Lecture 8: Principal component analysis (PCA)

Nils Olovsson
nils.olofsson@igp.uu.se

Introduction to data science and Python programming for medical research
Uppsala University

2025

UPPSALA
UNIVERSITET

Content

PCA

PCA

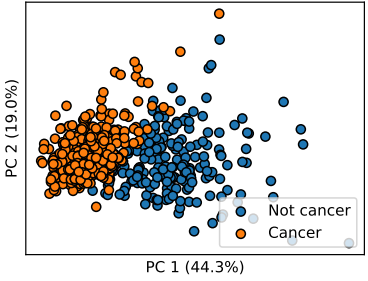
PCA: Problem

Have very high dimensional data that we want to plot.

Or we might want to extract features used to classify data points or create model used to generate samples from the data distribution using only a few parameters.

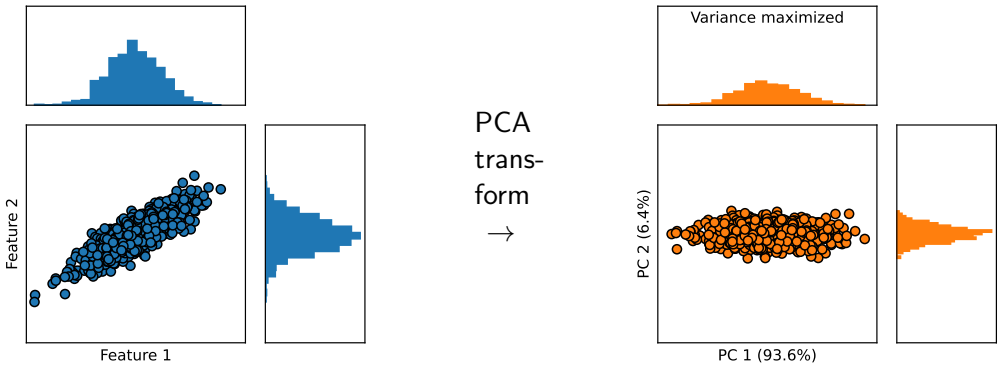
E.g. cancer dataset with 30 features.

PCA can be used to **reduce the dimensionality** of this dataset to 2 such that we can plot it.



PCA: Maximizing variance

PCA can reduce the number of dimensions by finding a representation of the data that keeps as much variance as possible.



PCA: Minimizing distance

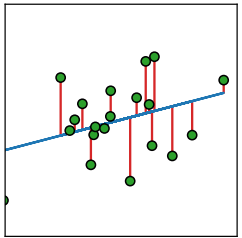
PCA can also be viewed as a **minimization problem**. Model is the same as in linear regression! A line (or hyperplane)

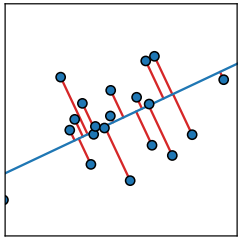
Linear regression

Minimize **vertical distance** between model and data.
Predict y from x.

PCA

Minimize **perpendicular, Euclidean, distance** between model and data.
Relationship between x and y.





PCA: Principal components (PC)

Vectors

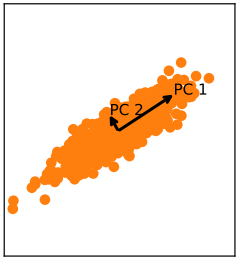
Principal components are **vectors**, $\mathbf{v}_i$, that create a data-oriented coordinate system.

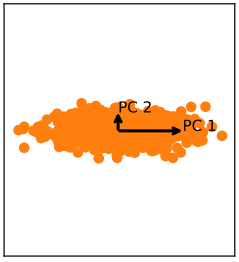
Linear transform

Can be seen as a **rotation or re-orientation** of the data points.

Variance

Each **vector** is associated with a **variance** perpendicular to its direction.





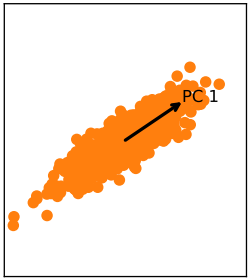
PCA: Dimensionality reduction (`transform(P)`)

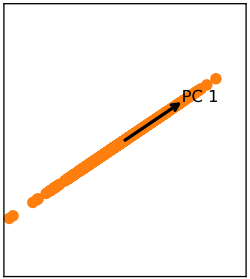
Dimensionality reduction with PCA is achieved by **projecting** data points on the first PC vectors.

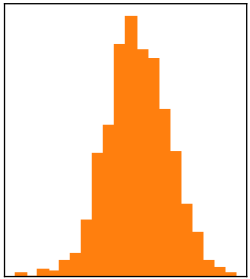
This **embeds** the data in the PCA coordinate system.

The projection is calculated using the **dot product** of a PC vector, $\mathbf{v}_i$, and a data point, $\mathbf{p}$.

$x_i = \mathbf{v}_i \cdot \mathbf{p}$







PCA: Generate sample from model (`inverse_transform(X)`)

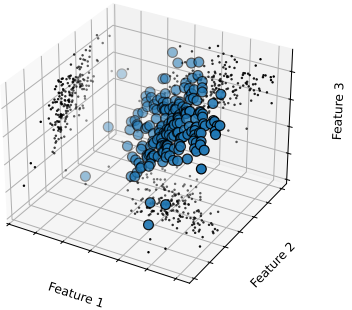
A sample in original data space can be generated from the model by using all or a subset of of PCs.
Start with the mean value of the dataset, μ, and add offsets along the PC kept in the model vectors.
$\mathbf{p} = \mu + a\mathbf{v}_1 + b\mathbf{v}_2 + \cdots$

PCA: Higher dimensions

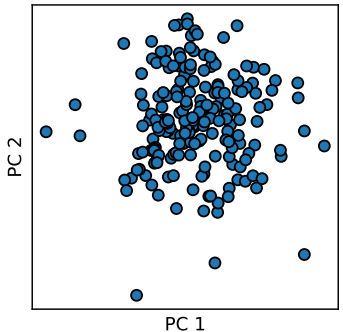
Find PC vectors but instead of two 2D vectors we get N ND vectors.

$$\mathbf{v}_{ND} = [v_1, v_2, \cdots v_N]$$

When projecting data on subset of PC vectors they are no longer projected on a line but on a **hyperplane**. In 3D a hyperplane is a 2D plane.



