Introduction to Python

A Crash Course on Python

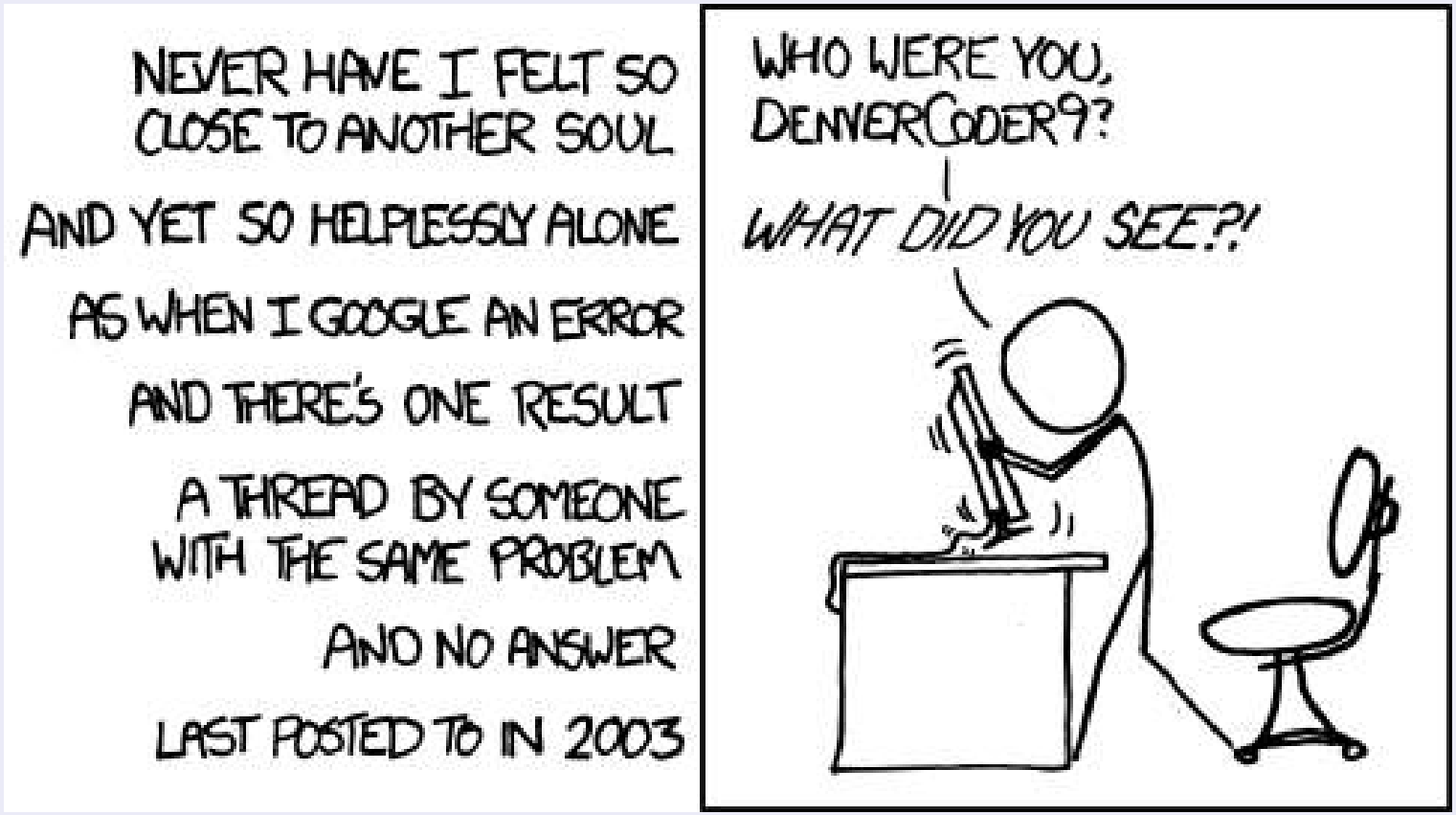
Alexandre Perera,[1,2] PhD

[1]Centre de Recerca en Enginyeria Biomèdica (CREB)
Departament d'Enginyeria de Sistemes, Automatica i Informatica Industrial (ESAII)
Universitat Politecnica de Catalunya

[2]Centro de Investigacion Biomedica en Red en Bioingenieria, Biomateriales y Nanomedicina (CIBER-BBN)
Alexandre.Perera@upc.edu

Introduction to Python 2019

Before We Start ....



Fasten seat belts and download your seminar material from this link.
https://goo.gl/gq3biq

Contents I

Contents II

Section 1

Getting Started With Python

First step

STEP 1

Start the interpreter and type in

```
>>> print("Hello, world")
Hello, world
```

Welcome to Python,
you just executed your first Python instruction, congratulations!

Second step

STEP 2

To get yourself started, type the following stack of instructions

```
>>> a = 3
>>> b = 2*a
>>> type(b)
<class 'int'>
>>> print(b)
6
>>> a*b
18
>>> b = 'hello'
>>> type(b)
<class 'str'>
>>> b + b
'hellohello'
>>> 2*b
'hellohello'
```

Second step

STEP 2

To get yourself started, type the following stack of instructions

```
>>> a = 3
>>> b = 2*a
>>> type(b)
<class 'int'>
>>> print(b)
6
>>> a*b
18
>>> b = 'hello'
>>> type(b)
<class 'str'>
>>> b + b
'hellohello'
>>> 2*b
'hellohello'
```

Observe that

- We do not declare variables (hurrah!!!!!)
- Variable type may be changed on the fly (hurrah!!!, hurrah!!!)
- There is a way to overload operators (hurrah!, hurrah!, hurrah!!!)
- There is a function that tell us the type of a variable.

Types

Integer

```
>>> 1+1
2
>>> a=4
```

Float

```
>>> c=2.1
>>> 3.5/c
1.6666666666666665
```

Boolean

```
>>> 3 > 4
False
>>> test = (3 > 4)
>>> test
False
>>> type(test)
<class 'bool'>
```

Complex

```
>>> a=1.5+0.5j
>>> a.real
1.5
>>> a.imag
0.5
>>> import cmath
>>> cmath.phase(a)
0.3217505543966422
```

Basic Calculator

A Python shell can therefore replace your pocket calculator, with the basic arithmetic operations +, -, *, /, % (modulo) natively implemented:

```
>>> 7 * 3.
21.0
>>> 2**10
1024
>>> 8 % 3
2
```

WARNING!

Integer Division (different in Python 3)

```
>>> 3/2
1.5
```

Use floats

```
>>> 3 / 2.
1.5
>>> a = 3
>>> b = 2
>>> a / b
1.5
>>> a / float(b)
1.5
```

Lists

Python provides many efficient types of containers, in which collections of objects can be stored.

Lists

A list is an ordered collection of objects, that may have different types. For example

```
>>> l = [1, 2, 3, 4, 5]
>>> type(l)
<class 'list'>
```

Lists

accessing individual objects contained in the list:

```
>>> l[2]
3
```

Counting from the end with negative indices:

```
>>> l[-1]
5
>>> l[-2]
4
```

Warning Indexing starts at 0

```
>>> l[0]
1
```

Lists

Slicing

```
>>> l
[1, 2, 3, 4, 5]
>>> l[2:4]
[3, 4]
```

Warning

Warning Note that **l[start:stop]** contains the elements with indices i such as start $\leq$ i $<$ stop (i ranging from start to stop-1). Therefore, **l[start:stop] has (stop-start) elements**.

Lists

Slicing syntax: l[start:stop:step]

All slicing parameters are optional:

```
>>> l
[1, 2, 3, 4, 5]
>>> l[3:]
[4, 5]
>>> l[:3]
[1, 2, 3]
>>> l[::2]
[1, 3, 5]
```

Lists

The elements of a list may have different types:

```
>>> l = [3, 2+3j, 'hello']
>>> l
[3, (2+3j), 'hello']
>>> l[1], l[2]
((2+3j), 'hello')
```

Lists

Python offers a large panel of functions to modify lists, or query them. Here are a few examples; for more details, see
http://docs.python.org/tutorial/datastructures.html#more-on-lists

Add and remove elements

```
>>> l = [1, 2, 3, 4, 5]
>>> l.append(6)
>>> l
[1, 2, 3, 4, 5, 6]
>>> l.pop()
6
>>> l
[1, 2, 3, 4, 5]
>>> l.extend([6, 7]) # extend l, in-place
>>> l
[1, 2, 3, 4, 5, 6, 7]
>>> l = l[:-2]
>>> l
[1, 2, 3, 4, 5]
```

Lists

Reverse list

```
>>> r = l[::-1]
>>> r
[5, 4, 3, 2, 1]
```

Concatenate and repeat

```
>>> r + l
[5, 4, 3, 2, 1, 1, 2, 3, 4, 5]
>>> 2 * r
[5, 4, 3, 2, 1, 5, 4, 3, 2, 1]
```

Sort (in-place)

```
>>> r.sort()
>>> r
[1, 2, 3, 4, 5]
```

Note

Methods and Object-Oriented Programming

The notation r.method() (r.sort(), r.append(3), l.pop()) is our first example of object-oriented programming (OOP). Being a list, the object r owns the method function that is called using the notation '.'
No further knowledge of OOP than understanding the notation '.' is necessary for going through this tutorial.

Note

Discovering methods in ipython tab-completion (press tab)

```
In [1]: r.
r.append   r.extend   r.insert   r.remove   r.sort
r.count    r.index    r.pop      r.reverse
```

PAY ATTENTION

NEXT SET OF SLIDES ARE VERY IMPORTANT!!!

Mutable and immutable types

Immutable types

- integer
- float
- complex
- boolean
- strings

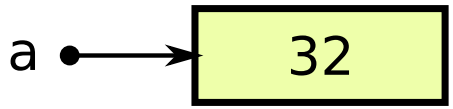
Mutable

- Lists

Immutable types

Create an immutable element

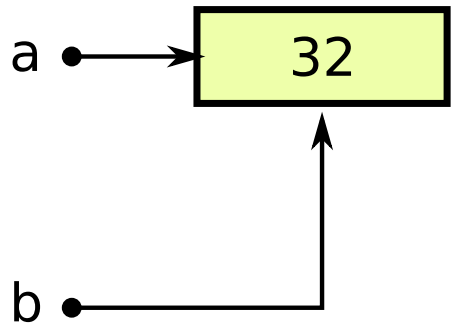
```
>>> a=32
```



Immutable types

"copy" it

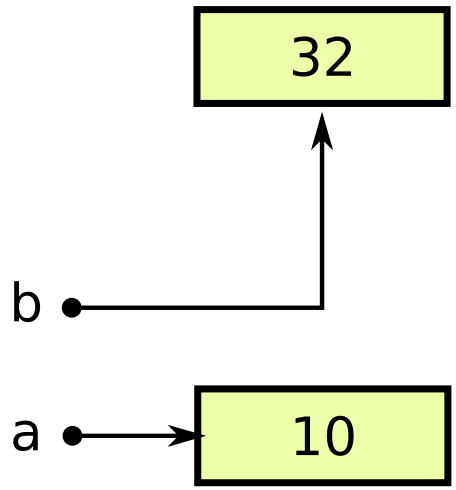
```
>>> a=32
>>> b=a
```



Immutable types

Change the original object

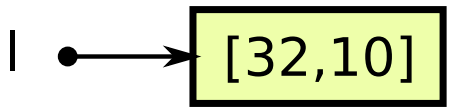
```
>>> a=32
>>> b=a
>>> a=10
>>> b
32
```



Mutable types

Create a mutable type

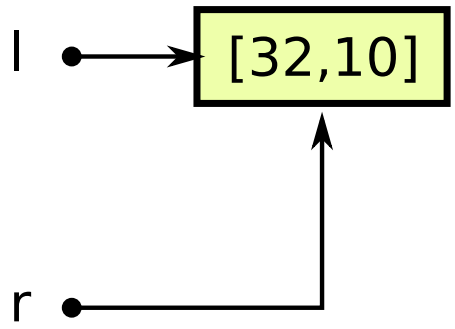
```
>>> l=[32,10]
```



Mutable types

"Copy" it

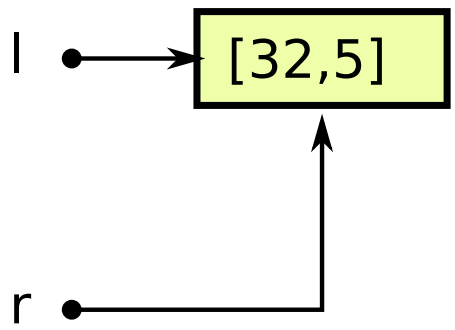
```
>>> l=[32,10]
>>> r=l
```



Mutable types

Change the original object

```
>>> l=[32,10]
>>> r=l
>>> l[1]=3
>>> r
[32, 3]
```



Challenge

1 minute challenge

Create a list A, create a list B that contains A, copy the list B into C, modify A and check C value

Visited Types

Already seen types

- boolean
- integer
- float
- complex
- string
- list

Pending Types

- Dictionary
- Tuple
- Set

Dictionary

A dictionary is basically an efficient table that maps keys to values. It is an unordered container:

```
>>> tel = {'emmanuelle': 5752, 'sebastian': 5578}
>>> tel['francis'] = 5915
>>> tel
{'emmanuelle': 5752, 'sebastian': 5578, 'francis': 5915}
>>> tel['sebastian']
5578
>>> tel.keys()
dict_keys(['emmanuelle', 'sebastian', 'francis'])
>>> tel.values()
dict_values([5752, 5578, 5915])
>>> 'francis' in tel
True
```

Dictionary

A dictionary can have keys (resp. values) with different types:

```
>>> d = {'a':1, 'b':2, 3:'hello'}
>>> d
{'a': 1, 'b': 2, 3: 'hello'}
```

Tuples

The elements of a tuple are written between parentheses, or just separated by commas:

```
>>> t = 12345, 54321, 'hello!'
>>> t[0]
12345
>>> t
(12345, 54321, 'hello!')
>>> u = (0, 2)
```

if/then/else

If

```
>>> if 2**2 == 4:
...     print('Obvious!')
...
Obvious!
```

Blocks are delimited by indentation

```
>>> a = 10
>>> if a == 1:
...     print(1)
... elif a == 2:
...     print(2)
... else:
...     print('A lot')
...
A lot
```

Conditional Expressions

if object:

Evaluates to False:

- any number equal to zero (0, 0.0, 0+0j)
- an empty container (list, tuple, set, dictionary, ...)
- False, None

Evaluates to True:

- everything else (User-defined classes can customize those rules by overriding the special **nonzero** method.)

Tests identity: both sides are the same object:

```
>>> 1 is 1.
False
>>> a = 1
>>> b = 1
>>> a is b
True
```

Tests equality, with logics:

```
>>> 1==1.
True
```

For any collection b: b contains a

```
>>> b = [1, 2, 3]
>>> 2 in b
True
>>> 5 in b
False
```

If b is a dictionary, this tests that a is a key of b.

for/range

Iterating with an index:

```
>>> for i in range(4):
...     print(i)
...
0
1
2
3
```

But most often, it is more readable to iterate over values:

```
>>> for word in ('cool', 'powerful', 'readable'):
...     print('Python is %s' % word)
...
Python is cool
Python is powerful
Python is readable
```

while/break/continue

Typical C-style while loop (Mandelbrot problem):

```
>>> z = 1 + 1j
>>> while abs(z) < 100:
...     z = z**2 + 1
...
```

Break out of enclosing for/while loop:

```
>>> z = 1 + 1j
>>> while abs(z) < 100:
...     if z.imag == 0:
...         break
...     z = z**2 + 1
```

Continue the next iteration of a loop.:

```
>>> a = [1, 0, 2, 4]
>>> for element in a:
...     if element == 0:
...         continue
...     print(1. / element)
...
1.0
0.5
0.25
```

Advanced iteration

Iterate over any sequence

You can iterate over any sequence (string, list, keys in a dictionary, lines in a file, ...):

```
>>> vowels = 'aeiou'
>>> for i in 'powerful':
...     if i in vowels:
...         print(i)
...
o
e
u
```

```
>>> message = "Hello how are you?"
>>> message.split() # returns a list
['Hello', 'how', 'are', 'you?']
>>> for word in message.split():
...     print(word)
...
Hello
how
are
you?
```

Few languages (in particular, languages for scientific computing) allow to loop over anything but integers/indices. With Python it is possible to loop exactly over the objects of interest without bothering with indices you often don't care about.

Keeping track of enumeration number

Common task is to iterate over a sequence while keeping track of the item number.

Could use while loop with a counter as above. Or a for loop:

```
>>> words = ('cool', 'powerful', 'readable')
>>> for i in range(0, len(words)):
...     print(i, words[i]),
...
0 cool
(None,)
1 powerful
(None,)
2 readable
(None,)
```

But Python provides enumerate for this:

```
>>> words = ('cool', 'powerful', 'readable')
>>> for index, item in enumerate(words):
...     print(index, item,)
...
0 cool
1 powerful
2 readable
```

Looping over a dictionary

Use iteritems(python2) or items():

```
>>> d = {'a': 1, 'b':1.2, 'c':1j}
>>> for key, val in d.items():
...     print('Key: %s has value: %s' % (key, val))
...
Key: a has value: 1
Key: b has value: 1.2
Key: c has value: 1j
```

List comprehensions

Natural math

$$k = \left\{x^2, x \in \{0, 1, 2, 3\}\right\}$$

List comprehensions

Natural math

$$k = \left\{x^2, x \in \{0, 1, 2, 3\}\right\}$$

Translates to

```
>>> k=[x**2 for x in range(4)]
>>> k
[0, 1, 4, 9]
```

Challenge

5 minutes challenge

Compute the decimals of π using the Wallis formula:

$$\pi = 2 \prod_{i=1}^{\infty} \frac{4i^2}{4i^2 - 1}$$

Section 2

Functions

Function definition

Function blocks must be indented as other control-flow blocks.

```
In [56]: def test():
    ....:     print('in test function')
    ....:
    ....:

In [57]: test()
in test function
```

Return statement

Functions can optionally return values.

```
In [6]: def disk_area(radius):
   ...:     return 3.14 * radius * radius
   ...:

In [8]: disk_area(1.5)
Out[8]: 7.064999999999995
```

Structure:

- the def keyword;
- is followed by the function's name, then
- the arguments of the function are given between brackets followed by a colon.
- the function body ;
- and return object for optionally returning values.
- By default, functions return None.

Parameters

Mandatory parameters (positional arguments)

```
In [81]: def double_it(x):
    ....:     return x * 2
    ....:

In [82]: double_it(3)
Out[82]: 6

In [83]: double_it()
---------------------------------------------------------------------
TypeError                                 Traceback (most recent ca

/Users/cburns/src/scipy2009/scipy_2009_tutorial/source/<ipython con

TypeError: double_it() takes exactly 1 argument (0 given)
```

Parameters

Optional parameters (keyword or named arguments)

```
In [84]: def double_it(x=2):
   ....:     return x * 2
   ....:
In [85]: double_it()
Out[85]: 4
In [86]: double_it(3)
Out[86]: 6
```

Warning

```
In [124]: bigx = 10
In [125]: def double_it(x=bigx):
   .....:     return x * 2
   .....:
In [126]: bigx = 1e9  # Now really big
In [128]: double_it()
Out[128]: 20
```

Functions are objects

Functions are first-class objects, which means they can be:

- assigned to a variable
- an item in a list (or any collection)
- passed as an argument to another function

Example

```
In [38]: va = variable_args

In [39]: va('three', x=1, y=2)
args is ('three',)
kwargs is {'y': 2, 'x': 1}
```

Challenge

10 min challenge: Fibonacci

Write a function that displays the n first terms of the Fibonacci sequence, defined by:

$u_0 = 1; u_1 = 1$

$u_{(n+2)} = u_{(n+1)} + u_n$

Section 3

NumPy

NumPy

Python has built-in:

containers: lists (costless insertion and append), dictionaries (fast lookup)

high-level number objects: integers, floating point

Numpy is:

- extension package to Python for multi-dimensional arrays
- closer to hardware (efficiency)
- designed for scientific computation (convenience)

Snippet

```
import numpy as np
a = np.array([0, 1, 2, 3])
a
array([0, 1, 2, 3])
```

NumPy

For example:

An array containing:

- values of an experiment/simulation at discrete time steps
- signal recorded by a measurement device, e.g. sound wave
- pixels of an image, grey-level or colour
- 3-D data measured at different X-Y-Z positions, e.g. MRI scan
- ...

Memory-efficient container that provides fast numerical operations.

```
In [1]: l = range(1000)
In [2]: %timeit [i**2 for i in l]
1000 loops, best of 3: 403 us per loop
In [3]: a = np.arange(1000)
In [4]: %timeit a**2
100000 loops, best of 3: 12.7 us per loop
```

Creating Arrays

1-D Array

```
a = np.array([0, 1, 2, 3])
a
array([0, 1, 2, 3])
a.ndim
1
a.shape
(4,)
len(a)
4
```

30 seconds challenge

Is an np.array mutable?

Creating Arrays

Common arrays

```
a = np.ones((3, 3))  # reminder: (3, 3) is a tuple
a
array([[ 1.,  1.,  1.],
       [ 1.,  1.,  1.],
       [ 1.,  1.,  1.]])
b = np.zeros((2, 2))
b
array([[ 0.,  0.],
       [ 0.,  0.]])
c = np.eye(3)
c
array([[ 1.,  0.,  0.],
       [ 0.,  1.,  0.],
       [ 0.,  0.,  1.]])
d = np.diag(np.array([1, 2, 3, 4]))
d
array([[1, 0, 0, 0],
       [0, 2, 0, 0],
       [0, 0, 3, 0],
       [0, 0, 0, 4]])
```

Section 4

Machine learning scientific kit

Scikits

Avaliable kits

The list of available kits can be found at
http://scikits.appspot.com/scikits.

Scikits

Main scikits

scikit-aero	Aeronautical engineering calculations in Python.
scikit-image	Image processing routines for SciPy.
scikit-rf	Object Oriented Microwave Engineering.
audiolab	A python module to make noise from numpy arrays (sic).
timeseries	Time series manipulation.
learn	Machine learning Sci-kit.

Introduction

Datasets

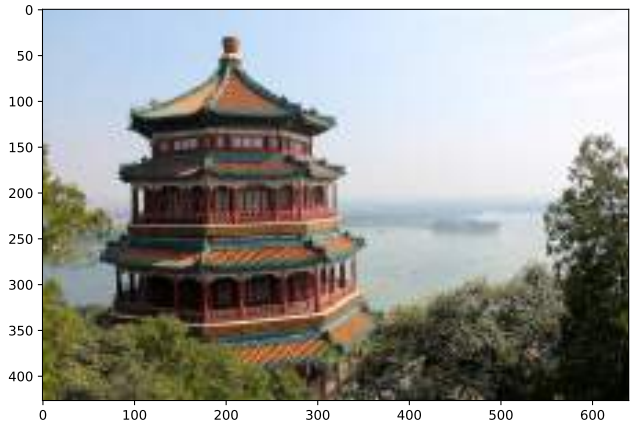
There are three different datasets in sklearn:

1. Sample images.
2. Toy Datasets.
3. Sample Generators.

Image sets

The scikit also embed a couple of sample JPEG images (china and flower) published under Creative Commons license by their authors.

```
import numpy as np
import pylab as pl
from sklearn.datasets import load_sample_ima
china = load_sample_image("china.jpg")
```



Toy datasets

load_boston() Regression Load and return the boston house-prices dataset.

load_iris() Classification Load and return the iris dataset.

load_diabetes() Regression Load and return the diabetes dataset.

load_digits() Classification Load and return the digits dataset.

load_linnerud() Multivariate Regression Load and return the linnerud dataset.

These functions return a *bunch* (which is a dictionary that is accessible with the *dict.key* syntax). All datasets have at least two keys,

- data, containing an array of shape n samples $\times$ n features and
- target, a numpy array of length n features, containing the targets.

