Visual Analytics

Giuseppe Santucci

Dimensionality reduction (v4b)

Thanks to Enrico Bertini

Outline

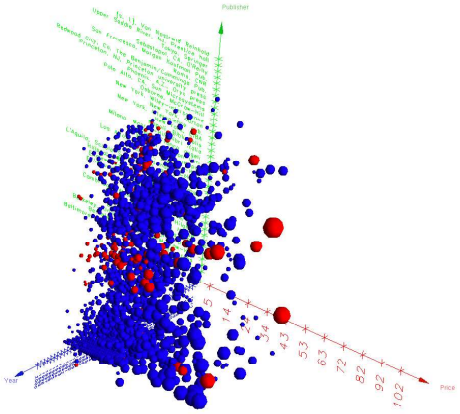
- Motivations
- PCA (Principal Component Analysis)
- MDS (Multidimensional Scaling)
- t-SNE (t-distributed Stochastic Neighbor Embedding)
- Comparison

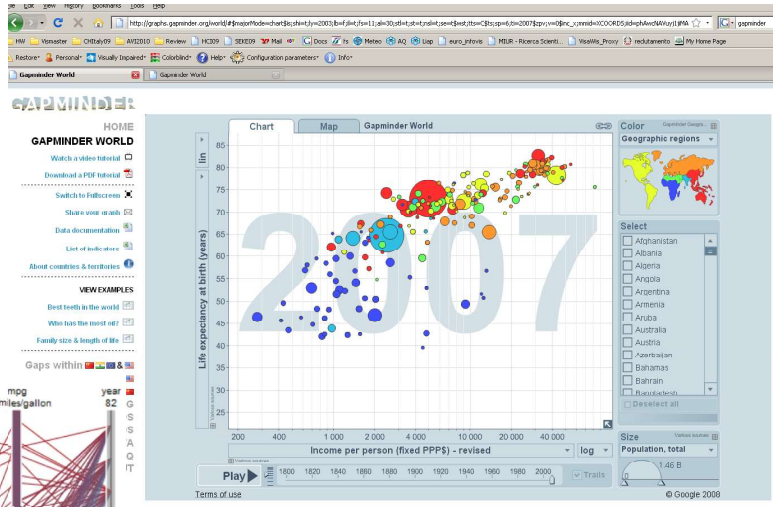
How do we deal with and visualize multidimensional data?

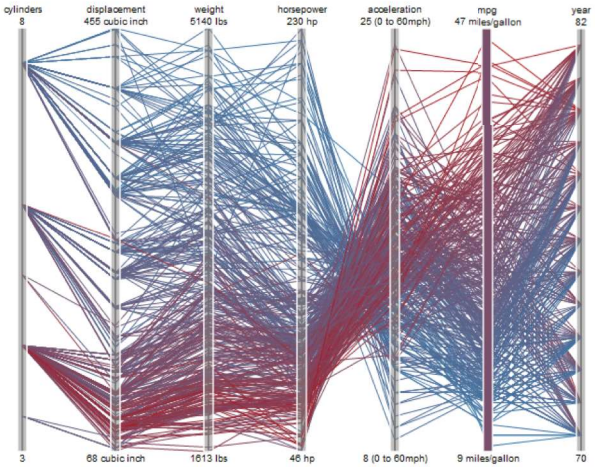
- Ideally we would like to see relationships between data items in the multidimensional (nD) space
  - clusters,
  - trends,
  - outliers,
  - etc...

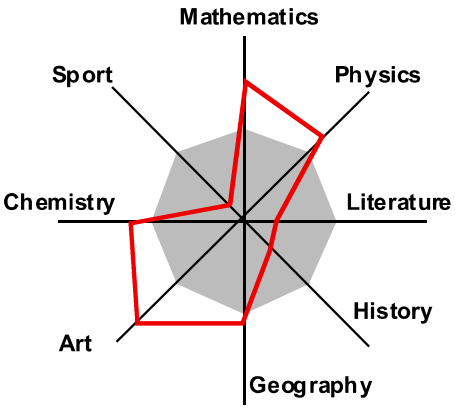
- Most visualization techniques allow for visualizing a very limited number of variables

