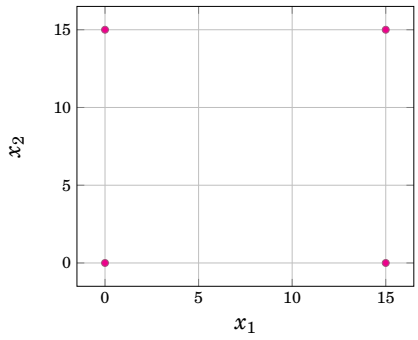


Fig. 4. The same data after being multiplied by the 2×3 projection matrix $\mathbf{A}$.

☞ **Note:** In the next sections, we will assume that $d = D$ unless otherwise specified. We will study how to compress from D principal components to d later.

1.5 Minimising error ⇔ project variance maximisation – intuition for one PC

In this section, we see what statistical properties the PCs must satisfy in order to minimise the reconstruction error. To simplify the derivations, we assume we want to project to 1D data, spanned by a single PC $\mathbf{u}$. Lemmas will be generalised in another section. We assume the original data points are D-dimensional but they will be visualised as 2D for simplicity. Finally, we assume that the data have been centred.

To simplify the derivations in this section, we assume zero-mean ($\bar{\mathbf{x}}_i = 0$) data.

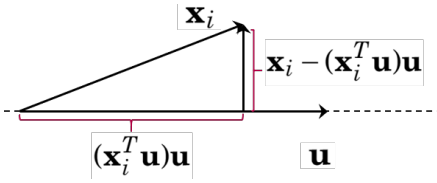


Fig. 5. Given a data vector $\mathbf{x}_i$ and a (unit vector) PC loading $\mathbf{u}$, the reconstruction error for $\mathbf{x}_i$ is $\left\|\mathbf{x}_i - (\mathbf{x}_i^\top\mathbf{u})\mathbf{u}\right\|^2$.

For data projected along a single line spanned by $\mathbf{u}$, maximising the variance of the projected data minimises the reconstruction error.

LEMMA 1.4 (variance maximisation ⇔ reconstruction error minimisation for one PC)**.** *Maximising the variance* $\frac{1}{n}\sum_{i=1}^{n}(\boldsymbol{u}^T\boldsymbol{x}_i)^2$ *for the projected data, is equivalent to minimising the total* $reconstruction\ error$ *(a.k.a. Mean Squared Error), which is defined in as:*

$$MSE = \frac{1}{n}\sum_{i=1}^{n}\left\|\boldsymbol{x}_i - (\boldsymbol{x}_i^\top\boldsymbol{u}_1)\boldsymbol{u}_1\right\|^2$$

Proof. For one data vector $\mathbf{x}_i$, the squared reconstruction error is:

$$\begin{aligned}
\left\|\mathbf{x}_i - (\mathbf{x}_i^\top\mathbf{u})\mathbf{u}\right\|^2 &= \left(\mathbf{x}_i - (\mathbf{x}_i^\top\mathbf{u})\mathbf{u}\right)^\top\left(\mathbf{x}_i - (\mathbf{x}_i^\top\mathbf{u})\mathbf{u}\right) \\
&= \mathbf{x}_i^\top\mathbf{x}_i - \mathbf{x}_i^\top\mathbf{u}(\mathbf{x}_i^\top\mathbf{u}) - (\mathbf{x}_i^\top\mathbf{u})\mathbf{u}^\top\mathbf{x}_i + (\mathbf{x}_i^\top\mathbf{u})\mathbf{u}^\top\mathbf{u}(\mathbf{x}_i^\top\mathbf{u}) \\
&= \|\mathbf{x}_i\|^2 - 2(\mathbf{x}_i^\top\mathbf{u})^2 + (\mathbf{x}_i^\top\mathbf{u})^2 \\
&= \|\mathbf{x}_i\|^2 - (\mathbf{x}_i^\top\mathbf{u})^2
\end{aligned}$$

, as $(\mathbf{x}_i^\top\mathbf{u}1)$ is a scalar, $\mathbf{x}_i^\top\mathbf{u} = \mathbf{u}^\top\mathbf{x}_i$, and $\mathbf{u}^\top\mathbf{u} = \|\mathbf{u}\|_1^2 = 1$. To obtain the MSE we sum over the n data points:

$$MSE = \frac{1}{n}\sum_{i=1}^{n}\left\|\mathbf{x}_i - (\mathbf{x}_i^\top\mathbf{u})\mathbf{u}\right\|^2 = \frac{1}{n}\left(\sum_{i=1}^{n}\|\mathbf{x}_i\|^2 - \sum_{i=1}^{n}(\mathbf{x}_i^\top\mathbf{u})^2\right)$$

The former term is constant and the latter (assuming centred data) is $\sigma_{\mathbf{u}}^2$. The proof is done. □

1.6 Mathematical properties of PC's

Before we derive the other PCs, we describe the properties they must satisfy. Again, let $\mathbf{X} = [\mathbf{x}_1, \ldots, \mathbf{x}_D] \in \mathbb{R}^{D \times n}$ be the (centred) data matrix and $\mathbf{x}_i \in \mathbb{R}^n$ a feature vector. Also, by convention, we rank the covariance matrix $\mathbf{\Sigma}$ eigenvalues in descending order; $\lambda_1 \geq \ldots \lambda_D \geq 0$[1]. The 3 properties are:

For these expressions to be true, features must be centred and in columns.

1. Each of the PC $\mathbf{z}_j \in \mathbb{R}^n$ ($\mathbf{z}_j$'s are often called "scores") is a linear combination of the original feature vectors (columns of $\mathbf{X}$).

$$\begin{aligned} \mathbf{z}_1 &= u_{11}\mathbf{x}_1 + u_{21}\mathbf{x}_2 + \ldots + u_{D1}\mathbf{x}_D = \mathbf{X}\mathbf{u}_1 \\ \mathbf{z}_2 &= u_{21}\mathbf{x}_1 + u_{22}\mathbf{x}_2 + \ldots + u_{D2}\mathbf{x}_D = \mathbf{X}\mathbf{u}_2 \\ &\vdots \\ \mathbf{z}_D &= u_{D1}\mathbf{x}_1 + u_{D2}\mathbf{x}_2 + \ldots + u_{DD}\mathbf{x}_D = \mathbf{X}\mathbf{u}_D \end{aligned} \tag{1.14}$$

   More compactly, the scores can be found by the multiplication: $\mathbf{X}$ is centred.

$$\begin{bmatrix} | & | & & | \\ \mathbf{z}_1 & \mathbf{z}_2 & \ldots & \mathbf{z}_D \\ | & | & & | \end{bmatrix} = \mathbf{X} \begin{bmatrix} | & | & & | \\ \mathbf{u}_1 & \mathbf{u}_2 & \ldots & \mathbf{u}_D \\ | & | & & | \end{bmatrix} \tag{1.15}$$

   It makes sense for every weight vector $\mathbf{u}_j = \begin{bmatrix} u_{j1} \\ \vdots \\ u_{jD} \end{bmatrix}$ to impose $\|\mathbf{u}_j\|^2 = \sum_{i=1}^{D} \mathbf{u}_{ij}^2 = 1$, $i = 1, \ldots, n$ so that we have some sort of scale. By expanding the feature columns, we find that the entries of the j-th PC score $\mathbf{z}_j$ are given by:

$$\begin{bmatrix} z_{j1} \\ z_{j2} \\ \vdots \\ z_{jn} \end{bmatrix} = u_{j1} \begin{bmatrix} X_{11} \\ X_{21} \\ \vdots \\ X_{n1} \end{bmatrix} + u_{j2} \begin{bmatrix} X_{12} \\ X_{22} \\ \vdots \\ X_{n2} \end{bmatrix} + \ldots + u_{jD} \begin{bmatrix} X_{1D} \\ X_{2D} \\ \vdots \\ X_{nD} \end{bmatrix} \tag{1.16}$$

$$\therefore \begin{bmatrix} z_{1j} \\ z_{2j} \\ \vdots \\ z_{nj} \end{bmatrix} = \begin{bmatrix} u_{j1}X_{11} + u_{j2}X_{12} + \ldots + u_{jD}X_{1D} \\ u_{j1}X_{21} + u_{j2}X_{22} + \ldots + u_{jD}X_{2D} \\ \vdots \\ u_{j1}X_{n1} + u_{j2}X_{n2} + \ldots + u_{jD}X_{nD} \end{bmatrix} \tag{1.17}$$

2. PCs are ranked in descending variance order:

$$\sigma^2(\mathbf{z}_1) \geq \sigma^2(\mathbf{x}_2) \geq \ldots \geq \sigma^2(\mathbf{x}_D) \tag{1.18}$$

3. PCs are mutually uncorrelated (i.e. their mutual information is zero):

$$cov(\mathbf{z}_i, \mathbf{z}_j) = 0, \quad i \neq j \tag{1.19}$$

change z indexes in fig below

[1] Eigenvalues are non-negative since the matrix is real and symmetric

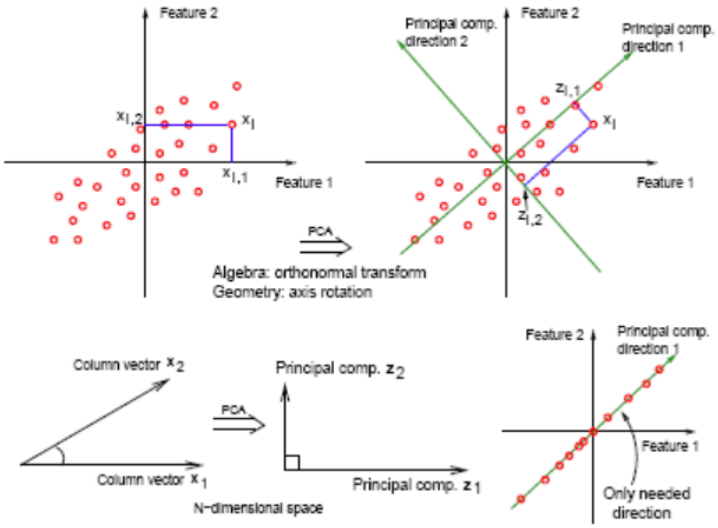


Fig. 6. Observation $\mathbf{x}_i = [x_{i1}\ x_{i2}]$ is projected on a 2D basis spanned by PC "loadings" $\mathbf{u}_1$, $\mathbf{u}_2$. The projected components along the basis vectors are z_{i1} and z_{i2} respectively. In the end, "scores" $\mathbf{z}_1 = [z]$

We saw that to find each PC loading $\mathbf{u}_j$ we rely on two important propertes – the variance of the projection of all observations on each loading $\mathbf{u}_j$ must be maximised and each PC score vector $\mathbf{z}_j$ is uncorrelated to all previous ones. Exception is the first PC vector $\mathbf{z}_1$ whose only requirement is that its variance is maximised. So we reach the following definition.