Data Generators

A number of functions exists to create the most esoteric data distributions:

make_classification() n-class classification problem (+ multilabel).

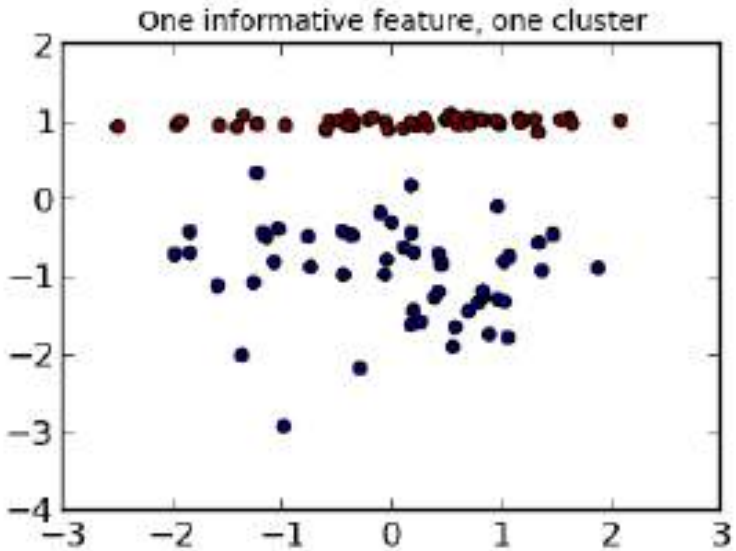
make_regression() Generate a regression problem.

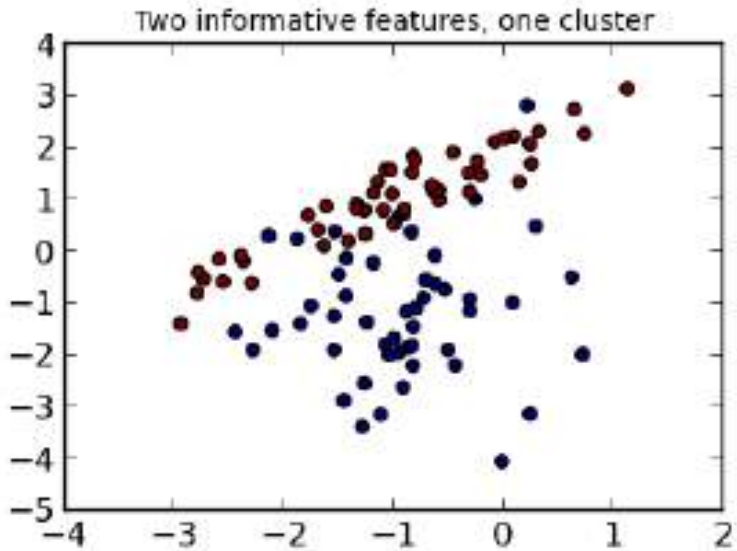
make_swiss_roll() Generate swiss roll datasets.

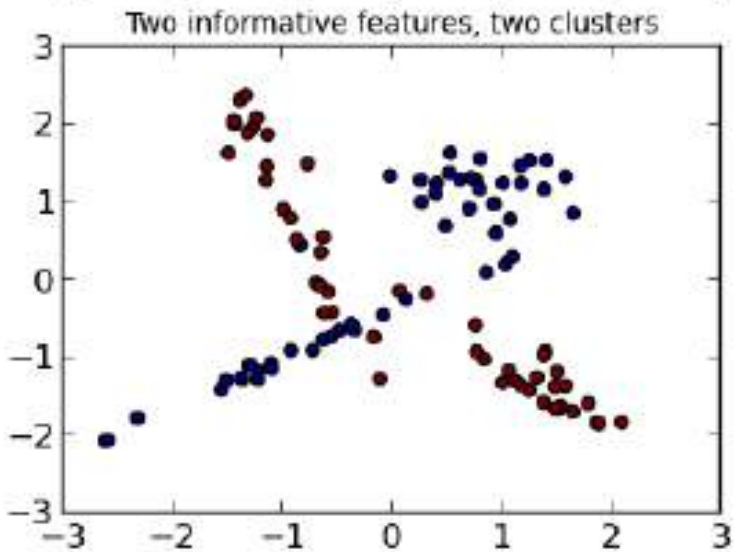
make_s_curve Generates S curve datasets.

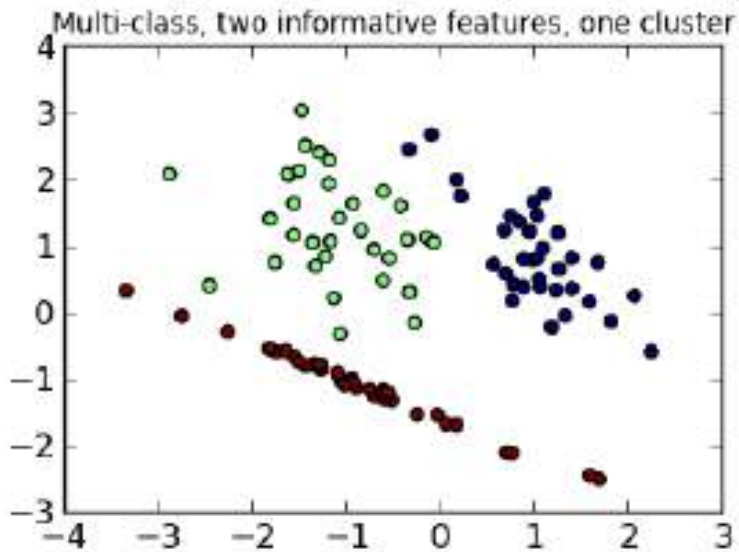
All of them returning a tuple (X, y) consisting of a n samples $\times$ n features numpy array X and an array of length n samples containing the targets y.

Data Generators









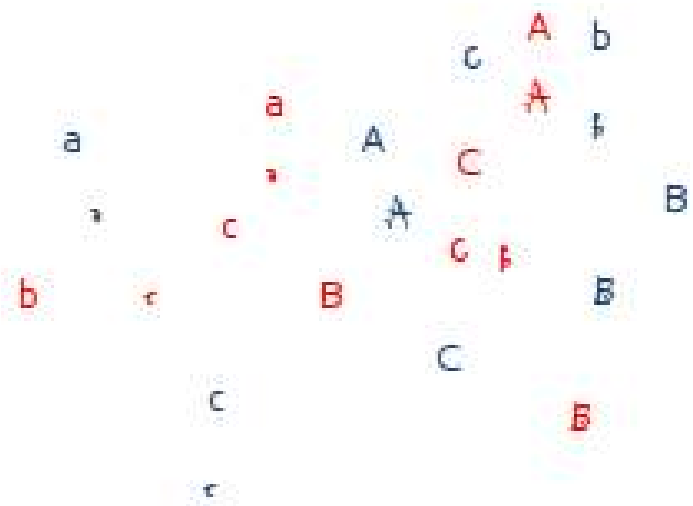
Clustering

Or Cluster Analysis, is the task of grouping a set of objects in such a way that objects in the same groups (clusters) are more similar to each other than to those in other groups.

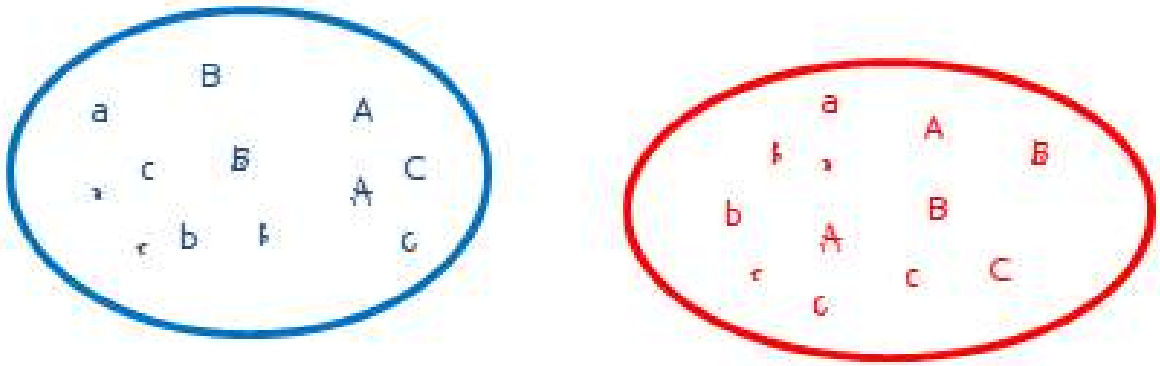
It is useful in a large amount of applications:

- Exploratory analysis.
- Machine Learning & Pattern Recognition.
- Image analysis.
- Bioinformatics.
- Market Analysis.

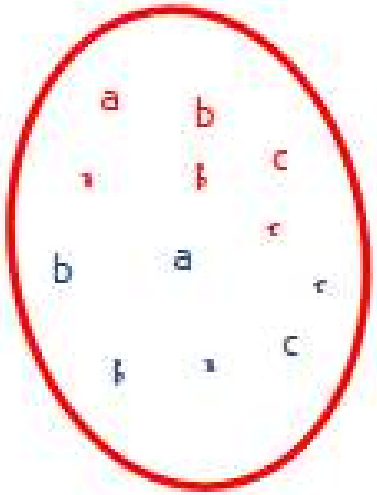
Clustering

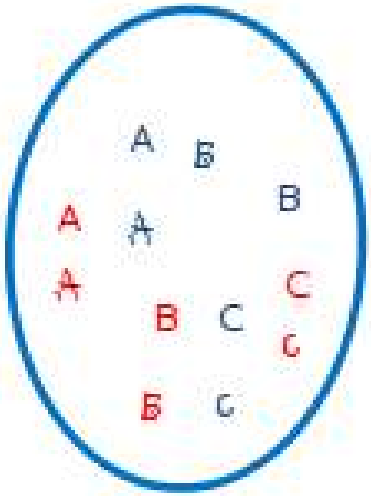


Clustering

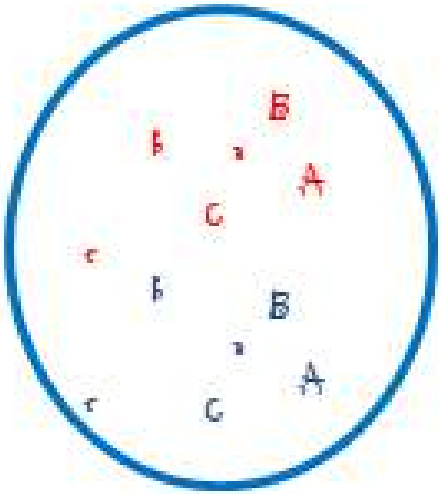


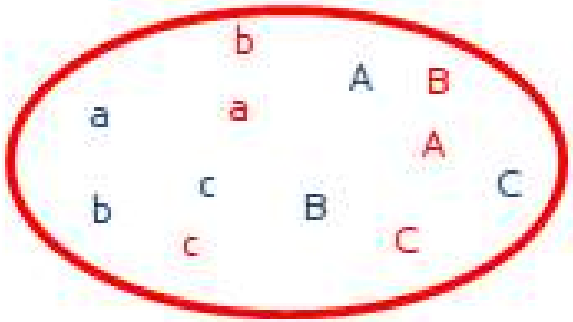
Clustering



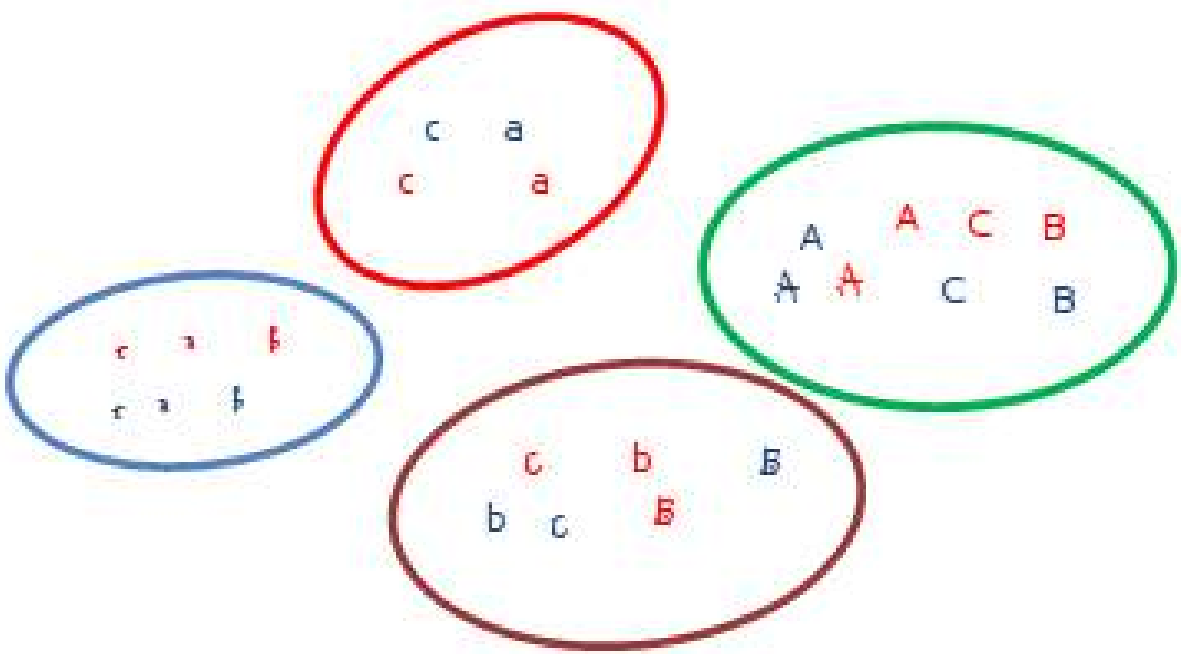


Clustering





Clustering



Cluster Analysis

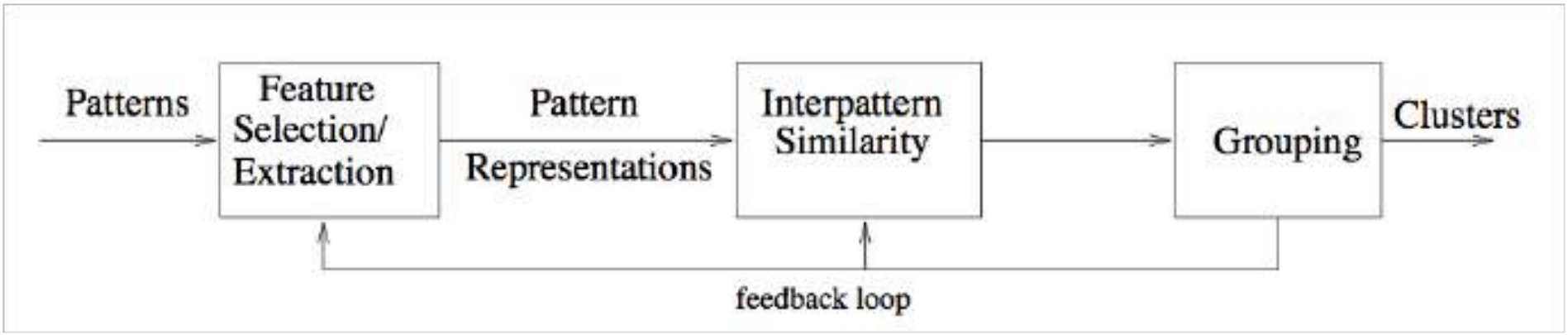
Clustering

Partitioning of a data set into subsets (clusters). Individuals in each subset share some common trait according to some defined distance measure The most typical trait is proximity

Data clustering is a common technique for statistical data analysis. There are a number of techniques available

- Hierarchical (e.g. agglomerative clustering)
- Partitional (e.g. k-means clustering)
- Graph base (Spectral)

Clustering


Patterns
Feature Selection/ Extraction
Pattern Representations
Interpattern Similarity
Grouping
Clusters
feedback loop

(di)Similarity

Metric definition $d(x, y)$.

- $d(x, y) \geq 0$
- $d(x, y)$ iff $x == y$
- $d(x, y) = d(y, x)$
- $d(x, y) \leq d(x, z) + d(z, y)$

And...

- $d(ax, ay) = |a|d(x, y)$, d is a norm $||x - y||$

The most common form of distance metric:

$$||x - y||_{p/r} = \left(\sum_{i=1}^{D} |x_i - y_i|^p \right)^{1/r}$$

$$||x - y||_e = \left(\sum_{i=1}^{D} |x_i - y_i|^2 \right)^{1/2} \quad (1)$$

k-means

k-means

The most common clustering algorithm, (J. B. MacQueen, 1967)

1. Ask user how many clusters are desired.

k-means

k-means

The most common clustering algorithm, (J. B. MacQueen, 1967)

1. Ask user how many clusters are desired.
2. Randomly guess k cluster center locations.

k-means

k-means

The most common clustering algorithm, (J. B. MacQueen, 1967)

1. Ask user how many clusters are desired.
2. Randomly guess k cluster center locations.
3. Each pattern is assigned to the closest center.

k-means

k-means

The most common clustering algorithm, (J. B. MacQueen, 1967)

1. Ask user how many clusters are desired.
2. Randomly guess k cluster center locations.
3. Each pattern is assigned to the closest center.
4. A New center is assigned as the centroid of the assigned patterns.

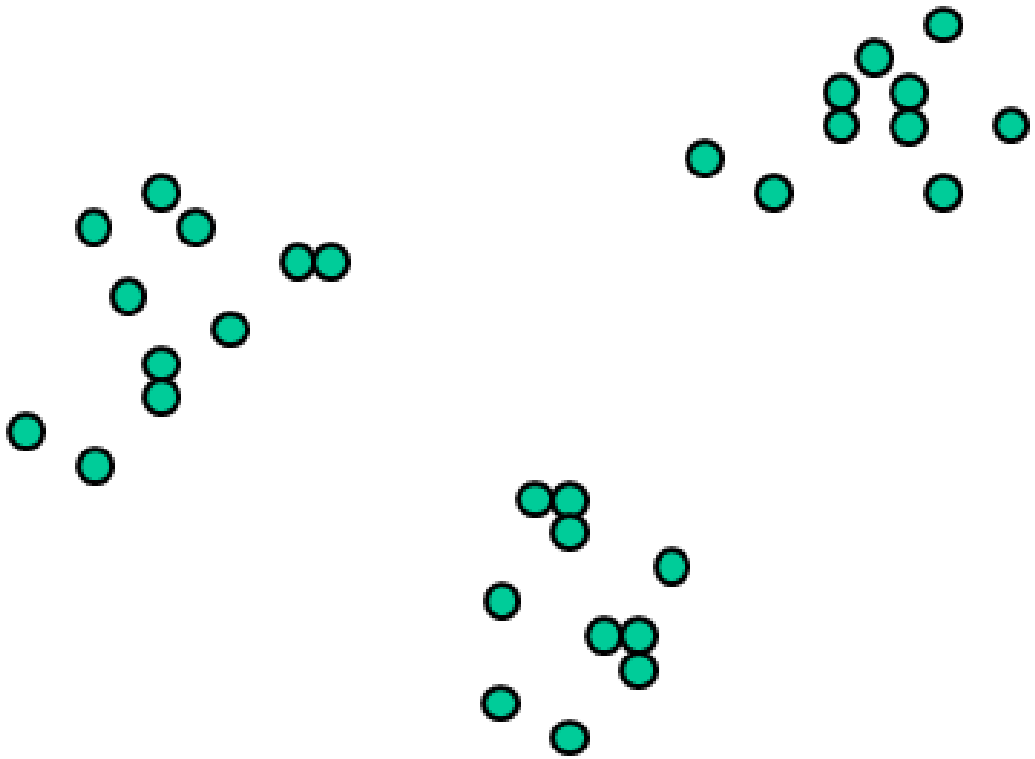
k-means

k-means

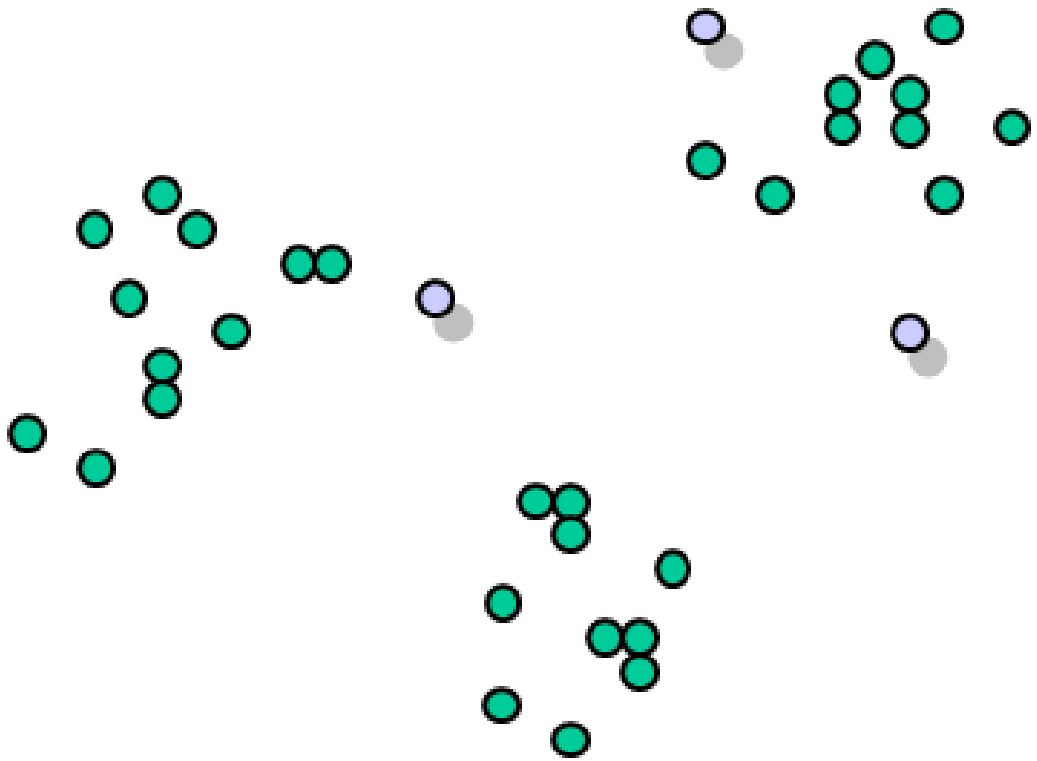
The most common clustering algorithm, (J. B. MacQueen, 1967)

1. Ask user how many clusters are desired.
2. Randomly guess k cluster center locations.
3. Each pattern is assigned to the closest center.
4. A New center is assigned as the centroid of the assigned patterns.
5. Repeat from (3).

