Topic #3

Data analysis

Overview

Fiddling with signals and frequencies

sampling, frequency space representations, filters and filter properties, convolution theorem

Spectral analysis

(windowed) Fourier, Hilbert, wavelets, coherence measures

Multidimensional representations

PCA, ICA, SVD, k-means

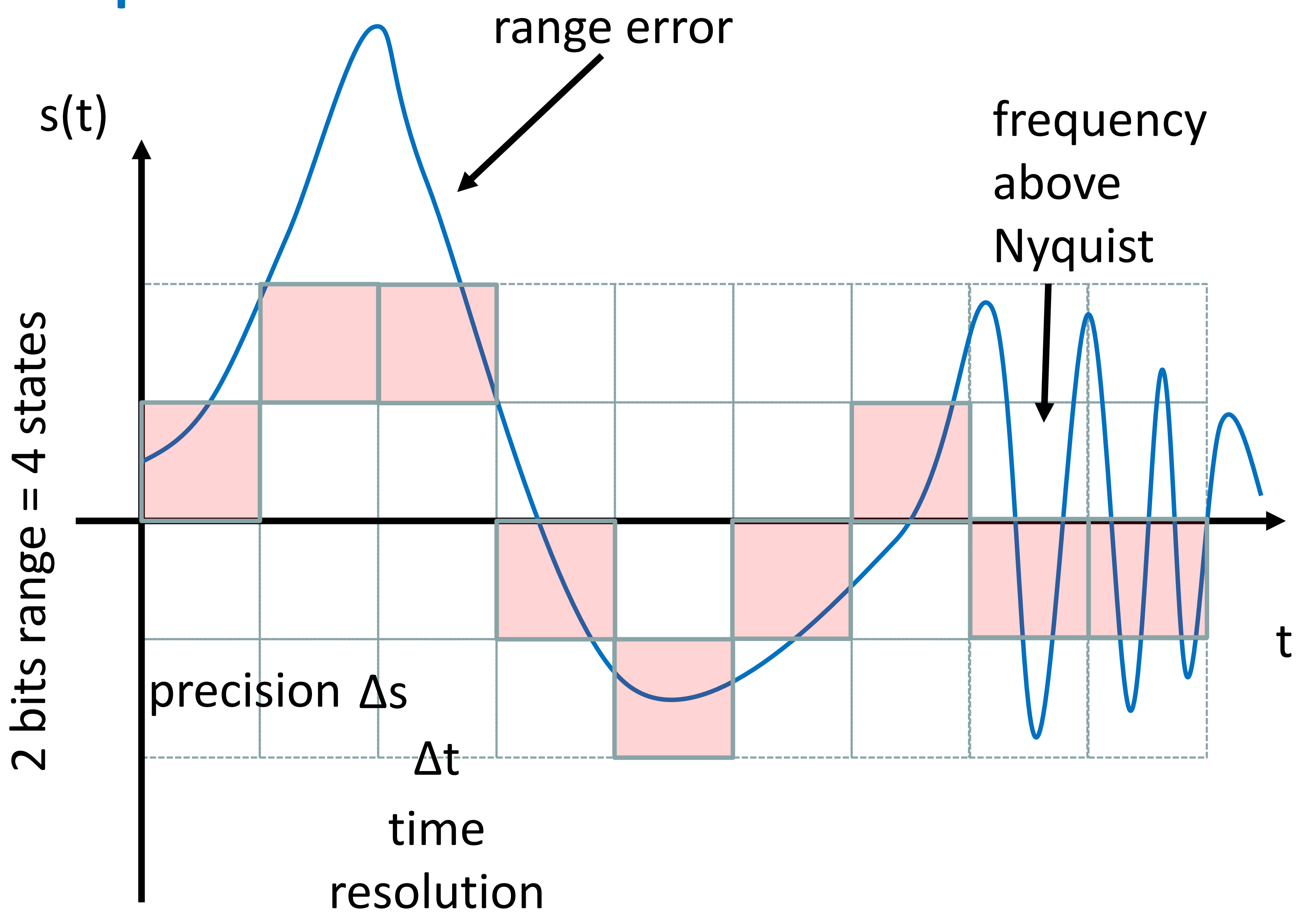
Classification

ROC, k-NN, SVM

Fiddling with signals and frequencies

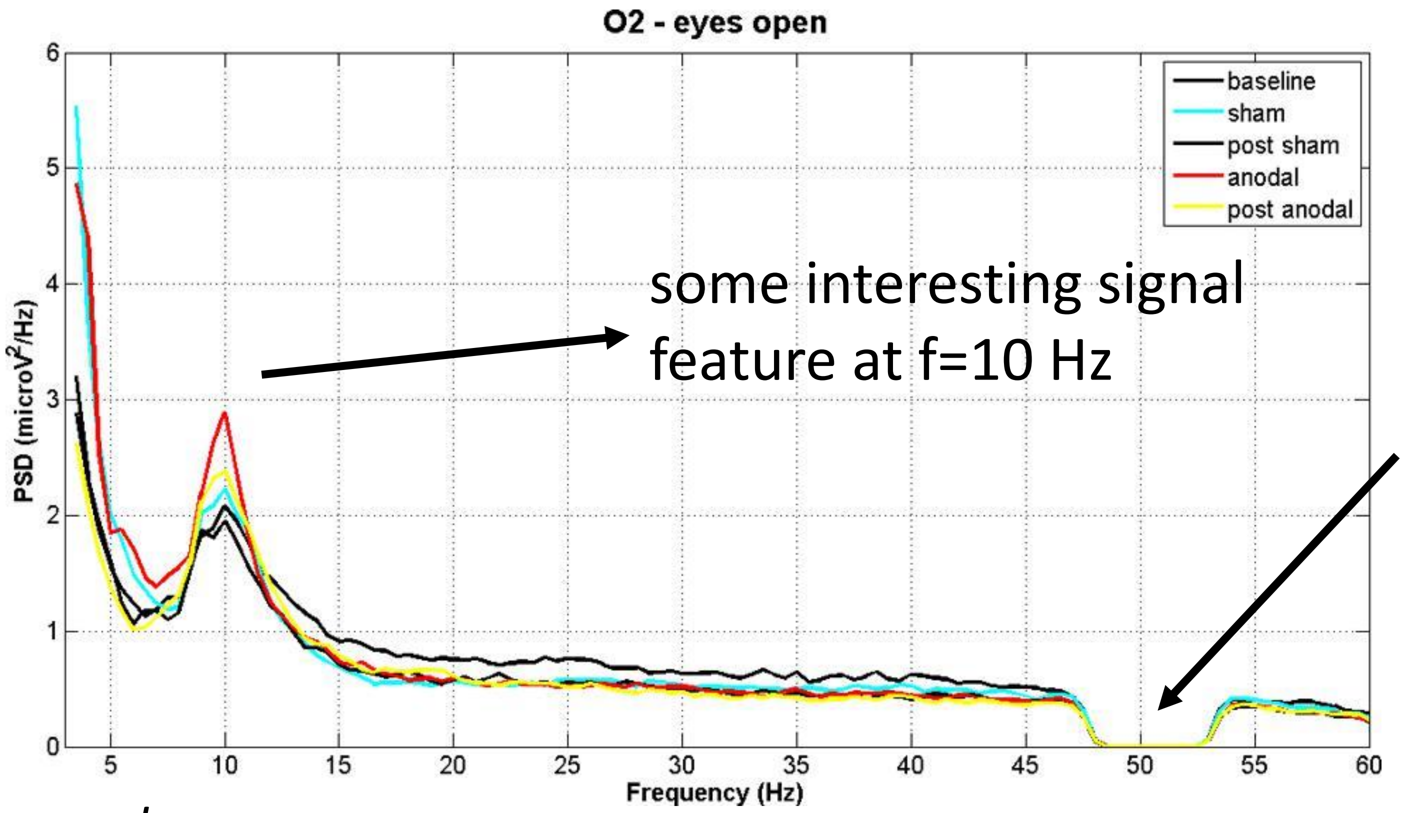
Sampling

Sampled signals have a limited time resolution and a limited range and precision



Reminder: the Fourier transform

Signals s(t) can also be represented in Fourier space as **complex coefficients S(f)**. Transform forth and back by (inverse) **Fourier transform**. Visualize a Fourier-transformed signal as **power spectral density** (remember lecture/exercise in Theo. Neurosciences):



some interesting signal feature at f=10 Hz

…there‘s a hole in the bucket, dear Liza, dear Liza!