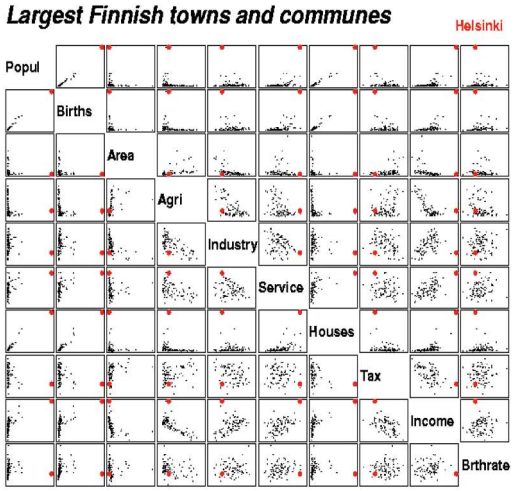
Radar plots, 2D,3D+ scatterplots, etc.







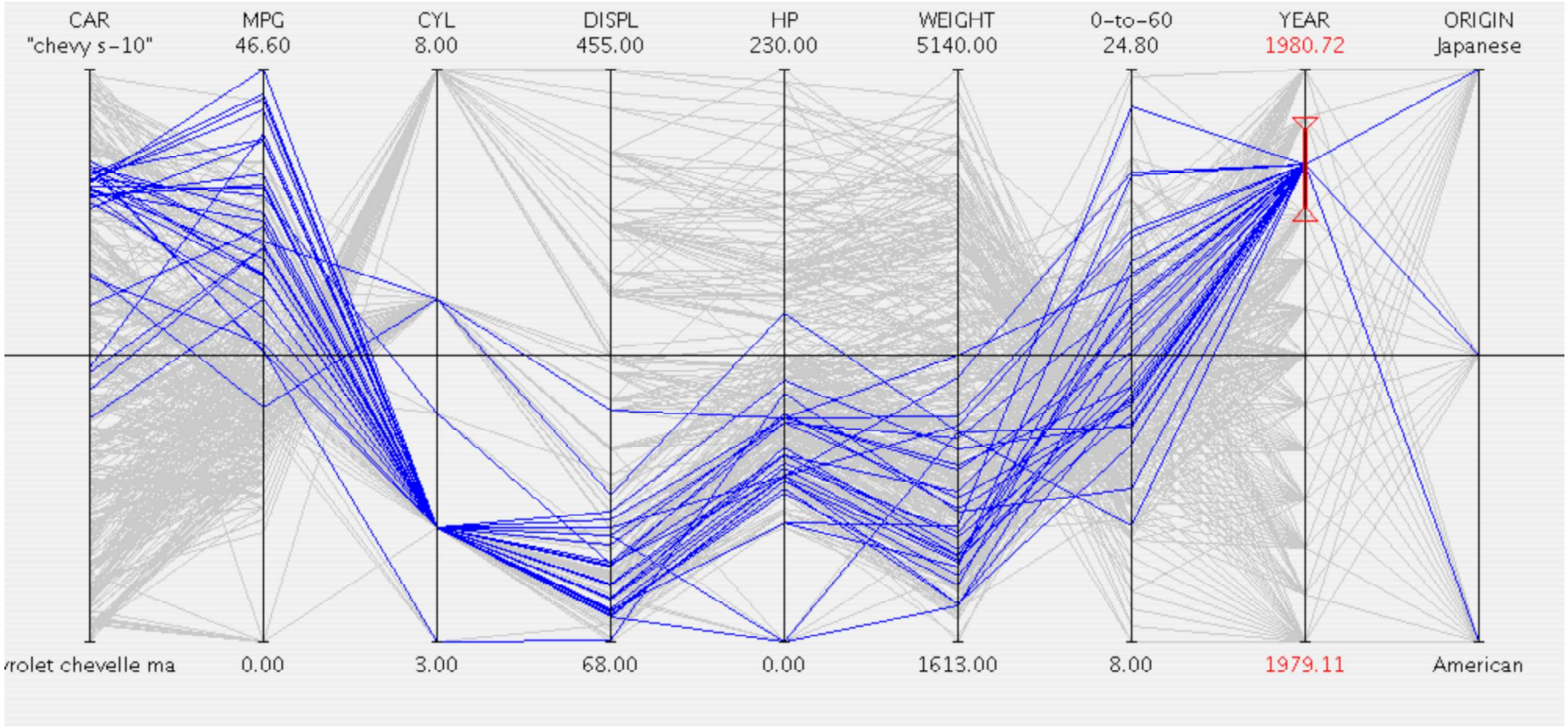




Proximity Data

- Sometime we want to visualize data in a way that proximity reveals similarity between data objects
- The first, intuitive source of similarity is the Euclidean distance
- We need a visual representation that allows us to say “these objects are similar”
- Clusters!