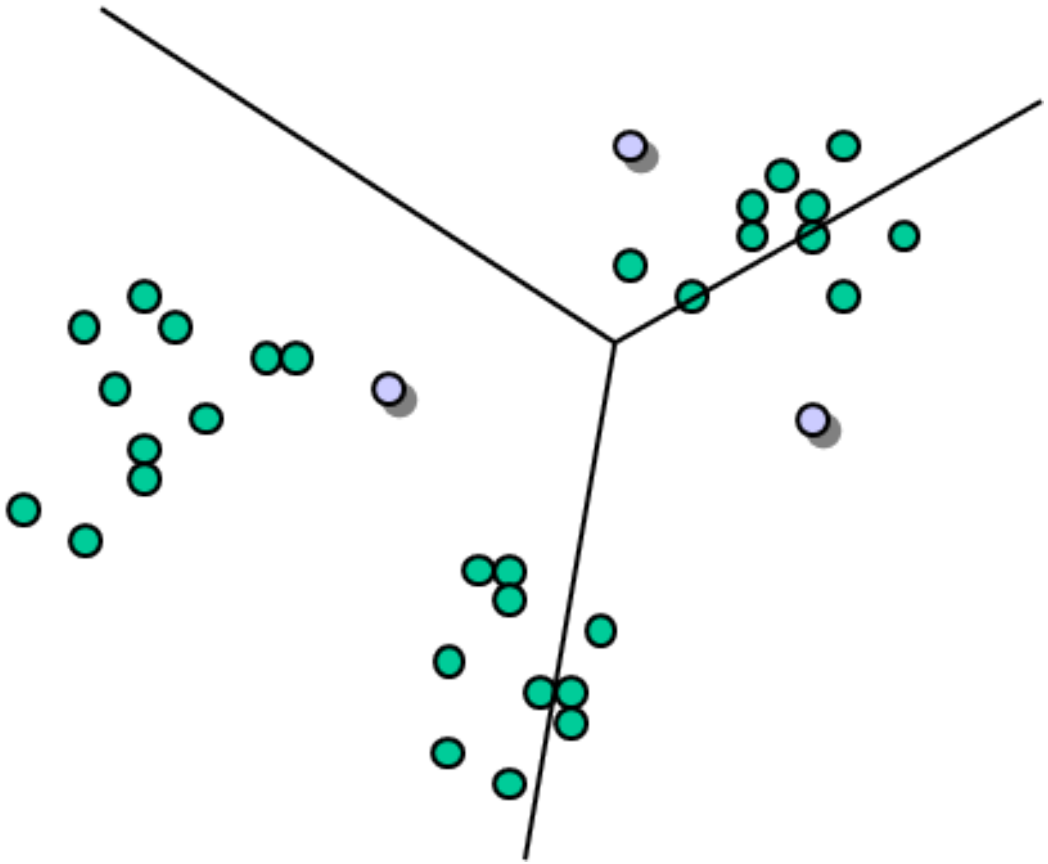
kmeans



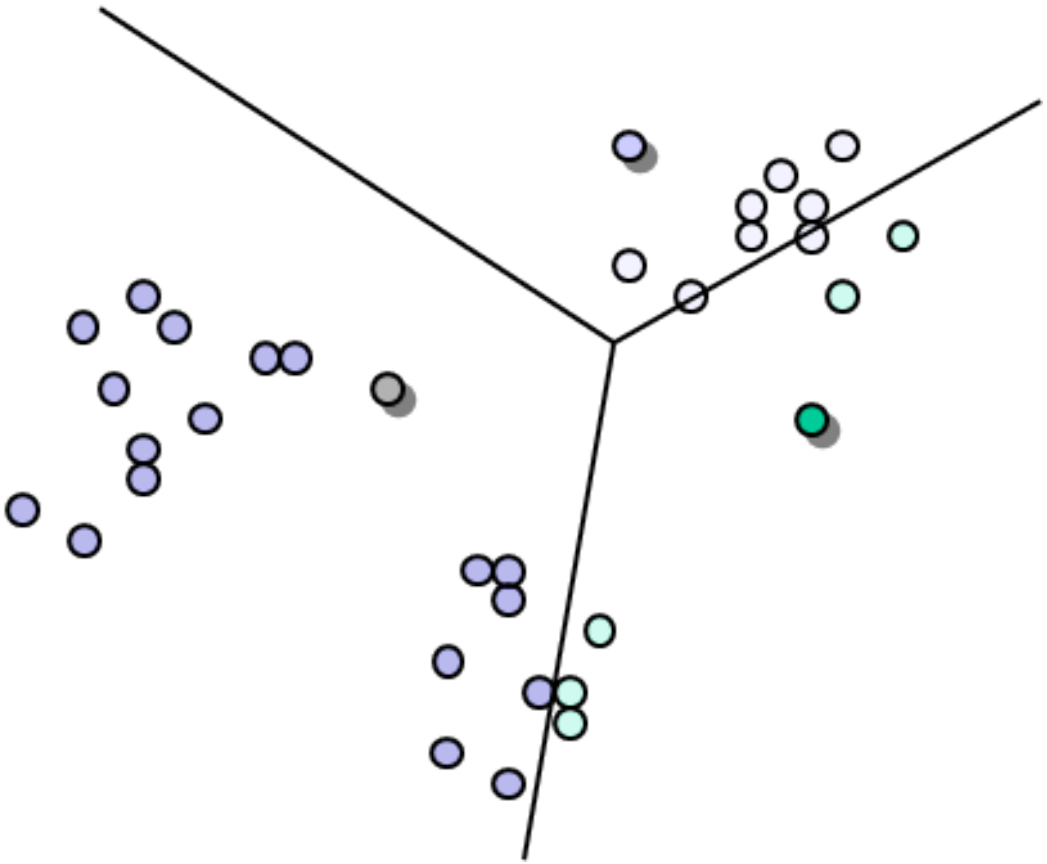
kmeans



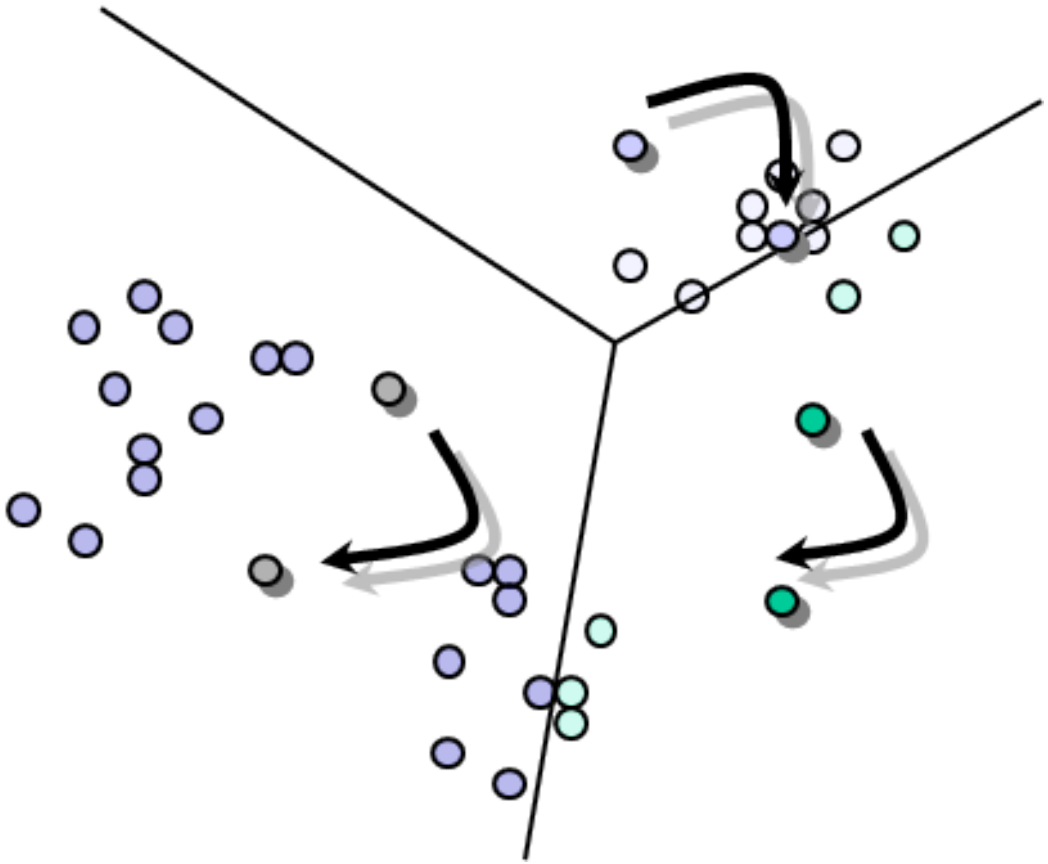
kmeans



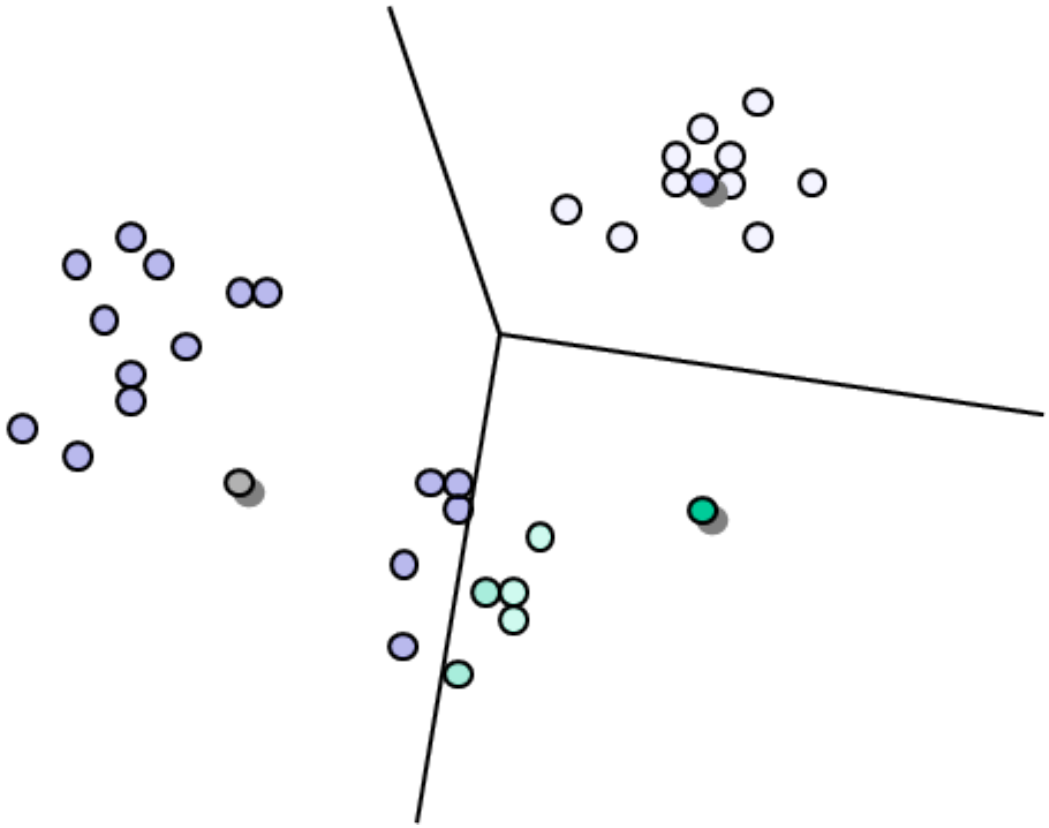
kmeans



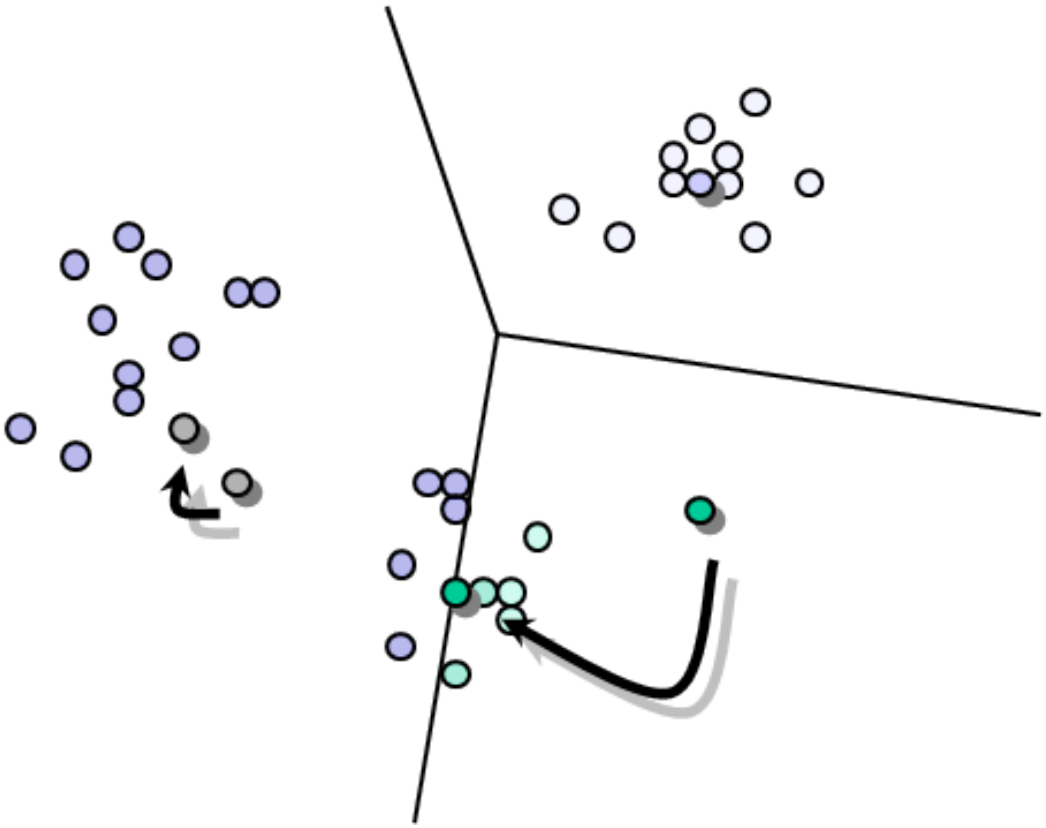
kmeans



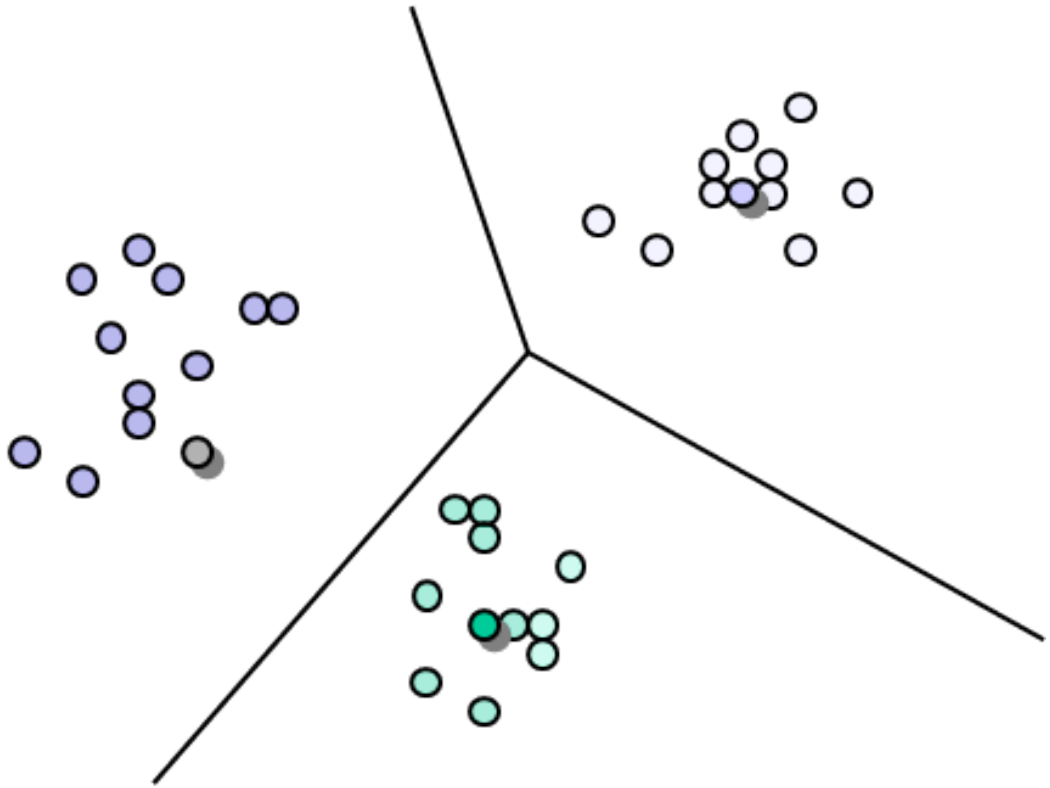
kmeans



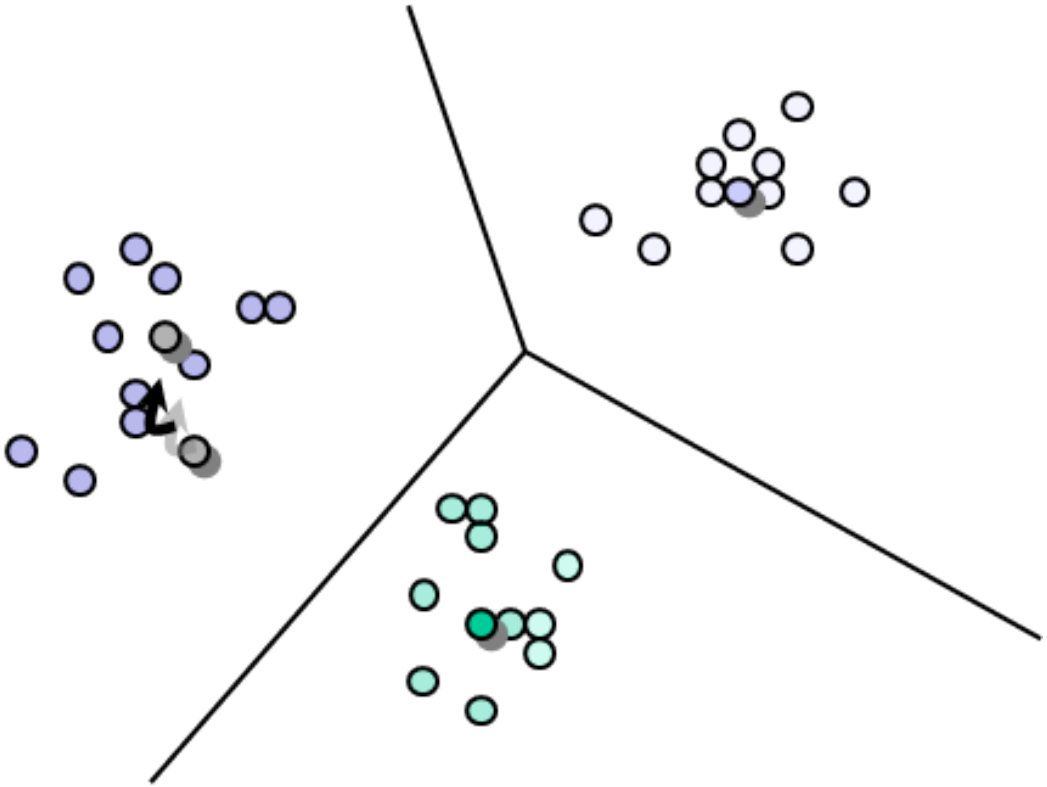
kmeans



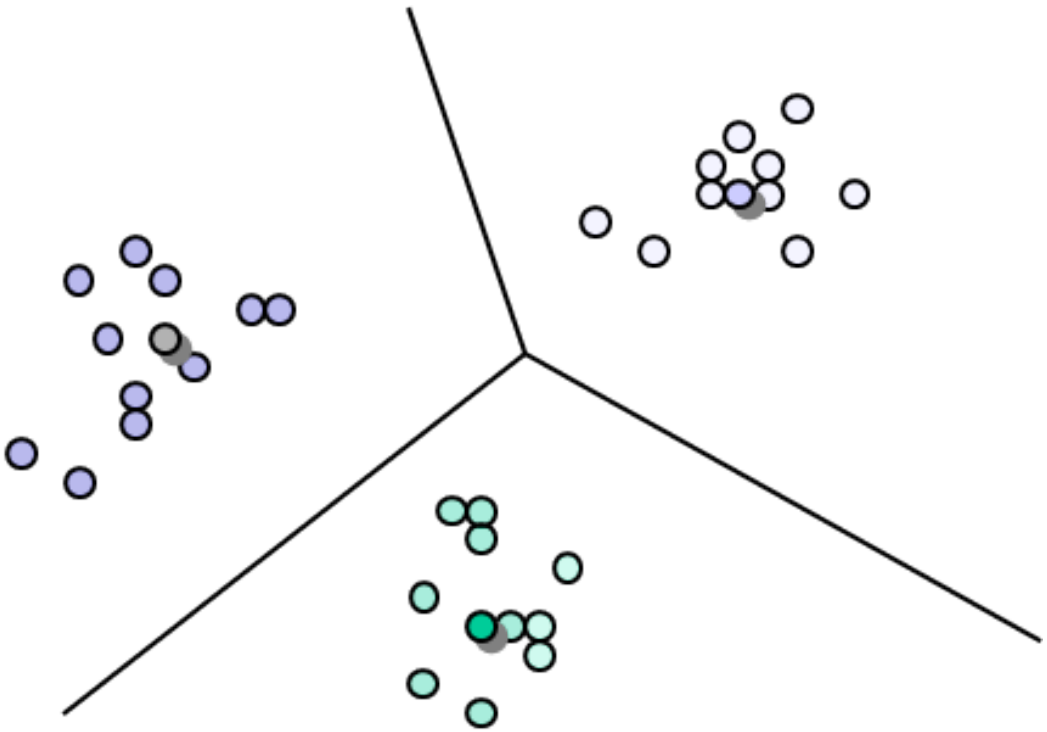
kmeans



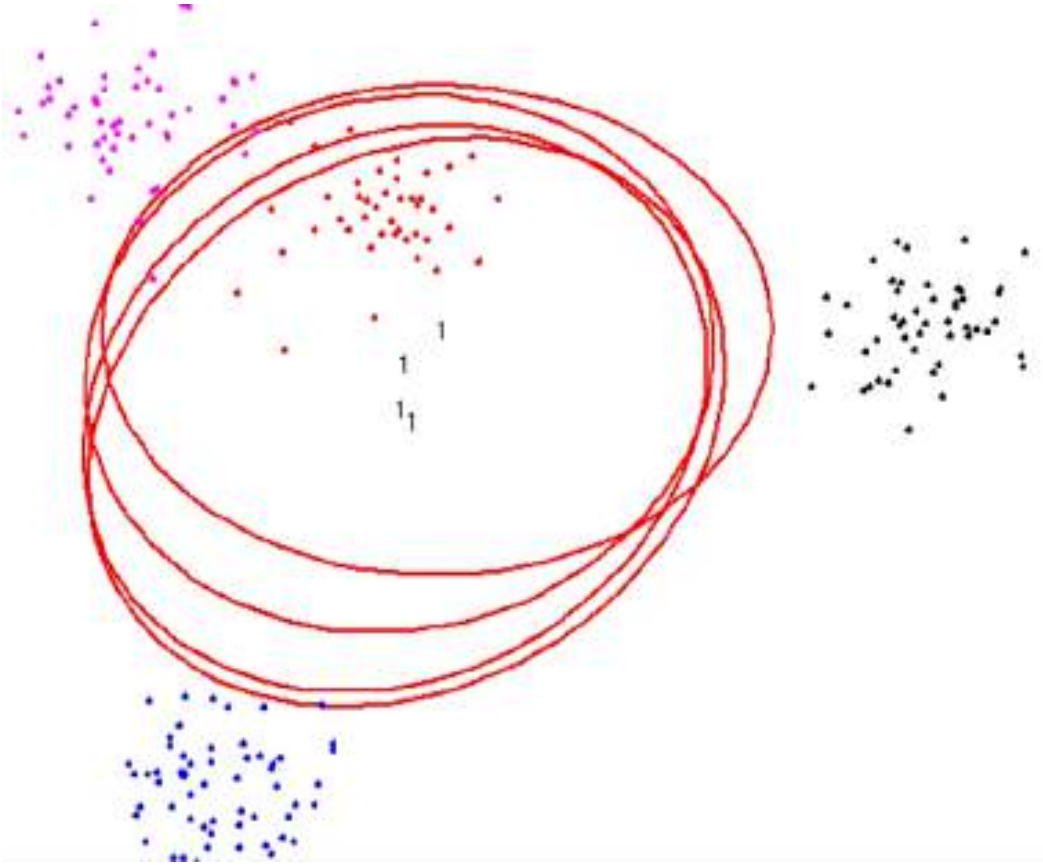
kmeans



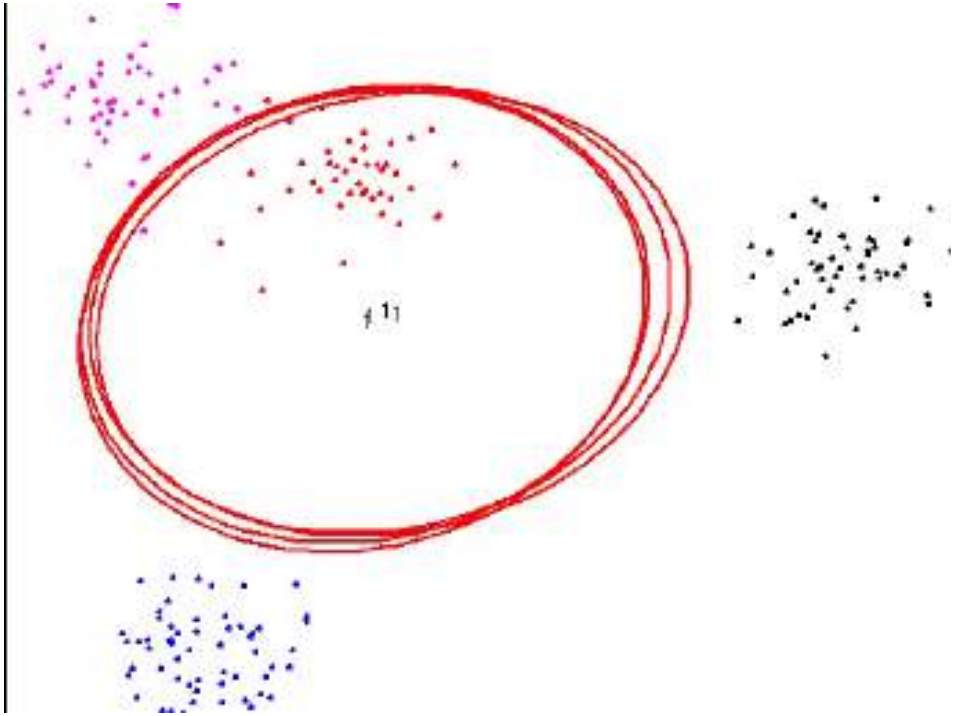
kmeans

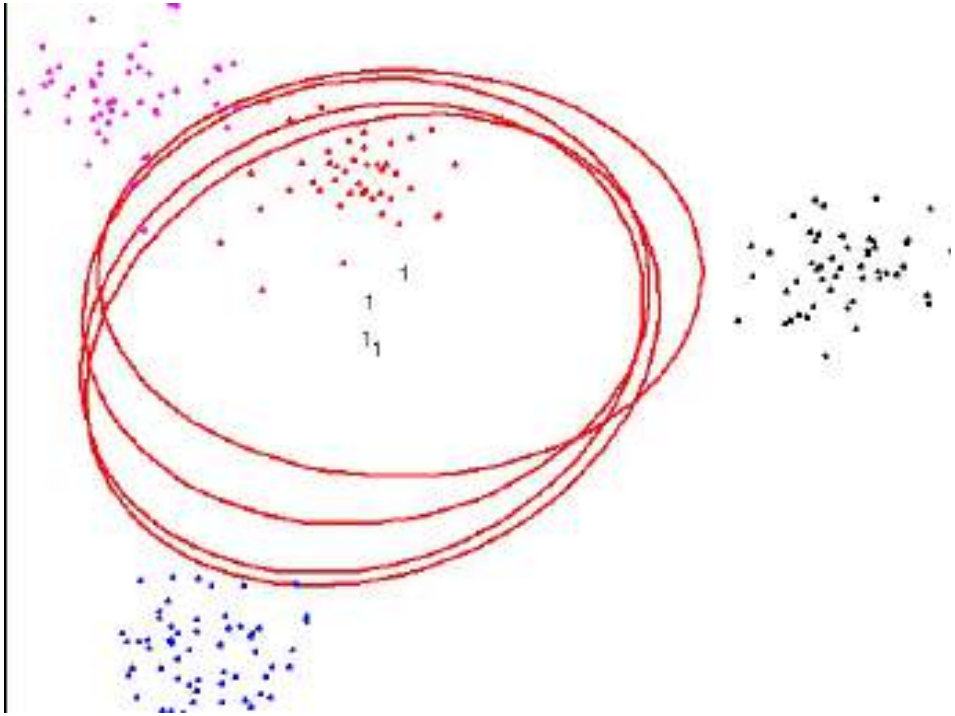


kmeans

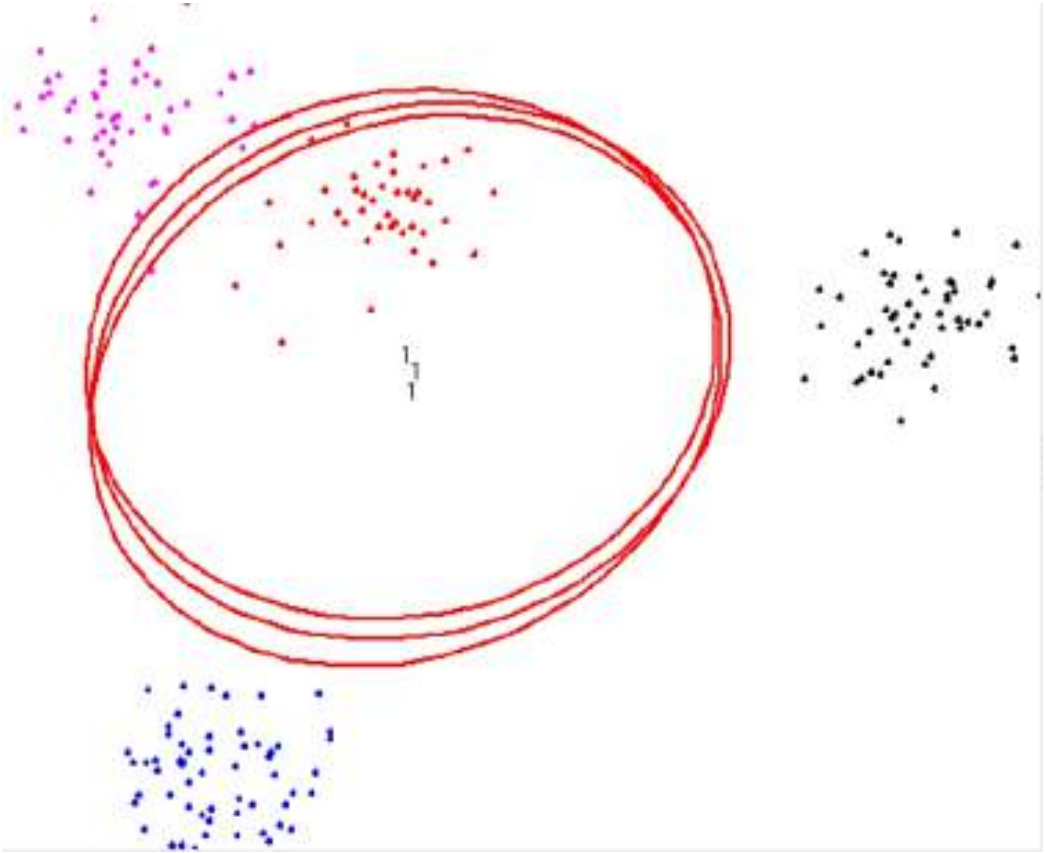


Convergence

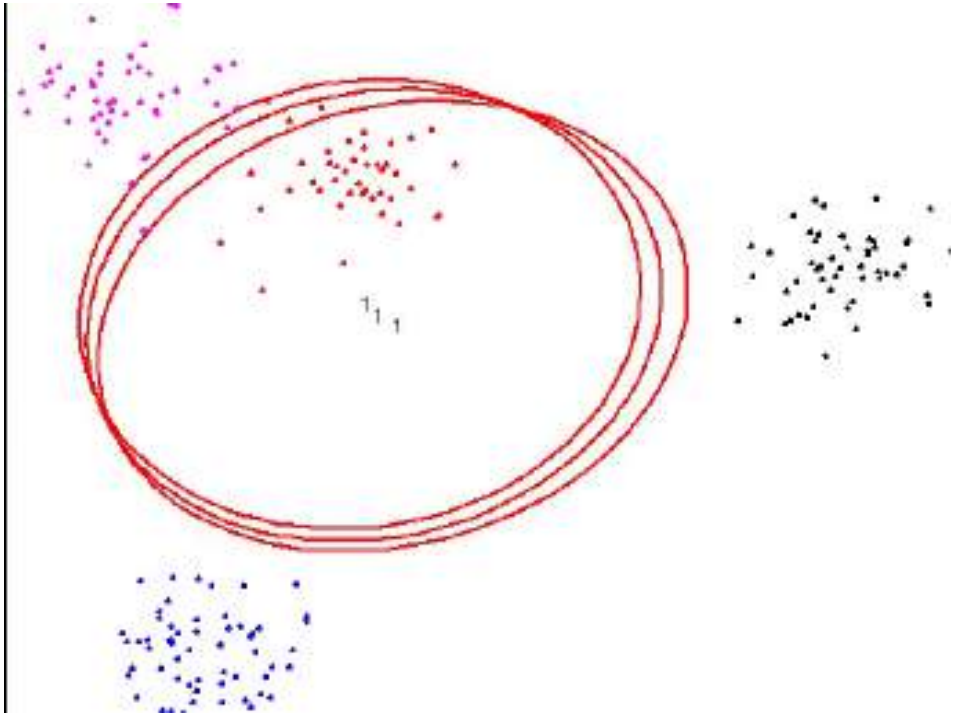


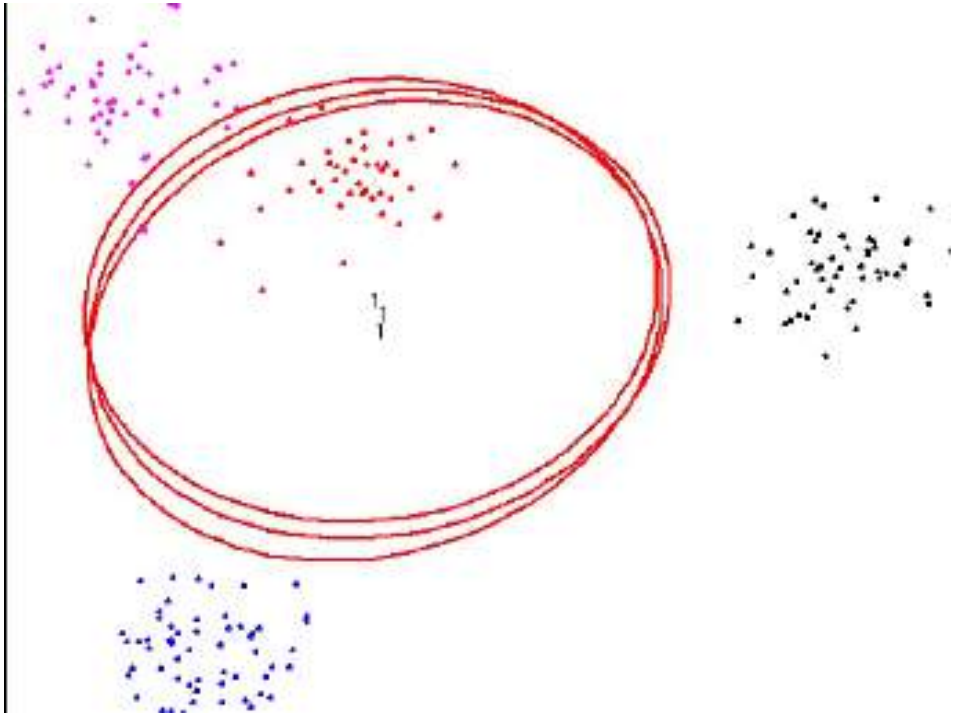


Number of Clusters

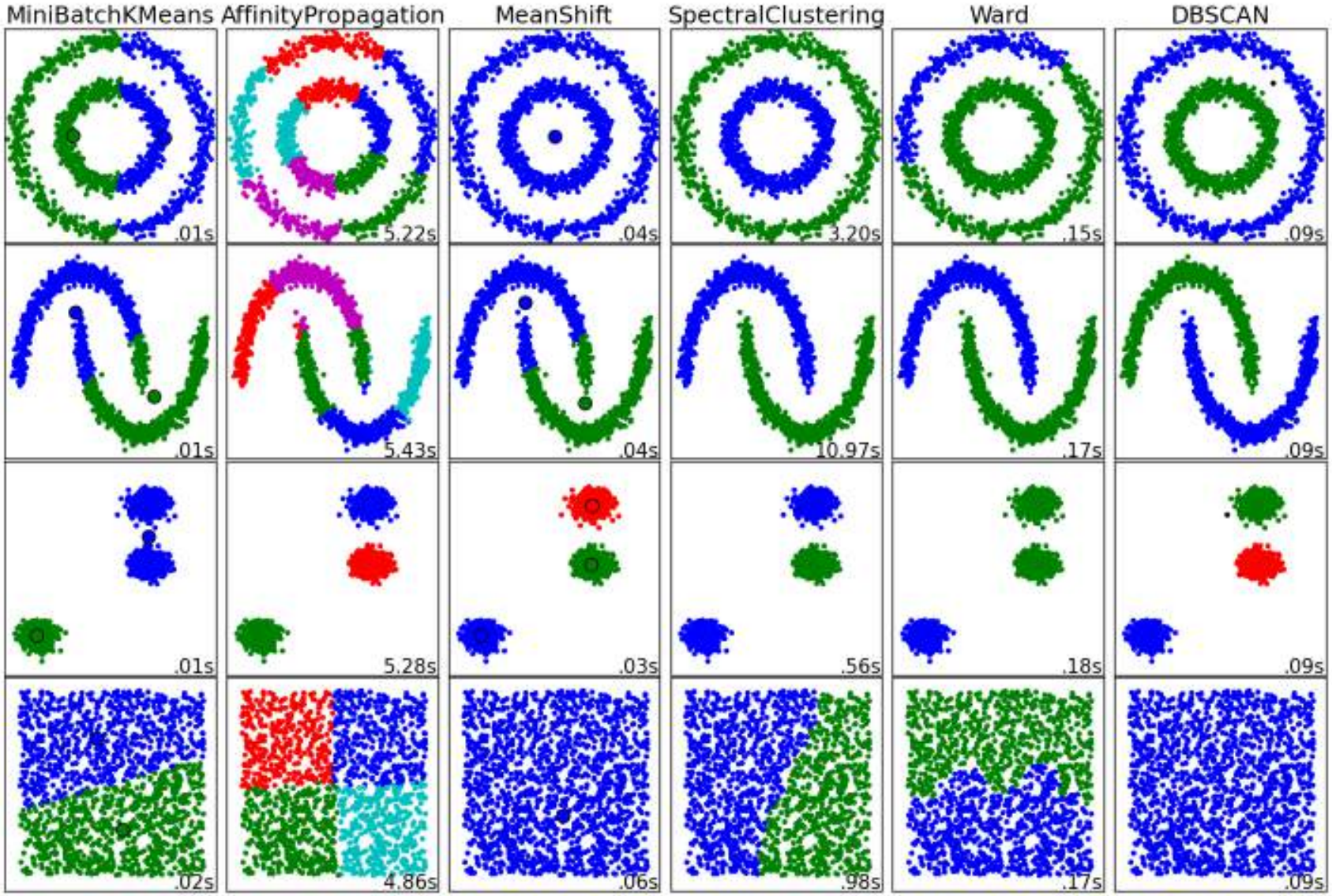


Number of Clusters





Clustering algorithms in sklearn

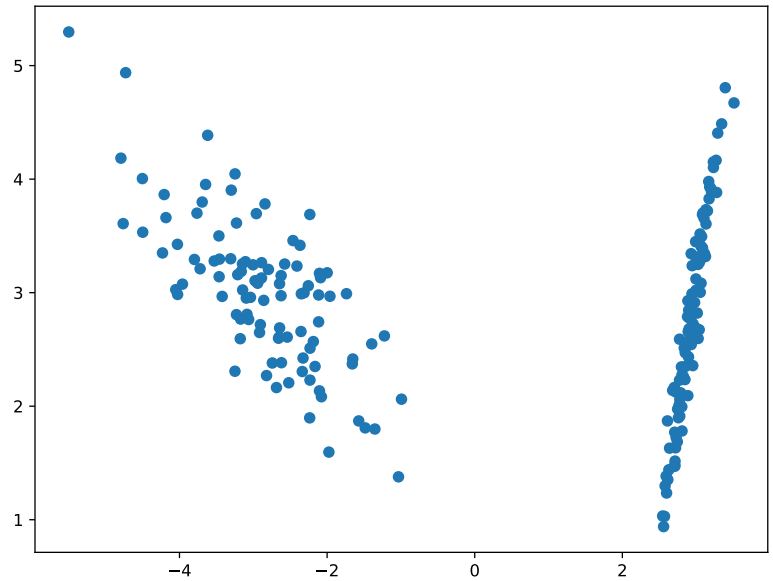


Learning Clustering in python

Let's build a simple clustering problem: Use `make_classification()` for building a two clusteres data as in the picture.

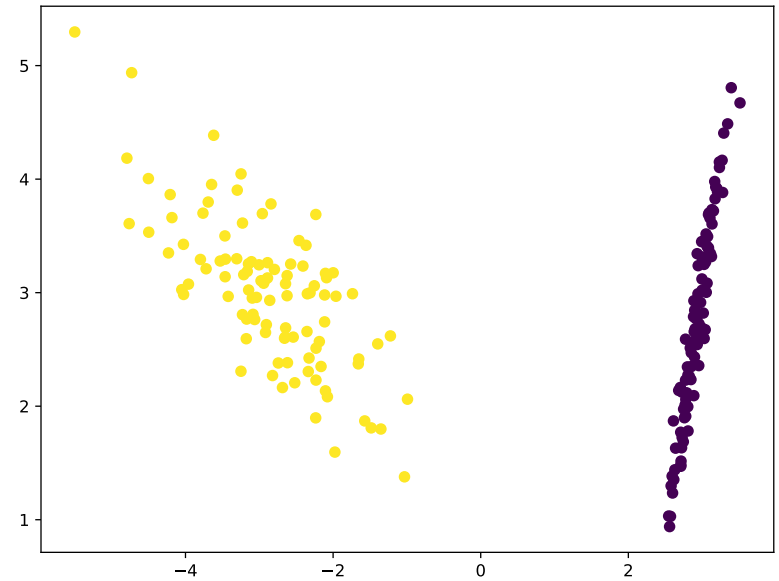
Could we automagically assign each sample to a cluster? Open your ipython and:

- Use k-means.



Learning Clustering in python

```
from sklearn.cluster import KMeans
cl = KMeans(n_clusters=2)
cl.fit(X)
pl.scatter(X[:,0],X[:,1], c=cl.label
```



Feature Extraction: subspace methods

Most feature extraction methods are based on subspace methods

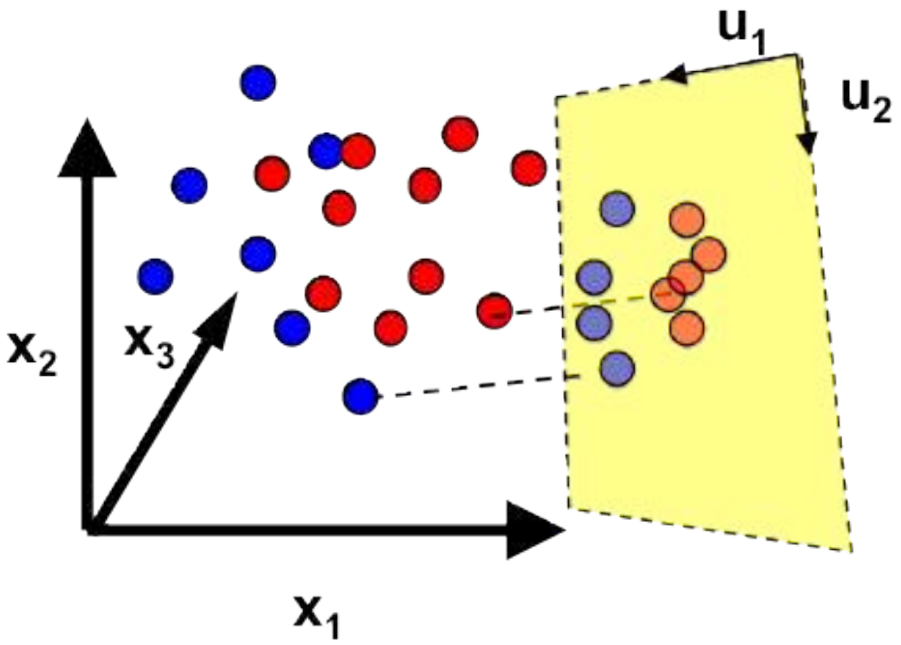
- Find subspace within the feature space
- Avoid curse of dimensionality
- Make use of some good features of the data structure

Feature Extraction: subspace methods