PCA transform $\rightarrow$



PCA: Units on the axes

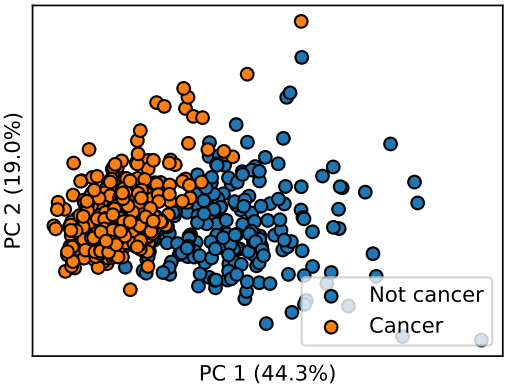
The axes after the transform represent fractions of change in the original data features!

Do NOT write

Generally, do not write **units on these axes** unless you know what you are doing!
Since we do not write units, we should not write any **values on the axes** either!

Do write

Write only the **fraction of the variance** the PC vector represents!



PCA: How many components do we need?

Visualization

Visualization is a special case were we want to project or data on a screen or on paper. Typically keep **2 components** but 1 and 3 can be valid as well.

Modeling

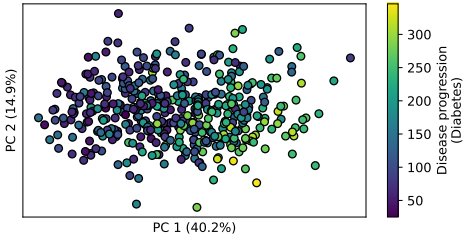
Each PC is associated with the variance in the data in decreasing order.
Do not explicitly specify number of components to keep but rather **how much of the total variance** we we want the model to keep!

Python

Python

Python: PCA for visualization

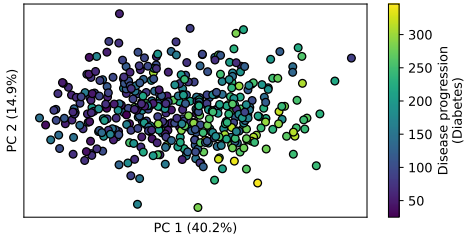
```python
import numpy as np
import matplotlib.pyplot as plt
from sklearn import datasets
from sklearn.decomposition import PCA
# Load data, log, fit PCA, and transform data
X, y = datasets.load_diabetes(return_X_y=True)
pca = PCA(n_components=2)
X_pca = pca.fit_transform(X)
# Create figure, plot projected data points
fig = plt.figure(figsize=(6,3))
ax = fig.gca()
sc =ax.scatter(X_pca[:, 0],
               X_pca[:, 1],
               marker='o',
               c = y,
               cmap = 'viridis',
               edgecolor='k')
# Indicate explained variance on axes
v = 100.0 * pca.explained_variance_ratio_
ax.set_xlabel(f'PC 1 ({v[0]:.1f}%)')
ax.set_ylabel(f'PC 2 ({v[1]:.1f}%)')
plt.colorbar(sc, label='Disease progression\n(Diabetes)')
#plt.show()
print("Diabetes dataset")
print(f"Number of samples:  {X.shape[0]}")
print(f"Number of features: {X.shape[1]}")
```



Python: PCA for modeling

```python
import numpy as np
import matplotlib.pyplot as plt
from sklearn import datasets
from sklearn.decomposition import PCA
# Load data, log, fit PCA, and transform data
X, y = datasets.load_diabetes(return_X_y=True)
pca = PCA(0.90)
X_pca = pca.fit_transform(X)
# Create figure, plot projected data points
fig = plt.figure(figsize=(6,3))
ax = fig.gca()
sc =ax.scatter(X_pca[:, 0],
               X_pca[:, 1],
               marker='o',
               c = y,
               cmap = 'viridis',
               edgecolor='k')
# Indicate explained variance on axes
v = 100.0 * pca.explained_variance_ratio_
ax.set_xlabel(f'PC 1 ({v[0]:.1f}%)')
ax.set_ylabel(f'PC 2 ({v[1]:.1f}%)')
plt.colorbar(sc, label='Disease progression\n(Diabetes)')
#plt.show()
print("Diabetes dataset")
print(f"Number of samples:  {X.shape[0]}")
print(f"Number of features: {X.shape[1]}")
```

The trasformed data contains more than 2 dimensions but we can still plot using the two most principal components.

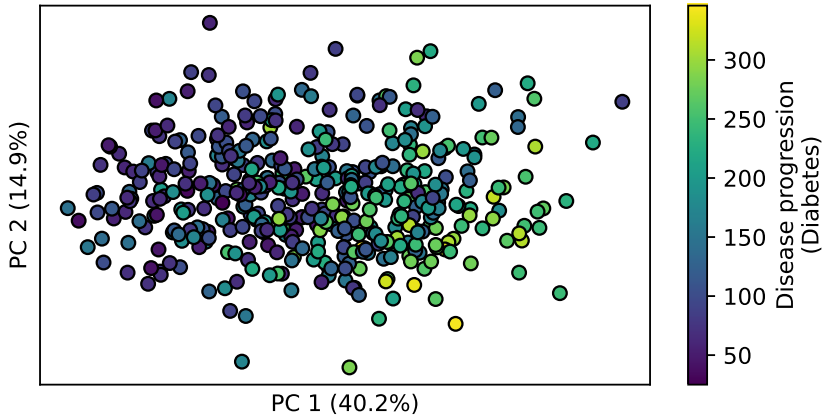


Examples

Examples

Examples: Diabetes

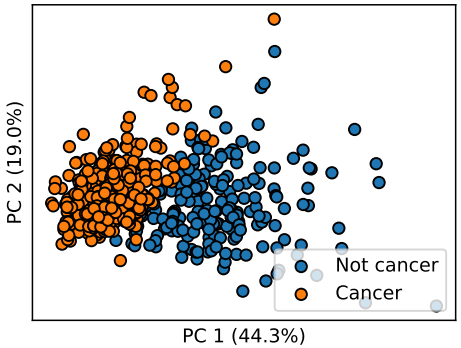
We can observe how the first PC seems to model the general direction of disease progression.