Proximity in the Car Data Set ?



Similarity

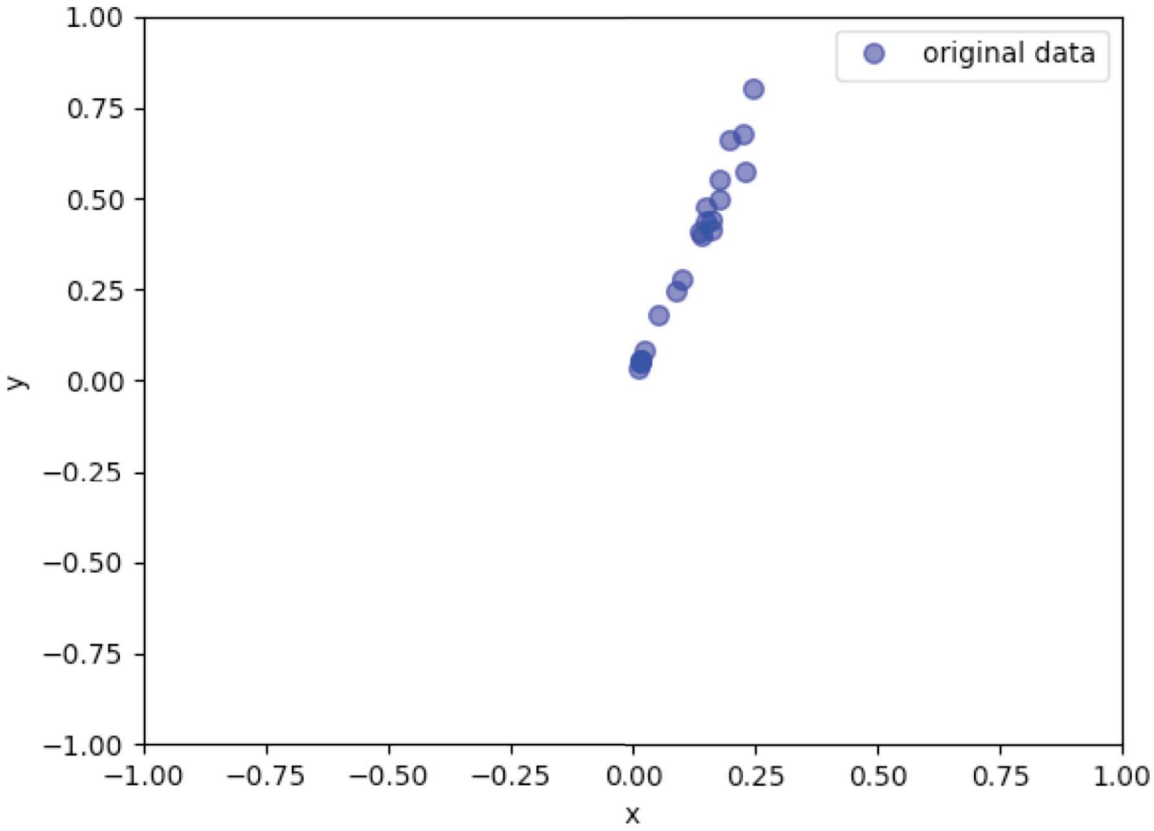
- The notion of similarity is crucial
- In some cases it's straightforward:
  e.g., values of the car data set: Euclidean distance
  - (What about non numerical attribute?)
- In other cases similarity must be defined according to some data transformation: similarity between documents, images, songs, molecules, route distance (vs. Euclidean distance), etc.