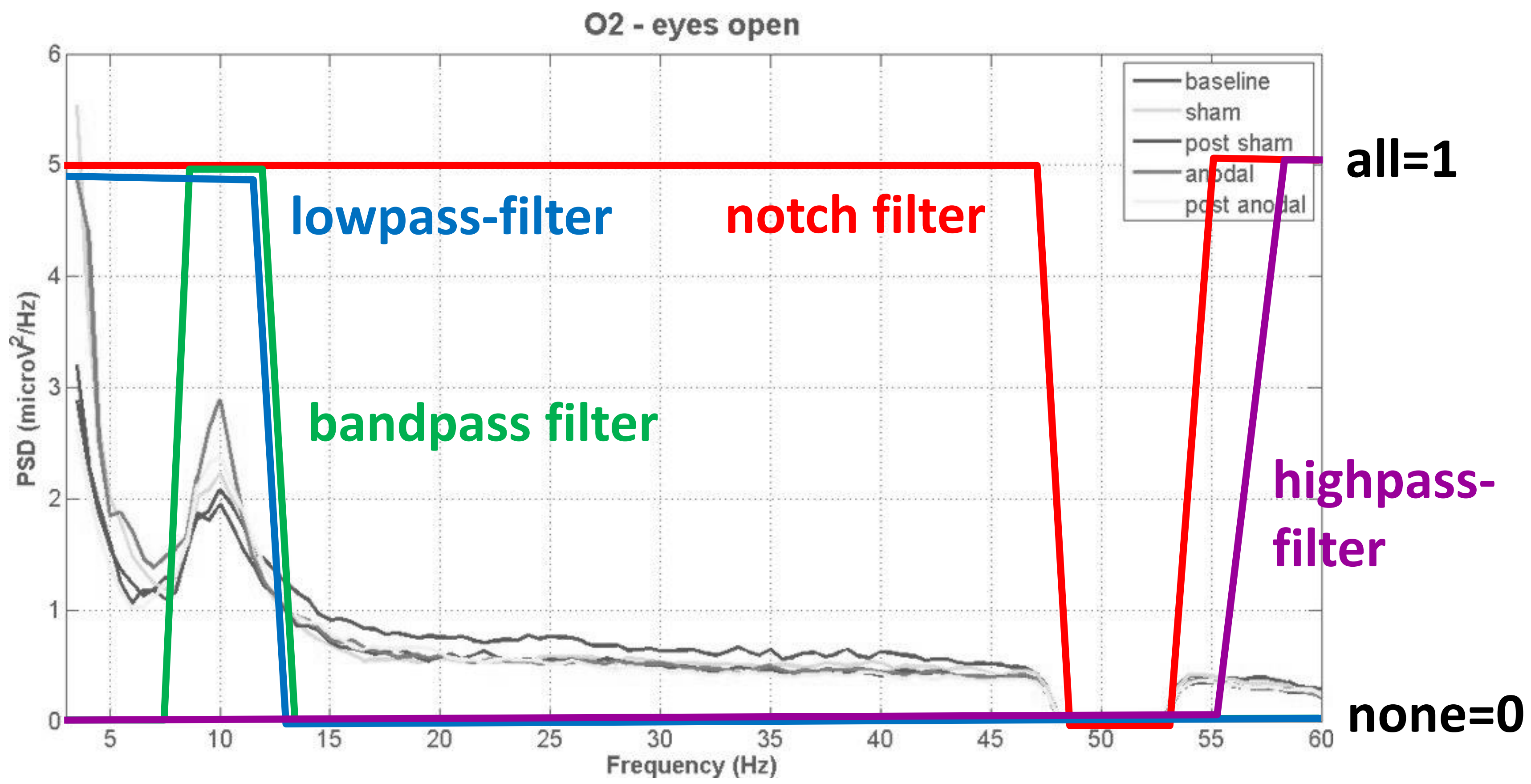
https://en.wikipedia.org/wiki/There%27s_a_Hole_in_My_Bucket

source: neuroimage.usc.edu

How can we extract signals at frequency ranges of interest, or put holes in the spectrum of the data?

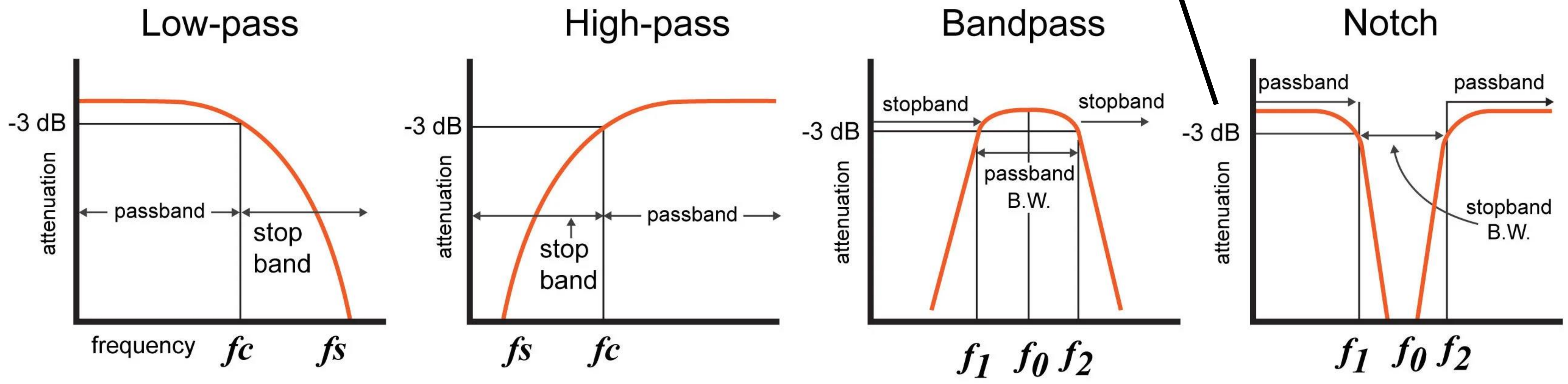
→ **Filters!**

Visualizing in frequency space what a filter does…



Names and props

amplitude drops to ~70%



https://www.allaboutcircuits.com/technical-articles/an-introduction-to-filters/

dB = decibel: defined as **10 log(P2/P1) dB** for **power ratio** P2 vs. P1

…therefore, **20 log(A2/A1) dB** for **amplitude ratios**!

(note that mathematical „log“ is numpyically „log10“!)

Quality factor:

Q=f0/(f2-f1)

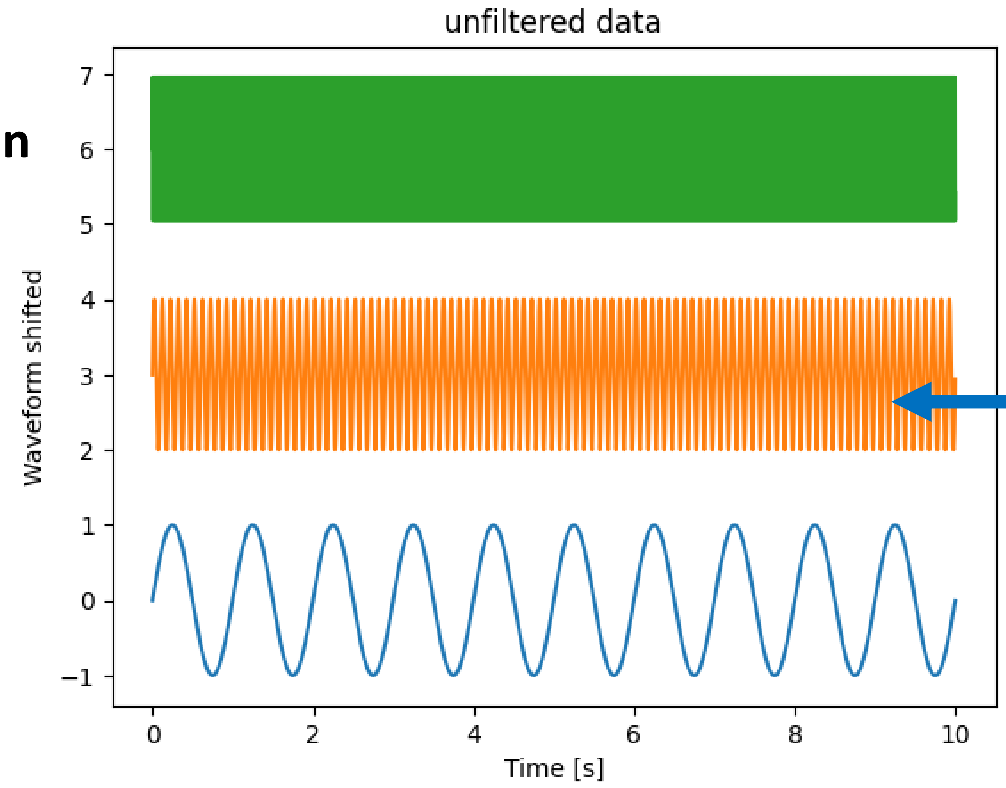
Filter order:

~ slope of decay

Filtering with Python

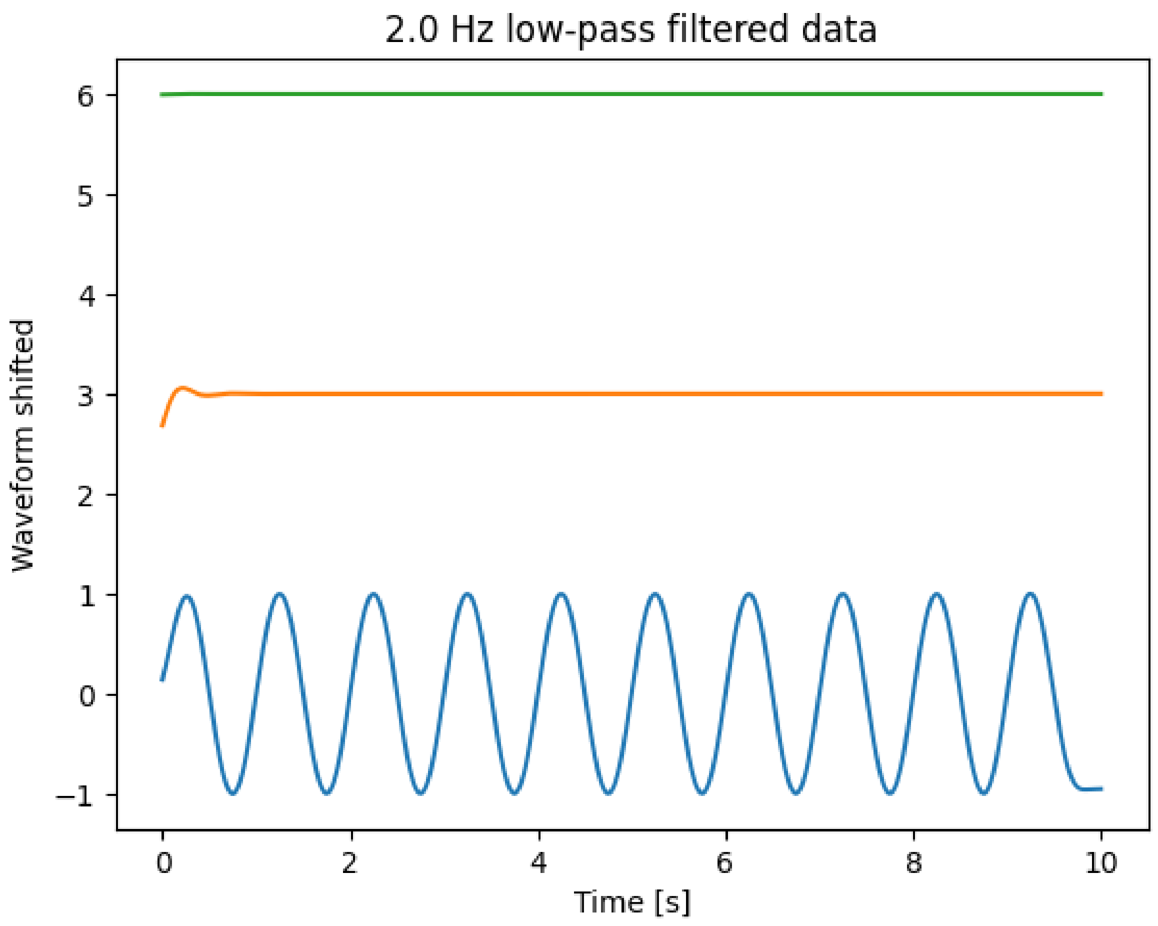
We like the butterworth filter provided by the **scipy.signal** module. One uses **butter** to construct the filter, and **filtfilt** to apply the constructed filter to a time series:

Sample signals used in the following slides:



Low-pass filter

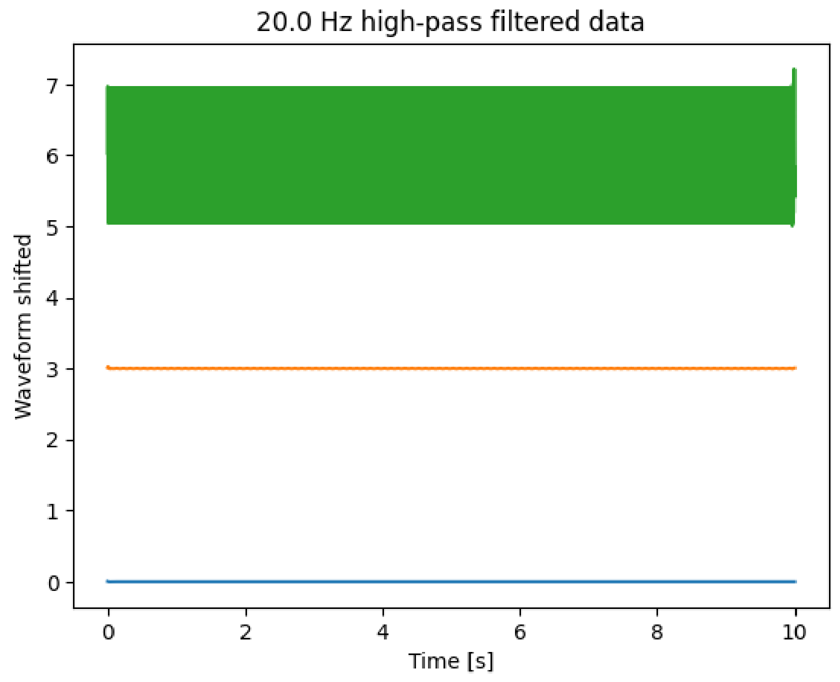
```python
from scipy import signal

lowpass_frequency: float = 2.0  # Hz

# Nint : The order of the filter.
# Wn : The critical frequency or frequencies. For lowpass anc
ers, Wn is a length-2 sequence.
# For a Butterworth filter, this is the point at which the ga
# For digital filters, if fs is not specified, Wn units are r
us in half cycles / sample and defined as 2*critical frequenc
# For analog filters, Wn is an angular frequency (e.g. rad/s)
# btype{'lowpass', 'highpass', 'bandpass', 'bandstop'}, optic
# fs float, optional : The sampling frequency of the digital
b_low, a_low = signal.butter(
    N=4, Wn=lowpass_frequency, btype="lowpass", fs=samples_pe
)
sin_low_lp = signal.filtfilt(b_low, a_low, sin_low)
sin_mid_lp = signal.filtfilt(b_low, a_low, sin_mid)
sin_high_lp = signal.filtfilt(b_low, a_low, sin_high)
```

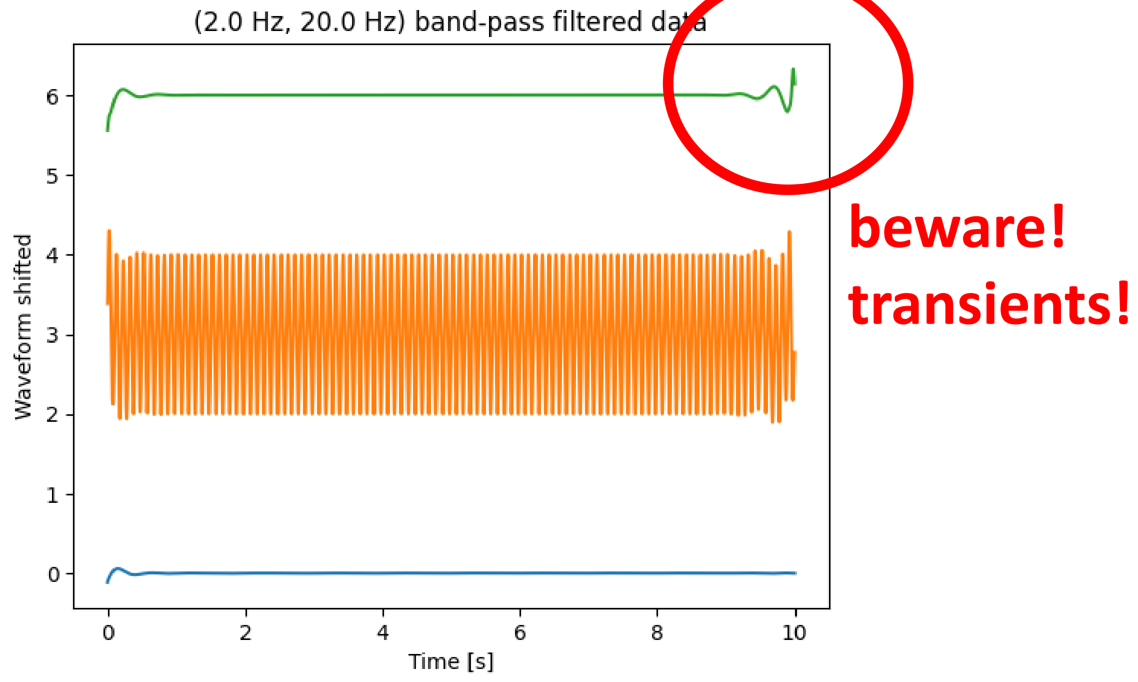


High-pass and bandpass

```
highpass_frequency: float = 20.0   # Hz

b_high, a_high = signal.butter(
    N=4, Wn=highpass_frequency, btype="highpass",
fs=samples_per_second
)
```

```
lowpass_frequency: float = 2.0   # Hz
highpass_frequency: float = 20.0   # Hz

b_band, a_band = signal.butter(
    N=4,
    Wn=(lowpass_frequency, highpass_frequency),
    btype="bandpass",
    fs=samples_per_second,
)
```