Examples: Breast cancer (morphology)

```python
import numpy as np
import matplotlib.pyplot as plt
from sklearn import datasets
from sklearn.preprocessing import
    StandardScaler
from sklearn.decomposition import PCA
# Load data, log, fit PCA, and transform data
X, y = datasets.load_breast_cancer(return_X_y=
    True)
X = StandardScaler().fit_transform(X)

pca = PCA(n_components=2)
X_pca = pca.fit_transform(X)
# Create figure, plot projected data points
fig = plt.figure(figsize=(4,3))
ax = fig.gca()
ax.scatter(X_pca[y==0, 0], X_pca[y==0, 1],
           marker='o', color='tab:blue',
           edgecolor='k', label='Not cancer')
ax.scatter(X_pca[y==1, 0], X_pca[y==1, 1],
           marker='o', color='tab:orange',
           edgecolor='k', label='Cancer')
# Indicate explained variance on axes
v = 100.0 * pca.explained_variance_ratio_
ax.set_xlabel(f'PC 1 ({v[0]:.1f}%)')
ax.set_ylabel(f'PC 2 ({v[1]:.1f}%)')
ax.legend(loc='lower right')
#plt.show()
```

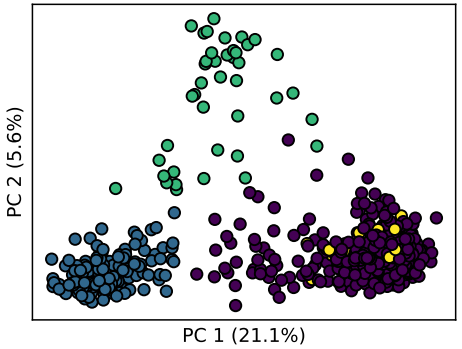
We can almost separate all true cancer cases from the non-cancer ones even when the data has been reduced to only two PCs.



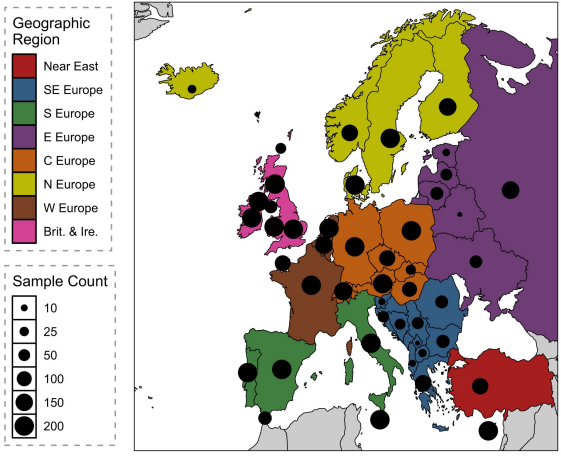
Examples: Breast cancer (omics)

```python
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
from sklearn.decomposition import PCA
# Load data, get X and y
filename = 'CAFs.txt'
filepath = os.path.join('..', '..', 'data',
     filename)
df = pd.read_csv(filepath, sep='\t')
X = df.values[:,0:(df.shape[1]-1)]
y = df.values[:,df.shape[1]-1]
# Log transform X, fit PCA, transform
X = np.log(X + 1)
pca = PCA(n_components=2)
X_pca = pca.fit_transform(X)
# Create figure, plot projected data points
fig = plt.figure(figsize=(4,3))
ax = fig.gca()
ax.scatter(X_pca[:,0], X_pca[:, 1],
           marker='o', c=y.astype(int),
           edgecolor='k')
# Indicate explained variance on axes
v = 100.0 * pca.explained_variance_ratio_
ax.set_xlabel(f'PC 1 ({v[0]:.1f}%)')
ax.set_ylabel(f'PC 2 ({v[1]:.1f}%)')
#plt.show()
```

Three of the four cell types can still be clearly separated even when the data has been reduced to two PCs.

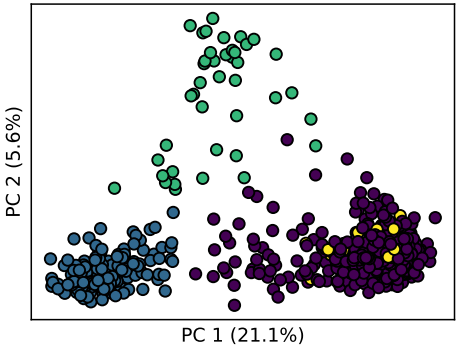


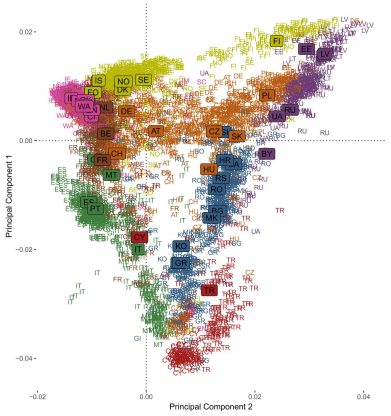
Examples: Map of Europe



Adapted from Gilbert *et al.* 2022. (Can you spot the mistakes?)

Examples: Triangular shape of PCA

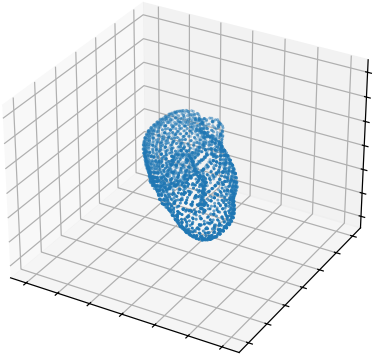




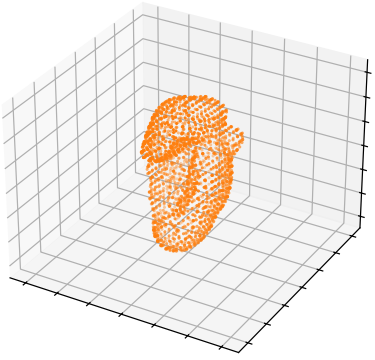
Adapted from Gilbert *et al.* 2022.