Most feature extraction methods are based on subspace methods

- Find subspace within the feature space
- Avoid curse of dimensionality
- Make use of some good features of the data structure

- Principal Component Analysis
- Linear Discriminant Analysis
- Independent Component Analysis



Principal Component Analysis

Most popular and commonly used methods. Sometimes included in the "Factor Analysis" methods.

- It's the second most main tool for visualization
  - (First is always to plot the signals!)

Principal Component Analysis

Most popular and commonly used methods. Sometimes included in the "Factor Analysis" methods.

- It's the second most main tool for visualization
  - (First is always to plot the signals!)
- Main goal of PCA is to capture main directions of variance in input space.

Principal Component Analysis

Most popular and commonly used methods. Sometimes included in the "Factor Analysis" methods.

- It's the second most main tool for visualization
  - (First is always to plot the signals!)
- Main goal of PCA is to capture main directions of variance in input space.
- Dimensionality reduction:

Principal Component Analysis

Most popular and commonly used methods. Sometimes included in the "Factor Analysis" methods.

- It's the second most main tool for visualization
  - (First is always to plot the signals!)
- Main goal of PCA is to capture main directions of variance in input space.
- Dimensionality reduction:
  - Allows for projecting the dataset onto a low dimensional subspace.

Principal Component Analysis

Most popular and commonly used methods. Sometimes included in the "Factor Analysis" methods.

