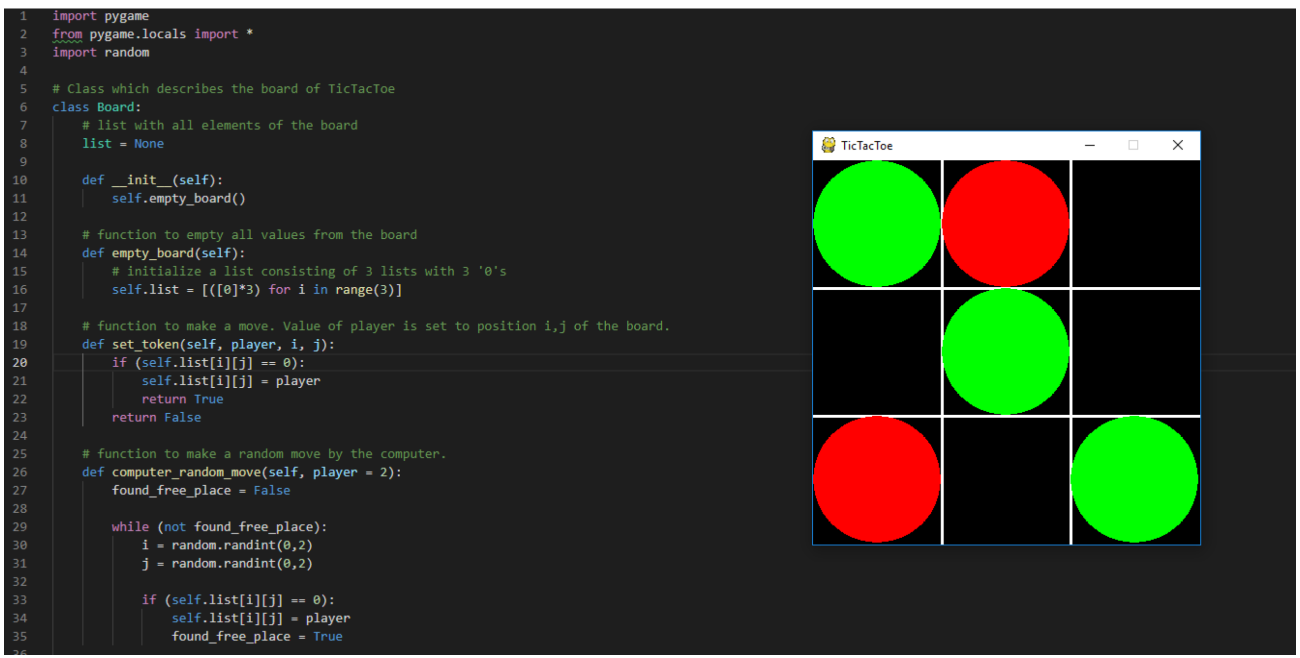
Introduction to Python Programming (3/3)

Institute of Medical Informatics

University of Lübeck

Last session

- Introduction of basic Python concepts
- Basic Python exercises
- Implementation of a Tic Tac Toe game



Contents of the session

- Array and list indexing
- **[Exercise 1]** Shuffling in unison
- Multidimensional array manipulation
- **[Exercise 2]** Standard normalization
- Evaluation of supervised learning algorithms
- **[Exercise 3]** Confusion matrix, precision and recall
- k-means clustering
- **[Exercise 4]** k-means implementation

Array and list indexing (1/4)

The elements of arrays and lists can be accessed using the **brackets** [...] notation.

Examples:

```
>>> import numpy as np
>>> T = np.arange(5) # Creates array([0, 1, 2, 3, 4])
>>> T[0] # First element of T
0
>>> T[2] # 3rd element of T
2
>>> T[-1] # Last element of T
4
>>> T[-2] # Last-but-one element of T
3
```

Array and list indexing (2/4)

It is possible to access specific elements of an array using **slice indexing.** Given an array or list T:

```
T[start:end:increment]
```

returns all elements of T between the indices `start` (included) and `end` (excluded) by intervals of `increment`.

Note 1: `start`, `end` and `increment` are all optional (integer) arguments. If not specified, `increment` is equal to 1 by default.

Note 2: this syntax still works if T is a multidimensional array

Array and list indexing (3/4)

Slice indexing examples:

```
>>> import numpy as np
>>> T = np.arange(5) # Creates array([0, 1, 2, 3, 4])
>>> T[1:4] # Elements of T between the 2nd and 4th
array([1,2,3])
>>> T[:3] # All elements of T until the 3rd one
array([0,1,2])
>>> T[2:] # All elements of T from the 3rd to the end
array([2,3,4])
>>> T[1:4:2] # All second elements of T between the 2nd and 4th
array([1,3])
>>> T[:] # Returns all elements of T; equivalent to T[::] and T
array([0,1,2,3,4])
```

Array and list indexing (4/4)

Extracting a sub-array can also be performed using **list indexing**. The list can contain either boolean or integers.