https://medium.com/towards-data-science/why-pca-looks-triangular-a642daac721a

Examples: Coordinate system alignment of 3D data



PCA
trans-
form
$\rightarrow$



Examples: Principal plane of 3D respiratory motion

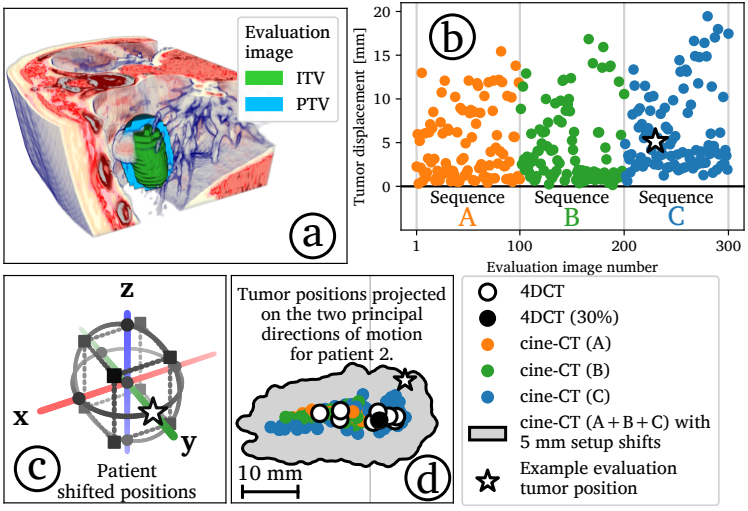


Image processing

Image processing

Image processing: Flatten image to vector

The images in a dataset needs to be flattened into vectors to be analyzed with PCA.

Note that by flattening the image like this we loose the geometry represented by the image and the fact that neighboring pixels are connected.

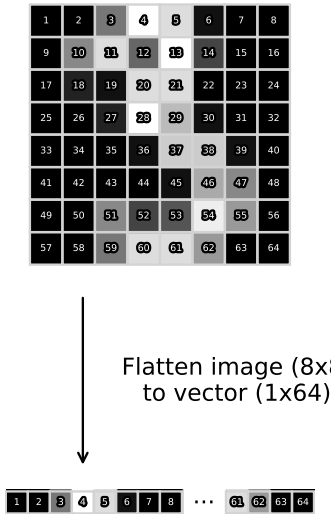
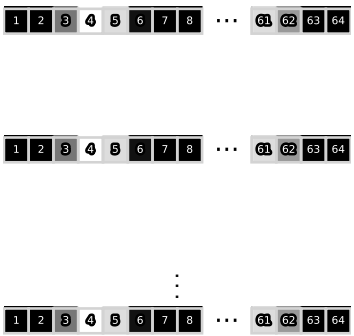


Image processing: PCA embedding

Put flattened images into data matrix, *e.g.* by vertical stacking using `np.vstack`.



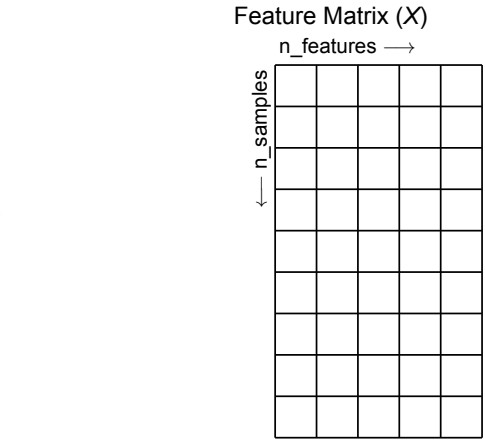


Image processing: Unpack vector to image

Any output from the PCA model using `inverse_transform(...)` will be in the form of a flat vector.

To create an image from the vector we reshape it to the shape of the original images used to create the PCA model.

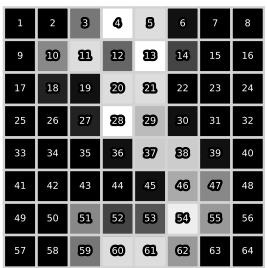


Image processing: Example - Interpolate

Digits dataset in sklearn contains 1 797 8×8 images of **handwritten digits**.

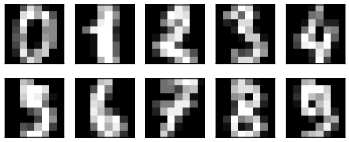
Flatten these into vectors of 64 features and apply PCA.

```
pca = PCA(n_components=2)
pca.fit_transform(X)
```

Plot how they are grouped in a 2D PCA projection.

Can sample this PCA-space to generate new images that captures shape of digits.

```
pca.inverse_transform(y)
```



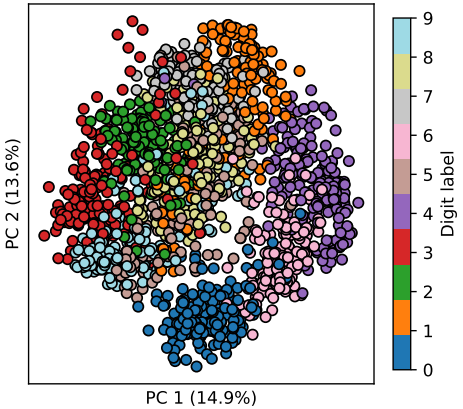


Image processing: Example - Interpolate

Digits dataset in sklearn contains 1 797 8×8 images of **handwritten digits**.

Flatten these into vectors of 64 features and apply PCA.

```
pca = PCA(n_components=2)
pca.fit_transform(X)
```

Plot how they are grouped in a 2D PCA projection.