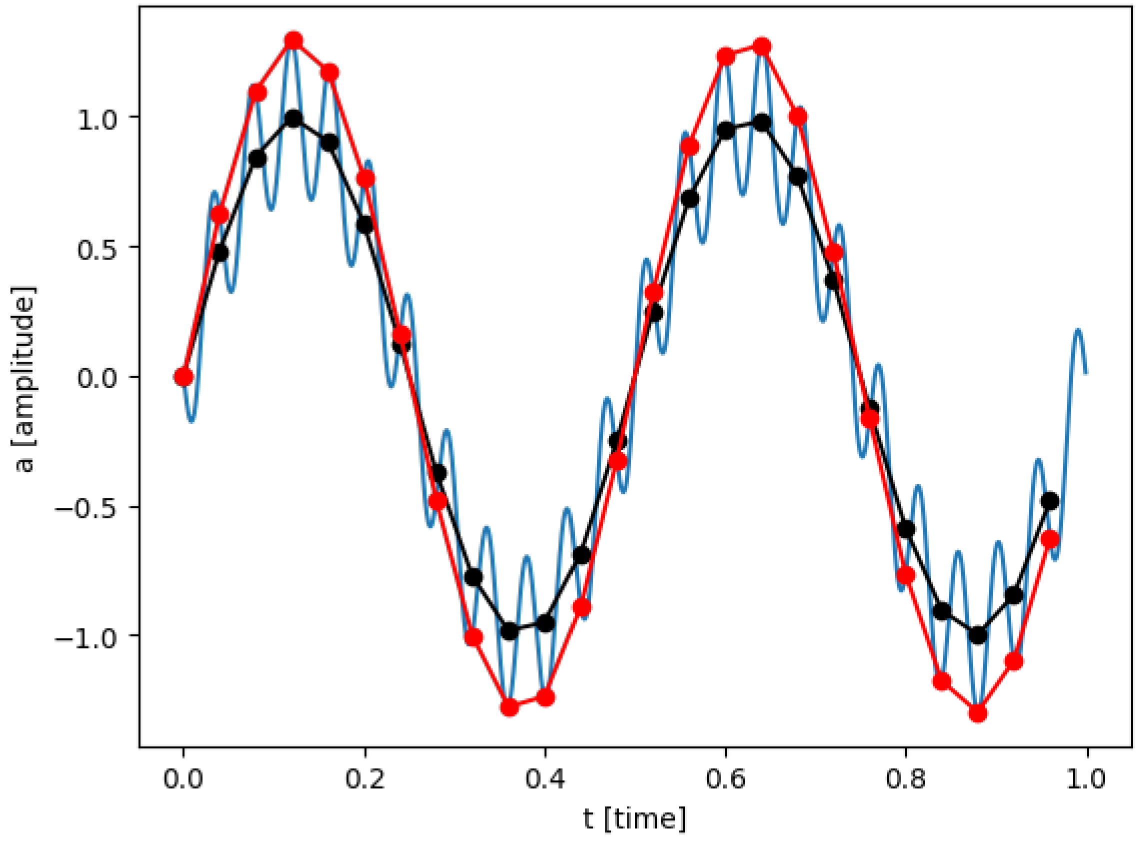




Filter caveats!

- Filters imply phase shifts. To compensate, combine for- and backward filtering. → Live coding
- Filter first before downsampling (see example).
- To inspect a filter, filter a white noise signal and plot PSD.
- Take care, transients at start and end of signal
- The more parameter you specify, the more difficult is it to design a filter

$$a(t) = \sin(2 * 2\pi t) - 0.3\sin(23 * 2\pi t)$$



blue: original signal, sampled at 2500 Hz
red: downsampled to 25 Hz
black: first filtered, then downsampled to 25 Hz

Kernels as filters

Convolutions can act as filters on time series. The kernel $k(\tau)$ determines filter properties.

$$s_f(t) = \int k(\tau)s(t-\tau)d\tau$$

The convolution theorem states that in Fourier space, convolutions are expressed by multiplication of the transformed signal and filter.

$$\tilde{s}_f(\omega) = \tilde{k}(\omega)\tilde{s}(\omega)$$

If you transform a filter into Fourier space, you can investigate its properties by considering it a ‚mask‘ for your time series representation.

You can use the convolution theorem to perform convolutions efficiently, using FFT.

$$s_f(t) = \tilde{k}(\widetilde{\omega)\tilde{s}}(\omega)$$

Example: low-pass filter

More information:

https://davrot.github.io/pytutorial/scipy/scipy.signal_butterworth/

Spectral analysis

→ **ANDA tutorial…**

Switch presentations

Wavelet Transform in Python

One can use the **pywt** module, and requires essentially only two commands for creating a ‚mother wavelet‘ and applying it to the time series of interest:

```
# The wavelet we want to use...
mother = pywt.ContinuousWavelet("cmor1.5-1.0")
# ...applied with the parameters we want:
complex_spectrum, frequency_axis = pywt.cwt(
    data=test_data, scales=wave_scales, wavelet=mother, sampling_period=dt
)
```

However, working with the wavelet transform requires to think about the scales or frequency bands, their spacing, proper definition of time/frequency resolution, taking care of the cone-of-interest etc…

Full code at: https://davrot.github.io/pytutorial/pywavelet/

More information:

https://davrot.github.io/pytutorial/pywavelet/

Multidimensional representations

Introduction

Neural recordings often yield a large number of signals $x_i(t)$.

Typically, these signals contain a mixture of (internal and external) sources $s_j(t)$.
Example: One EEG signal contains the activity of millions of neurons.

Goal: find the neural sources s(t) contained in the signals x(t)

Also:

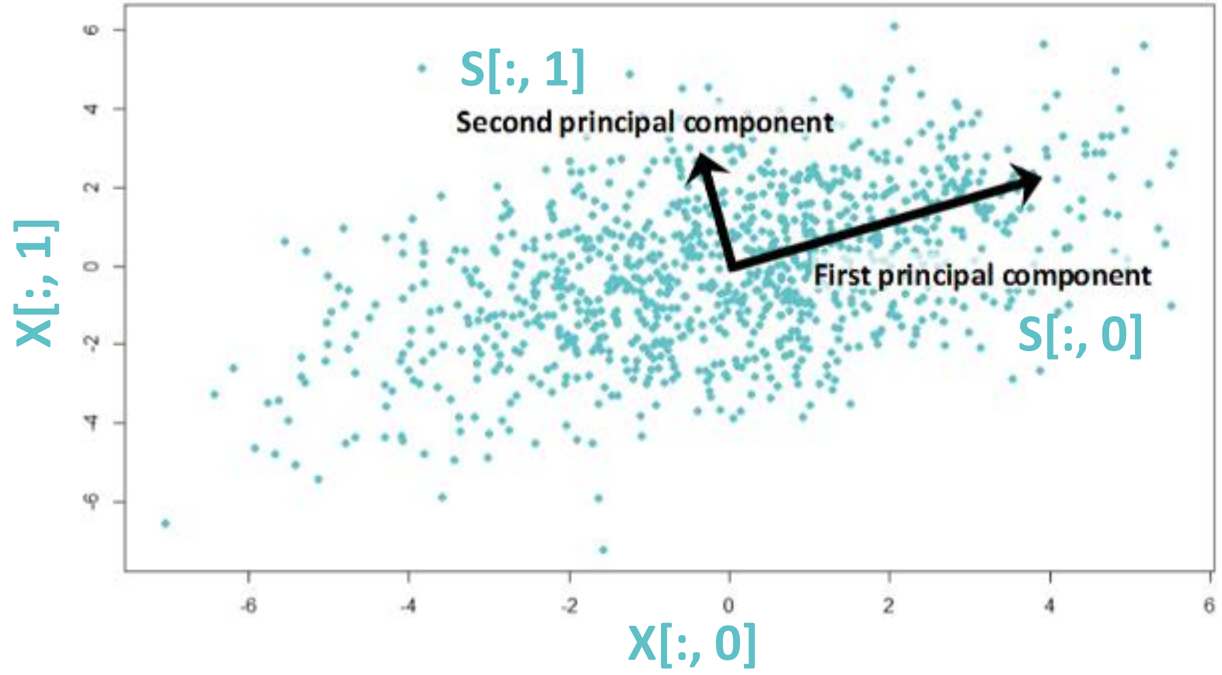
- Assessment of dimensionality of a representation
- Dimensionality reduction. Get the principal components.
- Remove common sources (common reference, line noise, heartbeat artifacts, etc.)
- ...

PCA – principal component analysis

Find sources which are uncorrelated with each other. Uncorrelated means that the source vectors S will be orthogonal to each other.

PCA finds matrix W_{PCA} such that X is explained by $X = S W_{PCA}$.

$W_{PCA}^{-1} = W_{PCA}^{T}$, so $S = X W_{PCA}^{T}$



Example: n signals of duration t:

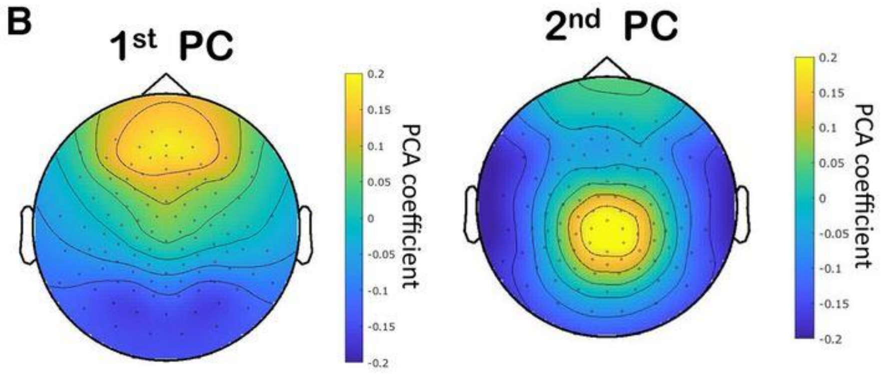
S: (t x n) – n source vectors

W_{PCA}: (n x n) – mixture matrix

X: (t x n) – n observation vectors

Visualization:

$W_{PCA}[k, :]$ shows how the k-th component contributes to the n observations:



PCA – principal component analysis: Python

Use **class PCA** from **sklearn.decomposition** module:

```
class sklearn.decomposition.PCA(n_components=None, *, copy=True, whiten=False, svd_solver='auto', tol=0.0, iterated_power='auto', n_oversamples=10, power_iteration_normalizer='auto', random_state=None)
```

- After defining an instance, you can use **fit** for fitting a transform, and **transform** for transforming X to S.
- **fit_transform** combines these steps, and **inverse_transform** does the transfrom from S to X.
- The attribute **components_** will contain the PCA transformation components
- Components will be sorted with descending (explained) variance.

```
from sklearn.decomposition import PCA

# transform x to s
pca = PCA()
s = pca.fit_transform(x)
w_pca = pca.components_

# transform s to x
x_recover = pca.inverse_transform(s)
also_x_recover = s@w_pca
```