DEFINITION 1.7 (principal components)**.** *We define the principal components* $\boldsymbol{u}_1,\ldots,\boldsymbol{u}_D$ *such that:*

$$\begin{aligned}
\text{First PC } \boldsymbol{z}_1 &:= \text{linear combination } \boldsymbol{u}_1^\top \boldsymbol{X} \text{ that maximises } \sigma^2(\boldsymbol{X}\boldsymbol{u}_1) \\
&\text{s.t. } \boldsymbol{u}_1^\top \boldsymbol{u}_1 = 1 \\
\text{Second PC } \boldsymbol{z}_2 &:= \text{linear combination } \boldsymbol{u}_2^\top \boldsymbol{X} \text{ that maximises } \sigma^2(\boldsymbol{X}\boldsymbol{u}_2) \\
&\text{s.t. } \boldsymbol{u}_2^\top \boldsymbol{u}_2 = 1 \text{ and } \sigma(\boldsymbol{X}\boldsymbol{u}_2, \boldsymbol{X}\boldsymbol{u}_1) = 0 \\
&\vdots \\
j\text{-th PC } \boldsymbol{z}_j &:= \text{linear combination } \boldsymbol{u}_j^\top \boldsymbol{X} \text{ that maximises } \sigma^2(\boldsymbol{u}_j^\top \boldsymbol{X}) \\
&\text{s.t. } \boldsymbol{u}_j^\top \boldsymbol{u}_j = 1 \text{ and } \sigma(\boldsymbol{X}\boldsymbol{u}_j, \boldsymbol{X}\boldsymbol{u}_i) = 0 \quad \forall i < j
\end{aligned}$$

Remember, the definition on the left holds assuming the features are in columns – some authors stack them as rows so the product would become $\mathbf{u}_j^\top \mathbf{X}$. The variance lemma remains the same though.

Before we derive how loading (direction) vectors must be chosen, it's important to note that the variance of each PC $\mathbf{z}_j$ can be written as a prouduct involving the covariance matrix $\boldsymbol{\Sigma}$.

LEMMA 1.5 (Variance along a PC in terms of covariance matrix)**.** *The variance of centred data projected on a PC* $\boldsymbol{u}_j$*, i.e. on* $\boldsymbol{z}_j = \boldsymbol{X}\boldsymbol{u}_j$ *can be written as:*

$$\sigma^2(\boldsymbol{z}_j) = \boldsymbol{u}_j^\top \boldsymbol{\Sigma} \boldsymbol{u}_j \tag{1.20}$$

The covariance between score vectors $\boldsymbol{z}_i, \boldsymbol{z}_j$ *can be written as:*

$$\sigma(\boldsymbol{z}_i, \boldsymbol{z}_j) = \boldsymbol{u}_i^\top \boldsymbol{\Sigma} \boldsymbol{u}_j = \boldsymbol{u}_i^\top \boldsymbol{\Sigma} \boldsymbol{u}_j \tag{1.21}$$

(since $\boldsymbol{\Sigma}$ *is symmetric).*

Proof. Recall that the data matrix $\mathbf{X}$ contains the features as columns and the observations as row, therefore $\mathbf{X}^\top$ contains the observations as columns.

Let's consider $\mathbf{z}_j$. In Eq. (1.17), the i-th row z_{ij} is the dot product $\mathbf{u}_j^\top \mathbf{X}_i^\top$, where $\mathbf{X}_i^\top$ is the i-th column of $\mathbf{X}^\top$, therefore the i-th observation vector. Then, $\mathbf{z}_j$ can alternatively be written as:

$$\begin{bmatrix} z_{1j} \\ z_{2j} \\ \vdots \\ z_{nj} \end{bmatrix} = \begin{bmatrix} \mathbf{u}_j^\top \mathbf{X}_1^\top \\ \mathbf{u}_j^\top \mathbf{X}_2^\top \\ \vdots \\ \mathbf{u}_j^\top \mathbf{X}_n^\top \end{bmatrix}$$

It's trivial to show that $\mathbf{z}_j$ is zero-centred so its variance is simply the squared sum of its elements:

$$\begin{aligned} \sigma^2(\mathbf{z}_j) &= \frac{1}{n-1}\sum_{i=1}^{n} z_{ij}^2 \\ &= \frac{1}{n-1}\sum_{i=1}^{n} (\mathbf{u}_j^\top \mathbf{X}_i^\top)(\mathbf{u}_j^\top \mathbf{X}_i^\top) \\ &= \frac{1}{n-1}\sum_{i=1}^{n} \mathbf{u}_j^\top \mathbf{X}_i^\top \mathbf{X}_i \mathbf{u}_j \qquad (\mathbf{a}^\top \mathbf{b} = \mathbf{b}^\top \mathbf{a}) \\ &= \frac{1}{n-1}\mathbf{u}_j^\top \mathbf{X}^\top \mathbf{X} \mathbf{u}_j \\ &= \mathbf{u}_j^\top \mathbf{\Sigma} \mathbf{u}_j \end{aligned}$$

We can prove Eq. (1.21) exactly the same way. □

1.6.1 Deriving the first PC

Given the data matrix $\mathbf{X}$, we will see how $\mathbf{u}_1$ must be chosen to satisfy the properties of the first PC loading, particularly to maximise $\sigma(\mathbf{z}_1)$ subject to $\|\mathbf{u}_1\| = \mathbf{u}_1^\top \mathbf{u}_1 = 1$. In Eq. (1.20), we also derived that $\sigma(\mathbf{z}_1) = \sigma(\mathbf{X}\mathbf{u}_1) = \mathbf{u}_1^\top \mathbf{\Sigma} \mathbf{u}_1$, so we can prove the following.

COROLLARY 1.4 (1st PC loading)**.** *The first PC loading $\boldsymbol{u}_1$ is the eigenvector of the covariance matrix $\mathbf{\Sigma}$ corresponding to its highest eigenvalue λ_1.*

Proof. The first PC maximises $f(\mathbf{u}_1) = \sigma_{\mathbf{X}\mathbf{u}_1} = \mathbf{u}_1^\top \mathbf{\Sigma} \mathbf{u}_1$ subject to $g(\mathbf{u}_1) := \mathbf{u}_1^\top \mathbf{u}_1 - 1 = 0$ so our constrained optimisation problem is:

$$\max(\mathbf{u}_1^\top \mathbf{\Sigma} \mathbf{u}_1) \qquad \text{given } \mathbf{u}_1^\top \mathbf{u}_1 = 1 \tag{1}$$

The $\mathbf{u}_1$ that satisfies Eq. (1) can be found by introducing a Lagrange multiplier λ such the Lagrangian cost function $\mathscr{L}(\mathbf{u}_1, \lambda) = f(\mathbf{u}_1) - \lambda g(\mathbf{u}_1)$ is maximised:

$$\begin{gathered} \max(\mathscr{L}(\mathbf{u}_1, \lambda)) = \max\left(\mathbf{u}_1^\top \mathbf{\Sigma} \mathbf{u}_1 - \lambda(\mathbf{u}_1^\top \mathbf{u}_1 - 1)\right) \Rightarrow \\ \frac{\partial f(\mathbf{u}_1, \lambda)}{\partial \lambda} = 0, \quad \frac{\partial f(\mathbf{u}_1, \lambda)}{\partial \mathbf{u}_1} = 0 \Rightarrow \\ \mathbf{u}_1^\top \mathbf{u}_1 - 1 = 0 \qquad (2) \\ \mathbf{\Sigma}\mathbf{u}_1 - \lambda \mathbf{u}_1 = (\Sigma - \lambda \mathbf{I}_D)\mathbf{u}_1 = 0 \qquad (3) \end{gathered}$$

Pre-multiplying the last equation by $\mathbf{u}_1^\top$ and using Lem. 1.5 yields:

$$\begin{gathered} \mathbf{u}_1^\top \mathbf{\Sigma} \mathbf{u}_1 = \lambda \mathbf{u}_1^\top \mathbf{u}_1 \Rightarrow \\ \lambda = \sigma^2(\mathbf{z}_1) \qquad (4) \end{gathered}$$

Eq. (2) and Eq. (3) tell us that the desired vector $\mathbf{u}_1$ is an eigenvector of the covariance matrix Σ and λ is its first and largest eigenvalue. □

1.7 Deriving the other PCs

Before deriving the second PC loading, recall from Cor. 1.4 that $\mathbf{\Sigma}\mathbf{u}_1 = \lambda \mathbf{u}_1$. Next, we explore what zero correlation between two PC scores $\mathbf{z}_i$, $\mathbf{z}_j$ imposes on the geometry of loadings $\mathbf{u}_i$, $\mathbf{u}_j$.