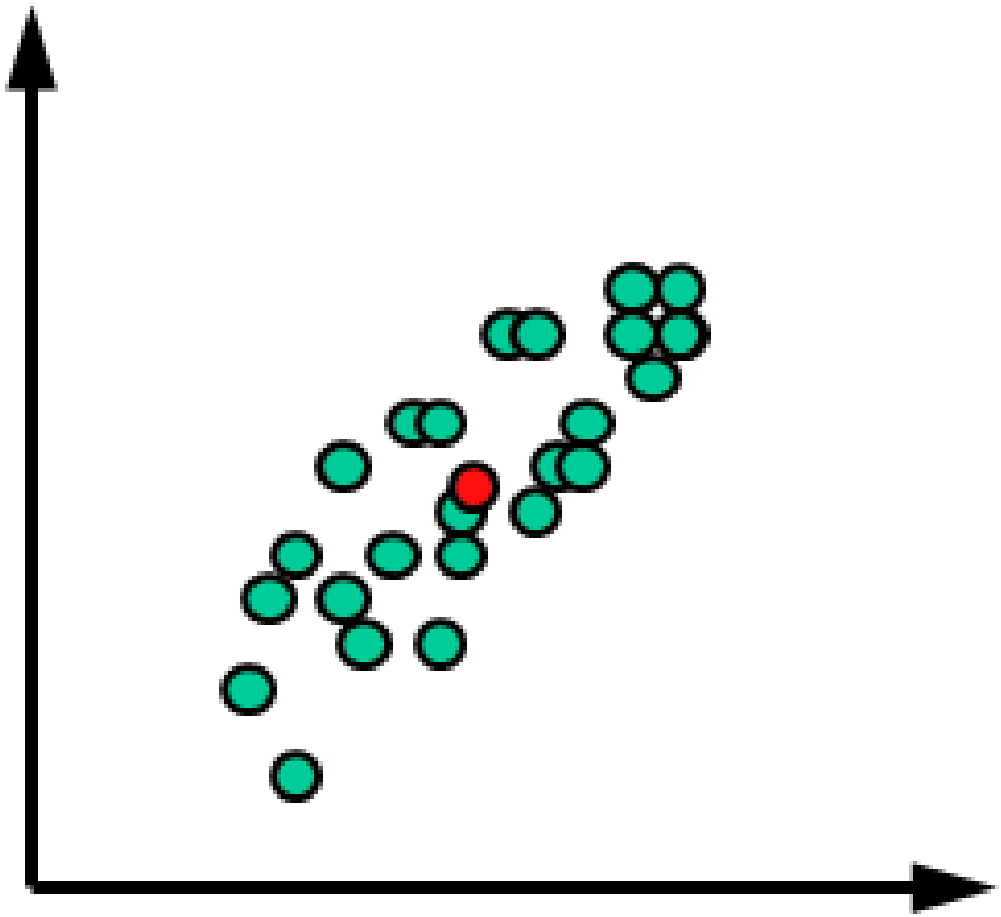
- It's the second most main tool for visualization
  - (First is always to plot the signals!)
- Main goal of PCA is to capture main directions of variance in input space.
- Dimensionality reduction:
  - Allows for projecting the dataset onto a low dimensional subspace.
  - Form of compression, or data modeling.

Principal Component Analysis

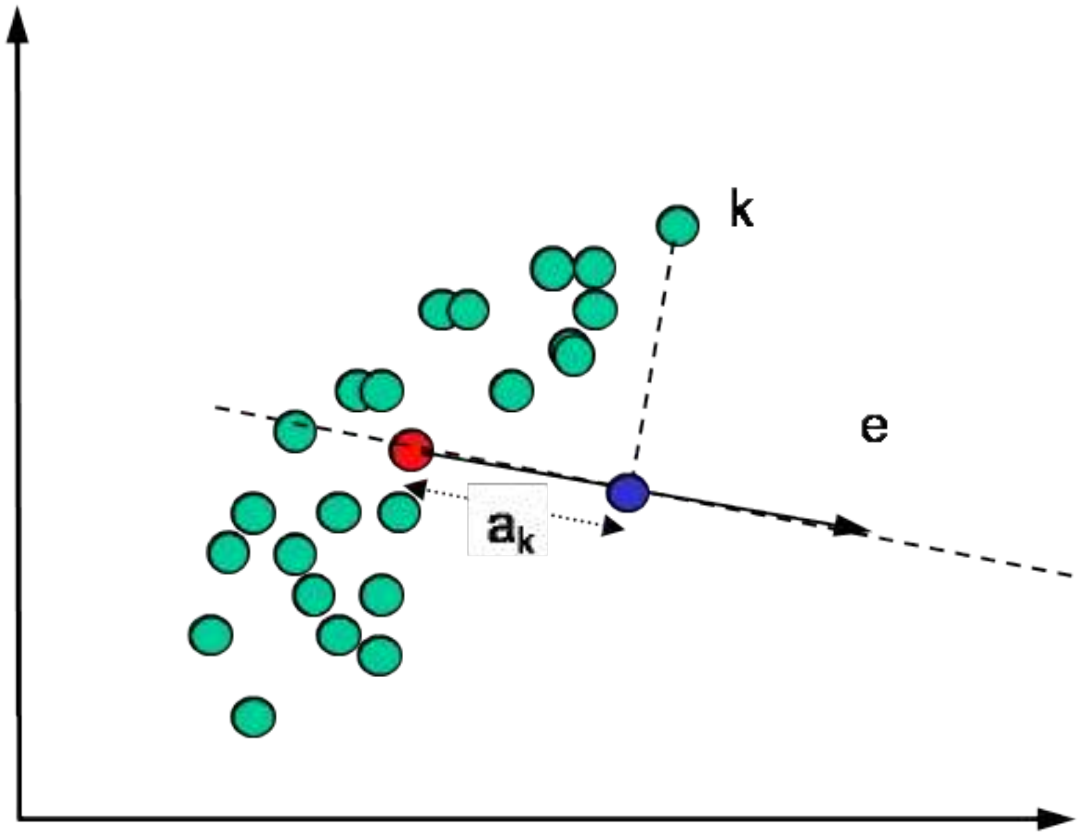
Most popular and commonly used methods. Sometimes included in the "Factor Analysis" methods.

- It's the second most main tool for visualization
  - (First is always to plot the signals!)
- Main goal of PCA is to capture main directions of variance in input space.
- Dimensionality reduction:
  - Allows for projecting the dataset onto a low dimensional subspace.
  - Form of compression, or data modeling.
  - Filtering.