Can sample this PCA-space to generate new images that captures shape of digits.

```
pca.inverse_transform(y)
```

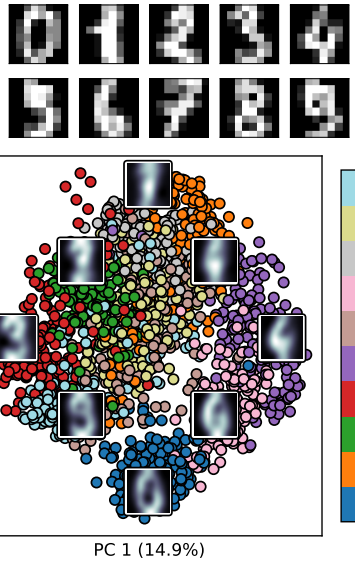
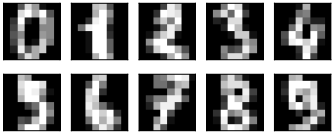


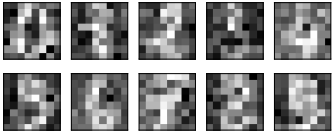
Image processing: Example - Noise reduction

```python
import numpy as np
import matplotlib.pyplot as plt
from sklearn.decomposition import PCA
from sklearn.datasets import load_digits
def plot_noisy_digits(
    X: np.ndarray,
    plot_index: int,
) -> None:
    fig, axs = plt.subplots(2,5,figsize=(5,2))
    axs = axs.flatten()
    for i in range(10):
        image = X[i,:].reshape((8,8))
        axs[i].imshow(image, cmap='gray')
        self._hide_plot_axes_and_ticks(axs[i])
    self.save_and_close_figure(
      image_number_in_sequence=plot_index)
    return
# Load digits and apply noise
X = load_digits().data
rng = np.random.default_rng(seed=0)
X_noise = rng.normal(X, 4)
# PCA, transform noisy data and inverse
pca = PCA(0.5).fit(X_noise)
y = pca.transform(X_noise)
X_inv = pca.inverse_transform(y)
# Plot original, noisy and noise reduced
plot_noisy_digits(X, 0)
plot_noisy_digits(X_noise, 1)
plot_noisy_digits(X_inv, 2)
```

Original images.



Images with added noise.



Noisy images transformed with PCA, then inverse transformed to get images with less noise.

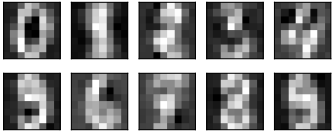


Image processing: Example - Eigenfaces (1/3)

Eigenfaces is a classical example of using PCA to model images of human faces.

The dataset, available in sklearn, consists of N images of size AxB and represents different images of the faces of Z (at one time well known) people.

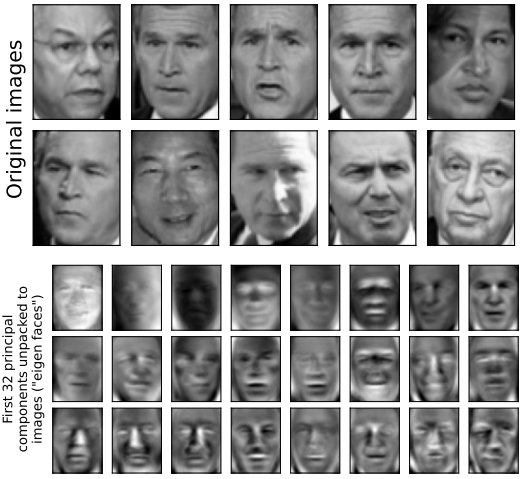


Image processing: Example - Eigenfaces (2/3)

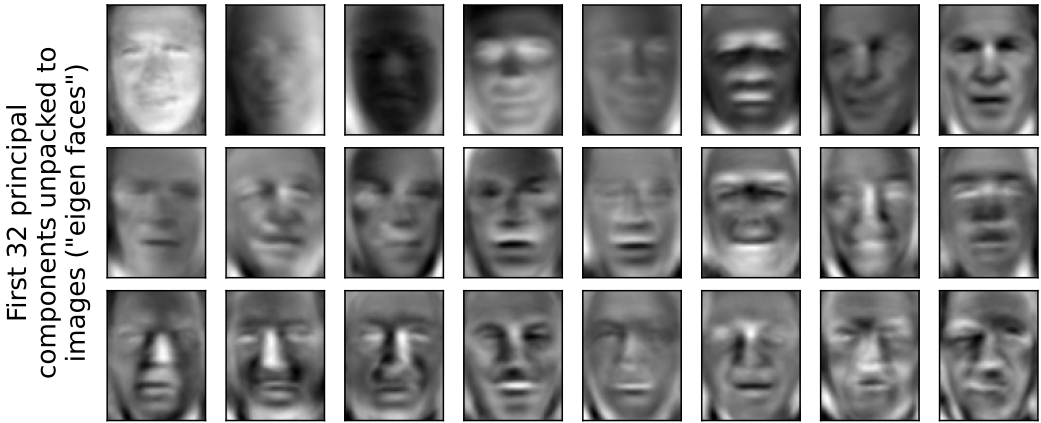
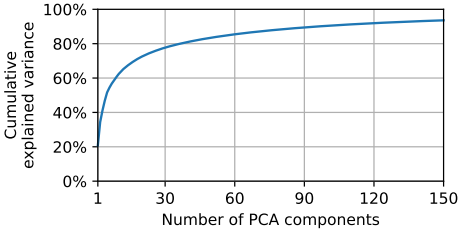
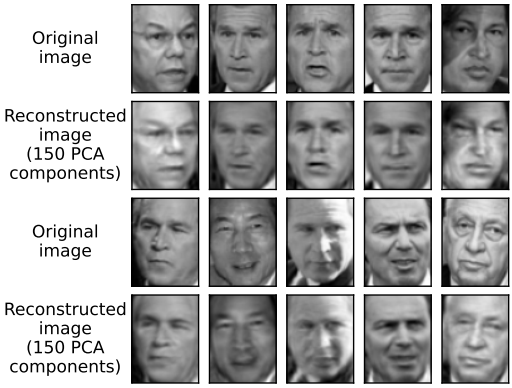


Image processing: Example - Eigenfaces (3/3)

With 150 PCA components we keep nearly 90% of the explained variance.

If we project an image on these 150 components and then reconstruct it using these 150 PCA features using `pca.inverse_transform(y)` we get results that are very similar to the original images!





Summary

Summary

Summary: PCA

Very versatile method with lots of utility for different purposes.

- Dimensionality reduction
- Noise reduction
- Data transformation
- Generate by sampling data points in PCA space

Covariance

Covariance

Covariance: In depth

Now we have a good understanding of what PCA is and what it does.
But how do we actually find the PC vectors?
They are the so called **eigenvectors** of the **covariance matrix**!