COROLLARY 1.5 (orthonormal loading vectors)**.** *The loading vectors $\boldsymbol{u}_1,\ldots,\boldsymbol{u}_D$ form an orthonormal basis.*

Proof. We have already imposed that $\|\mathbf{u}_j\| = 1$. Because of the eigenvector corollary, $\boldsymbol{\Sigma}\mathbf{u}_1 = \lambda_1\mathbf{u}_1$. Also, one of the properties is that $\sigma(\mathbf{z}_i,\mathbf{z}_j) = 0$. Therefore, using Eq. (1.21):

$$\sigma(\mathbf{z}_1,\mathbf{z}_2) = \mathbf{u}_1^\top\boldsymbol{\Sigma}\mathbf{u}_2 = \lambda_1\mathbf{u}_1^\top\mathbf{u}_2 = 0 \Rightarrow$$

$$\mathbf{u}_1^\top\mathbf{u}_2 = 0$$

We have shown the result for $\mathbf{u}_2$ given $\mathbf{u}_1$ but it holds for any $\mathbf{u}_j$ given $\mathbf{u}_i$, $\quad i < j$. □

To derive the j-th PC loading $\mathbf{u}_j$, we use the same variance maximisation property, however we add the fact that the current PC score it's uncorrelated to all previous PCs, i.e. $\sigma(\mathbf{z}_i,\mathbf{z}_j) = \sigma(\mathbf{X}\mathbf{u}_i,\mathbf{X}\mathbf{u}_j) = 0 \quad \forall 1 < j$. Recall from Eq. (1.21) that $\sigma(\mathbf{X}\mathbf{u}_i,\mathbf{X}\mathbf{u}_j) = \mathbf{u}_i^\top\boldsymbol{\Sigma}\mathbf{u}_j$ and that covariance needs to be zero. We will show how $\mathbf{u}_2$ is computed given $\mathbf{u}_1$.

$$\max(\mathbf{u}_2^\top\boldsymbol{\Sigma}\mathbf{u}_2)$$

$$\text{given } \mathbf{u}_2^\top\mathbf{u}_2 = 1$$

$$\text{and } \mathbf{u}_1^\top\boldsymbol{\Sigma}\mathbf{u}_2 = 0$$

However, we deduced that $\mathbf{u}_1^\top\boldsymbol{\Sigma}\mathbf{u}_2 = 0$ implies $\mathbf{u}_1^\top\mathbf{u}_2 = 0$. Therefore if λ,μ the Lagrange multipliers, the Lagrange multiplier we want to maximise it:

$$\mathscr{L}(\mathbf{u}_1,\mathbf{u}_2) = \mathbf{u}_2^\top\boldsymbol{\Sigma}\mathbf{u}_2 - \lambda\mathbf{u}_2^\top\mathbf{u}_2 - \mu\mathbf{u}_2^\top\mathbf{u}_1 = 0$$

Differentiate w.r.t. $\mathbf{u}_2$ and set the derivative to zero:

$$\frac{\partial\mathscr{L}}{\partial\mathbf{u}_2} = 0 \Rightarrow \boldsymbol{\Sigma}\mathbf{u}_2 - \lambda\mathbf{u}_2 - \mu\mathbf{u}_1 = 0 \tag{4}$$

Pre-multiply by $\mathbf{u}_1$

$$\mathbf{u}_1^\top\boldsymbol{\Sigma}\mathbf{u}_2 - \lambda\mathbf{u}_1^\top\mathbf{u}_2 - \mu\mathbf{u}_1^\top\mathbf{u}_1 = 0 \Rightarrow \mu = 0$$

,since $\sigma(\mathbf{z}_1,\mathbf{z}_2) = \mathbf{u}_1^\top\boldsymbol{\Sigma}\mathbf{u}_2 = 0$ (Eq. (1.21) and property 3), $\mathbf{u}_1^\top\mathbf{u}_2 = 0$ (Cor. 1.5) and $\mathbf{u}_1^\top\mathbf{u}_1 = 1$. Plugging in $\mu = 0$ back in Eq. (4), we find exactly that λ is an eigenvalue λ_2 of $\boldsymbol{\Sigma}$ corresponding to eigenvector $\mathbf{u}_2$:

$$\boldsymbol{\Sigma}\mathbf{u}_2 = \lambda_2\mathbf{u}_2 \Rightarrow$$

$$\mathbf{u}_2^\top\boldsymbol{\Sigma}\mathbf{u}_2 = \lambda_2\mathbf{u}_2^\top\mathbf{u}_2 \Rightarrow$$

$$\lambda_2 = \sigma^2(\mathbf{z}_2)$$

Recall also that $\lambda \neq \lambda_1$ and $\lambda_1 \geq \lambda_2$ due to property 2 of PCs. Similarly we derive:

$$\boldsymbol{\Sigma}\mathbf{u}_j = \lambda_j\mathbf{u}_j, \quad \lambda_j = \sigma^2(\mathbf{z}_j)$$

$$\boldsymbol{\Sigma}\mathbf{u}_{j+1} = \lambda_{j+1}\mathbf{u}_{j+1}, \quad \lambda_{j+1} = \sigma^2(\mathbf{z}_{j+1})$$

$$\ldots$$

$$\boldsymbol{\Sigma}\mathbf{u}_D = \lambda_D\mathbf{u}_D, \quad \lambda_D = \sigma^2(\mathbf{z}_D)$$

The eigenvectors $\mathbf{u}_j$ corresponding to each eigenvalue λ_j are arranged in descending order, i.e. $\lambda_j \geq \lambda_{j+1}$ since $\lambda_j = \sigma^2(\mathbf{z}_j)$ and due to property 2. We have arrived at the following conclusion.

LEMMA 1.6 (how each PC is computed given the data matrix)**.** *Given a data matrix $\boldsymbol{X}_{n\times D}$, where n is the number of observations and D the features (dimensions) of each one, then each PC loading (direction of the projection data) is the j-th eigenvalue of the covariance matrix $\boldsymbol{\Sigma}$, i.e.*

$$\boldsymbol{\Sigma}\boldsymbol{u}_j = \lambda_j\boldsymbol{u}_j \tag{1.22}$$

THe j-th eigenvalue is the variance of the data projected on loading $\boldsymbol{u}_j$, i.e. the variance of vector $\boldsymbol{z}_j$:

$$\lambda_j = \sigma^2(\boldsymbol{z}_j) = \sigma^2(\boldsymbol{X}\boldsymbol{u}_j) \tag{1.23}$$

The variance expresses how much information each direction $\mathbf{u}_j$ "holds", i.e. its significance. So when we perform PCA we can only keep the first d components based on their total relative significance $\sum_{i=1}^{d}\lambda_i / \sum_{i=1}^{D}\lambda_i$.

1.7.1 The relative significance of the first d components

When we want to project the original D dimensional data to a d-dimensional space, $d \leq D$, then due to Eq. (1.23) the part of variance explained by the first d PCs is:

$$R = \frac{\sum\limits_{i=1}^{d}\sigma^2(\mathbf{z}_i)}{\sum\limits_{i=1}^{D}\sigma^2(\mathbf{z}_i)} = \frac{\sum\limits_{i=1}^{d}\lambda_i}{\sum\limits_{i=1}^{D}\lambda_i} \tag{1.24}$$

, where λ_i, $\lambda_1 \geq \lambda_2 \geq \ldots \geq \lambda_D$ are the eigenvalues of the covariance matrix $\boldsymbol{\Sigma}$ after we centred our data, i.e. the mean of each feature (column) of $\mathbf{X}$ is zero.

There are two common rules of thumb to select d.

1. Require the variance explained by the first d components to be equal at least to a percentage r. Then we simply choose the minimum integer d such that:

$$d = \arg\min\left(100\sum_{i=1}^{d}\lambda_i / \sum_{i=1}^{D}\lambda_i\right) \geq r, \quad 0 < r < 100$$

2. Find the "elbow" of the `scree plot`. The scree plot is simply the (discrete) plot of the value of each eigenvalue $\lambda_1, \ldots, \lambda_D$ on the y-axis against their index i on the x-axis. The elbow is the index of what it looks like the inflexion point of that plot, i.e. index before the plot starts to flatten. The contribution of the eigenvalues beyond the elbow is considered insignificant. An example is illustrated below.

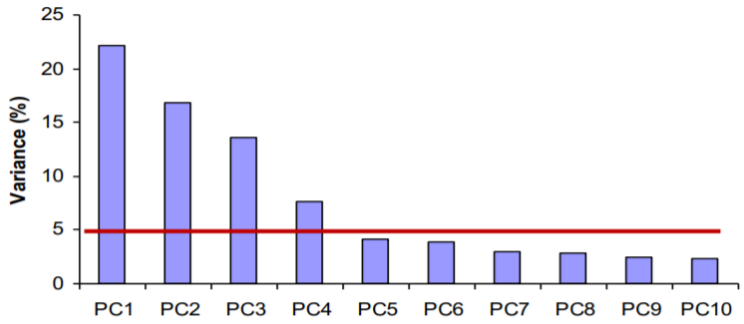


Fig. 7. Scree plot of some hypothetical 10-dimensional data. In this case, we could choose $i = 4$ as the elbow, ignoring the contributions of $PC_5, \ldots, PC_{10}$.

1.7.2 Reconstruction of the original data

Sometimes it's useful to reconstruct the original non-centred data $\mathbf{X}$ given the PC scores $\mathbf{z}_1, \ldots, \mathbf{z}_D$. To see how, re-write Eq. (1.15) by setting $\mathbf{X}_c$ as $\mathbf{X}$, to make it clear that is denoted the centred data. Also set $\mathbf{Z} := \begin{bmatrix}\mathbf{z}_1 & \ldots & \mathbf{z}_D\end{bmatrix}$, $\mathbf{U} := \begin{bmatrix}\mathbf{u}_1 & \ldots & \mathbf{u}_D\end{bmatrix}$. As a reminder, $\mathbf{U}$ stores the eigenvectors of $\boldsymbol{\Sigma}$ in descending order. Then:

$$\mathbf{Z} = \mathbf{X}_c\mathbf{U} \Rightarrow$$

$$\mathbf{X}_c = \mathbf{Z}\mathbf{U}^\top$$

We obtain the approximation $\hat{\mathbf{X}}$ of the original data by adding its mean to each column of $\mathbf{X}_c$:

$$\hat{\mathbf{X}} = \mathbf{X}_c + \bar{\mathbf{X}} = \mathbf{Z}\mathbf{U}^\top + \bar{\mathbf{X}}, \quad \bar{\mathbf{X}} = \begin{bmatrix} | & | & | \\ \bar{\mathbf{x}_1} & \ldots & \bar{\mathbf{x}_D} \\ | & | & | \end{bmatrix} \tag{1.25}$$

If we use all D basis vectors, like above, the error $\left\|\hat{\mathbf{X}}-\mathbf{X}\right\|^2$ should be very close to zero (practically it's not exactly zero due to floating point). If we perform a lossy reconstruction, i.e. use $d < D$ componenents, then we can discard the $D-d$ last columns of $\mathbf{U}$, and the last $D-d$ columns of $\mathbf{Z}$ (but we don't discard columns of $\mathbf{X}_c$, notice the product size is still $n \times D$):

$$\hat{\mathbf{X}} = \mathbf{X}_c + \begin{bmatrix}\mathbf{z}_1 & \dots & \mathbf{z}_d\end{bmatrix}\begin{bmatrix}\mathbf{u}_1 & \dots & \mathbf{u}_d\end{bmatrix}^\top \tag{1.26}$$

As we saw in the next section, the information retention of the first d components is $\sum_{i=1}^{d}\lambda_i / \sum_{i=1}^{D}\lambda_i$. The larger the better but at the same time the more components, the larger the dimensionality of the projected data, which is a undesired. The basics of PCA have been presented and in the next section there is a summary of the algorithm.