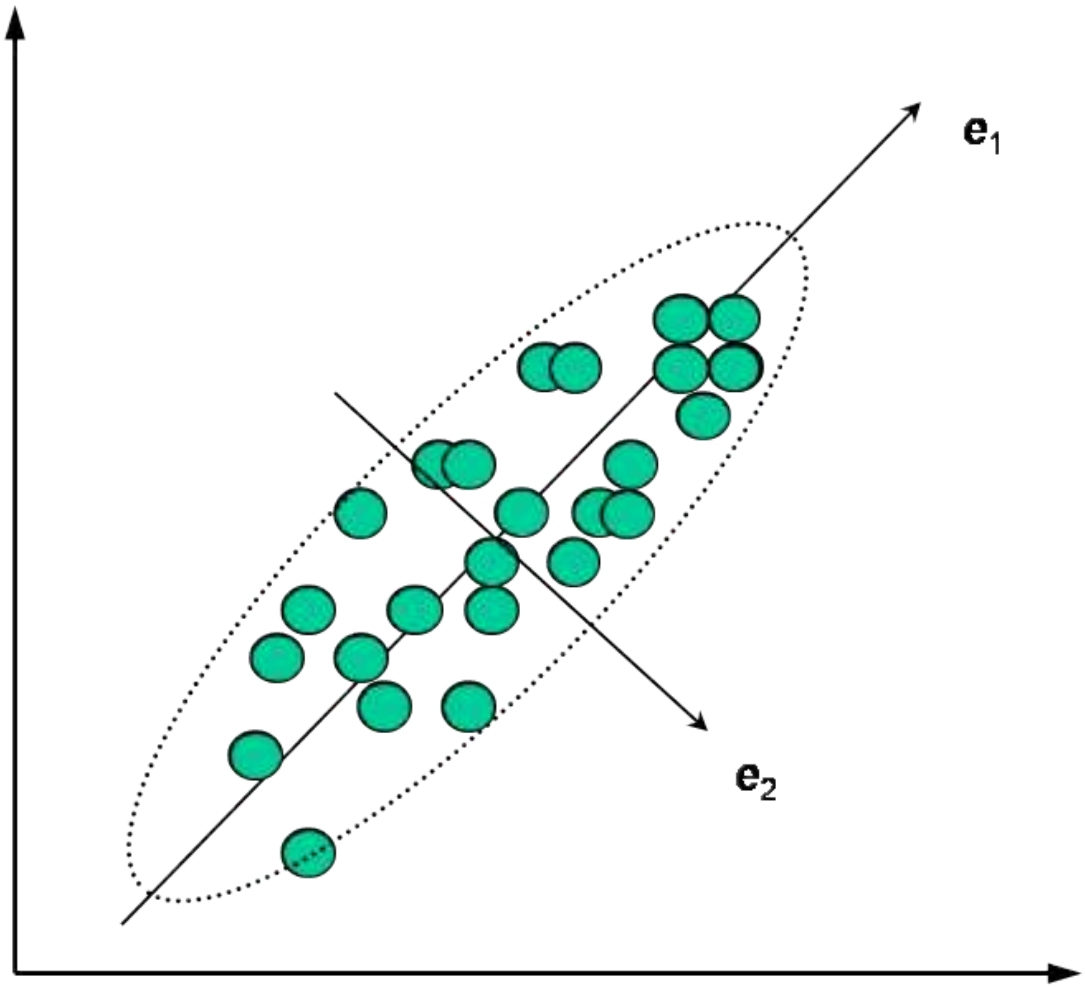
Principal Component Analysis



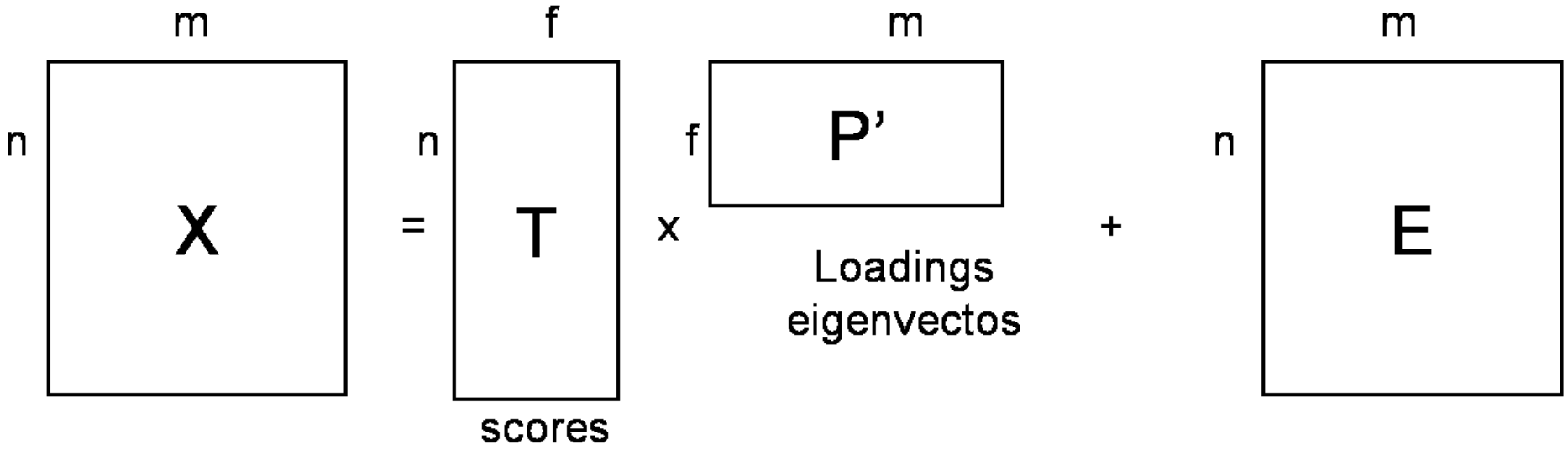
Principal Component Analysis



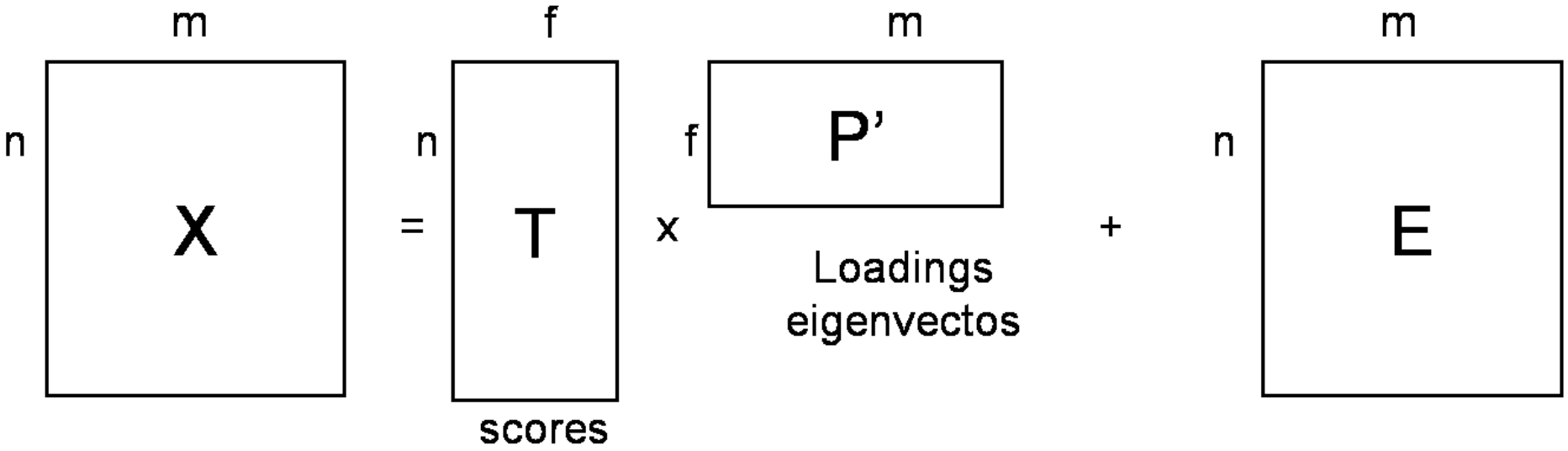
PCA

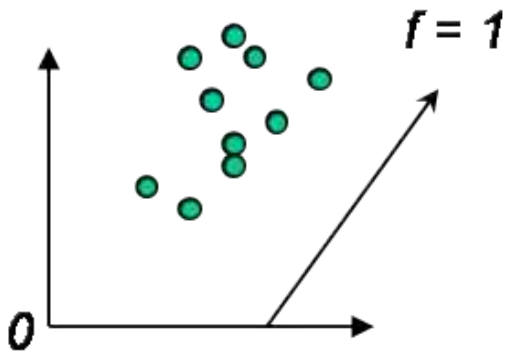


PCA



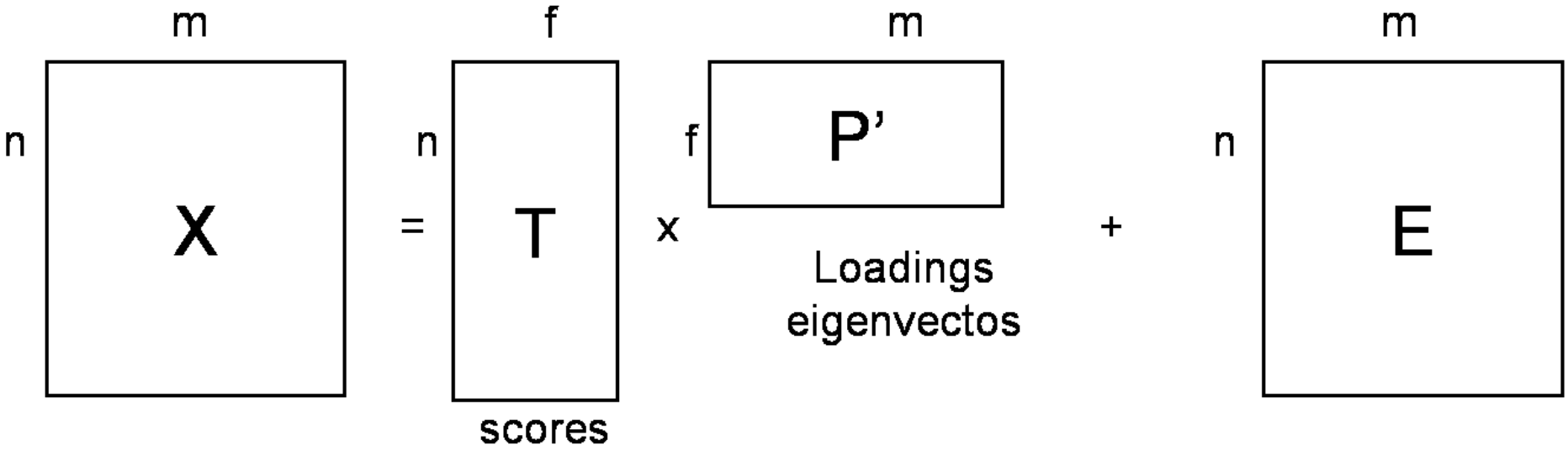
PCA

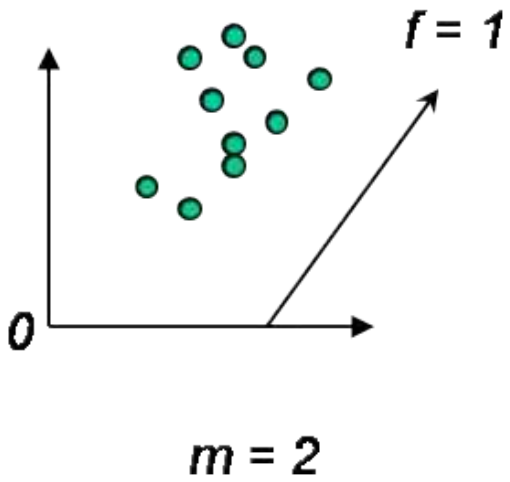


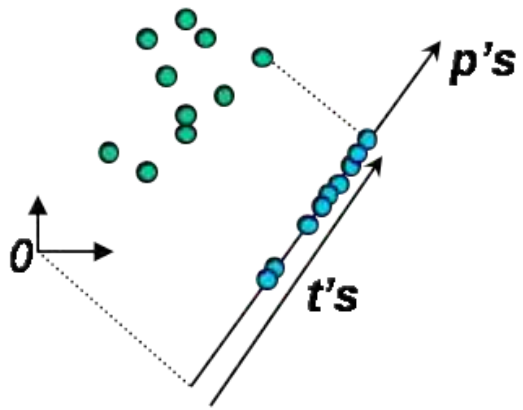


m = 2

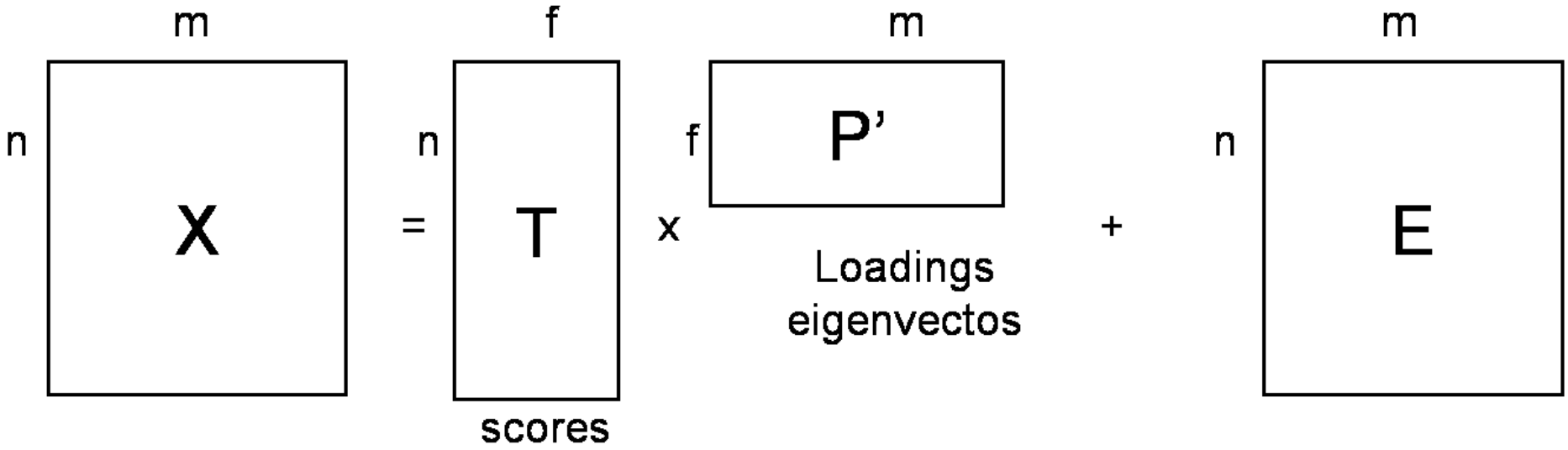
PCA

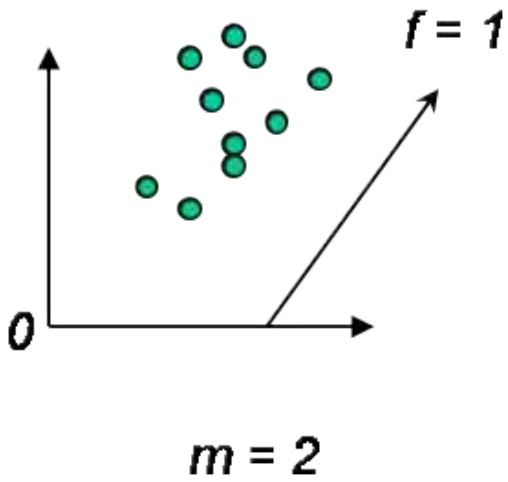


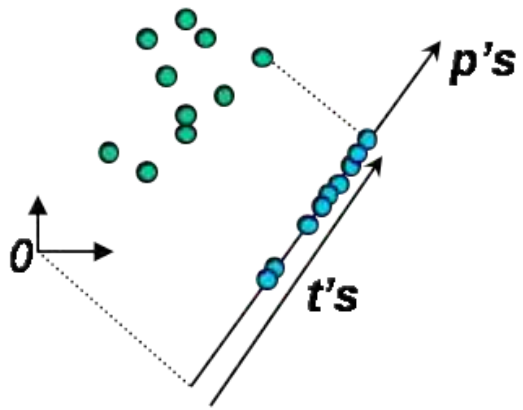


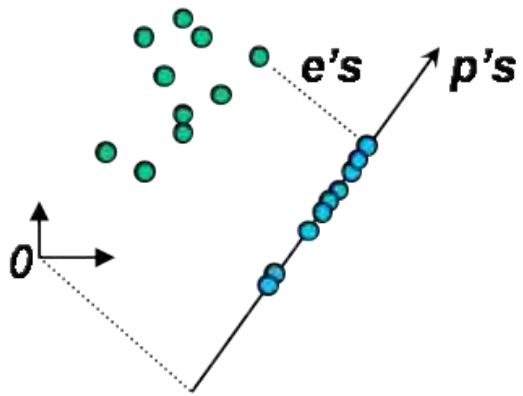


PCA

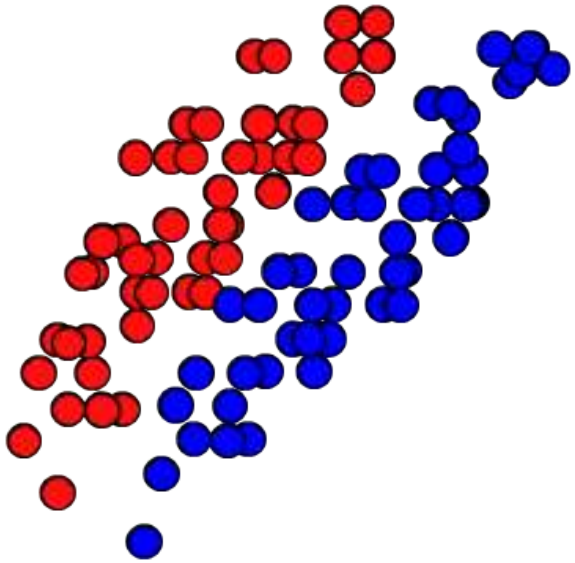

m
f
m
m
n
X
=
n
T
x
f
P'
+
n
E
scores
Loadings
eigenvectos


f = 1
0
m = 2

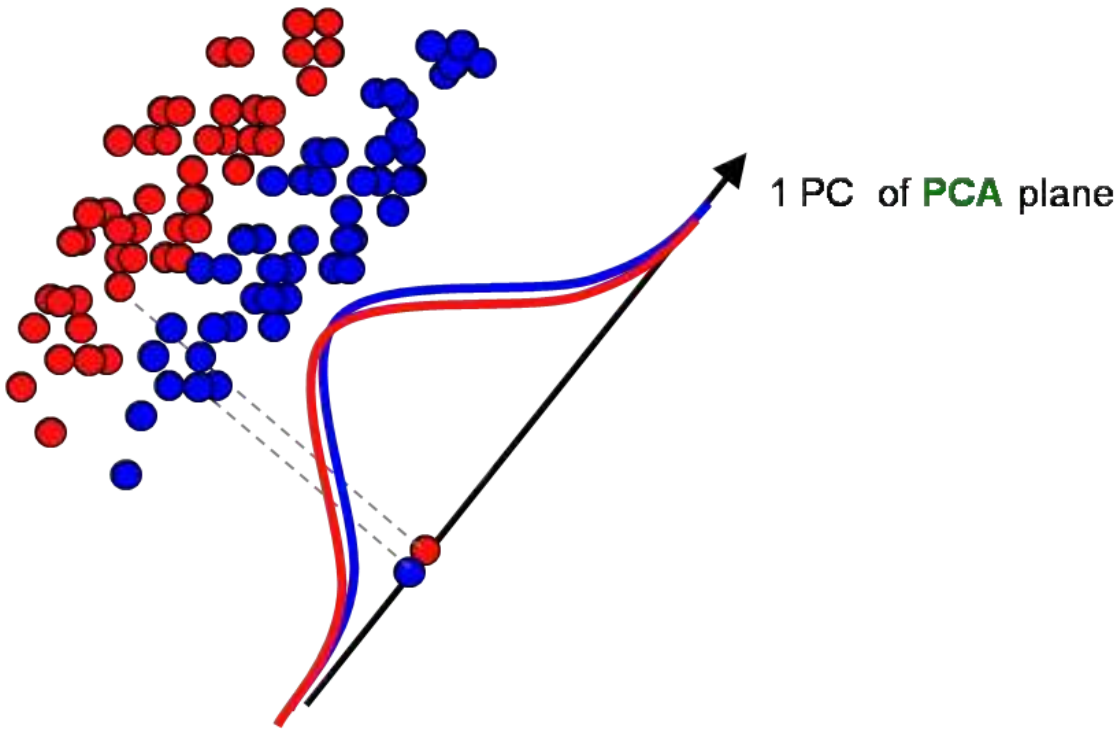

p's
0
t's


e's
p's
0

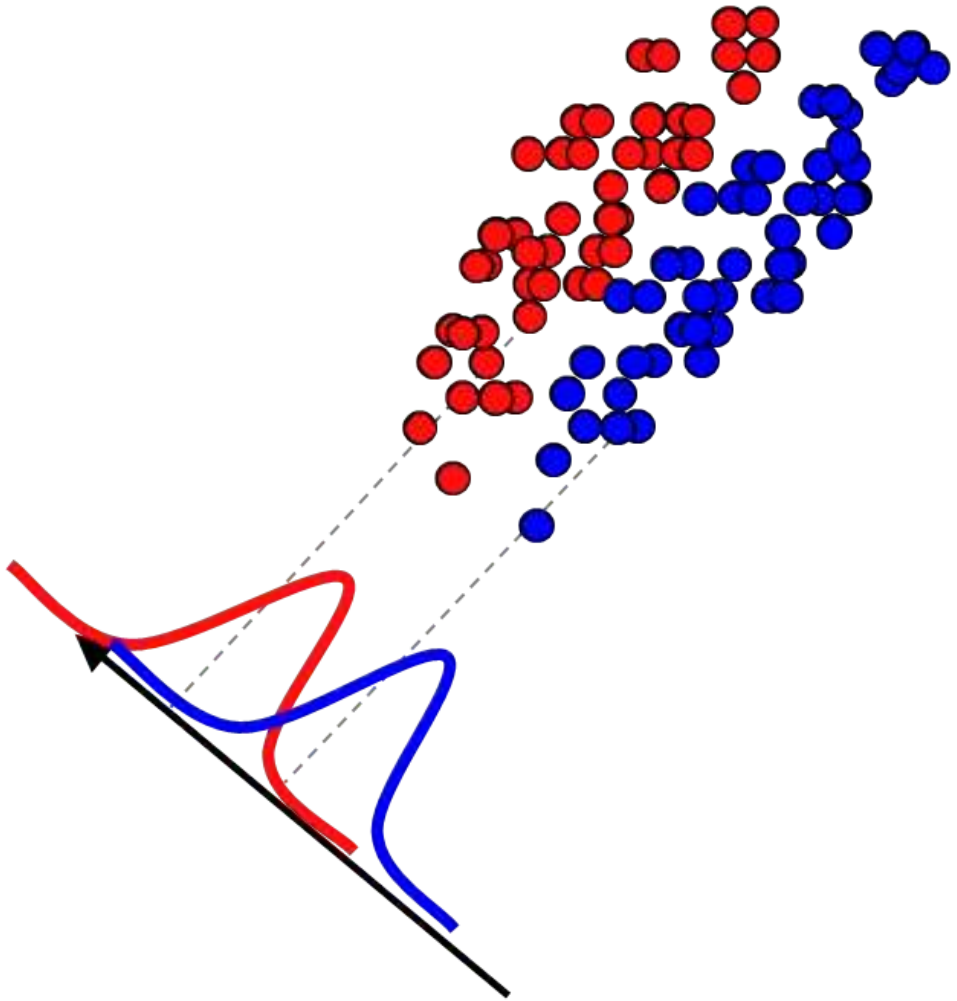
LDA vs PCA



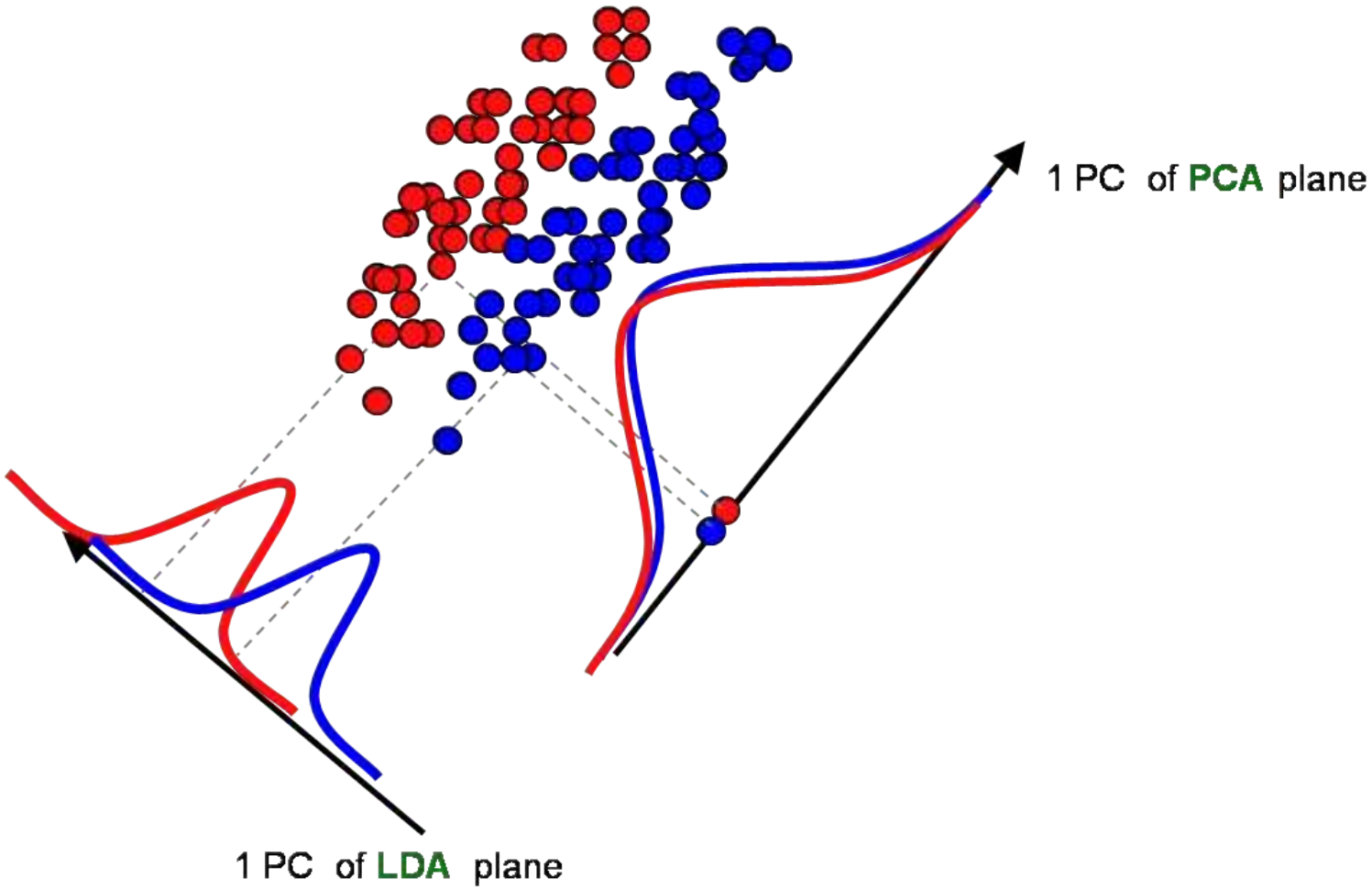
LDA vs PCA



LDA vs PCA



LDA vs PCA



The Iris dataset

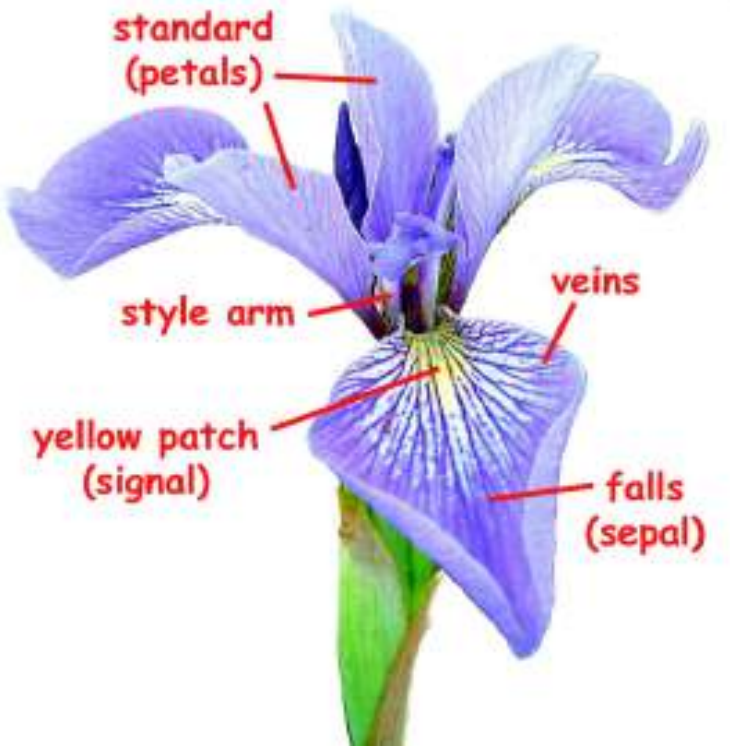
Iris dataset

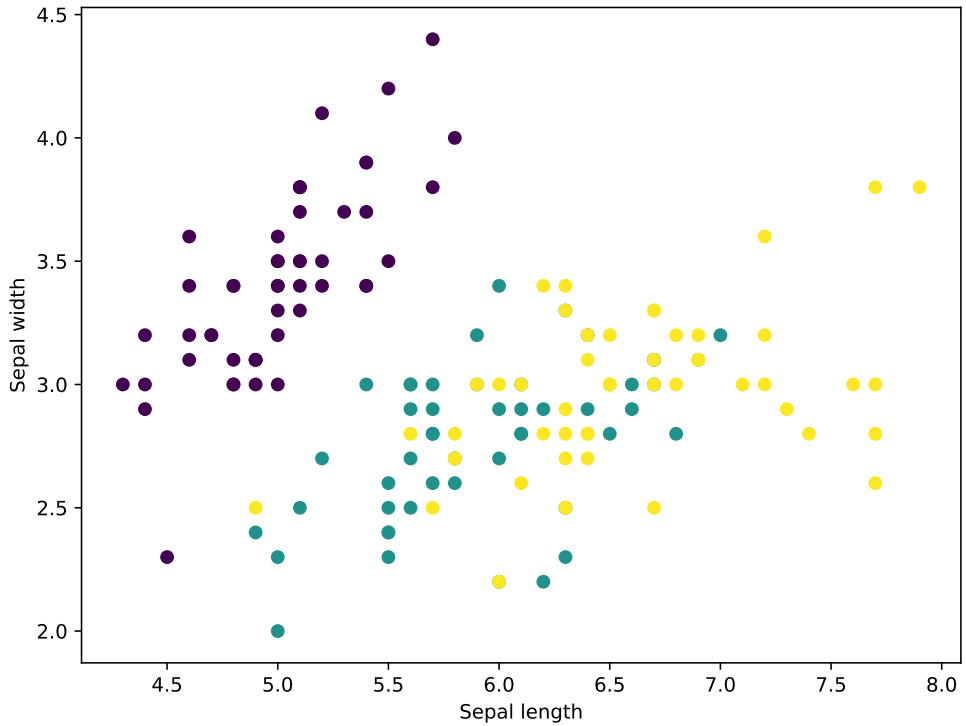
The iris dataset is a classical classification task consisting in identifying 3 different types of irises (Setosa, Versicolour, and Virginica) from their petal and sepal length and width.

```
>>> import numpy as np
>>> from sklearn import datasets
>>> import matplotlib.pylab as pl
>>> iris = datasets.load_iris()
>>> X = iris.data
>>> y = iris.target
>>> X[:2,:]
array([[ 5.1,  3.5,  1.4,  0.2],
       [ 4.9,  3. ,  1.4,  0.2]])
>>> np.unique(y)
array([0, 1, 2])
```

```
pl.scatter(X[:, 0], X[:, 1], c=y)
pl.xlabel('Sepal length')
pl.ylabel('Sepal width')
```

Iris Dataset





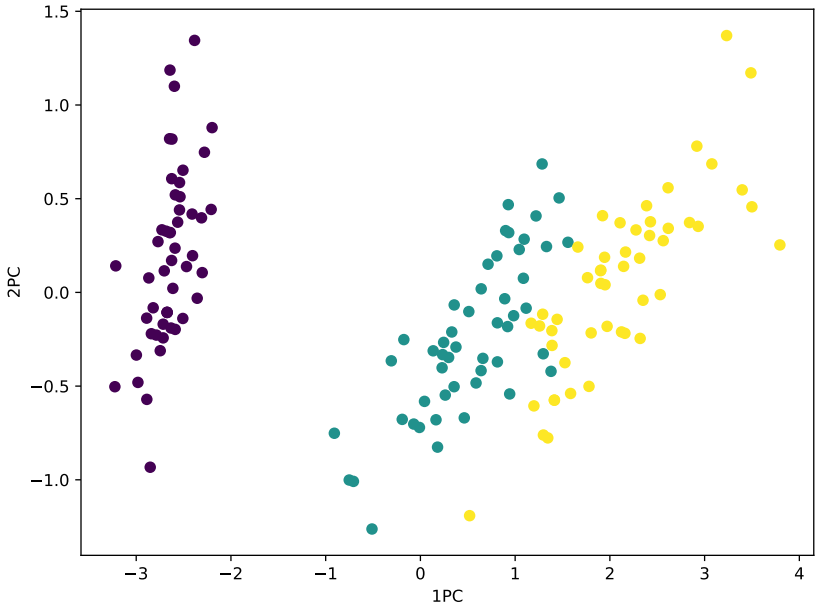
PCA in python

It's very easy now to construct the PCA projection:

```
>>> from sklearn import decomposition
>>> pca = decomposition.PCA(n_components=2)
>>> pca.fit(X)
PCA(copy=True, iterated_power='auto', n_components=2,
random_state=None,
  svd_solver='auto', tol=0.0, whiten=False)
>>> T = pca.transform(X)
```

PCA in python

```
import pylab as pl
pl.scatter(T[:, 0], T[:, 1], c=y)
pl.xlabel('1PC')
pl.ylabel('2PC')
```



Supervised Learning

Supervised Learning

Consists in learning the link between two datasets:

- An observed data X and
- A variable y usually called target or labels.

The k-Nearest Neighbour rule

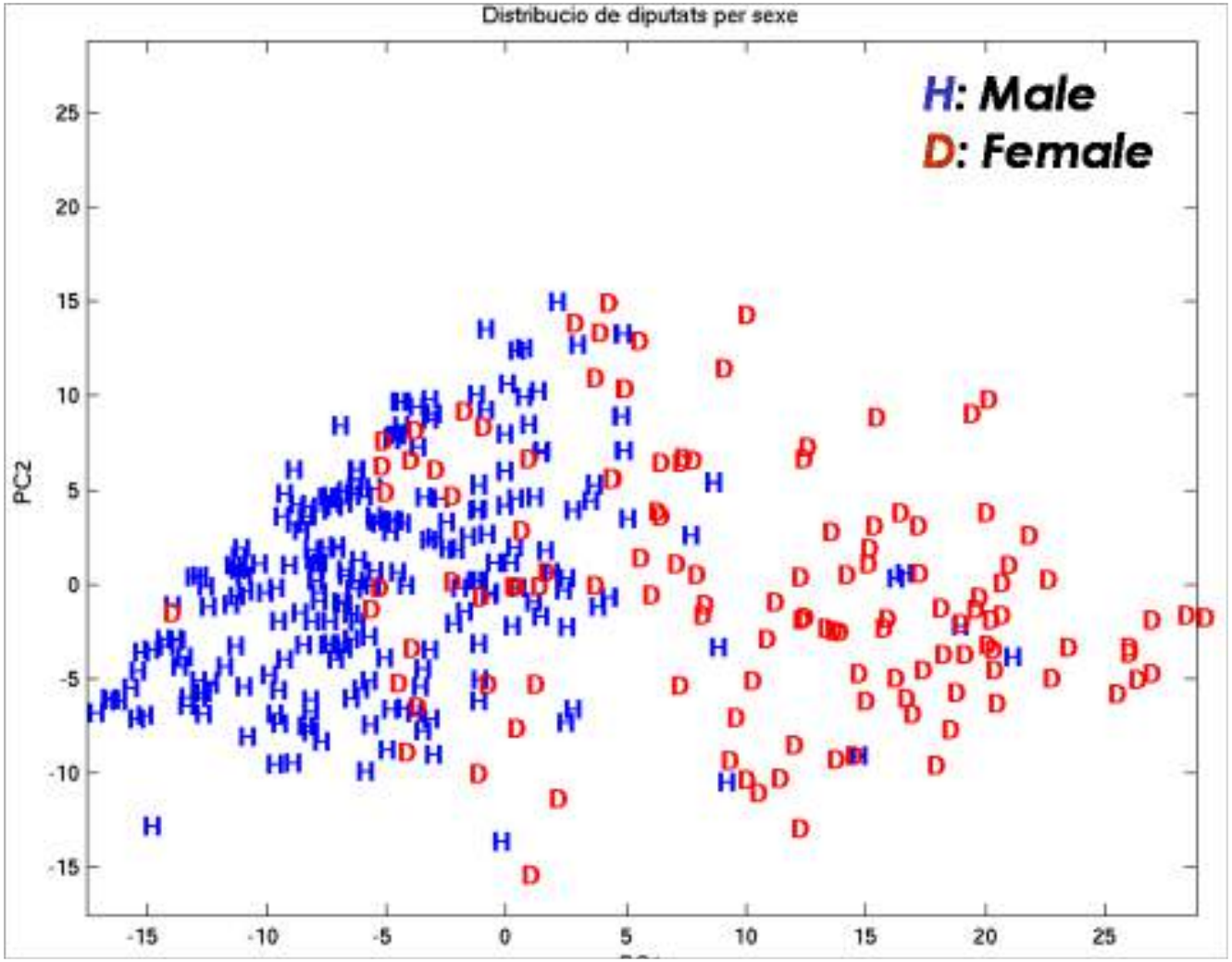
k-NN is a very intuitive lazy learning rule:

- Let X_u be an unknown sample
- Let $D^n = \{X_1, \cdots, X_n\}$ be a set of n labeled prototypes (Training Set).
- Let $D^k = \{X_1, \cdots, X_k\}$ be the k closest prototypes to x_u with $Y^k = \{y_1, \cdots, y_n\}$.
- Assign to x_u the class Y_u that appears most on D^k

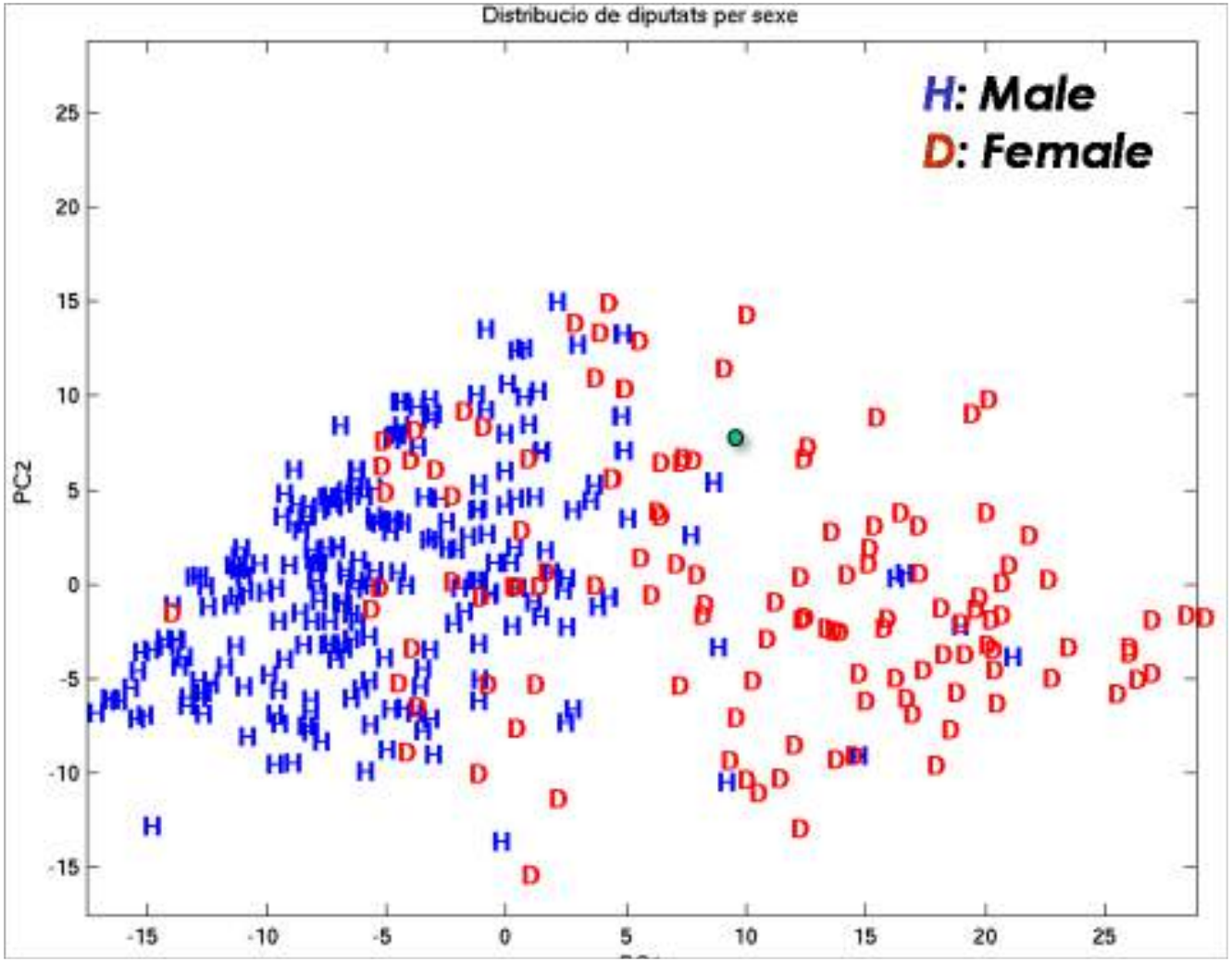
lazy?