Examples:

```
>>> import numpy as np
>>> T = np.array([12,5,-3,7,24])
>>> b = [True,False,False,True,True] # List of booleans
>>> T[b]
array([12,7,24])
>>> idx1 = [3,2,0,4,1] # List of integers with same length than T
>>> T[idx1]
array([7,-3,12,24,5])
>>> idx2 = [2,3,0] # List of integers shorter than the length of T
>>> T[idx2]
array([-3,7,12])
```

Exercise 1: Shuffling in Unison

Write a Python function `shuffleInUnison` which shuffles (i.e. re-orders) the elements of two arrays of same length using the **same** random permutation:

```
shuffledT1, shuffledT2 = shuffleInUnison(T1,T2)
```

With:

- **[input]** `T1` and `T2`: arrays assumed to have same length
- **[output]** `shuffledT1` and `shuffledT2`: randomly shuffled input arrays/lists using the same permutation

Tip: for random related functions, use the `numpy.random` package

Multidimensional array manipulation (1/3)

Some useful options to initialize a multidimensional array:

- Direct initialization with `numpy.array`

```
>>> import numpy as np
>>> T = np.array([[1,2],[3,4]]) # 2x2 array containing the values 1, 2, 3, 4
```

- Initialization with numpy functions

```
>>> import numpy as np
>>> T0 = np.zeros((2,2),dtype=int) # 2x2 array to zero with integer type
>>> T1 = np.ones((100,50,200),dtype=float) # 3D array to one with float type
>>> TRand = np.random.rand(10,20) # 2x2 array of random float (default) values
```

Multidimensional array manipulation (2/3)

Multidimensional array **indexing** and **slicing** works the same way than for the 1D case. In particular, for a N-dimensional array T it is possible to use the following syntax to obtain a specific slice of T:

```
T[start1:end1:incr1,start2:end2:incr2,…,startN:endN:incrN]
```

Examples:

```
>>> import numpy as np
>>> T = np.random.rand(10,5,20,10,dtype=float) # 4D array of
random float values
>>> s1 = T[-2,3,15,0] # Float element at position (8,3,15,0)
>>> s2 = T[:,3,:,:] # 3D slice of shape (10,20,10)
>>> s3 = T[5,0:5:2,9,:] # 2D slice of shape (3,10)
>>> s4 = T[1:4,:,:,-1] # 3D slice of shape (3,5,20)
```

Multidimensional array manipulation (3/3)

Numpy arrays are **mutable**: it is possible to change their values after initialization.

Examples:

```
>>> import numpy as np
>>> T = np.zeros((2,2)) # 2x2 array of random float values
>>> T[0,0] = 1; print(T)
array([[1.,0.],
       [0.,0.]])
>>> T[:,1] += 3; print(T) # Increment by 3 the 2nd column of T
array([[1.,3.],
       [0.,3.]])
>>> T[1,:] = -2; print(T) # Set the 2nd line of T to -2
array([[1.,3.],
       [-2.,-2.]])
```

Exercise 2: Standard Normalization (1/2)

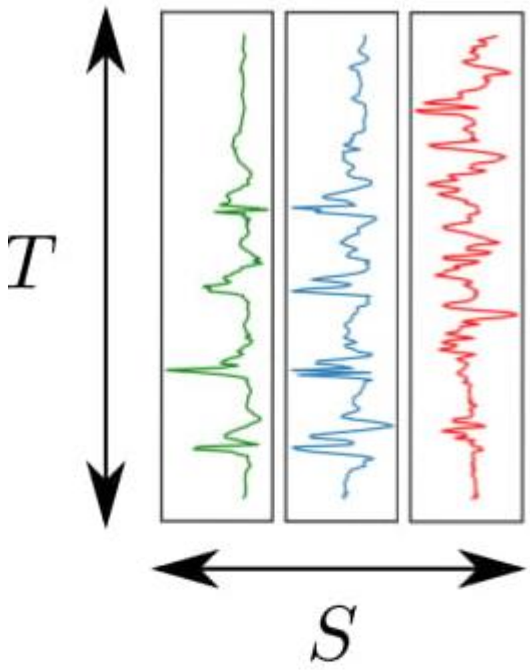
Multimodal time-series data records can be represented as a 2D array of size T x S with:

- T: number of timestamps (i.e. duration of the data record)
- S: number of sensor channels

It is usually needed to perform pre-processing operations on the raw data records. **Standard normalization** is one of them:

$$X \leftarrow \frac{X - \mu}{\sigma}$$

with $\mu = mean(X)$ and $\sigma = std(X)$



Exercise 2: Standard Normalisation (2/2)

1) Initialize a 2D array of random values between –1 and 1 - called `data` - of size `(T,S) = (1000,3)`

2) Multiply each column of `data` with the following respective coefficients: `[10,0.5,100]`

3) Write a function `standardNormalization` which takes a 2D array as input and returns the input 2D array with a standard normalization **independently** performed on all its sensor channels:

```
normalisedData = standardNormalisation(data)
```

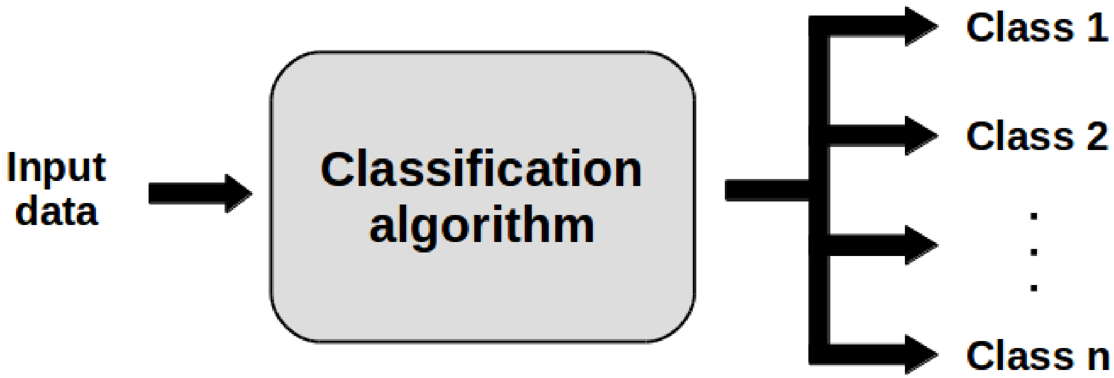
With:

- **[input]** `data`: 2D array (of implied dimensions nbTimestamps x nbSensors)
- **[output]** `normalisedData`: data with all its sensor channels standard-normalised independently

Test your function on the 2D array `data` previously defined.

Evaluation of supervised learning algorithms (1/5)

A pattern recognition problem can be translated into a **classification** problem, with **classes** = objects to recognize.

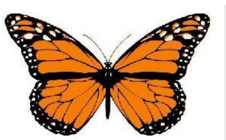


The most popular way to obtain a proper classification algorithm is to use a **supervised learning approach**.

Supervised approaches require a **training set** with its associated **labels**.

Evaluation of supervised learning algorithms (2/5)

Each element of a **labeled dataset** is associated to one **label**. A label indicates **to which class the example belongs to**.

Training examples	Associated labels
	butterfly
	dolphin
	leopard
•••	•••

Example: Caltech101 dataset

Integers between 0 and N-1 are usually used as labels, where N is the total number of classes.

For each example, the classifier outputs a **prediction** which is also an integer between 0 and N-1.

Evaluating the classifier can be done by **comparing the "true" labels to the estimated ones**.

Evaluation of supervised learning algorithms (3/5)

One handy tool for the evaluation of classification algorithms: **confusion matrix.**

A confusion matrix **counts and compares** the estimations of a trained classification algorithm to the true labels.

Estimated Labels

True Labels

	Class 1	**Class 2**	**...**	**Class n**
Class 1	**793**	34	...	29
Class 2	105	**567**	...	78
...	...	...	...	...
Class n	87	14	...	**1025**

Evaluation of supervised learning algorithms (4/5)

It is possible to compute evaluation metrics from a confusion matrix. For a **binary** classification problem:

- **Accuracy** = (TP+TN) / (TP+TN+FP+FN)
- **Precision** = p = TP / (TP+FP)
- **Recall** = r = TP / (TP+FN)
- **F1-score** = 2*p*r / (p+r)

TP = True Positive
FN = False Negative
TN = True Negative
FP = False Positive

Estimated Labels

True Labels	Class +	Class -
Class +	**TP**	FN
Class -	FP	**TN**

Evaluation of supervised learning algorithms (5/5)

For a **multiclass** classification problem (n classes with n≥3), the **overall precision** (**recall**) can be obtained by averaging class precisions (respectively class recalls) over the n classes.