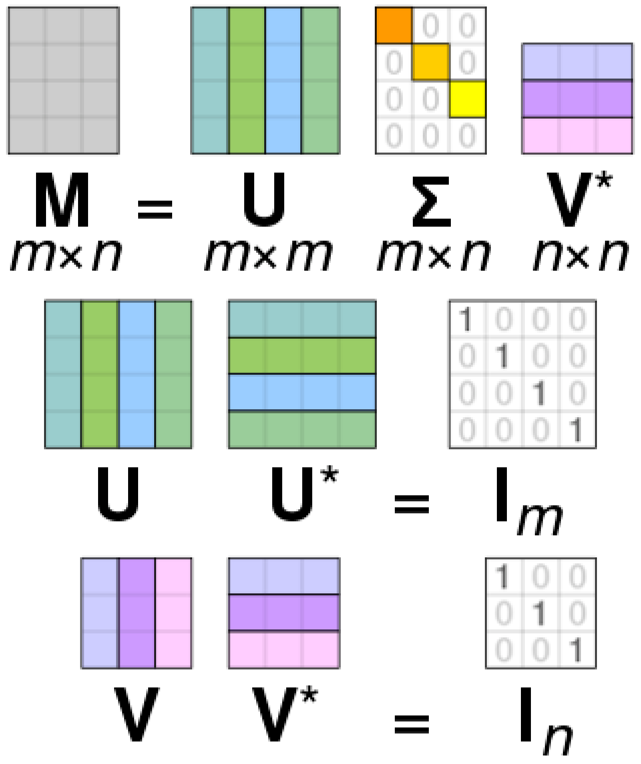
Take care! Instead of $X = S\, W_{PCA}$, the transform is also often defined as $X' = W_{PCA}\, S'$. This makes X', S' (n x t) instead of (t x n) matrices!

SVD – Singular Value Decomposition

The singular value decomposition decmposes a matrix **M** into two unitary matrices U and V, and a diagonal matrix ∑: **$M = U \Sigma V^*$**

Assumptions are $U^T U = UU^T = I$, and $V^T V = VV^T = I$ with I being the unit matrix.

Relation to PCA: Consider m denotes ‚time' t, and n <=t. Then M are the observations X, V* will be W_{PCA}, and $S = U \Sigma$ the uncorrelated principal components, related via: $\mathbf{X = S\, W_{PCA}}$.



ICA – independent component analysis

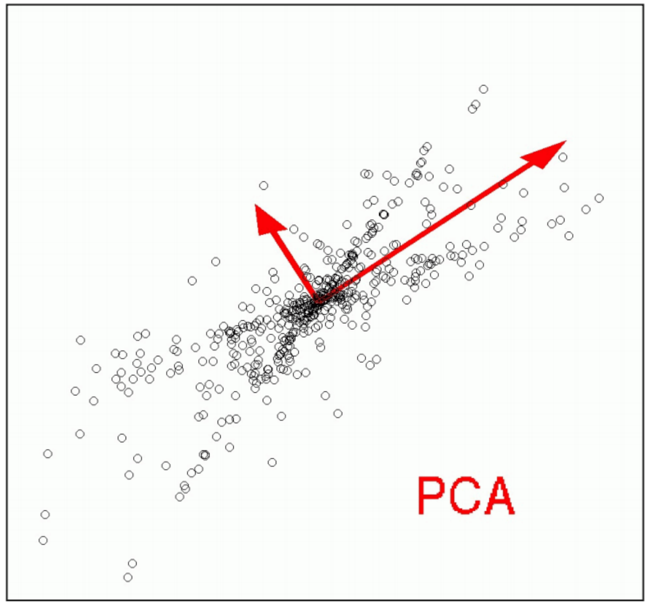
ICA assumes also a linear mixture of ‚sources' via $X = S\,W_{ICA}$. However, here the goal is to find sources which are statistically independent to each other.

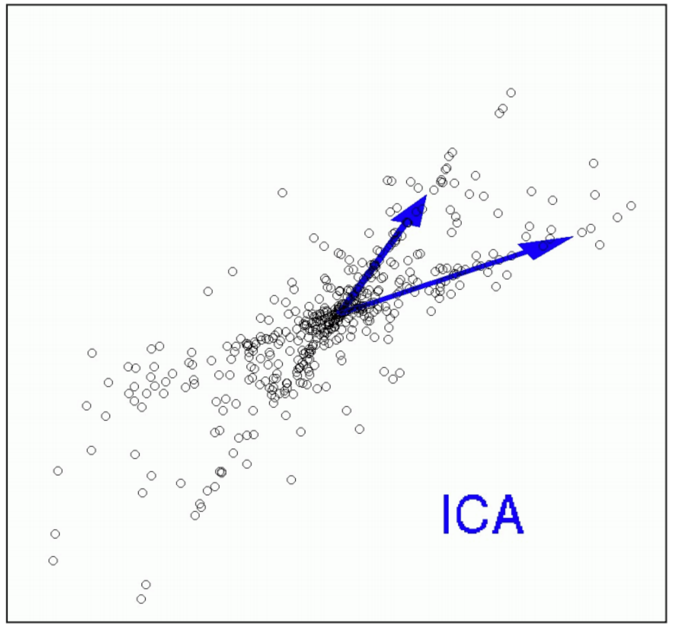
The ICA transform is not unique and depends on the independence criterion!

When might ICA be more appropriate than PCA?
→ **Example:**

Independence criteria:

- minimization of mutual information
- maximization of non-Gaussianity





ICA – independent component analysis: Python

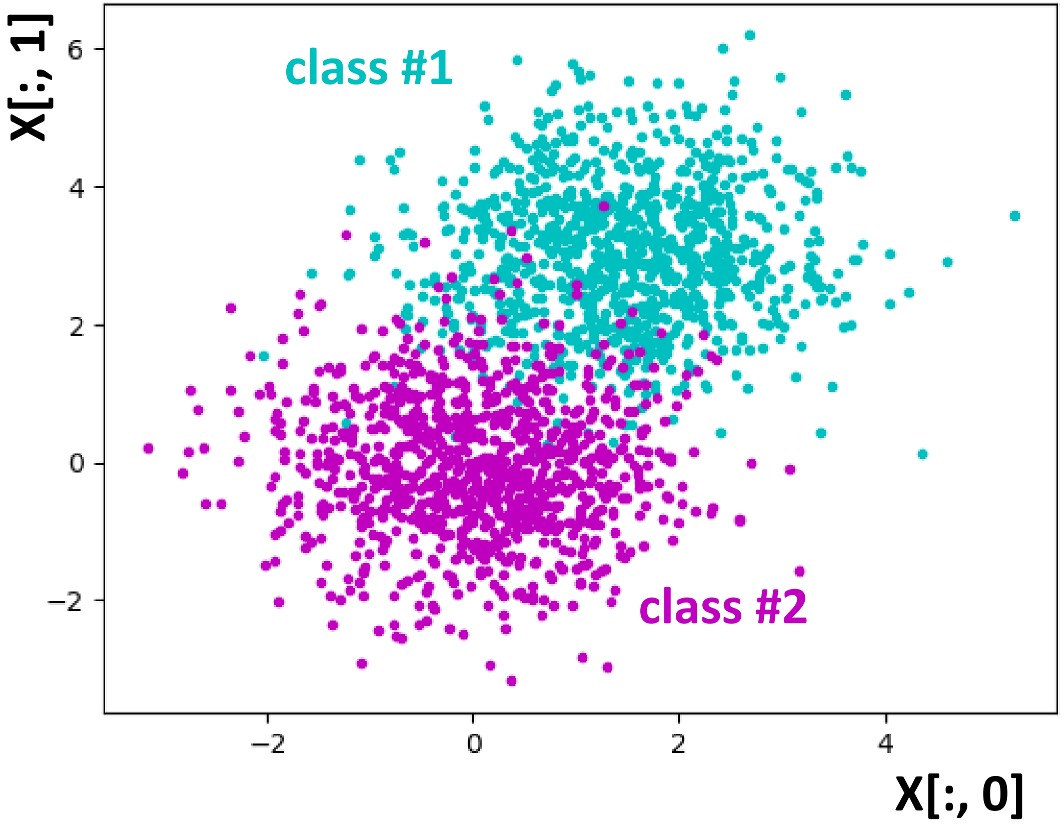
Use **class FastICA** from **sklearn.decomposition** module. The usage is very similar to **PCA**.

```python
from sklearn.decomposition import FastICA

# transform x to s
ica = FastICA()
s = ica.fit_transform(x)
w_ica = ica.components_

# transform s to x
x_recover = ica.inverse_transform(s)
```

Clustering

Motivation

We have **multidimensional samples X** and expect that they stem from different ‚classes', e.g. spike waveforms where spikes from one particular cell constitute one class. Samples from a particular class should have smaller distance than samples stemming from different classes:



The k-means Clustering Algorithm

Given an initial set of k means $m_1^{(1)}$, ..., $m_k^{(1)}$ (see below), the algorithm proceeds by alternating between two steps:[7]

1. **Assignment step**: Assign each observation to the cluster with the nearest mean: that with the least squared Euclidean distance.[8] (Mathematically, this means partitioning the observations according to the Voronoi diagram generated by the means.)