- It does not process the dataset until there is a request to classify an unlabeled sample.
- They use to provide lower complexity costs in the training phase.
- Can be computationally expensive on recall phase.

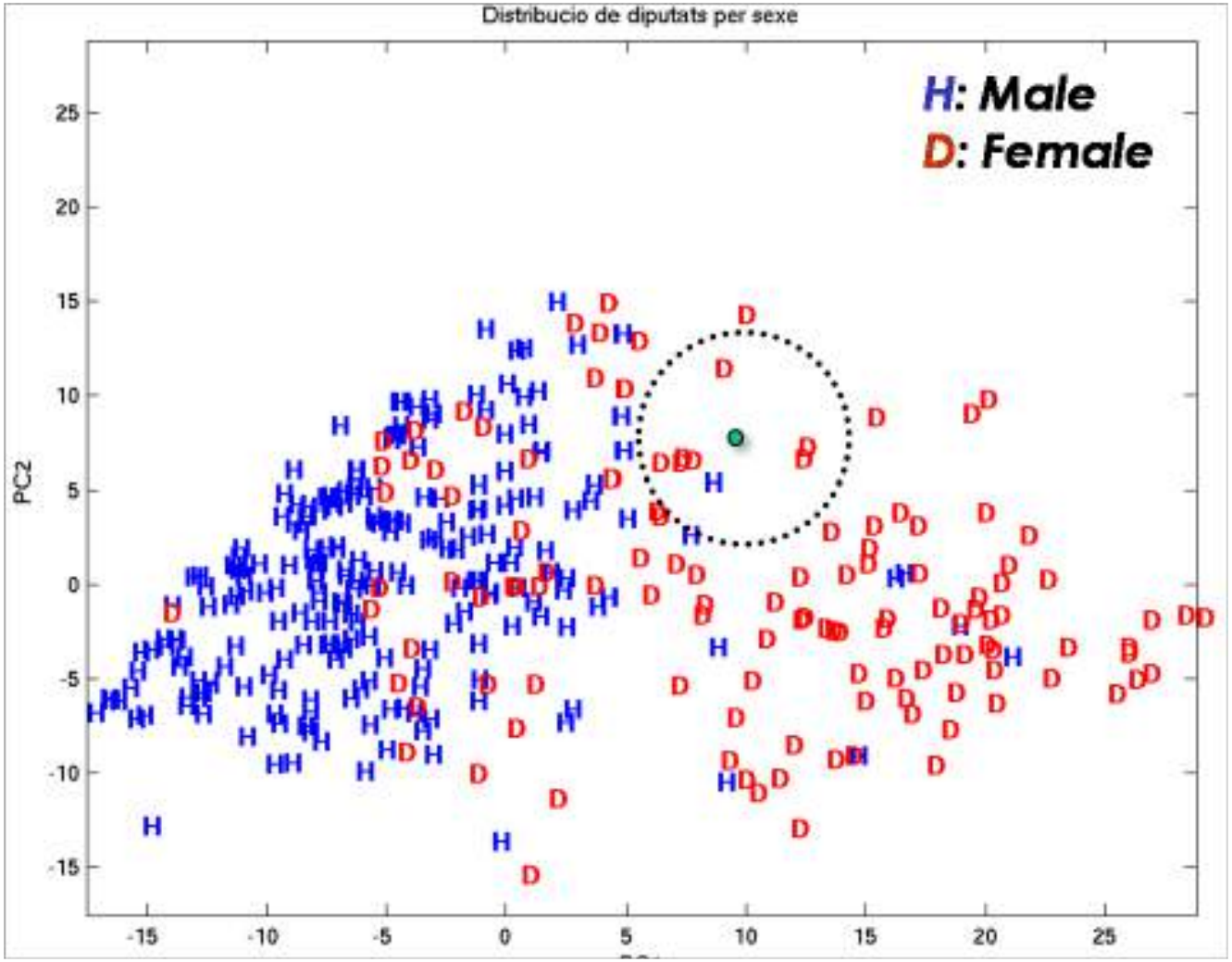
k-NN


Distribucio de diputats per sexe
H: Male
D: Female
PC2
25
20
15
10
5
0
-5
-10
-15
-15
-10
-5
0
5
10
15
20
25

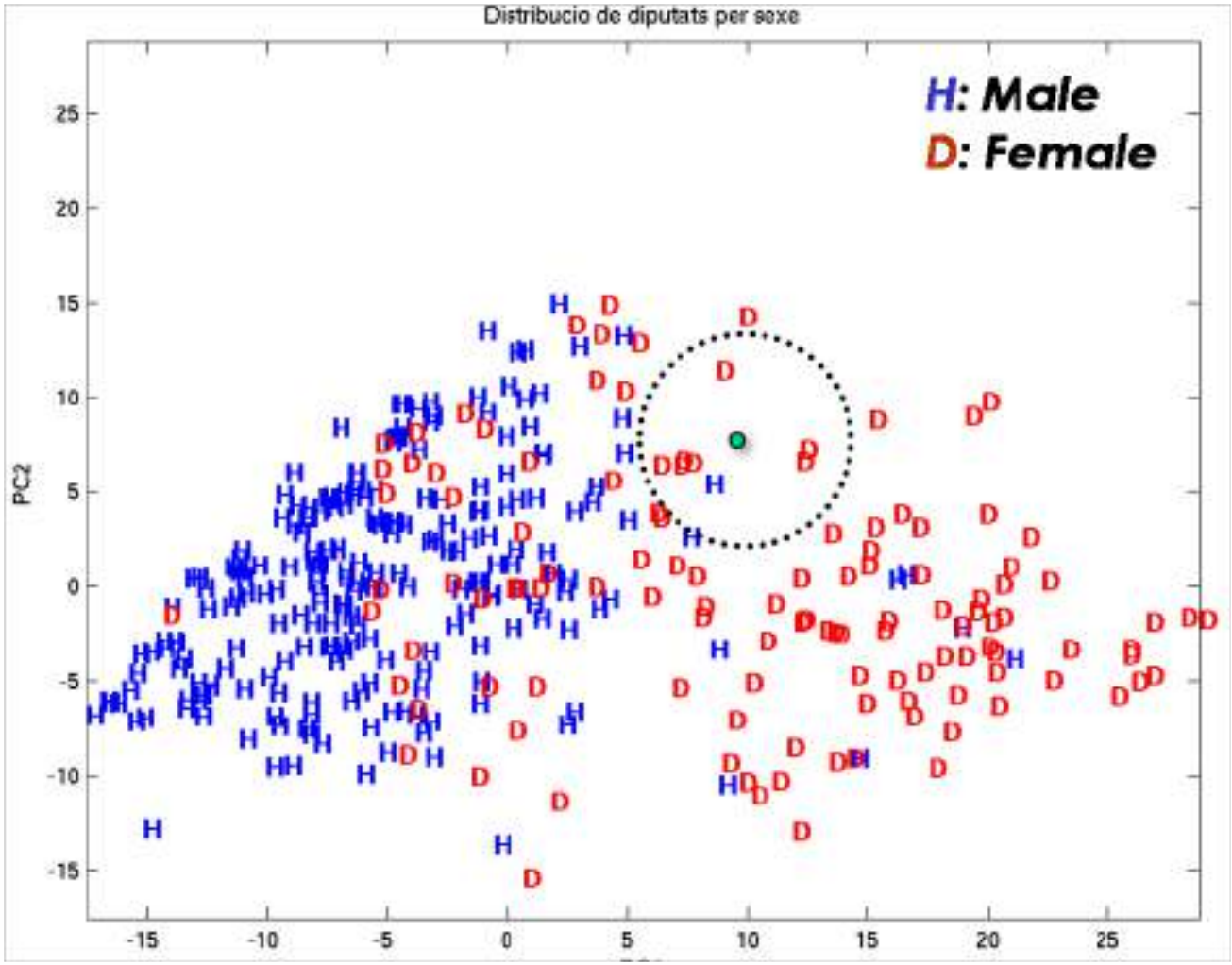
k-NN


Distribucio de diputats per sexe
H: Male
D: Female
PC2
25
20
15
10
5
0
-5
-10
-15
-15
-10
-5
0
5
10
15
20
25

k-NN

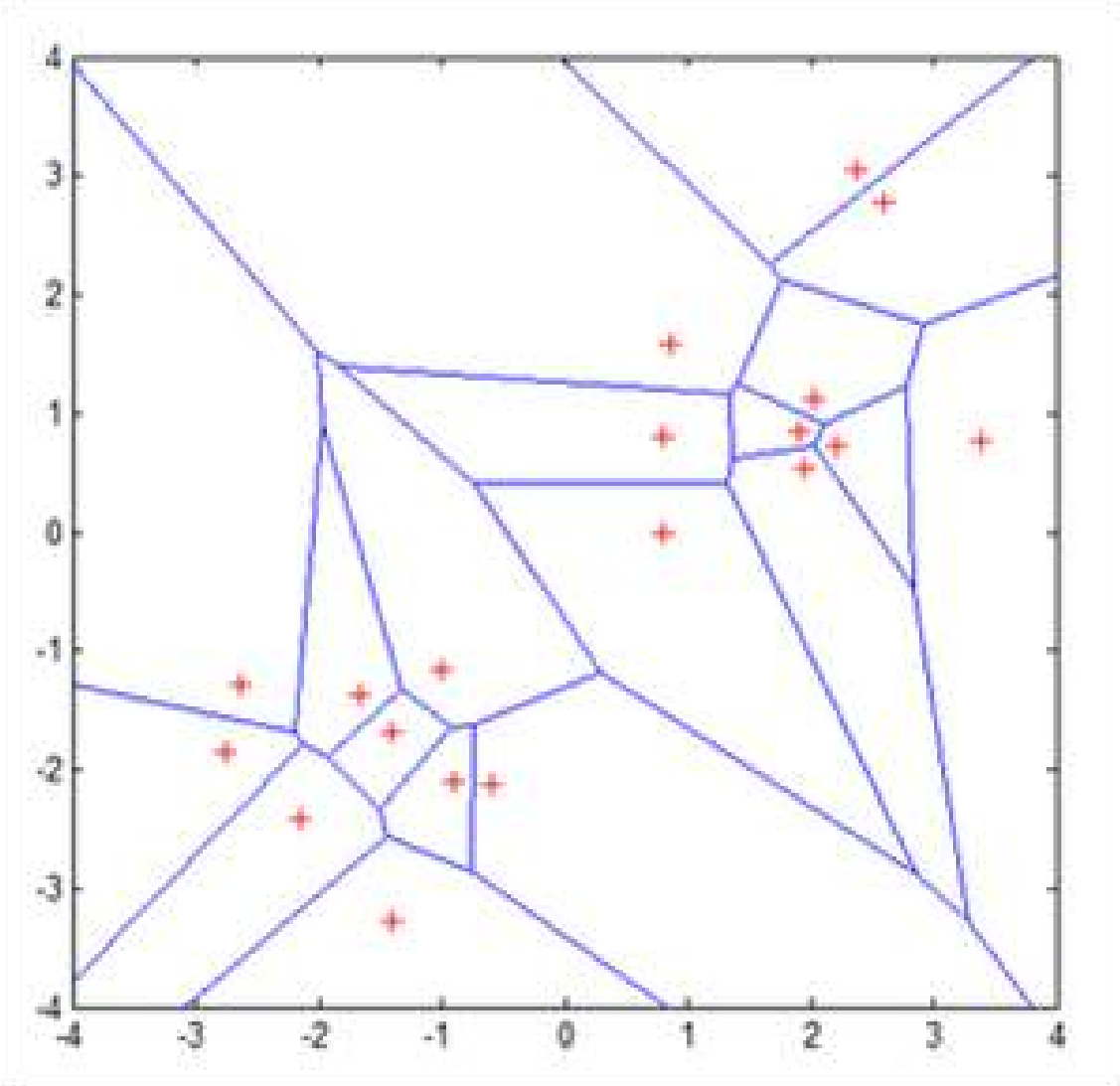

Distribucio de diputats per sexe
H: Male
D: Female
PC2
25
20
15
10
5
0
-5
-10
-15
-15
-10
-5
0
5
10
15
20
25

k-NN

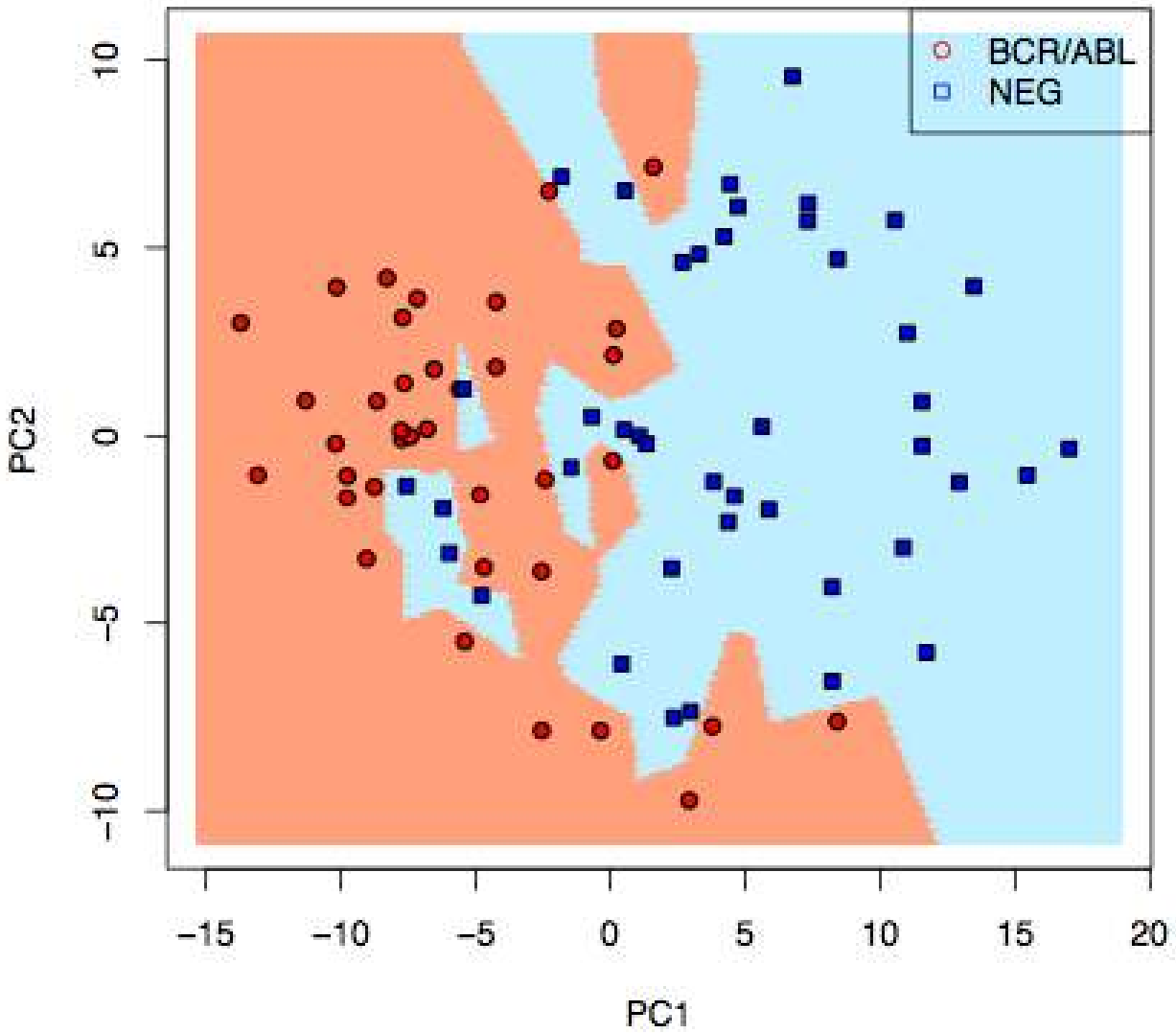


DDDDDDH = D

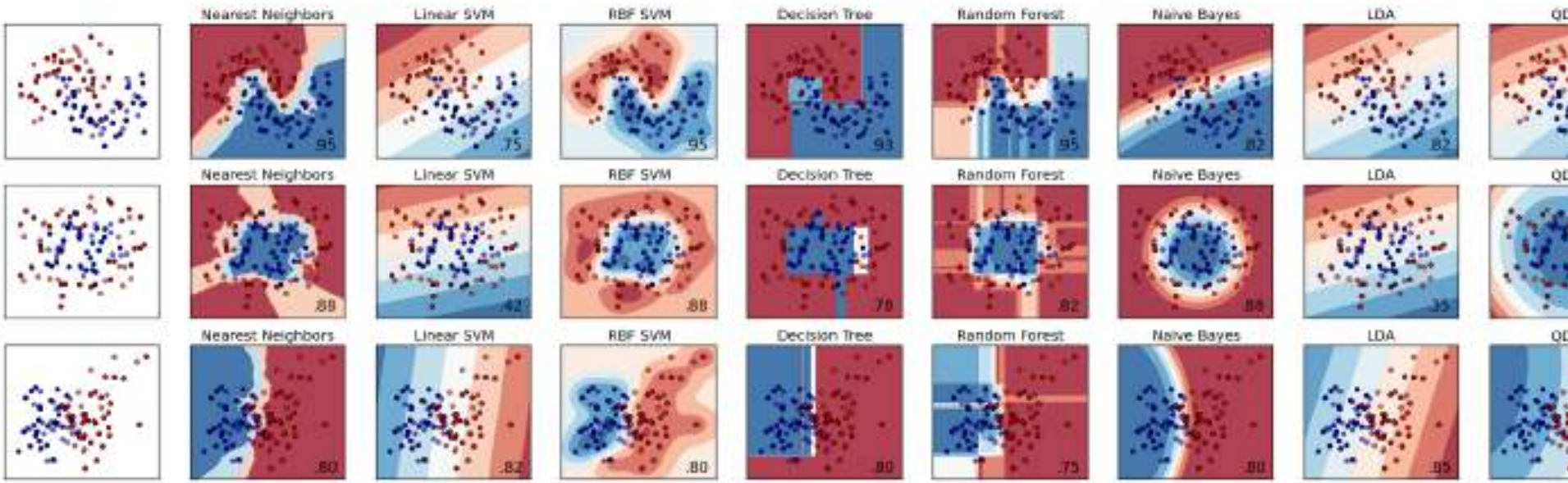
Voronoi Tessellation



Voronoi Tessellation



Classification algorithms in sklearn



k-nearest neighbours

```
>>> from sklearn import neighbors
>>> knn = neighbors.KNeighborsClassifier(7)
>>> knn.fit(T, y)
KNeighborsClassifier(algorithm='auto', leaf_size=30,
metric='minkowski',
           metric_params=None, n_jobs=1, n_neighbors=7, p=2,
           weights='uniform')
>>> knn.predict([[0.1, 0.2]])
array([1])
>>> yPred = knn.predict(T)
>>> (yPred==y).mean()
0.9799999999999998
```

Support Vector Classification

```
>>> from sklearn import svm
>>> C,gamma = 1,0.7
>>> svcg = svm.SVC(kernel='rbf', gamma=gamma, C=C).fit(T, y)
>>> svcg.predict([[-.7,7]])
array([2])
>>> yPred = svcg.predict(T)
>>> (yPred==y).mean()
0.95333333333333337
```

Section 5

Extra Slides

Decorators as function wrapper

Function can be decorated by using the decorator syntax for functions:

```
@mydecorator            # (2)
def function():         # (1)
    pass
```

```
def mydecorator(f)
    return f()
def function():                 # (1)
    pass
function = mydecorator(function)    # (2)
```

Decorators as function wrappers

Example

```
def helloSolarSystem(original_function):
    def new_function():
        original_function()  # the () after "original_function" causes original_function to be called
        print("Hello, solar system!")
    return new_function

def helloGalaxy(original_function):
    def new_function():
        original_function()  # the () after "original_function" cause original_function to be called
        print("Hello, galaxy!")
    return new_function

@helloGalaxy
@helloSolarSystem
def hello():
    print ("Hello, world!")

# Here is where we actually *do* something!
hello()
```

Checkout the result of this structure

Debug with decorators

Just for fun

```
def debug(f):
    def my_wrapper(*args,**kwargs):
        call_string = "%s called with *args: %r, **kwargs: %r " % (f.__name__, args, kwargs)
        ret_val=f(*args,**kwargs)
        call_string+=repr(ret_val)
        if debugging:
            print(call_string)
        return ret_val
    return my_wrapper

@debug
def recursive(k):
    if k>1:
        return k*recursive(k-1)
    else:
        return 1

debugging=False
recursive(3)
debugging=True
recursive(3)
```

Scripts

First script

A sequence of instructions that are executed each time the script is called.
Instructions may be e.g. copied-and-pasted from the interpreter (but take care to respect indentation rules!).

```
message = "Hello how are you?"
for word in message.split():
    print(word)
```

Scripts

in Ipython, the syntax to execute a script is %run script.py. For example,

```
In [1]: %run test.py
Hello
how
are
you?

In [2]: message
Out[2]: 'Hello how are you?'
```

From de command line

```
mv->mv-PC:~/Curs_Python\$ python test.py
Hello
how
are
you?
```

Scripts

Standalone scripts may also take command-line arguments

in file.py:

```
import sys
print(sys.argv)
```

when executed

```
$ python file.py test arguments
['file.py', 'test', 'arguments']
```

Modules

Importing objects from modules

```
In [1]: import os

In [2]: os
Out[2]: <module 'os' from '/usr/lib/python2.6/os.pyc'>

In [3]: os.listdir('.')
Out[3]:
['conf.py',
 'basic_types.rst',
 'control_flow.rst',
 'functions.rst',
 'python_language.rst',
 'reusing.rst',
 'file_io.rst',
 'exceptions.rst',
 'workflow.rst',
 'index.rst']
```

Try to check how many functions are there in os with tab-completion and ipython

Modules

Alternatives to full import

Import only some functions

```
In [4]: from os import listdir
```

Or a shorthand

```
In [5]: import numpy as np
```

Modules

Actually, all the scientific computing tools we are going to use are modules:

```
>>> import numpy as np # data arrays
>>> np.linspace(0, 10, 6)
array([  0.,   2.,   4.,   6.,   8.,  10.])
>>> import scipy # scientific computing
```

My own module

```
"A demo module."

def print_b():
    "Prints b."
    print('b')
def print_a():
    "Prints a."
    print('a')
c = 2
d = 2
```

```
In [1]: import demo
In [2]: demo.print_a()
a
In [3]: demo.print_b()
b
```

Try this in ipython

```
In [4]: demo?
In [5]: who
In [6]: whos
In [7]: dir(demo)
In [8]: demo.    #tab-completion
```

Modules

Warning:Module caching

‘main’ and module loading

A script and a Module

```
def print_a():
    "Prints a."
    print('a')


if __name__ == '__main__':
    print_a()
```

```
In [12]: import demo2
In [13]: %run demo2
a
```

Input and Output

To write in a file:

```
>>> f = open('workfile', 'w') # opens the workfile file
>>> type(f)
<type 'file'>
>>> f.write('This is a test \nand another test')
>>> f.close()
```

Input and Output

To read from a file

```
In [1]: f = open('workfile', 'r')

In [2]: s = f.read()

In [3]: print(s)
This is a test
and another test

In [4]: f.close()
```

Input and Output

Iterating over a file

```
In [6]: f = open('workfile', 'r')

In [7]: for line in f:
...:     print(line)
...:
...:
This is a test
and another test
In [8]: f.close()
```