The **accuracy** is provided by **Trace(M) / Sum(M)** with M confusion matrix.

Estimated Labels

M =

	...	Class i	...	Class n
...	**...**	FP_i	...	...
Class i	FN_i	$\mathbf{TP_i}$	FN_i	FN_i
...	...	FP_i	...	...
Class n	...	FP_i	...	**...**

True Labels

Exercise 3: Confusion matrix, precision, recall (1/2)

Write a function `confusionMatrix` which computes and returns a confusion matrix given two vectors of true and estimated labels:

```
confMat = confusionMatrix(nbClasses,trueLabels,estimatedLabels)
```

With:

- **[input]** `nbClasses`: integer giving the total number of classes
- **[input]** `trueLabels`: vector of integers containing true labels (elements assumed to be between 0 and `nbClasses-1`)
- **[input]** `estimatedLabels`: vector of integers containing estimated labels (assumed to have the same length as `trueLabels`; elements between 0 and `nbClasses-1`)
- **[output]** `confMat`: 2D array of size `nbClasses x nbClasses` containing the confusion matrix of the classification problem

Exercise 3: Confusion matrix, precision, recall (2/2)

Write the functions `precision` and `recall` which compute and returns the overall precision and recall given a confusion matrix:

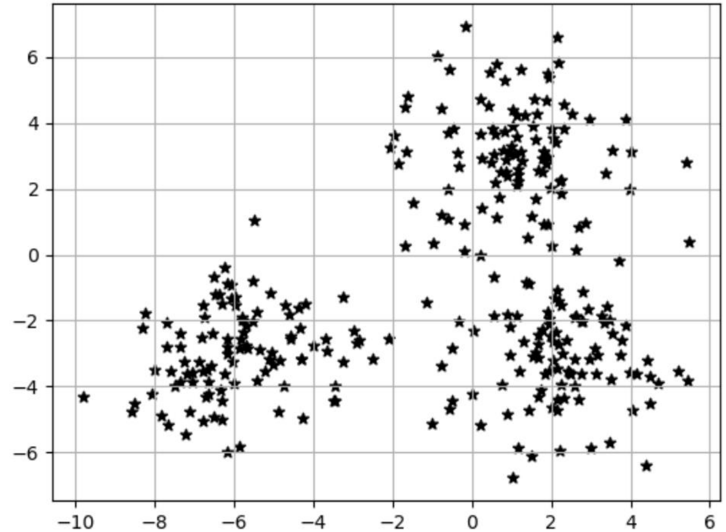
```
p = precision(confusionMatrix)
r = recall(confusionMatrix)
```

With:

- **[input]** `confMat`: 2D array of size `nbClasses x nbClasses` containing the confusion matrix of the classification problem
- **[output]** `p / r`: overall precision / recall of the classifier

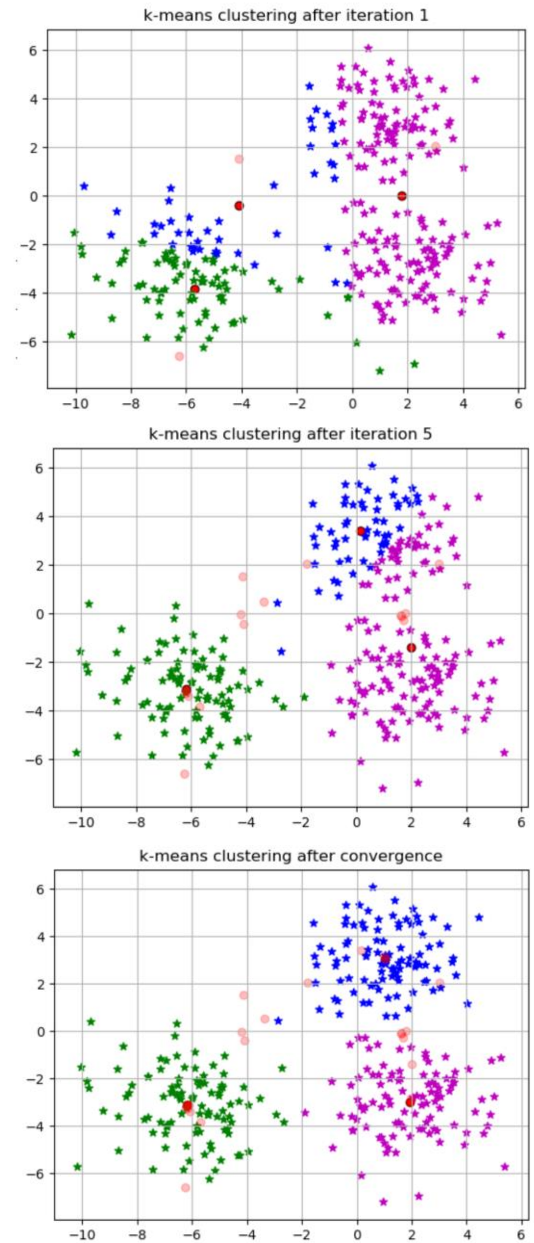
K-means clustering (1/3)