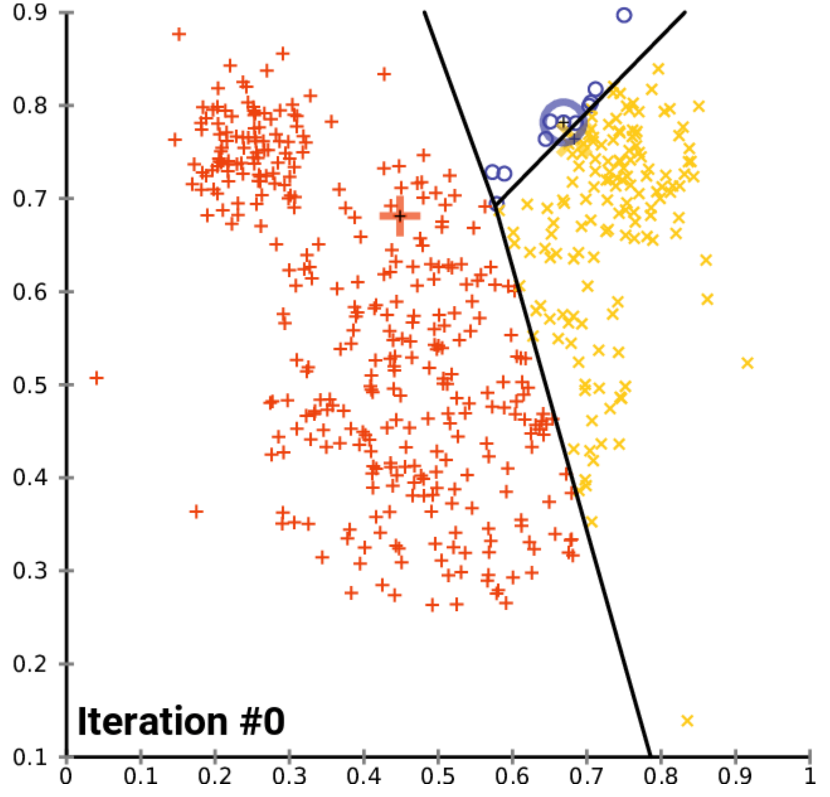
$$S_i^{(t)} = \left\{ x_p : \left\| x_p - m_i^{(t)} \right\|^2 \le \left\| x_p - m_j^{(t)} \right\|^2 \ \forall j, 1 \le j \le k \right\},$$

   where each x_p is assigned to exactly one $S^{(t)}$, even if it could be assigned to two or more of them.

2. **Update step**: Recalculate means (centroids) for observations assigned to each cluster.

$$m_i^{(t+1)} = \frac{1}{\left|S_i^{(t)}\right|} \sum_{x_j \in S_i^{(t)}} x_j$$

description and animation from Wikipedia



The k-means Clustering Algorithm: Python

sklearn.cluster.KMeans and its fit

```
class sklearn.cluster.KMeans(n_clusters=8, *, init='k-means++', n_init='warn', max_iter=
300, tol=0.0001, verbose=0, random_state=None, copy_x=True, algorithm='lloyd')
```

K-Means clustering.

Attribute:

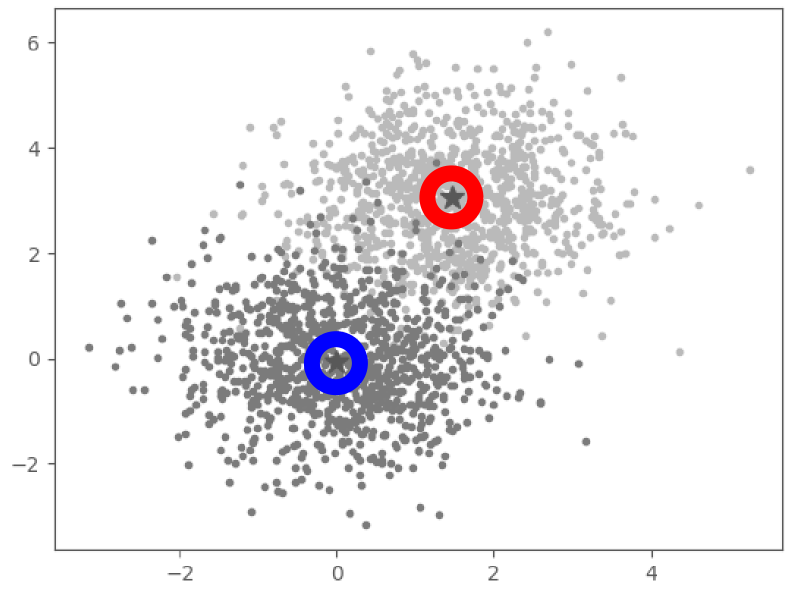
cluster_centers_ : ndarray of shape (n_clusters, n_features) Coordinates of cluster centers. If the algorithm stops before fully converging (see tol and max_iter), these will not be consistent with labels_.

Method:

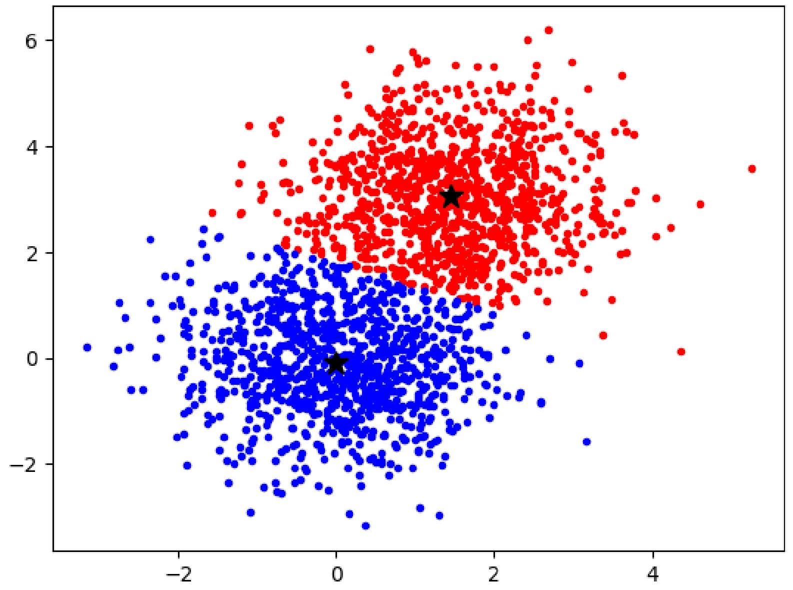
```
fit(X, y=None, sample_weight=None)
```

Compute k-means clustering **X**: {array-like, sparse matrix} of shape (n_samples, n_features) Training instances to cluster. It must be noted that the data will be converted to C ordering, which will cause a memory copy if the given data is not C-contiguous. If a sparse matrix is passed, a copy will be made if it's not in CSR format.

cluster_centers_



labels_



More information:

https://davrot.github.io/pytutorial/scikit-learn/overview/
https://davrot.github.io/pytutorial/scikit-learn/pca/
https://davrot.github.io/pytutorial/scikit-learn/fast_ica/
https://davrot.github.io/pytutorial/scikit-learn/kmeans/

Classification

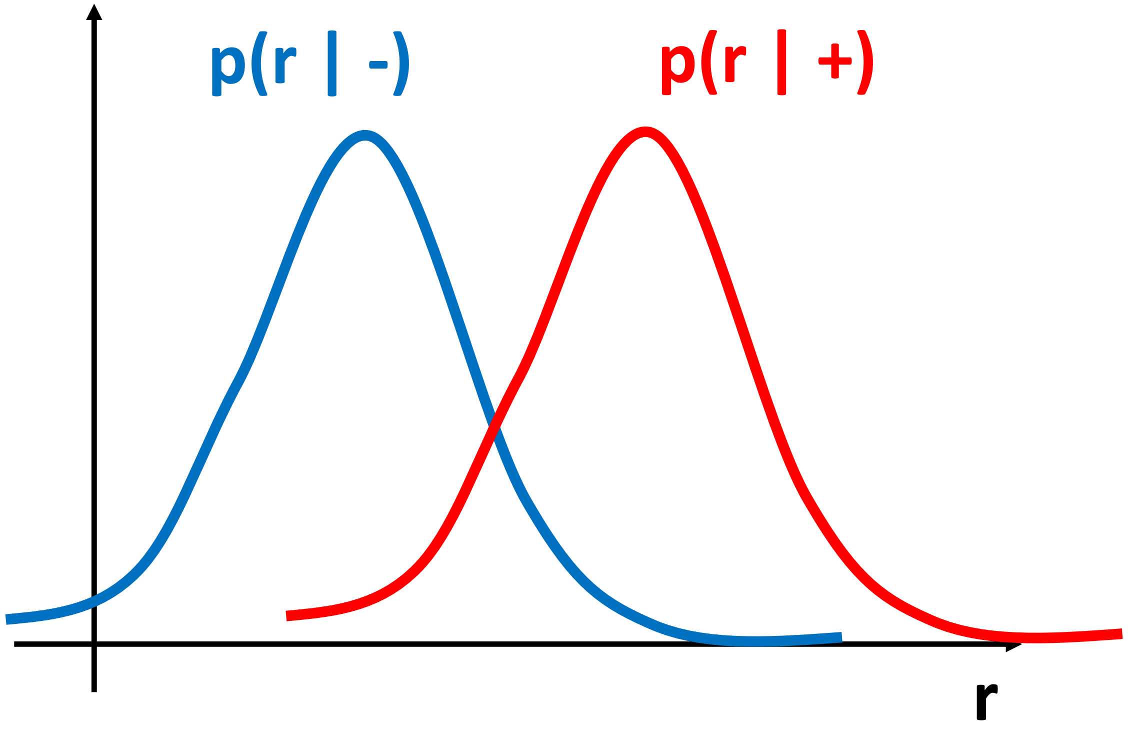
Classification yields information about information in data…

- **Receiver-operator-characteristics (ROC):** a simple tool for quick inspection for both simple and complex data sets
- **K-nearest-neighbor classifier (kNN):** easy to implement, suited for a quick inspection
- **Support vector machine (SVM):** an almost state-of-the-art tool for (non-)linear classification of large data sets. Very useful if you don‘t want to fire up your deep network and NVidia GPU for every almost trivial problem…

Important: For classification, you need a **training data set**, and a **test data set**. Each data set contains (a large number of) **samples** together with their **labels**. You are **not allowed to use the test set for training**.