Challenge

10 Minutes challenge

Write a script that reads a file with a column of numbers and calculates the min, max and sum

Challenge

10 minutes challenge

Write a module that performs basic trigonometric functions using Taylor expansions

OS module: Operating system functionality

Directory and file manipulation

Current directory:

```
In [17]: os.getcwd()
Out[17]: '/Users/cburns/src/scipy2009/scipy_2009_tutorial/source'
```

List a directory:

```
In [31]: os.listdir(os.curdir)
Out[31]:
['.index.rst.swo',
 '.python_language.rst.swp',
 '.view_array.py.swp',
 '_static',
 '_templates',
 'basic_types.rst',
 'conf.py',
 'control_flow.rst',
 'debugging.rst',
 ...
```

OS module: Operating system functionality

Make a directory

```
In [32]: os.mkdir('junkdir')
In [33]: 'junkdir' in os.listdir(os.curdir)
Out[33]: True
```

Rename the directory:

```
In [36]: os.rename('junkdir', 'foodir')
In [37]: 'junkdir' in os.listdir(os.curdir)
Out[37]: False
In [38]: 'foodir' in os.listdir(os.curdir)
Out[38]: True
In [41]: os.rmdir('foodir')
In [42]: 'foodir' in os.listdir(os.curdir)
Out[42]: False
```

Delete a file:

```
In [44]: fp = open('junk.txt', 'w')
In [45]: fp.close()
In [46]: 'junk.txt' in os.listdir(os.curdir)
Out[46]: True
In [47]: os.remove('junk.txt')
In [48]: 'junk.txt' in os.listdir(os.curdir)
Out[48]: False
```

os.path: path manipulations

os.path provides common operations on pathnames.

```
In [70]: fp = open('junk.txt', 'w')
In [71]: fp.close()
In [72]: a = os.path.abspath('junk.txt')
In [73]: a
Out[73]: '/Users/cburns/src/scipy2009/scipy_2009_tutorial/source/junk.txt'
In [74]: os.path.split(a)
Out[74]: ('/Users/cburns/src/scipy2009/scipy_2009_tutorial/source','junk.txt')
In [78]: os.path.dirname(a)
Out[78]: '/Users/cburns/src/scipy2009/scipy_2009_tutorial/source'
In [79]: os.path.basename(a)
Out[79]: 'junk.txt'
In [80]: os.path.splitext(os.path.basename(a))
Out[80]: ('junk', '.txt')
In [84]: os.path.exists('junk.txt')
Out[84]: True
In [86]: os.path.isfile('junk.txt')
Out[86]: True
In [87]: os.path.isdir('junk.txt')
Out[87]: False
In [88]: os.path.expanduser('~/local')
Out[88]: '/Users/cburns/local'
In [92]: os.path.join(os.path.expanduser('~'), 'local', 'bin')
Out[92]: '/Users/cburns/local/bin'
```

Other OS services

Running an external command

```
In [3]: os.system('ls *.tex')
commondefs.tex        CursP_1.tex  CursP_3.tex
CursP_4.tex     format.tex   header.tex
```

Walking a directory

```
In [4]: for dirpath, dirnames, filenames in
os.walk(os.curdir):
   ...:     for fp in filenames:
   ...:         print(os.path.abspath(fp))
   ...:
/home/Dropbox/Curs_Python/CursP_3.log
/home/Dropbox/Curs_Python/CursP_4.out
/home/Dropbox/Curs_Python/syllabus.odt
/home/Dropbox/Curs_Python/format.tex
/home/Dropbox/Curs_Python/CursP_3.pdf
/home/Dropbox/Curs_Python/tags
/home/Dropbox/Curs_Python/CursP_3.vrb
```

glob: Pattern matching on files

```
In [5]: import glob
In [6]: glob.glob('*.tex')
Out[6]:
['format.tex',
 'CursP_4.tex',
 'header.tex',
 'CursP_1.tex',
 'CursP_3.tex',
 'commondefs.tex']
```

sys module: system-specific information

```
In [8]: import sys
In [9]: sys.platform
Out[9]: 'linux2'
In [10]: sys.version
Out[10]: '2.7.3 (default, Aug  1 2012, 05:14:39) \n[GCC 4.6.3]'
In [11]: sys.prefix
Out[11]: '/usr'
```

Object-oriented programming

OOP

We are not going to use OOP in this course, but we provide some snippets of code just to know the structure of class declaration

Object-oriented programming

Class Declaration

```
>>> class Student(object):
...     def __init__(self, name):
...         self.name = name
...     def set_age(self, age):
...         self.age = age
...     def set_major(self, major):
...         self.major = major
...
>>> anna = Student('anna')
>>> anna.set_age(21)
>>> anna.set_major('physics')
```

Class extension

```
>>> class MasterStudent(Student):
...     internship = 'mandatory, from March to June'
...
>>> james = MasterStudent('james')
>>> james.internship
'mandatory, from March to June'
>>> james.set_age(23)
>>> james.age
23
```

Exceptions

Exceptions are raised by errors in Python:

```
In [1]: 1/0
---------------------------------------------------------------------------
ZeroDivisionError: integer division or modulo by zero
In [2]: 1 + 'e'
---------------------------------------------------------------------------
TypeError: unsupported operand type(s) for +: 'int' and 'str'
In [3]: d = {1:1, 2:2}
In [4]: d[3]
---------------------------------------------------------------------------
KeyError: 3
In [5]: l = [1, 2, 3]
In [6]: l[4]
---------------------------------------------------------------------------
IndexError: list index out of range
In [7]: l.foobar
---------------------------------------------------------------------------
AttributeError: 'list' object has no attribute 'foobar'
```

Catching exceptions

try/except

```
In [8]: while True:
 ....:     try:
 ....:         x = int(raw_input('Please enter a number: '))
 ....:         break
 ....:     except ValueError:
 ....:         print('That was no valid number.  Try again...')
 ....:
 ....:
Please enter a number: a
That was no valid number.  Try again...
Please enter a number: 1

In [9]: x
Out[9]: 1
```

Catching exceptions

try/finally

Important for resource management (e.g. closing a file)

```
In [10]: try:
   ....:     x = int(raw_input('Please enter a number: '))
   ....: finally:
   ....:     print('Thank you for your input')
   ....:
   ....:
Please enter a number: a
Thank you for your input
---------------------------------------------------------------------------
ValueError: invalid literal for int() with base 10: 'a'
```

There are many tricks with the exceptions, but they are out of the scope of these slides

Parameters

More involved example implementing python's slicing:

```
In [98]: def slicer(seq, start=None, stop=None, step=None):
   ....:     """Implement basic python slicing."""
   ....:     return seq[start:stop:step]
   ....:

In [101]: rhyme = 'one fish, two fish, red fish, blue fish'.split()

In [102]: rhyme
Out[102]: ['one', 'fish,', 'two', 'fish,', 'red', 'fish,', 'blue', 'fish']

In [103]: slicer(rhyme)
Out[103]: ['one', 'fish,', 'two', 'fish,', 'red', 'fish,', 'blue', 'fish']

In [104]: slicer(rhyme, step=2)
Out[104]: ['one', 'two', 'red', 'blue']

In [105]: slicer(rhyme, 1, step=2)
Out[105]: ['fish,', 'fish,', 'fish,', 'fish']

In [106]: slicer(rhyme, start=1, stop=4, step=2)
Out[106]: ['fish,', 'fish,']
```

Parameters and mutability

5 minutes challenge, solution

```
>>> def try_to_modify(x, y, z):
...     x = 23
...     y.append(42)
...     z = [99] # new reference
...     print(x)
...     print(y)
...     print(z)
...
>>> a = 77    # immutable variable
>>> b = [99]  # mutable variable
>>> c = [28]
>>> try_to_modify(a, b, c)
23
[99, 42]
[99]
>>> print(a)
77
>>> print(b)
[99, 42]
>>> print(c)
[28]
```

Global variables

Variables declared outside the function can be referenced within the function:

```
In [114]: x = 5

In [115]: def addx(y):
   .....:     return x + y
   .....:

In [116]: addx(10)
Out[116]: 15
```

But..

This doesn't work:

```
x=5
In [117]: def setx(y):
   .....:     x = y
   .....:     print('x is %d' % x)
   .....:
   .....:
In [118]: setx(10)
x is 10
In [120]: x
Out[120]: 5
```

This works:

```
x=5
In [121]: def setx(y):
   .....:     global x
   .....:     x = y
   .....:     print('x is %d' % x)
   .....:
   .....:
In [122]: setx(10)
x is 10
In [123]: x
Out[123]: 10
```

Variable number of parameters

Special forms of parameters:

- *args any number of positional arguments packed into a tuple
- **kwargs any number of keyword arguments packed into a dictionary

```
In [35]: def variable_args(*args, **kwargs):
   ....:     print 'args is', args
   ....:     print 'kwargs is', kwargs
   ....:

In [36]: variable_args('one', 'two', x=1, y=2, z=3)
args is ('one', 'two')
kwargs is {'y': 2, 'x': 1, 'z': 3}
```

Docstrings

Documentation about what the function does and it's parameters. General convention:

```
In [67]: def funcname(params):
   ....:     """Concise one-line sentence describing the function.
   ....:
   ....:     Extended summary which can contain multiple paragraphs.
   ....:     """
   ....:     # function body
   ....:     pass
   ....:

In [68]: funcname?
Type:              function
Base Class: <type 'function'>
String Form:       <function funcname at 0xeaa0f0>
Namespace:  Interactive
File:              /home/alex/Curs_Python/.../<ipython console>
Definition: funcname(params)
Docstring:
    Concise one-line sentence describing the function.

    Extended summary which can contain multiple paragraphs.
```

sklearn.metrics

sklearn.metrics

roc_curve	Compute Receiver operating characteristic (ROC).
precision_recall_curve	Compute precision-recall pairs for different probability thresholds.
accuracy_score	Accuracy classification score.
confusion_matrix	Confusion matrix.
matthews_corrcoef	Matthews Correlation coefficient
classification_report	Build a text report showing the main classification metrics.

Precision and Recall

Precision

How many selected items are relevant?

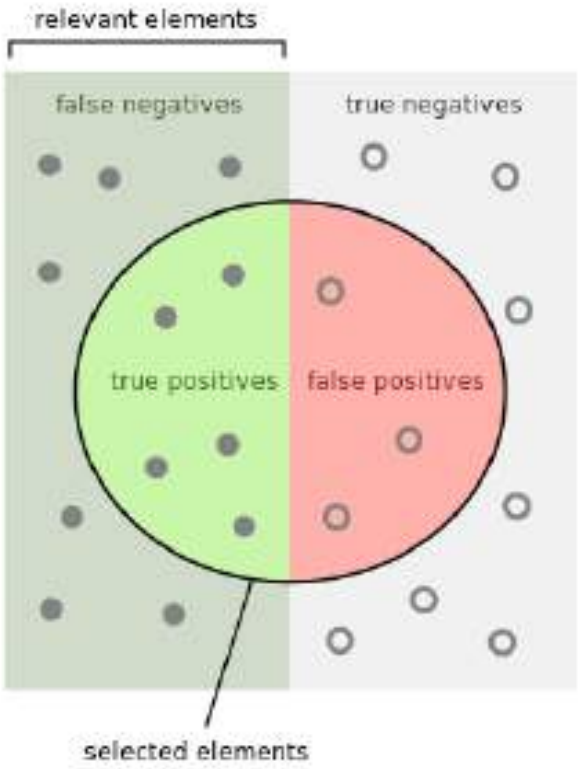
Recall

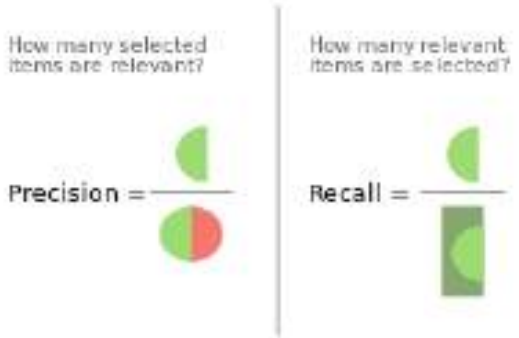
How many selected items are selected?

F1

Harmonic mean of precision and recall

$F_1 = 2\frac{P \cdot R}{P+R}$





sklearn.metrics

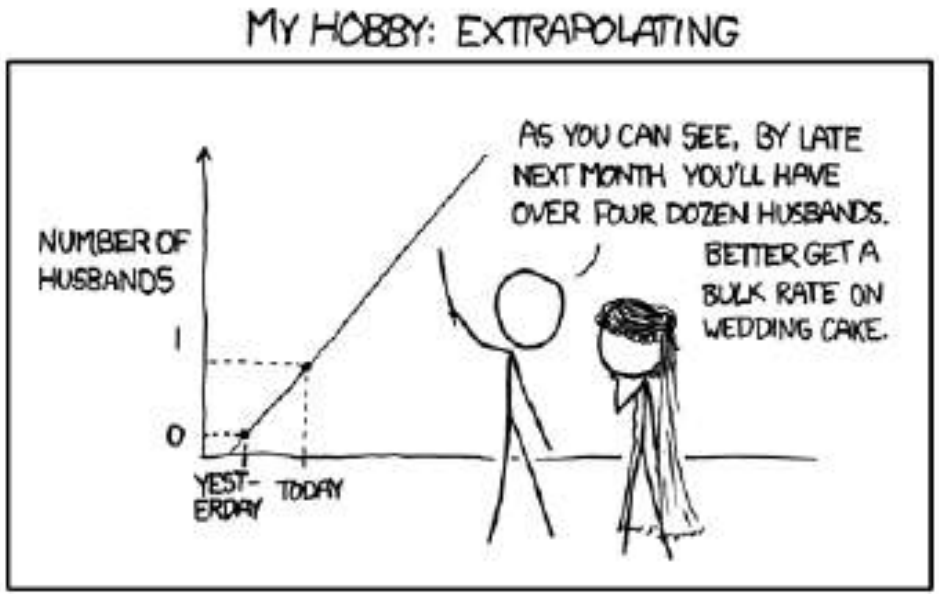
```
>>> from sklearn import metrics
>>> metrics.recall_score(yPred,y,average=None)
array([ 1.        ,  0.92156863,  0.93877551])
>>> metrics.confusion_matrix(yPred,y)
array([[50,  0,  0],
       [ 0, 47,  4],
       [ 0,  3, 46]])
```

```
metrics.classification_report(yPred,y)
             precision    recall  f1-score   support

          0       1.00      1.00      1.00        50
          1       0.94      0.92      0.93        51
          2       0.92      0.94      0.93        49

navg / total       0.95      0.95      0.95       150
```

Cross-validation



Validity and Over-Fitting

In multivariate predictive models, over-fitting occurs when a large number of predictor variables is fit to a small N of subjects. A model may "fit" well or perfectly, even if no real relationship. Simon, JNCI 2003

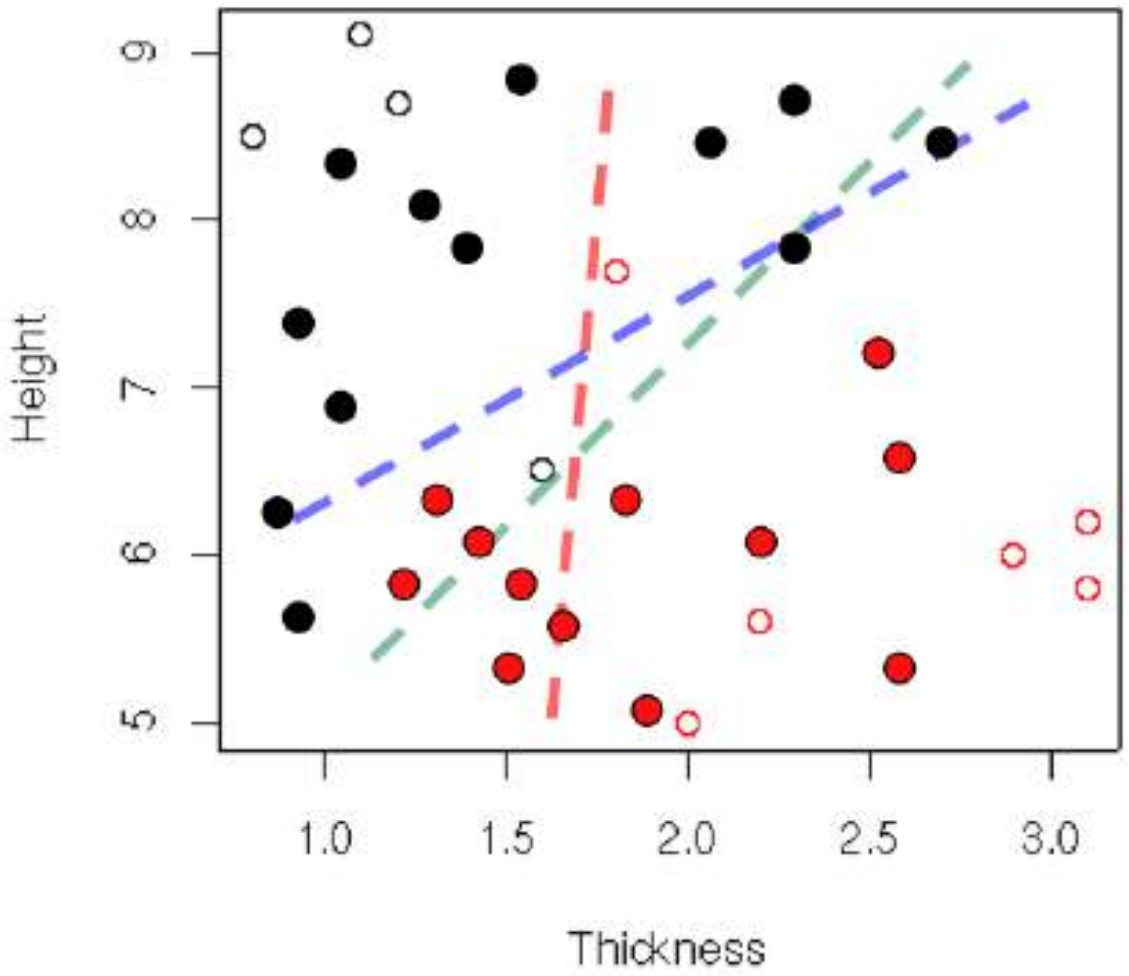
Validity and Over-Fitting

In multivariate predictive models, over-fitting occurs when a large number of predictor variables is fit to a small N of subjects. A model may "fit" well or perfectly, even if no real relationship. Simon, JNCI 2003

Direct Consequence of Over-fitting

Model performance results are not reproducible in a new set of data.

Over-Fitting



Validation Schemes

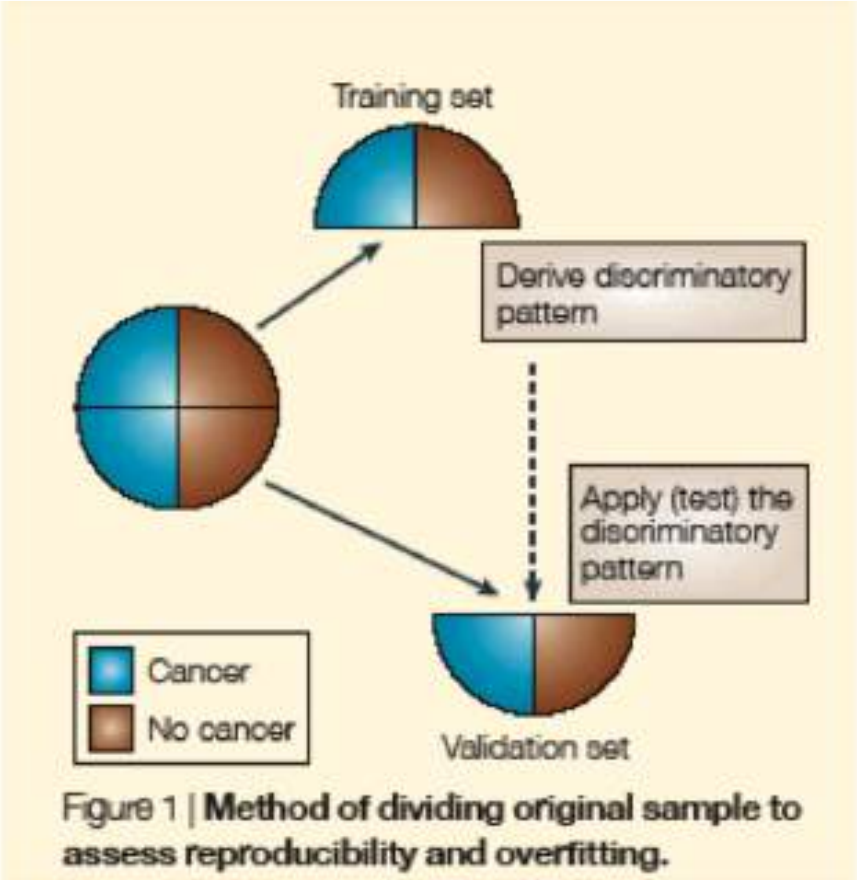


Figure 1 | **Method of dividing original sample to assess reproducibility and overfitting.**

Ransohoff. Nat Rev Cancer 2004

Cross Validation

sklearn.cross_validation.train_test_split

Split arrays or matrices into random train and test subsets.

```
>>> #from sklearn import cross_validation
>>> #xTrain, xVal, yTrain, yVal = cross_validation.train_test_split(X,
y, test_size=0.4)
>>> from sklearn.model_selection import train_test_split
>>> xTrain, xVal, yTrain, yVal =  train_test_split(X, y, test_size =
0.4)
>>> print(xTrain.shape,xVal.shape)
(90, 4) (60, 4)
```

Cross Validation

sklearn.cross_validation.train_test_split

Split arrays or matrices into random train and test subsets.

```
>>> #from sklearn import cross_validation
>>> #xTrain, xVal, yTrain, yVal = cross_validation.train_test_split(X,
y, test_size=0.4)
>>> from sklearn.model_selection import train_test_split
>>> xTrain, xVal, yTrain, yVal =  train_test_split(X, y, test_size =
0.4)
>>> print(xTrain.shape,xVal.shape)
(90, 4) (60, 4)
```

```
>>> from sklearn import neighbors
>>> knn = neighbors.KNeighborsClassifier(7).fit(xTrain, yTrain)
>>> yPredVal = knn.predict(xVal)
>>> (yPredVal==yVal).mean()
0.9833333333333328
```

Cross-validation metrics

Can we compute some statistics on the performance of the classifiers?
`cross_val_score` exists!

```
>>> from sklearn.model_selection import cross_val_score
>>> knn = neighbors.KNeighborsClassifier(1)
>>> recalls = cross_val_score(knn, X, y, cv = 6)
>>> recalls
array([ 0.96296296,  1.        ,  0.875     ,  0.91666667,  1.
,  1.        ])
>>> recalls.mean()
0.95910493827160492
>>> recalls.std()
0.048143449042612647
```

Leave one out

LOO

Each training set is constructed by taking all samples except sample k and the model validates over k. Then just iterate through k.

```
>>> from  sklearn.model_selection import LeaveOneOut
>>> from sklearn import svm
>>> loo = LeaveOneOut()
>>> cl = svm.SVC(kernel='linear', C=1)
>>> recalls = [(y[indVal] ==
cl.fit(X[indTrain,:],y[indTrain]).predict(X[indVal,:])).mean() for
indTrain, indVal in loo.split(X)]
>>> len(recalls)
150
>>> recalls[:5]
[1.0, 1.0, 1.0, 1.0, 1.0]
```

k-nearest neighbours vs. SVC Challenge

Challenge

1. Get all variables from Iris dataset.
2. Compute a 2D-PCA model of the Iris dataset.
3. Compute and plot the decision boundaries for a k-NN and SVC classifier of your choice.

Hint: use np.mesgrid()