- **Cluster analysis** = one branch of unsupervised learning
- **Goal**: find patterns in data without any external information (labels) to determine the structure of the data
- One of the most famous (and simple) clustering approach is **k-means clustering**.

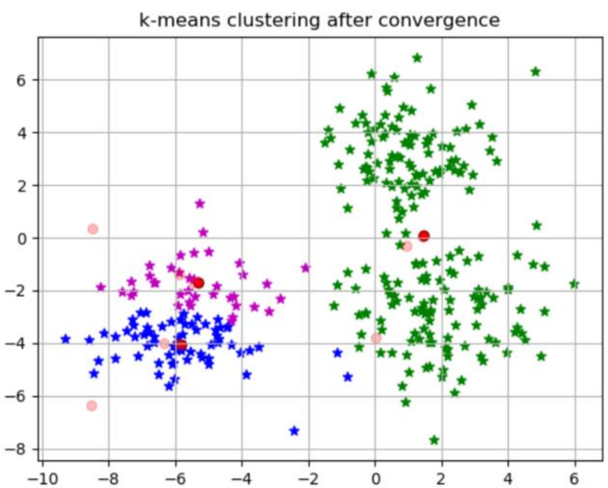


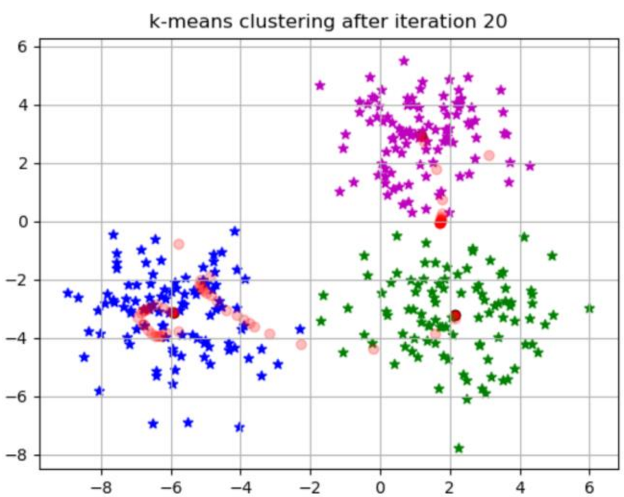
K-means clustering (2/3)

- **Input:**
  - X: d-dimensional dataset
- **Hyper-parameters:**
  - k: number of clusters to find
  - iterMax: number of maximum iterations
- **Pseudo-code**:
  - Initialise k cluster centers at random
  - While no convergence and nbIter < iterMax:
    - Attribute each point of X to its closest cluster center
    - Compute the new centers as the average of the points in the cluster
- **Note**: convergence = the positions of the cluster centers no longer change



K-means clustering (3/3)





- It is possible to mathematically prove that the k-means algorithm **always converges** …
- … but not always towards an optimal point!
- The initialisation of the starting cluster centers has a big impact on the **final cluster centers obtained after convergence** + **convergence time**.
- To make the convergence easier, possible to use **simple heuristic rules** (e.g. boundary on the min/max values for the initial coordinates of the centers)

Exercise 4: K-means implementation

- A main function implementing the algorithm has already been written in `1.3-kMeans-exercise.py`
- Write the code for the following functions:
  - `initialiseCenters(...)`: create k cluster centers at random in a d-dimensional space.
  - `checkConvergence(...)`: check if the k-means algorithm has converged.
  - `attributeCluster(...)`: associate each point of a dataset to its closest cluster center.
  - `computeNewCenters(...)`: compute new cluster centers given a dataset and cluster labels.
- All details regarding expected input and output arguments are provided in the script.

Conclusion

This lecture provided a short and incomplete introduction to Python.

Many Python libraries that will not be introduced but might be relevant in this lecture:

- **Pandas**: data management and analysis tools
- **Scikit-learn**: basic machine learning and data analysis tools
- **Matplotlib**: data plotting tools
- **Tensorflow** and **Keras**: deep-learning tools
- …

Next lecture: introduction to the first scenario about **Activity Monitoring**