Receiver-Operator Characteristics

The situation: one recorded signal r, two potential causes „+“ or „-“:

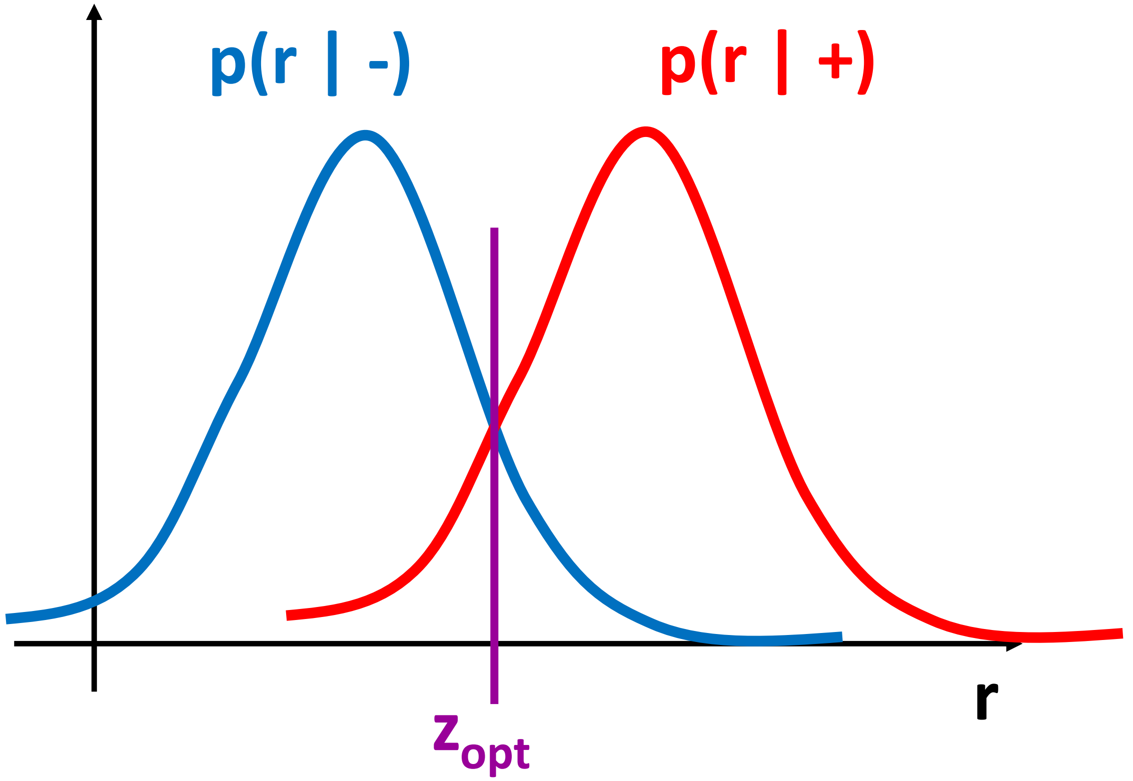


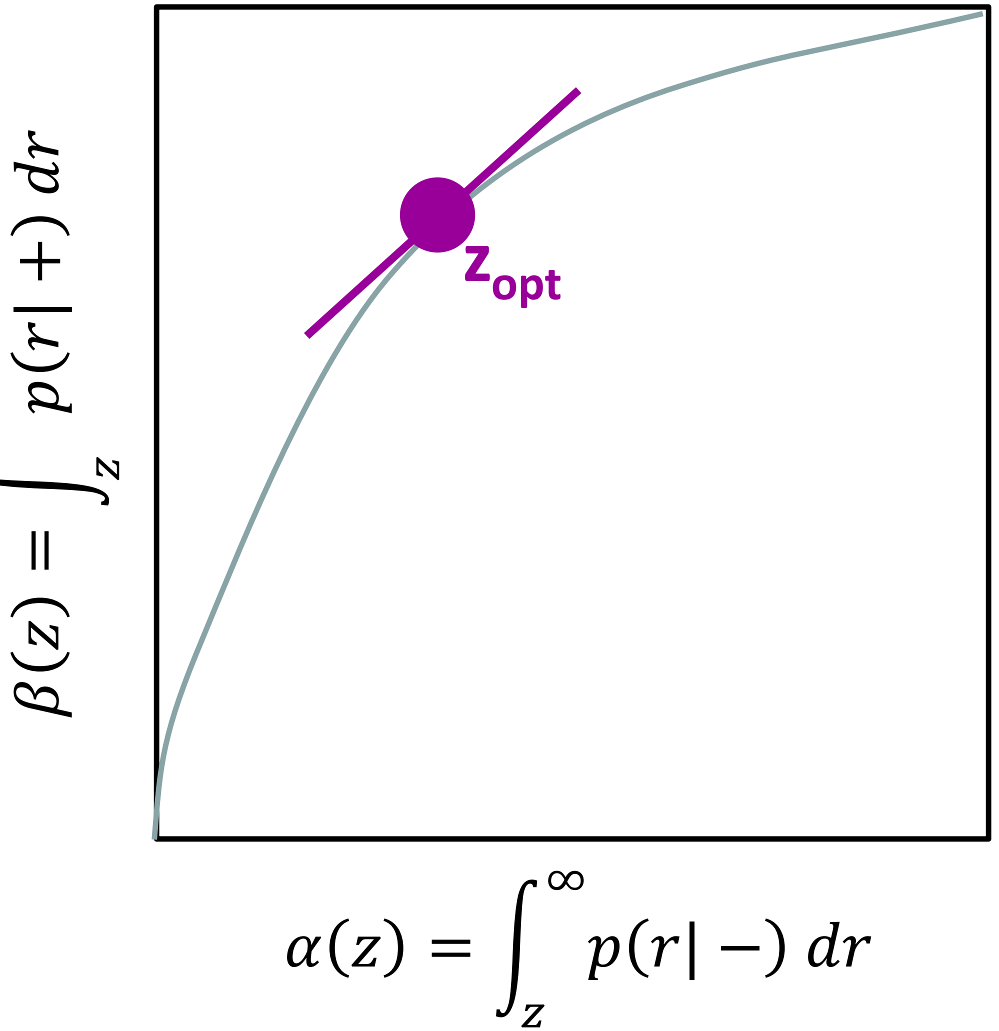
How can we distinguish between „+“ and „-“?

Simplest estimator: use threshold z, if sample r_0 is smaller than z, attribute to „-“, otherwise to „+“

→ **Whiteboard!**

Can we find an **optimal z**? Yes, the idea is to plot the **true positives (β)** against the **false positives (α)** while changing z (ROC curve). Classification accuracy has a **maximum/minimum when the rates of change are equal** **(slope=1)**.