Covariance: Variance

We have previously seen variance for data in 1D.

$$\text{Var}(X) = \frac{1}{N}\Sigma_i(x_i - \mu)$$

$\mu =$ mean value of$X$

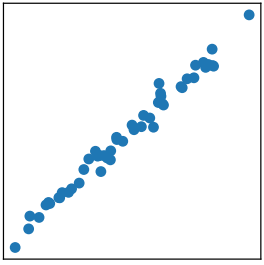
Covariance: Covariance

Covariance describes how features in our data are related or correlated.
If an increasing value along in one feature corresponds to an increasing, decreasing or no change in another.

$$\text{Cov}(X, Y) = \frac{1}{N}\Sigma_i(x_i - \mu_X)(y_i - \mu_Y)$$

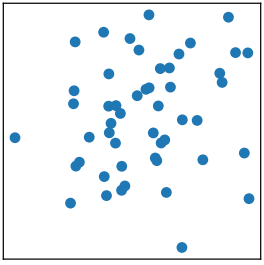
$$\mu_X = \text{mean value of} X$$
$$\mu_Y = \text{mean value of} Y$$

Covariance: Covariance matrix



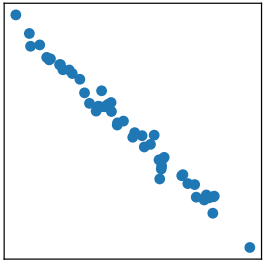
Positive correlation.

$$\begin{pmatrix} 1 & a \\ a & 1 \end{pmatrix}$$



No correlation.

$$\begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix}$$



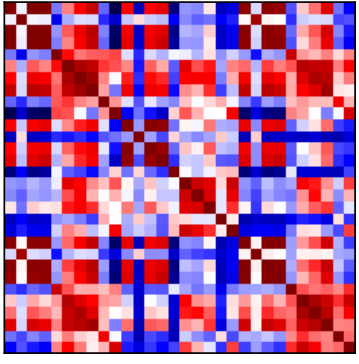
Negative correlation.

$$\begin{pmatrix} 1 & -a \\ -a & 1 \end{pmatrix}$$

Covariance: Python

```python
import numpy as np
import matplotlib.pyplot as plt
from sklearn.preprocessing import StandardScaler
from sklearn.datasets import load_breast_cancer

# Load breast cancer data set ("normalize" it)
X, y = load_breast_cancer(return_X_y=True)
X = StandardScaler().fit_transform(X)

# Compute covariance matrix
# Note transpose to get correct shape
C = np.cov(X.T)

print(f"Shape of data matrix X:\n{X.shape}\n")
print(f"Shape of covariance matrix C:\n{C.shape}")

# Draw the heatmap with the mask and correct aspect
    ratio
ax = plt.figure(figsize=(3, 3)).gca()
ax.imshow(C, cmap='seismic')
#plt.show()
```

```
>> Shape of data matrix X:
>> (569, 30)
>>
>> Shape of covariance matrix C:
>> (30, 30)
```



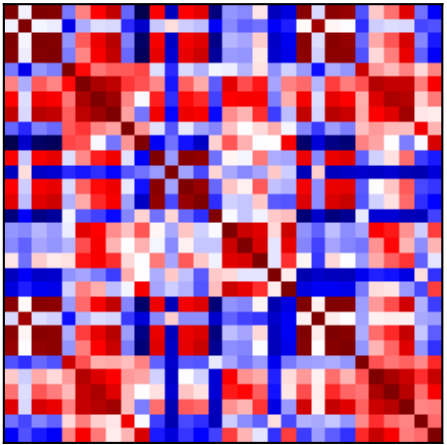
Eigendecomposition

Eigendecomposition

Eigendecomposition: Principal component vectors

The **PC vectors** are the **eigenvectors** of the **covariance matrix**.
Their associated **variances** are the corresponding **eigenvalues**.

The computation of these values is also called a **spectral** decomposition.



Eigendecomposition: Spectral theorem

Square, symmetric (positive definite) matrices A can be **factorized** as

$$A = QVQ^{-1}$$

Where the **columns of** Q are its **eigenvectors**, a set of **normalized, linearly independent vectors**. The diagonal of V contains their respective **eigenvalues**.

```python
import numpy as np
from sklearn.datasets import make_spd_matrix
rng = np.random.default_rng(seed=1)
# Create symmetric, positive definite matrix
A = make_spd_matrix(4)
# Eigen decomposition
v, Q = np.linalg.eig(A)
V = np.diag(v)
```

$$A = \begin{pmatrix} 1.33 & -0.14 & -1.01 & -1.32 \\ -0.14 & 0.70 & -0.10 & 0.08 \\ -1.01 & -0.10 & 1.88 & 1.60 \\ -1.32 & 0.08 & 1.60 & 2.33 \end{pmatrix}$$

$$v = \begin{pmatrix} 4.57 & 0.82 & 0.39 & 0.45 \end{pmatrix}$$

$$V = \begin{pmatrix} 4.57 & 0.00 & 0.00 & 0.00 \\ 0.00 & 0.82 & 0.00 & 0.00 \\ 0.00 & 0.00 & 0.39 & 0.00 \\ 0.00 & 0.00 & 0.00 & 0.45 \end{pmatrix}$$

$$Q = \begin{pmatrix} 0.46 & -0.36 & -0.67 & -0.46 \\ -0.02 & 0.81 & -0.04 & -0.58 \\ -0.57 & -0.45 & 0.28 & -0.63 \\ -0.68 & 0.12 & -0.69 & 0.23 \end{pmatrix}$$

Dot product between two **eigenvectors** is 0, *i.e.* they are all perpendicular to each other.

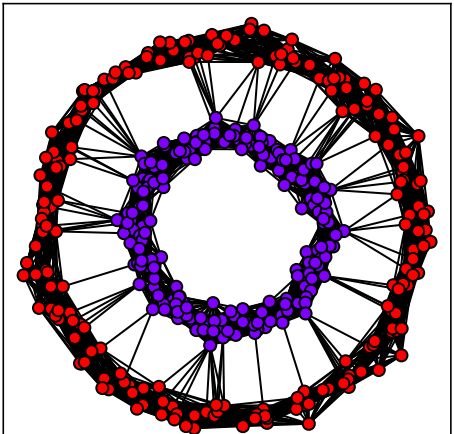
Transformed coordinate system!