1.8 Data pre-processing – feature whitening

The essential steps of PCA have been presented, but in real world one more pre-processing step, called whitening is often required. In practice, data contained in the rows of $\mathbf{X}$ might have different scales by one or more orders of magnitude or may be highly correlated. The aim of what's called `whitening` is

1. to make the features are less correlated with each other
2. and to make all features have the same variance.

Whitening is recommended when features have different orders of magnitude.

Aim 1 is addressed by the covariance of two scores being zero so it's already achieved in PCA. The problem when we have a data matrix where one feature $\mathbf{x}_1 \approx 10^3\mathbf{x}_2$, then for the the eigenvalue of $\mathbf{\Sigma}$ we'd have $\lambda_1 >> \lambda_2$, so feature $\mathbf{x}_1$ when projected to loading 1, $\mathbf{x}_1^\top\mathbf{u}_1$, will capture most of the significance just because its values are large. Whitening alleviates this.

DEFINITION 1.8 (feature whitening). *Whitening a feature $\boldsymbol{x}_j$ sets its mean to zero and its variance to unit and is expressed by:*

$$\boldsymbol{x}_{j,white} = \frac{\boldsymbol{x}_j - \bar{\boldsymbol{x}_j}}{\sigma_{\boldsymbol{x}_j}}, \quad \bar{\boldsymbol{x}_j} = \boldsymbol{1}_n \sum_{i=1}^{n}\frac{\boldsymbol{X}_{ij}}{n} \tag{1.27}$$

In our case, we assume that $\bar{\mathbf{x}_j}$ since we always centre the data as soon as we obtain it. So then $\sigma^2_{\mathbf{x}_{j,white}} = \sum^{n} x_{ij}^2/\sigma_{\mathbf{x}_j} = \sigma^2_{\mathbf{x}_j}/\sigma^2_{\mathbf{x}_j} = 1$.

1.9 The PCA algorithm

We summarise the steps to obtain the PCA of a data matrix $\mathbf{X}$, assuming $\mathbf{X}$ stores the observations are rows.

1. Centre the features of the data matrix using Eq. (1.8) to obtain matrix $\mathbf{X}_c$. Optionally, if the features have different orders of magnitiude, whiten them using Eq. (1.27).
2. Compute the covariance matrix $\mathbf{\Sigma} = \mathbf{X}_c^\top\mathbf{X}_c$.
3. Find the first d eigenvectors of $\mathbf{\Sigma}$ with largest eigenvalues $\mathbf{u}_1,\dots,\mathbf{u}_d$; we called those "loadings".
4. Find the projections $\mathbf{z}_1 = \mathbf{X}_C\mathbf{u}_1$, $\mathbf{z}_2 = \mathbf{X}_C\mathbf{u}_2$, …, $\mathbf{z}_d = \mathbf{X}_C\mathbf{u}_d$. These are called "scores" and they approximate the centred set, but spanned by the basis $\mathbf{u}_1,\dots,\mathbf{u}_d$. PCA returns the loads, scores, and the corresponding eigenvalues of the scores.

Let's verify the algorithm and the formulas for the PCs in Matlab. We'll be using Matlab's (not available in Octave) `pca` function to get the correct results and the eigendata (`eigs`) to follow the algorithm. For reference, the return of these two function is the following:

```
% X is n by D, i.e. observations are stacked as rows
[loads, pcs, pca_eval] = pca(X)
% V is the matrix with eigenvectors as columns,
% D diagonal with eigenvalues in descending order
[V,D] = eigs(A)
```

1. (*Create some data*) We'll be using some synthesised data whose each ovservation contains a weight, height, and IQ (w,h,iq). We'll create the data such that the weight is positively correlated with height and IQ is independent of both. The data consist of 1000 observations.

```
h = random('norm', 177, 5, 1000, 1);
w = h/2 - 15 + 10*random('uniform', 0, 1, 1000, 1);
iq = random('norm', 105, 10, 1000, 1);
X = [w, h, iq];
» X
ans =
    80.8160  179.8088   87.1759
    83.8719  179.7486  104.0231
    84.5937  188.9917  106.7827
    79.9716  179.2866  105.9784
    83.1234  180.2754  121.5708
    78.5173  173.9326  106.1138
    78.0839  176.9065  121.0178
    <-- omitted -->
```

2. (*Calculate covariance matrix*) Centre the data using Eq. (1.8) and then calculate $\mathbf{\Sigma}$.

```
n = size(X,1);
C = eye(n) - 1/n*ones(n,1)*ones(n,1)';
Xc = C*X;
S = Xc'*Xc;
» S
    S =
    1.0e+04 *
    1.4384    1.1808    0.0451
    1.1808    2.2948   -0.1340
    0.0451   -0.1340    9.3945
```

3. (*Find the eigenvalues of covariance matrix*) Recall the `eigs` command and what it returns. Eigenvectors of $\mathbf{\Sigma}$ are our loading vectors $\mathbf{u}_1, \mathbf{u}_2, \mathbf{u}_3$. We can also verify that their dot products are zero.

```
[evec,eval] = eigs(S);
u1 = evec(:,1); u2 = evec(:,2); u3 = evec(:,3);
» u1' * u2
ans =
    -6.9389e-18
```

4. (*Compute and verify and PC scores*) Use Eq. (1.14). Compare against `pca`'s values.

```
z1 = Xc*u1; z2 = Xc*u2; z3 = Xc*u3;
[loads, pcs, pca_evals] = pca(Xc);
» loads
    loads =
    0.0029    0.5745    0.8185
   -0.0184    0.8184   -0.5744
    0.9998    0.0133   -0.0130
» evec
    evec =
    0.0029    0.5745   -0.8185
   -0.0184    0.8184    0.5744
    0.9998    0.0133    0.0130
% 3rd eigenvector is flipped but it doesn't matter as they specify direction
» pcs - [z1 z2 z3]
    ans =
    -0.3908   -0.0888    0.1776
    -0.4263   -0.1066    0.2132
    -0.3908   -0.1243    0.1732
    -0.4041   -0.1599    0.2021
    -0.4241   -0.0910    0.1954
    <-- omitted -->
```

The difference between the PCs scores from scratch (`z1, z2, z3`) and Matlab's scores is small but

not exactly zero, possibly due to implementation discrepancies (?).

5. (*Reconstruct the original data*) Use Eq. (1.26). First we'll reconstruct the original using the full basis $\{\mathbf{u}_1, \mathbf{u}_2, \mathbf{u}_3\}$ (matrices $\mathbf{U}_3, \mathbf{Z}_3$) and then using $\{\mathbf{u}_1, \mathbf{u}_2\}$ (matrices $\mathbf{U}_2, \mathbf{Z}_2$).

```
U3 = [u1 u2 u3]; U2 = [u1 u2];
Z3 = [z1 z2 z3]; Z2 = [z1 z2];
X_recon_3 = Z3*U3' +...
    [mean(X(:,1))*ones(n,1), mean(X(:,2))*ones(n,1), mean(X(:,3))*ones(n,1)];
X_recon_2 = Z2*U2' +...
    [mean(X(:,1))*ones(n,1), mean(X(:,2))*ones(n,1), mean(X(:,3))*ones(n,1)];
» X_recon_3 - X
    ans =
    0.0853    0.2842    0.0995
    0.0995    0.2558    0.1421
    0.0995    0.2558    0.1137
    0.0853    0.2558    0.1137
    0.0853    0.2558    0.1137
    <-- omitted -->
» norm(X_recon_3 - X, 2)
    ans =
    8.6679e-12
» norm(X_recon_2 - X, 2)
    ans =
    78.0468
```

From the last answer, we can see that $\mathbf{U}_3$ reconstructs better the original data than $\mathbf{U}_2$. The three figures below visualise the loadings $\mathbf{u}_1, \mathbf{u}_2, \mathbf{u}_3$ and the centred data cloud. Notice how each loading captures the maximum variation of the data and how they are mutually orthogonal. Loadings have been exaggerated by 20 times so that they are visible.

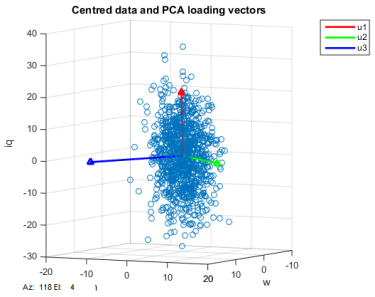


Fig. 8. The centred data and loadings in 3D.

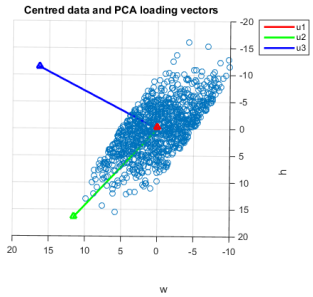


Fig. 9. The data and loadings in the x_1, x_2 plane.