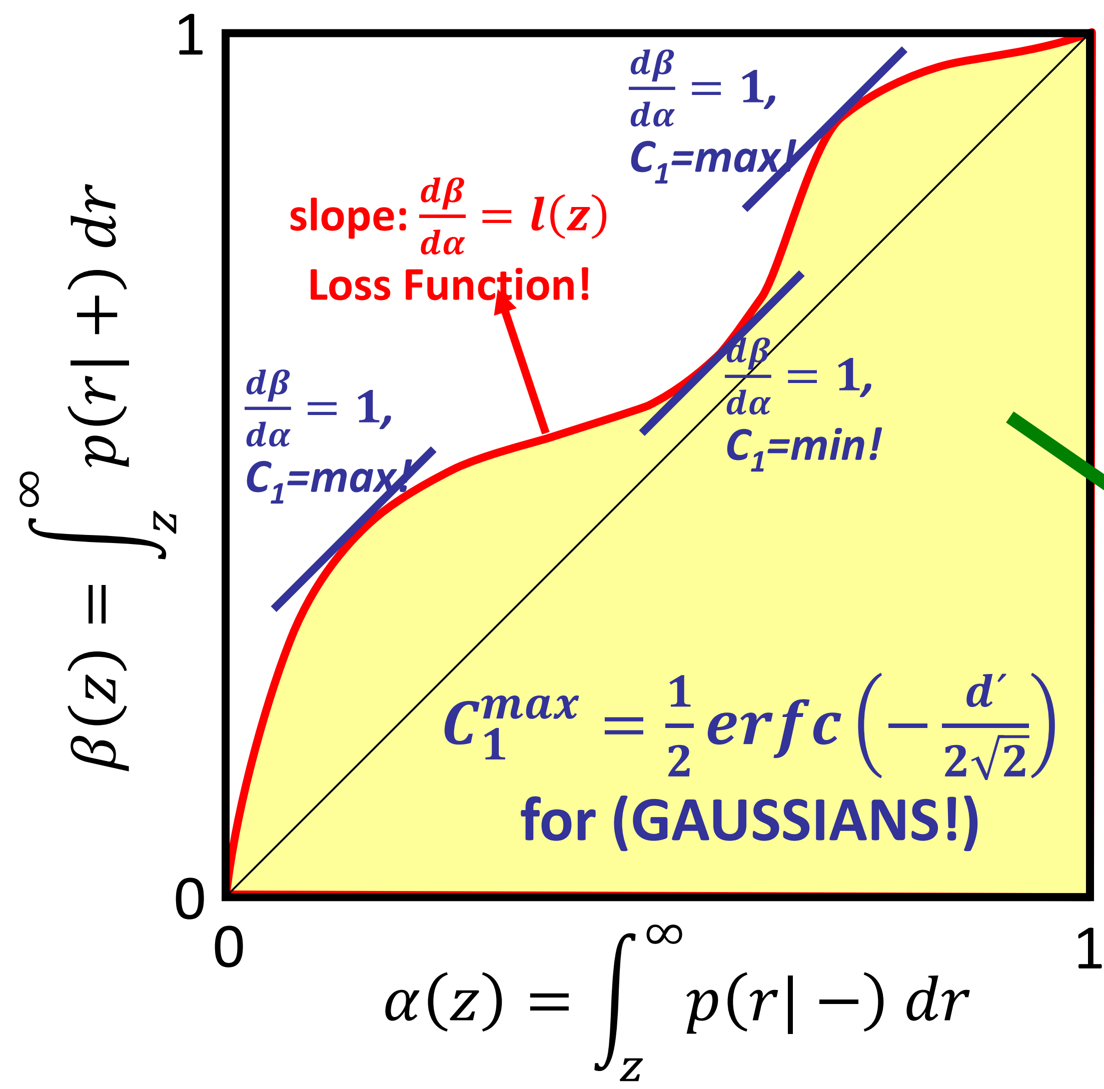




$$\beta(z) = \int_z^\infty p(r|+)\,dr$$

$$\alpha(z) = \int_z^\infty p(r|-)\,dr$$

Summary: ROC'n'Roll



…it's a nice tool for quick inspection how well a scalar variable allows to discriminate between two situations!

$$C_2 = \frac{1}{2} erfc\left(-\frac{d'}{2}\right)$$

for **GAUSSIANS!**

$$\boxed{C_2 = \int_0^1 \beta(\alpha)\,d\alpha}$$

Discriminability:
difference of means relative to std:

$d' := (r_+ - r_-)/\,\sigma$

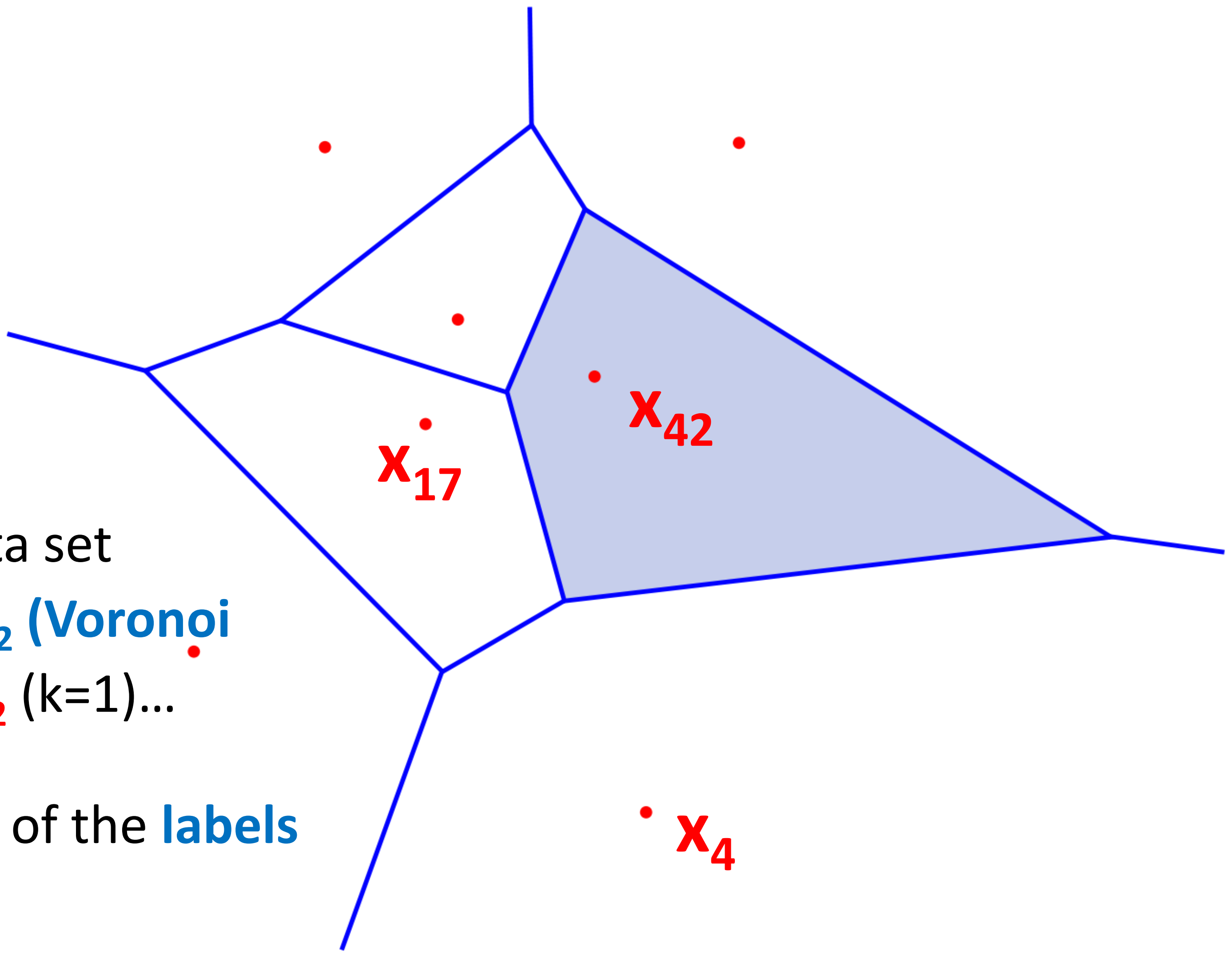
k-Nearest-Neighbour Classifier:

Super-easy to explain,
super-easy to implement,
super memory consuming!

The x_i are samples of the training data set with labels y_i.

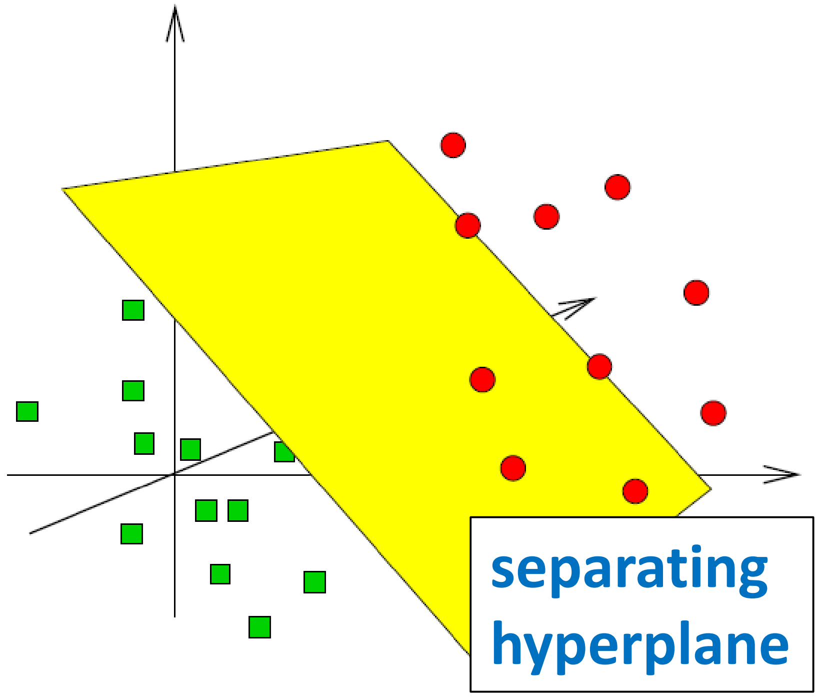
Every sample from the test data set **inside the neighborhood of x_{42} (Voronoi cell)** gets assigned the label y_{42} (k=1)…

…or the majority vote/mixture of the **labels of the k nearest neighbors**.



The support vector machine (SVM)

You know how a simple perceptron works (lecture Theoretical Neurosciences)? The SVM is doing the same thing, but **transforms the data into a higher-dimensional space** before it performs a **linear classification by using an appropriately placed separating hyperplane**:



Python tools for elementary classification tasks:

ROC and **kNN** – easy to code on your own (and a good training for you!)...

Learning an **SVM** is more tricky. scikit-learn provides you with a good tool:

```python
import numpy as np
import sklearn.svm  # type:ignore

data_train = np.load("data_train.npy")
data_test = np.load("data_test.npy")
label_train = np.load("label_train.npy")
label_test = np.load("label_test.npy")

svm = sklearn.svm.SVC()

svm.fit(X=data_train, y=label_train)
prediction = svm.predict(X=data_test)

performance = 100.0 * (prediction == label_test).sum() / prediction.shape[0]

print(f"Performance correct: {performance}%") # -> Performance correct: 95.4%
```

More information:

https://davrot.github.io/pytutorial/numpy/roc/
https://davrot.github.io/pytutorial/numpy/knn/
https://davrot.github.io/pytutorial/scikit-learn/svm/