Eigendecomposition: Spectral methods

We have already encountered spectral decomposition in spectral clustering and we will see it again in manifold learning and image segmentation.

Spectral methods in this course:

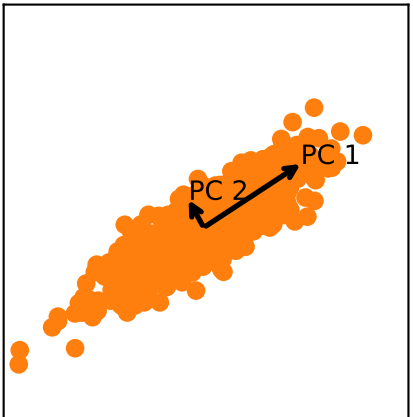
- PCA
- Spectral clustering
- Manifold learning
- Spectral image segmentation



Eigendecomposition: Implementing PCA from scratch (1/4)

Create a class that will contain all of our PCA stuff.

```python
import numpy as np

class PCA:

    def __init__(
        self,
        n_components: int,
    ):
        # Number of components kept by PCA model
        self.n_components = n_components

        # Results after fit
        self.explained_variance_ = -1
        self.mean_ = None
        self.components_ = None

        return
```



Eigendecomposition: Implementing PCA from scratch (2/4)

```
    def fit(
        self,
        X: np.ndarray,
    ) -> None:
        """ Fit the PCA model to data.
        """
        # Compute mean value
        self.mean_ = np.mean(X, axis=0)

        # Compute covariance matrix from data matrix
        C = np.cov(X.T)

        # Eigen decomposition
        v, Q = np.linalg.eigh(C)
        # Eigen values and vectors sorted in ascending order
        # Flip to get in largest eigen value-vector pair first
        v = np.flip(v)
        Q = np.fliplr(Q)

        # Store only the number of components
        v = v[0:self.n_components]
        Q = Q[:, 0:self.n_components]

        # Store results after fit
        self.explained_variance_ = v
        self.components_ = Q
        return
```

Need a function that fits the model to our data matrix.

Eigendecomposition: Implementing PCA from scratch (3/4)

```
1     def transform(
2         self,
3         x: np.ndarray,
4     ) -> np.ndarray:
5         """ Transform at data vector by projecting
       it on
6         the PC vectors.
7         """
8         n = self.n_components
9         y = np.zeros(n)
10        for i in range(n):
11            v = self.components_[:,i]
12            y[i] = np.dot(v, x)
13        return y
14
15    def inverse_transform(
16        self,
17        y: np.ndarray,
18    ) -> np.ndarray:
19        """ Can reproject compoents into the space
       of original data.
20        """
21        x = self.mean_
22        for i in range(self.n_components):
23            v = self.components_[:,i]
24            x += y[i] * v
25        return x
```

Can transform and inverse transform data.

$x_i = \mathbf{v}_i \cdot \mathbf{p}$

$\mathbf{p} = \mu + a\mathbf{v}_1 + b\mathbf{v}_2 + \cdots$

Eigendecomposition: Implementing PCA from scratch (4/4)

```python
import matplotlib.pyplot as plt
from sklearn.datasets import load_diabetes
from PCA import PCA

X, y = load_diabetes(return_X_y=True)

pca = PCA(2)
pca.fit(X)

y = pca.transform(X[0,:])
x = pca.inverse_transform(y)

print("Shape of data matrix X:")
print(X.shape)
print("Data after PCA should should have
      dimensionality 2")
print(y.shape)
print("Data after 'inverse trasnform' should have
      original dimensionality")
print(x.shape)
```

```
>> Shape of data matrix X:
>> (442, 10)
>> Data after PCA should should have
     dimensionality 2
>> (2,)
>> Data after 'inverse trasnform' should have
     original dimensionality
>> (10,)
```

Small example

Small example

Small example: Eigenvalues by hand (1/N)

Generally, for large matrices the eigenvalues and eigenvectors can not be calculated by hand.

For small matrices however it is possible.

We will calculate the eigenvalues, λ, for a 2D example. A is our covariance matrix and I is the identity matrix

$$A = \begin{pmatrix} a & b \\ c & d \end{pmatrix} \qquad I = \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix} .$$

The eigenvalue problem states that we want solve for λ such that this **determinant** is zero

$$|A - \lambda I| = 0$$

Small example: Eigenvalues by hand (2/N)

The determinant of a 2×2 matrix

$$M = \begin{pmatrix} m_{1,1} & m_{1,2} \\ m_{2,1} & m_{2,2} \end{pmatrix}$$

is defined as

$$|M| = \begin{vmatrix} m_{1,1} & m_{1,2} \\ m_{2,1} & m_{2,2} \end{vmatrix} = m_{1,1}m_{2,2} - m_{1,2}m_{2,1}$$

We apply this to our eigenvalue problem $|A - \lambda I| = 0$

$$\begin{vmatrix} a - \lambda & b \\ c & d - \lambda \end{vmatrix} = (a - \lambda)(d - \lambda) - bc = \lambda^2 - (a + d)\lambda + (ad - bc) = 0$$

Small example: Eigenvalues by hand (3/N)

The eigenvalue problem became a regular quadratic equation which we can solve!

$$\begin{vmatrix} a-\lambda & b \\ c & d-\lambda \end{vmatrix} = \lambda^2 - (a+d)\lambda + (ad-bc) = 0$$

We rewrite this

$$\lambda^2 + (-a-d)\lambda + (ad-bc) = 0$$
$$\Rightarrow$$
$$\lambda^2 + p\lambda + q = 0$$

so we can use the quadratic formula

$$\lambda = -\frac{p}{2} \pm \sqrt{\left(\frac{p}{2}\right)^2 - q}$$

Small example: Eigenvalues by hand (4/N)