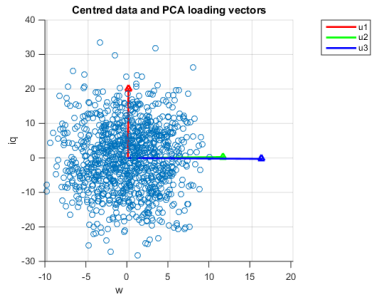


Fig. 10. The data and loadings in the x_1, x_3 plane.

The above steps are generalised in the Matlab function `my_pca`, found in A.1.

1.9.1 PCA via SVD

SVD (Singular value decomposition) is a method of decomposing a any matrix $\mathbf{X}_{n\times d}$ three matrices $\mathbf{U}, \mathbf{D}, \mathbf{V}$. It's very powerful and used in numerous applications. SVD is available in many libraries and frameworks and can help quickly compute the PCA essentially out of the box.

THEOREM 1.1 (SVD). *Any matrix $\boldsymbol{X}_{n\times d}$ can be written as a product of two orthogonal matrices $\boldsymbol{U}, \boldsymbol{V}$ and a diagonal one $\boldsymbol{D}$.*

$$\boldsymbol{X} = \boldsymbol{U}\boldsymbol{D}\boldsymbol{V}^\top \tag{1.28}$$

- $\boldsymbol{V}_{n\times D}$ *contains the eigenvectors of $\boldsymbol{X}'\boldsymbol{X}$ as columns. They are arranged from the one that corresponds to the one that corresponds to the largest eigenvalue to one that corresponds to the smallest.*
- $\boldsymbol{D}$ *A diagonal matrix that contains the singular values of $\boldsymbol{X}$ in descending order, i.e. the square*

roots of the eigenvalues of $\boldsymbol{X}^\top\boldsymbol{X}$*:*

$$\boldsymbol{D} = \begin{bmatrix} \sqrt{\lambda_1(\boldsymbol{X}^\top\boldsymbol{X})} & & \\ & \ddots & \\ & & \sqrt{\lambda_D(\boldsymbol{X}^\top\boldsymbol{X})} \end{bmatrix}, \quad \lambda_1(\boldsymbol{X}^\top\boldsymbol{X}) \geq \ldots \geq \lambda_D(\boldsymbol{X}^\top\boldsymbol{X}) \tag{1.29}$$

- $\boldsymbol{U}_{n\times D}$*:*

$$\boldsymbol{U} = \begin{bmatrix} | & | & | \\ \frac{\boldsymbol{X}\boldsymbol{u}_1}{\|\boldsymbol{X}\boldsymbol{u}_1\|} & \cdots & \frac{\boldsymbol{X}\boldsymbol{u}_D}{\|\boldsymbol{X}\boldsymbol{u}_D\|} \\ | & | & | \end{bmatrix} \tag{1.30}$$

Again, the assumption is that $\mathbf{X}$ has its columns centred. Then $\mathbf{X}^\top\mathbf{X}$ is the covariance matrix, and SVD computes its eigenvalues in $\mathbf{V}$, so the "loads" of PCA. Next, we simply compute the "scores" from Eq. (1.15) as:

$$\mathbf{Z} = \mathbf{X}\mathbf{V} \tag{1.31}$$

Alternatively, notice the $\mathbf{U}$ matrix already contains the scores as unit vectors so each score $\mathbf{z}_i$ can be obtained as:

$$\mathbf{z}_i = \mathbf{u}_i \|\mathbf{X}\mathbf{v}_i\| \tag{1.32}$$

Finally, we can keep the first d scores (columns) of $\mathbf{U}$ to retain the most useful information.

1.10 Interpreting and visualising PCA results

As defined before, each loading has the same size as an observation, i.e. $\mathbf{u} \in \mathbb{R}^d$. For instance, after performing PCA on a 3-feature dataset, we could have:

$$\mathbf{u}_1 = \begin{bmatrix} 0.36 \\ 0.91 \\ 0.18 \end{bmatrix} = 0.36 \begin{bmatrix} 1 \\ 0 \\ 0 \end{bmatrix} + 0.91 \begin{bmatrix} 0 \\ 1 \\ 0 \end{bmatrix} + 0.18 \begin{bmatrix} 0 \\ 0 \\ 1 \end{bmatrix} = 0.36\mathbf{e}_1 + 0.91\mathbf{e}_2 + 0.18\mathbf{e}_3,$$

$$\mathbf{u}_2 = \begin{bmatrix} -0.91 \\ 0.36 \\ 0 \end{bmatrix}$$

Therefore if our dataset consisted of e.g. weight, height, IQ measurements, the above equation would imply that feature 2 (height) carries the most information (variance)/ When we looked at an observation, its height would be the most "interesting" feature as it would carry most of the information, followed by its weight, followed by its IQ. $\mathbf{u}_2$ mostly conveys the weight.

Keep 2 PCs → plot in 2D.

Let's say that we want to approximate all measurements given the two first loadings $\mathbf{u}_1, \mathbf{u}_2$. Each score can be written as $\mathbf{z}_i = w_1\mathbf{u}_1 + w_2\mathbf{u}_2$, where w_1, w_2 some weights. If we were to plot $\mathbf{z}_1$ in 2D (since it's approximated by 2 loadings), we would plot point (w_1, w_2) in Cartesian coordinates.

1.11 Application 1. Analysing wine data.

The wine dataset by UCI is used, which contains 178 wine measurements, each one with 13 attributes, such as alcohol percentage, hue, etc. The dataset is found in file `wine_data.csv` as is adapted from the one at: https://github.com/anishagartia/PCA-of-Wine-Dataset to contain a header with the feature names. The original file contains also the class in the first column but that column has been removed. The exact same dataset can be loaded as:

```
from sklearn.datasets import load_wine
# ...
X = load_wine()
```

in Python, but we want to file to be passed as an argument instead.

A PCA wrapper which includes all the mandatory steps, as well as the optional (whitening) has been written at `wines/my_pca.py`. To run it, execute file `wines/run_my_pca.py` with the following arguments:

```
python run_my_pca.py dataset_csv n_pc_components_to_keep 0|1
```

, where 1 plots the first 2 PCs in 2D and 0 doesn't and `n_pc_components_to_keep` is an integer. The source is found in A.2. Running it, we obtain the following results. Note the 2D visualisation is very similar to the one in `sklearn`'s documentation page [2]. It's not exactly the same as the documentation splits the dataset in testing and training data and follows a slightly different pipeline.

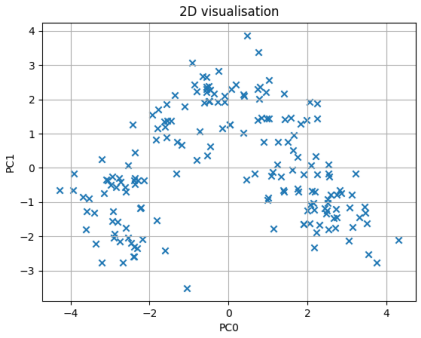


Fig. 11. The scores of the first 2 PCs for all wines. PC0 mainly depends on, flavanoids (see below) and for PC1 on colour intensity.

Regarding the most important features, the following are printed if we keep the first 3:

```
The most important attributes are:
Flavanoids,Color_intensity,Ash
```

1.12 Application 2. Eigenfaces.

Eigenfaces is a method that uses PCA to recognise faces. It is essentially PCA with *pixels as features*. It employs a face database, which stores various face images of various subjects (e.g. 8 images of each person). When a novel face is detected, it finds its closest match in the database and outputs the subject number. If the novel face does not belong to any of the subjects, it is added to the database.

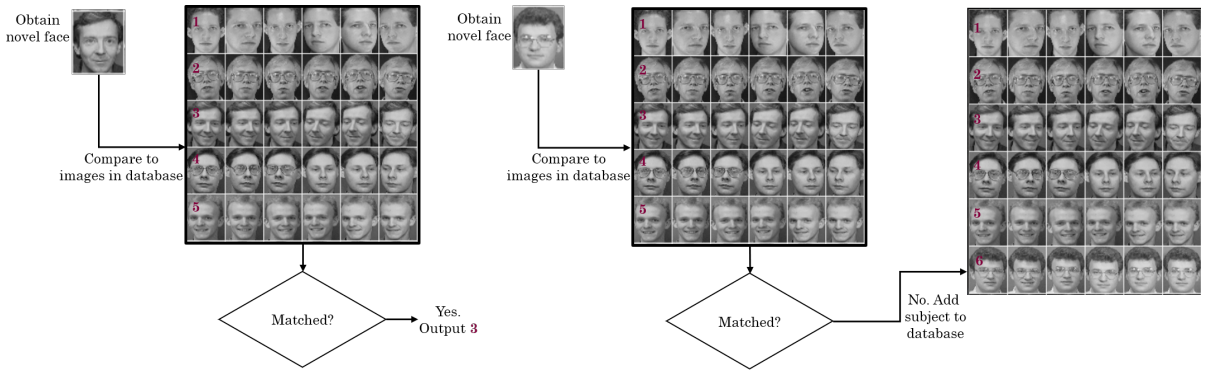


Fig. 12. Face recognition takes a new face as input and finds its best match in the database. Each row corresponds to a subject's stored faces, labelled with an integer.

The main question to answer in order to figure out how Eigenfaces work is "*how is matching accomplished?*". Before matching is performed, it requires that each face in the database and the novel face (the new face, the one we try to match) are all represented by their coordinates in a new orthonormal subspace instead of as pixel (training phase). Next, by comparing the distance between the coordinates of a novel face with those of the stored faces, matching is accomplished (prediction phase). Below are the steps for the training phase.

1. (*Compile the database*) Gather M face images. Each image must be labelled with the subject's number. Convert them to greyscale and resize them all to the same shape, e.g. $N \times N$. Flatten each image

[2] https://scikit-learn.org/stable/auto_examples/preprocessing/plot_scaling_importance.html#sphx-glr-auto-examples-preprocessing-plot-scaling-importance-py

row by row and store it in a row vector $\mathbf{X}_i \in \mathbb{R}^{N\times N}$. Stack those vectors horizontally to form the data matrix $\mathbf{X}$ of size $M \times N^2$:

$$\mathbf{X} = \begin{bmatrix} - & \mathbf{X}_1 & - \\ - & \mathbf{X}_2 & - \\ & \dots & \\ - & \mathbf{X}_M & - \end{bmatrix}, \quad \mathbf{X}_i \in \mathbb{R}^{1\times N^2} \tag{1.33}$$

2. (*Centre the features by subtracting the mean image*) PCA is applied on data with zero-mean features. The features of the dataset are in this case all pixels at positions $1, 2 \dots, N^2$ (columns of matrix $\mathbf{X}$) over all M faces. Therefore each feature (pixel) $\mathbf{X}_i \in \mathbb{R}^M$ will have its mean $\mathbf{u}_i$ subtracted from it (mean over all M images). The centred matrix $\mathbf{X}_c$ is defined as:

$$\mathbf{X}_c = \begin{bmatrix} | & | & & | \\ \mathbf{X}_1 - \boldsymbol{\mu}_1 & \mathbf{X}_2 - \boldsymbol{\mu}_2 & \dots & \mathbf{X}_{N^2} - \boldsymbol{\mu}_{N^2} \\ | & | & & | \end{bmatrix} = \mathbf{X} - \mathbf{1}_{M\times 1}\boldsymbol{\mu}^\top, \quad \boldsymbol{\mu} = \frac{1}{M}\mathbf{X}^\top \mathbf{1}_{M\times 1}, \quad \boldsymbol{\mu} \in \mathbb{R}^{N^2}$$

From Eq. (1.8), we know that we can rewrite the above equation to obtain the centred data matrix X_c through a single matrix multiplication as:

$$\mathbf{X}_c = \mathbf{C} \cdot \mathbf{X}, \quad \mathbf{C} = \mathbf{I}_M - \frac{1}{M}\mathbf{1}_{M\times 1}\mathbf{1}_{M\times 1}^\top \tag{1.34}$$

, where $\mathbf{1}_{M\times 1}$ contains only ones. Visually, subtracting the mean matrix gets rid of all the genetic features of each face and keeps the prominent ones (nose, lips, etc.).

Fig. 13. First sample $\mathbf{X}_1$ of the Olivetti faces dataset.

Fig. 14. The mean image $\boldsymbol{\mu}$ of Oliveti dataset images.

Fig. 15. The absolute value of the same sample after the mean is subtracted from it $|\mathbf{X}_1 - \boldsymbol{\mu}|$.

3. (*Compute the eigenvectors of the covariance matrix of $\mathbf{X}_c$, $\boldsymbol{\Sigma} = \mathbf{X}_c^\top \mathbf{X}_c$*) Recall that matrix $\mathbf{X}_c$ is of size $M \times N^2$ so $\boldsymbol{\Sigma} = \mathbf{X}_c^\top \mathbf{X}_c$ is of size $N^2 \times N^2$. If the original images were of size 64×64, then $\boldsymbol{\Sigma} \in \mathbb{R}^{3844\times 3844}$, which is too huge to perform eigen-analysis!

   From Lemma 1.3, recall that if we know the eigen-data $\mu_i, \mathbf{v}_i$ of $\mathbf{X}_c\mathbf{X}_c^\top$, then the eigen-data $\lambda_i, \mathbf{u}_i$ of $\mathbf{X}_c^\top\mathbf{X}$ are given by:

$$\lambda_i = \mathbf{u}_i, \qquad \mathbf{u}_i = \mathbf{X}_c^\top \mathbf{v}_i \tag{1.35}$$

   The eigenvectors of $\boldsymbol{\Sigma} = \mathbf{X}_c^\top \mathbf{X}_c$ are called `eigenfaces` and the space the K first eigenvectors span, i.e. $\{\mathbf{u}_1, \dots, \mathbf{u}_K\}$ is called `face space`.

4. (*Compute the eigenface space representation of each image*) Now since $\mathbf{u}_i$ form an orthonormal space, each face $\mathbf{X}_i$ can be represented as their linear combination, or $\mathbf{X}_i = w_{i1}\mathbf{u}_1 + \dots + w_{iM}\mathbf{u}_M = \mathbf{w}_i^\top \begin{bmatrix} \mathbf{u}_1 & \mathbf{u}_2 & \dots & \mathbf{u}_M \end{bmatrix}$. And since we want to keep only the first K eigenvectors[3], we can approximate $\mathbf{X}_i$ as the linear combination of $w_{i1}\mathbf{u}_1 + \dots + w_{iK}\mathbf{u}_K$. Then each j-th coefficient that describes face i, i.e. w_{ij}, is computed by the inner product $w_{ij} = \mathbf{X}_i^\top \mathbf{u}_j$. For instance, for face 1, $w_{11} = \mathbf{X}_1^\top \mathbf{u}_1$, $w_{12} = \mathbf{X}_1^\top \mathbf{u}_2, \dots w_{1K} = \mathbf{X}_1^\top \mathbf{u}_K$. Similarly, we can find the coefficients w_{2i} of face $\mathbf{X}_2$. More compactly, the coordinates of each face can be computed as the matrix rows of $\mathbf{W}$:

$$\mathbf{W}_i = \mathbf{X}_i \cdot \mathbf{U}, \quad \mathbf{U} = \begin{bmatrix} \mathbf{u}_1 & \dots & \mathbf{u}_K \end{bmatrix} \tag{1.36}$$

Coefficients w_{ij}, $j = 1, \dots, K$ are therefore the coordinates or the "fingerprint" of face i in the space spanned by $\{\mathbf{u}_1, \dots, \mathbf{u}_K\}$. When we obtain an "unseen" face, we simply find the closest match of those against each face in the database to do the matching as described below.

[3] The number K can be determined by comparing the relative significance of the first K eigenvectors to a threshold, e.g. 0.9. The relative significance was derived in section 1.7.1.

In the end, each face image $\mathbf{X}_i$ (whether it's stored in the database or novel) can be described by the vector $\mathbf{W}_i$:

$$\mathbf{X}_i \to \begin{bmatrix} w_{i1} \\ w_{i2} \\ \dots \\ w_{iK} \end{bmatrix} = \begin{bmatrix} \mathbf{X}_i^\top \mathbf{u}_1 \\ \mathbf{X}_i^\top \mathbf{u}_2 \\ \dots \\ \mathbf{X}_i^\top \mathbf{u}_K \end{bmatrix}$$

Note also that each $\mathbf{u}_i$ (eigenface) vector is of size $N^2 \times N^2$.

1. (*Compute face coordinates*) For each face $\mathbf{X}_i$, compute its "face space" coordinates, stored in the rows of $\mathbf{W}$ matrix:

$$\mathbf{W}_i = \mathbf{X}_i \cdot \begin{bmatrix} \mathbf{u}_1 & \dots & \mathbf{u}_K \end{bmatrix} \tag{1.37}$$

   Compute also the face space coordinates $\mathbf{W}$ of the novel face $\mathbf{X}$:

$$\mathbf{W} = \mathbf{X} \cdot \begin{bmatrix} \mathbf{u}_1 & \dots & \mathbf{u}_K \end{bmatrix} \tag{1.38}$$

2. (*Compute distances and output match*) Measure the distance[4] of the novel face representation with the representation of each face in the database:

$$d_i = \|\mathbf{W} - \mathbf{W}_i\|, \quad \mathbf{W}, \mathbf{W}_i \in \mathbb{R}^K$$

   Find index j that minimises d_i; $j = \arg\min d_i$ and output it. That's the matched subject.

To summarise, the training phase and prediction phase are listed as pseudocode below.

Algorithm 1 Eigenfaces training phase

procedure TRAIN($\mathbf{x}_1, \mathbf{x}_2, \dots, \mathbf{x}_M$)
  $\mathbf{X} = \begin{bmatrix} \mathbf{X}_1 \\ \vdots \\ \mathbf{X}_M \end{bmatrix}$ ▷ Stack the inputs in a matrix
  $\mathbf{X} \leftarrow \left(\mathbf{I}_M - \frac{1}{M}\mathbf{1}_{M\times 1}\mathbf{1}_{M\times 1}^\top\right)\mathbf{X}$ ▷ Centre each feature
  Compute $\mathbf{X}\mathbf{X}^\top$
  Compute the eigendata $\mu_i, \mathbf{v}_i$ of $\mathbf{X}\mathbf{X}^\top$
  Compute the eigenvectors of $\mathbf{X}^\top\mathbf{X}$ as $\mathbf{u}_i = \mathbf{X}^\top\mathbf{v}_i$ ▷ a.k.a. the eigenfaces
  Compute the eigenface representation of every stored face; $\mathbf{W} = \mathbf{X}\begin{bmatrix} \mathbf{u}_1 & \dots & \mathbf{u}_M \end{bmatrix}$
  Keep only the first K columns of $\mathbf{W}$ ▷ To compress each representation

Algorithm 2 Eigenfaces prediction phase

procedure PREDICT($\mathbf{x}_{new}$)
  $\mathbf{W} \leftarrow \mathbf{X}_{new}^\top \begin{bmatrix} \mathbf{u}_1 & \dots & \mathbf{u}_K \end{bmatrix}$ ▷ $\mathbf{u}_i$'s computed from training phase
  return $\arg\min_{i=1,\dots,M} \left(\text{dis}(\mathbf{W}, \mathbf{W}_i)\right)$

, where $dis(\mathbf{W}, \mathbf{W}_i)$ is the distance between $\mathbf{W}, \mathbf{W}_i$. For example, it can be the norm-2; $dis(\mathbf{W}, \mathbf{W}_i) = \|\mathbf{W} - \mathbf{W}_i\|$. A better distance metric is the Mahalanobis distance, however it is not discussed in this tutorial.