Finally we get our two eigenvalues for $A = \begin{pmatrix} a & b \\ c & d \end{pmatrix}$

$$\begin{aligned} \lambda &= -\frac{p}{2} \pm \sqrt{\left(\frac{p}{2}\right)^2 - q} \\ &= \frac{a+d}{2} \pm \sqrt{\frac{a^2 + d^2 + 2ad}{4} + bc - ad} \end{aligned}$$

The corresponding eigenvectors v can further be found by solving the specific eigenvalue equation. This turns into a system of two equations.

$$0 = (A - \lambda I)v = \begin{pmatrix} a-\lambda & b \\ c & d-\lambda \end{pmatrix} \begin{pmatrix} v_1 \\ v_2 \end{pmatrix} \Rightarrow \begin{cases} (a-\lambda)v_1 + bv_2 = 0 \\ cv_1(d-\lambda) + v_2 = 0 \end{cases}$$

Small example: Compare with Python

```python
from math import sqrt
import numpy as np
def calculateEigenvalues2D(
    A: np.ndarray,
) ->np.ndarray:
    """ Compute eigenvalues of 2x2 matrix A.
    """
    a = A[0,0]; b = A[0,1]
    c = A[1,0]; d = A[1,1]

    p = -(a + d)
    q = a * d - b * c

    l0 = -p / 2 + sqrt( (p / 2)**2 - q)
    l1 = -p / 2 - sqrt( (p / 2)**2 - q)
    return np.array([l0, l1])
# Generate data, calculate eigenvalues
rng = np.random.default_rng(seed=0)
cov = np.array([[2, 1], [1, 2]])
X = rng.multivariate_normal(mean=(0,0),
                            cov=cov, size=100)
X = X.T
A = np.cov(X)
l_my = calculateEigenvalues2D(A)
l_np = np.linalg.eigvals(A)

# Print the eigenvalues
print("Eigenvalues: ")
```

$$\begin{cases} A = \begin{pmatrix} a & b \\ c & d \end{pmatrix} \\ p = -a - d \\ q = ad - bc \\ \lambda = -\frac{p}{1} \pm \sqrt{\left(\frac{p}{2}\right)^2 - q} \end{cases}$$

```
>> Eigenvalues:
>> [2.87 0.88]
>> [2.87 0.88]
```

Limitations

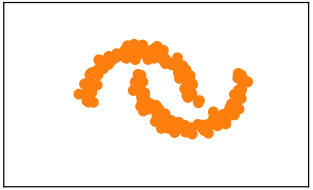
Limitations

Limitations: Normal distribution

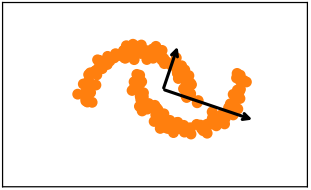
Assumes that data is from a multivariate normal distribution.

If the data is from some other distribution the PCA model will not actually capture this distribution.

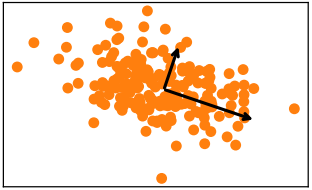
Data



PCA components

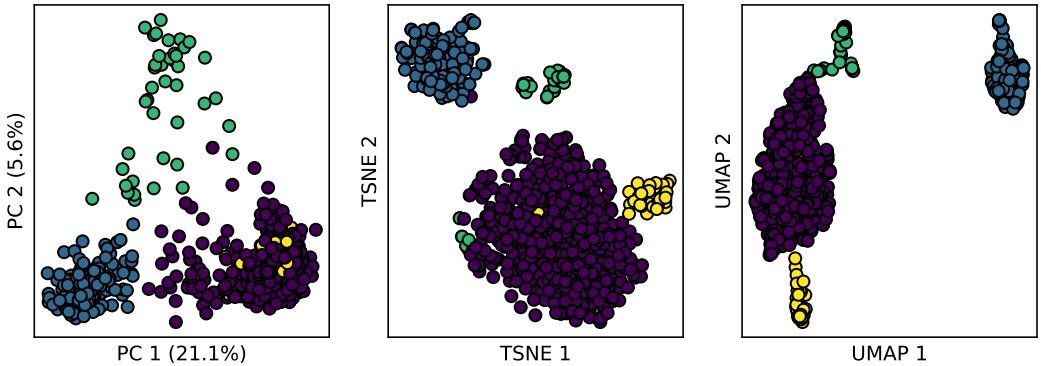


Samples from PCA model



Limitations: Linear model

Since the PCA is a linear model any non-linearities in the original data will not be captured by the model.



Limitations: Related methods

- kernel PCA
- Other PCAs
- Independent component analysis (ICA)

sklearn.decomposition

Matrix decomposition algorithms.

These include PCA, NMF, ICA, and more. Most of the algorithms of this module can be regarded as dimensionality reduction techniques.

User guide. See the Decomposing signals in components (matrix factorization problems) section for further details.

DictionaryLearning	Dictionary learning.
FactorAnalysis	Factor Analysis (FA).
FastICA	FastICA: a fast algorithm for Independent Component Analysis.
IncrementalPCA	Incremental principal components analysis (IPCA).
KernelPCA	Kernel Principal component analysis (KPCA).
LatentDirichletAllocation	Latent Dirichlet Allocation with online variational Bayes algorithm.
MiniBatchDictionaryLearning	Mini-batch dictionary learning.
MiniBatchNMF	Mini-Batch Non-Negative Matrix Factorization (NMF).
MiniBatchSparsePCA	Mini-batch Sparse Principal Components Analysis.
NMF	Non-Negative Matrix Factorization (NMF).
PCA	Principal component analysis (PCA).
SparseCoder	Sparse coding.
SparsePCA	Sparse Principal Components Analysis (SparsePCA).
TruncatedSVD	Dimensionality reduction using truncated SVD (aka LSA).
dict_learning	Solve a dictionary learning matrix factorization problem.
dict_learning_online	Solve a dictionary learning matrix factorization problem online.
fastica	Perform Fast Independent Component Analysis.
non_negative_factorization	Compute Non-negative Matrix Factorization (NMF).
sparse_encode	Sparse coding.

Limitations: Linear embedding

For visualizations we want to project the data on a 2D plane.

But the data might be better represented by by some other, non-planar, 2D embedding!

We will explore such methods in the next lecture on manifold learning.