1.12.1 Eigenfaces from scratch in Python

Eigenfaces were implemented in a repository called EZfaces. The core of the source is the `train` method, which implements Algorithm 1.

```
def train(self):
    """ Find the coordinates of each training image in the eigenface space """
    self._divide_dataset()
    # the matrix X to use for training
    X = np.array([v[0] for v in self.train_data.values()])
    # compute eig of MxN^2 matrix first instead of the N^2xN^2, N^2 >> M
    XXT = np.matmul(X, X.T)
```

[4] Euclidean distance is not the best metric to compute similarity for we use it for simplicity.

```
    eval_XXT, evec_XXT = np.linalg.eig(XXT)
    # sort eig data by decreasing eigvalue values
    idx_eval_XXT = eval_XXT.argsort()[::-1]
    eval_XXT = eval_XXT[idx_eval_XXT]
    evec_XXT = evec_XXT[idx_eval_XXT]
    # now compute eigs of covariance matrix (N^2xN^2)
    self.evec_XTX = np.matmul(X.T, evec_XXT)
    # coordinates of each face in "eigenface" subspace
    self.W = np.matmul(X, self.evec_XTX)
```

Here's what each line does:

- `X = np.array([v[0] for v in self.train_data.values()])` stores the centred training data in $\mathbf{X}$. Each image is stored as a row.
- `XXT = np.matmul(X, X.T)` and `eval_XXT, evec_XXT = np.linalg.eig(XXT)` computes the eigendata of $\mathbf{X}\mathbf{X}^\top$. Note that the latter is not the covariance matrix, it's an intermediate $M \times M$ matrix.
- `idx_eval_XXT = eval_XXT.argsort()[::-1]`, `eval_XXT = eval_XXT[idx_eval_XXT]`, `evec_XXT = evec_XXT[idx_eval_XXT]` sorts the eigenvalues in descending order and sorts the eigenvectors in the respective eigenvalue order.
- `self.evec_XTX = np.matmul(X.T, evec_XXT)` computes the eigenvectors of $\mathbf{X}^\top\mathbf{X}$ given those of $\mathbf{X}\mathbf{X}^\top$ according to Eq. (1.35).
- `self.W = np.matmul(X, self.evec_XTX)` computes the eigenface representation of image $\mathbf{X}$, i.e. its coordinates in the eigenface space, which is spanned by the columns of `self.evec_XTX`. For instance, if the first 9 values in the first row of $\mathbf{W}$ are `-262.24, 50.27, 78.78, -108.09, -31.16, -45.81, 4.19, -56.48, -33.50`, then it means that image $\mathbf{X}_1$ can be written as $-262.24\mathbf{u}_1 + 50.27\mathbf{u}_2 + 78.78\mathbf{u}_3 + \ldots$, where $\mathbf{U} = \begin{bmatrix}\mathbf{u}_1 & \ldots & \mathbf{u}_K\end{bmatrix}$ is the matrix of eigenvectors of $\mathbf{X}^\top\mathbf{X}$ (eigenfaces).

The figure below visualises the first 9 eigenfaces in scale $[0, 255]$ as 64×64 images and how $\mathbf{X}_1$ can be written as a linear combination of them.

Notice how the eigenfaces look like ghost image and not like actual human faces!

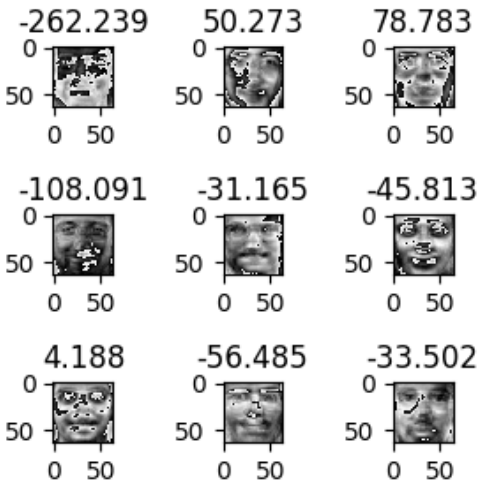


Fig. 16. Expressing $\mathbf{X}_1$ as the linear combination of the first 9 eigenfaces using `train` method of the EZfaces code.

This demonstrates how eigenfaces can be visualised. Full code of the main class of the EZfaces can be found in A.3.

A Appendices

A.1 Appendix Example

Listing 1: PCA simple implementation in Matlab/ Octave. (src/my_pca.m).

```
function [loads, scores, evals] = my_pca(X)
    n = size(X, 1);
    C = eye(n) - 1/n*ones(n,1)*ones(n,1)';
    X = C*X;
    S = X'*X;
    [loads, evals] = eigs(S);
    scores = [];
    for l=loads
        scores = [scores, X*l];
    end
end
```

A.2 Application 1 (wine analysis) source code.

Listing 2: PCA wrapper source code. (src/wines/my_pca.py).

```
from sklearn.decomposition import PCA
from sklearn.preprocessing import StandardScaler
import numpy as np
from matplotlib import pyplot as plt


##
# @brief Computers the PCA score, loadings, and eigenvalues
#
# @param X an n by D matrix, where n is the number of measurements
# @param n_compo how many of the most important scores to keep
# @param do_plot flag to plot the 2 first PCA scores in 2D
#
# @return scores, loadings, significance_ratios,
#         most_important_feature_indices
def pca_wrapper(X, n_comp = 2, do_plot = True):
    """
    A wrapper around sklearn's PCA.
    The notation follows the convention in my notes!
    """
    ### pre-processing
    # centre the data
    n = X.shape[0]
    C = np.eye(n) - 1/n*np.ones((n,1))
    X = C.dot(X)
    # whitening - variance to 1
    whiten = StandardScaler()
    X = whiten.fit_transform(X)
    ### execute PCA
    model = PCA(n_components = n_comp).fit(X)
    loads = model.components_
    scores = model.transform(X)
    sign_ratios = model.explained_variance_ratio_
    ### Post-processing and plotting
    # For each loading we keep, find its entry (feature)
    # with the highest abs value. That's the most important
    # feature of each loading. d is the #components
    d = model.components_.shape[0]
    ind_most_important = [np.abs(model.components_[i]).argmax()\
            for i in range(d)]
    if do_plot:
        plt.title('2D visualisation')
        plt.scatter(scores[:,0], scores[:,1], marker = 'x')
        plt.xlabel('PC0')
        plt.ylabel('PC1')
        plt.grid()
        plt.show()
    return scores, loads, sign_ratios, ind_most_important
```

Listing 3: PCA runner source code. (src/wines/run_my_pca.py).

```
#!/usr/bin/env python
from my_pca import pca_wrapper
import pandas as pd
import sys
import sklearn.datasets

if __name__ == '__main__':
    if len(sys.argv) == 1:
        raise SystemError("usage:\n\n."\
            "./<this_script> data_csv_file "\
```

```
            "number_of_pca_components 0|1 0|1\n, where 1 "\
            "shows the 2D plot and 0 doesn\'t")
    else:
        fname = sys.argv[1]
        n_comp = int(sys.argv[2])
        do_plot = int(sys.argv[3])
        with open(fname) as f:
            first_line = f.readlines()[0]
            try:
                [float(f) for f in first_line.split(',')]
                header = None
            except:
                header = 0

    X = pd.read_csv(fname, header = header)
    loads, scores, sign_ratios, feats =\
            pca_wrapper(X, n_comp = n_comp, do_plot = do_plot)
    if header is not None:
        with open(fname) as f:
            header = f.readlines()[0]
            feat_names = [first_line.split(',')[f] for f in feats]
        print('The most important attributes are:\n%s' %
                ','.join(feat_names))
    else:
        feats = [str(f) for f in feats]
        print('The most important attribute indexes are:\n%s' %
                ','.join(feats))
    import pdb; pdb.set_trace()
```

A.3 Application 2 (face recognition) source code.

Listing 4: Eigenfaces implementation code (src/face_classifier.py).

```
from sklearn.datasets import fetch_olivetti_faces
import numpy as np
import cv2
from collections import OrderedDict as OD
from sklearn.metrics import classification_report
from matplotlib import pyplot as plt
import pickle as pkl
import os
import time


class faceClassifier():
    def __init__(self, ratio = 0.85, K = 200, data_pkl = None, target_pkl = None):
        """__init__. Class constructor.

        Parameters
        ----------
        ratio :
            How much of the total data to use for training (0 to 1)
        K :
            How many eigenface space base vectors to keep in order to express each image
        data_pkl :
            Pickle serialised file that contains data (see export method)
        target_pkl :
            Pickle serialised file that contains label (see export method)
        """
        if data_pkl is not None:
            with open(data_pkl, 'rb') as f:
                self.data = pkl.load(f)
        else:
            self.data = None        # data vectors
        if target_pkl is not None:
            with open(target_pkl, 'rb') as f:
                self.target = pkl.load(f)
        else:
            self.target = None      # label (ground truth) vectors
        self.train_data = OD()      # maps sample index to data and label
        self.test_data = OD()       # maps sample index to data and label
        # how many eigenfaces to keep
        self.K = K
        # MxK matrix - each row stores the coords of each image in the eigenface space
        self.W = None
        self.classification_report = None # obtained from benchmarking
        # mean needed for reconstruction
        self._mean = np.zeros((1, 64*64), dtype=np.float32)
        if self.data is None and self.target is None: # no pre-loaded data
            self._load_olivetti_data()
        self._TRAIN_SAMPLE = 1
        self._PRED_SAMPLE = 0


    def __str__(self):
        M = len(self.data)
        Mtrain = len(self.train_data)
        Mtest = len(self.test_data)
        return "Loaded %d samples in total.\n"\
            "%d for training and %d for testing." % (M, Mtrain, Mtest)

```

```
59
60      def _load_olivetti_data(self):
61          """Load the Olivetti face data and save them in the class."""
62          data, target = fetch_olivetti_faces(return_X_y = True)
63          # data as floating vectors of length 64^2, ranging from 0 to 1
64          self.data = np.array(data)
65          # subject labels (sequential from to 0 to ...)
66          self.target = target
67
68
69      def _record_mean(self):
70          self._mean += np.mean(self.data, axis=0) # along columns
71
72
73      def _subtract_mean(self):
74          """
75          Make the mean of every column of self.data zero
76          """
77          self._record_mean()
78          M = self.data.shape[0]
79          C = np.eye(M) - 1/M*np.ones((M,1)) # centring matrix
80          self.data = np.matmul(C, self.data)
81
82
83      def _read_from_webcam(self, new_label, stream = 0):
84          """Takes face snapshots from webcam. Pass the new label of the subject
85          being photographed."""
86          print("Position your face in the green box.\n"
87                  "Press p to capture your face profile from slightly different
    angles,\n"
88                  "or q to quit.")
89          time.sleep(3)
90          # if stream == 0, try to open default webcam, else video from path
91          cap = cv2.VideoCapture(stream)
92          while True:
93              # Capture frame by frame
94              _, frame = cap.read()
95              grey = cv2.cvtColor(frame, cv2.COLOR_BGR2GRAY)
96              min_shape = min(grey.shape)
97              cv2.rectangle( frame, (0,0), (int(3*min_shape/4),
98                  int(3*min_shape/4)), (0,255,0), thickness = 4)
99              cv2.imshow('frame',frame)
100             k = cv2.waitKey(10) & 0xff
101             if k == ord('q'):
102                 break
103             elif k == ord('p'):
104                 im_cropped = grey[:int(3*min_shape/4), :int(3*min_shape/4)]
105                 cv2.destroyAllWindows()
106                 cv2.imshow("new data", im_cropped)
107                 cv2.waitKey(1500)
108                 cv2.destroyWindow("new data")
109                 x = self.img2vec(im_cropped)
110                 self.data = np.array([*self.data, np.array(x, dtype=np.float32)
    ])
111                 self.target = np.append(self.target, new_label)
112         cap.release()
113         cv2.destroyAllWindows()
114
115
116     def add_img_data(self, fpaths = [], from_webcam = False):
117         """add_img_data. Adds data and their labels to existing database.
118
119         Parameters
120         ----------
```

```
        fpaths : list
            fpaths list of image files. If empty, they're ignored
        from_webcam : bool
            from_webcam if True, opens webcam and lets the user capture face
    data
        """
        assert len(self.target) != 0, "No labels have been generated!"
        # create new label for new subject
        target_new = self.target[-1] + 1
        self.target = np.append(self.target, [target_new]*len(fpaths))
        # convert image to 64*64 data vector (ranging from 0 to 1)
        for i, f in enumerate(fpaths):
            im = cv2.imread(f)
            im = cv2.cvtColor(im, cv2.COLOR_BGR2GRAY)
            im = np.asarray(cv2.resize(im, dsize = (64,64))).ravel()
            # normalise from 0 to 1 - the range of original Olivetti data
            im = im/255
            self.data = np.array([*self.data, np.array(im, dtype=np.float32)])
        if from_webcam:
            self._read_from_webcam(new_label = target_new)
        self.data = np.array(self.data)


    def train(self):
        """ Find the coordinates of each training image in the eigenface space
    """
        self._divide_dataset()
        # the matrix X to use for training
        X = np.array([v[0] for v in self.train_data.values()])
        # compute eig of MxN^2 matrix first instead of the N^2xN^2, N^2 >> M
        XXT = np.matmul(X, X.T)
        eval_XXT, evec_XXT = np.linalg.eig(XXT)
        # sort eig data by decreasing eigvalue values
        idx_eval_XXT = eval_XXT.argsort()[::-1]
        eval_XXT = eval_XXT[idx_eval_XXT]
        evec_XXT = evec_XXT[idx_eval_XXT]
        # now compute eigs of covariance matrix (N^2xN^2)
        self.evec_XTX = np.matmul(X.T, evec_XXT)
        # coordinates of each face in "eigenface" subspace
        self.W = np.matmul(X, self.evec_XTX)
        self.W = self.W[:, :self.K]


    def _divide_dataset(self, ratio = 0.85):
        """Divides dataset in training and test (prediction) data"""
        if not 0 < ratio < 1:
            raise RuntimeError("Provide a ratio between 0 and 1.")
        training_or_test = [self._random_binary(ratio) for _ in self.data]
        self._subtract_mean()

        train_inds = [i for i,t in enumerate(training_or_test) if t == self.
    _TRAIN_SAMPLE]
        test_inds = [i for i,t in enumerate(training_or_test) if t == self.
    _PRED_SAMPLE]
        # {index: (data_vector, data_label)}, index starts from 0
        self.train_data = OD(                                       # ordered
    dict
                dict(zip(train_inds,                                # keys
            zip(self.data[train_inds,:], self.target[train_inds]))) # vals
                )
        self.test_data = OD(                                        # ordered
    dict
                dict(zip(test_inds,                                 # keys
            zip(self.data[test_inds,:], self.target[test_inds])))   # vals
```

```
            )


    def _random_binary(self, prob_of_1 = .5) -> int:
        """_random_binary. Randomly returns 0 or 1. Accepts probability
        to return 1 as input."""
        return np.round(np.random.uniform(.5, 1.5) - 1 + prob_of_1).astype(np.
    uint8)


    def get_test_sample(self) -> tuple:
        """ Get random training sample and its label. Returns (data vector,
    label) """
        Ntest = len(self.test_data)
        n = np.random.randint(0, Ntest)
        test_ind = [k for k in self.test_data.keys()][n]
        return self.test_data[test_ind] # data, label


    def classify(self, x_new:np.array) -> tuple:
        """classify. Classify an input data vector.

        Parameters
        ----------
        x_new : np.array
            Data vector

        Returns
        -------
        tuple
            containing the predicted data vector and its label (data, label)
        """
        train_inds = sorted([i for i in self.train_data.keys()])
        M = len(train_inds)
        # find eigenface space coordinates
        w_new = np.matmul(self.evec_XTX.T, x_new.T)
        w_new = w_new[:self.K]
        # if not match w/ itself else inf
        dists = [np.linalg.norm(w_new - self.W[i,:])
                if (np.linalg.norm(w_new - self.W[i,:]) > 0.0) else
                np.infty
                for i in range(M)]
        return (self.train_data[train_inds[np.argmin(dists)]][0], # data
                self.train_data[train_inds[np.argmin(dists)]][1]) # label


    def vec2img(self, x:list):
        """Converts an 1D data vector stored in the class to image."""
        x = np.array(x) + self._mean
        x = np.reshape(255*x, (64,64))
        return np.asarray(x, np.uint8)


    def img2vec(self, im) -> np.ndarray:
        """Converts an input greyscale image to an 1D data vector."""
        if not len(im.shape) == 2:
            raise RuntimeError("Provide a greyscale image as input.")
        x = np.asarray(cv2.resize(im, dsize=(64,64)), np.float32).ravel()
        x /= 255
        x = np.reshape(x, self._mean.shape)
        x -= self._mean
        # needed for eigenface coords dot product, do NOT delete this
        x = np.reshape(x, self.data[-1,:].shape)
        return x
```

```
241
242
243     def webcam2vec(self):
244         """ Opens webcam. The user can take a picture. Returns picture
245         as data vector"""
246         cap = cv2.VideoCapture(0)
247         while True:
248             # Capture frame by frame
249             _, frame = cap.read()
250             grey = cv2.cvtColor(frame, cv2.COLOR_BGR2GRAY)
251             min_shape = min(grey.shape)
252             cv2.rectangle( frame, (0,0), (int(3*min_shape/4),
253                 int(3*min_shape/4)), (0,255,0), thickness = 4)
254             cv2.imshow('frame',frame)
255             k = cv2.waitKey(10) & 0xff
256             if k == ord('q'):
257                 break
258             elif k == ord('p'):
259                 im_cropped = grey[:int(3*min_shape/4), :int(3*min_shape/4)]
260                 cv2.destroyAllWindows()
261                 cv2.imshow("new data", im_cropped)
262                 cv2.waitKey(1500)
263                 cv2.destroyWindow("new data")
264                 x = self.img2vec(im_cropped)
265                 break
266         cap.release()
267         cv2.destroyAllWindows()
268         return x
269
270
271     def benchmark(self, imshow = False, wait_time = 0.5, which_labels = []):
272         """benchmark. Iterates over each test sample and classifies it.
273         Genrates a classification report with all the classification metrics.
274
275         Parameters
276         ----------
277         imshow : bool
278             If True, show the actual vs predicted image, each for some times.
279         wait_time : float
280             How many seconds to show each actual vs predicted image for.
281         which_labels : list
282             Which labels to show. Useful when a new label was just added.
283         """
284         self.train()
285         lbl_actual = []
286         lbl_test = []
287         for ind_test, test_data_lbl in self.test_data.items():
288             # if we want to show only certain labels
289             if len(which_labels) != 0:
290                 if test_data_lbl[1] not in which_labels:
291                     continue
292             x_actual = test_data_lbl[0]
293             lbl_actual.append(test_data_lbl[1])
294             x_test, lbl = self.classify(x_actual)
295             lbl_test.append(lbl)
296             if imshow:
297                 fig=plt.figure(figsize=(64, 64))
298                 cols, rows = 2, 1
299                 ax1 = fig.add_subplot(rows, cols, 1)
300                 ax1.title.set_text('actual: %d' % test_data_lbl[1])
301                 plt.imshow(self.vec2img(x_actual))
302                 ax2 = fig.add_subplot(rows, cols, 2)
303                 ax2.title.set_text('predicted: %d' % lbl)
304                 plt.imshow(self.vec2img(x_test))
```

```
                plt.show(block=False)
                plt.pause(wait_time)
                plt.close()
        if len(lbl_actual) != 0 and len(lbl_test) != 0:
            self.classification_report = classification_report(y_true =
    lbl_actual,
                    y_pred = lbl_test)


    def export(self, dest_folder = '/tmp'):
        try:
            with open(os.path.join(dest_folder, 'data.pkl'), 'wb') as f:
                pkl.dump(self.data, f)
            with open(os.path.join(dest_folder, 'target.pkl'), 'wb') as f:
                pkl.dump(self.target, f)
            print("Wrote data and target as .pkl at:\n%s" % dest_folder)
        except Exception as e:
            print(e)
```