Multidimensionality reduction

- Moving from n dimensions to k<<n dimensions is useful for
  - compressing the data
  - using simple visualizations (e.g., 2D scatterplot)
- For visualization purposes we want to map data points **x** in R^n to points **y** in R^2 (o R^3) so that we can use these coordinates to place our data items and observe similarity
- The transformation should be done in a way that close proximity, i.e., Euclidean distance, in R^2 translates into similarity in R^n (and vice-versa)
  - Similarity may be defined in different ways
  - Some techniques use Euclidean distance in R^n (e.g., PCA, t-SNE)
  - Some techniques use user defined notion of similarity (e.g., MDS)

A toy dataset and a toy dimensionality reduction (2D->1D)

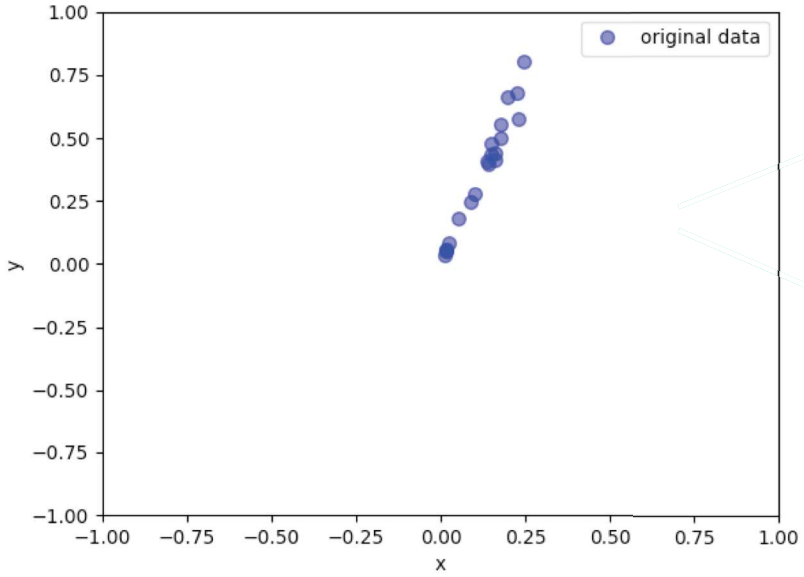
x	y
0.053552545	0.17827798
0.015198373	0.05072713
0.016663727	0.05466954
0.024896758	0.08309262
0.15002833	0.47698061
0.178904104	0.55430599
0.140067887	0.39466536
0.01781757	0.05690527
0.100065287	0.27632587
0.087911535	0.24468
0.147655514	0.43319142
0.199171712	0.66016647
0.162220477	0.43852329
0.163173454	0.41526889
0.244317024	0.80389978
0.175788208	0.4989565
0.011275933	0.03715916
0.136424559	0.40966642
0.228213934	0.57616316
0.226895748	0.6785588

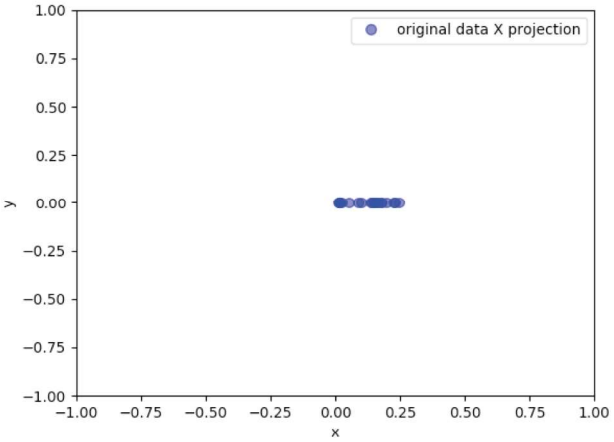


Simplest way: projection

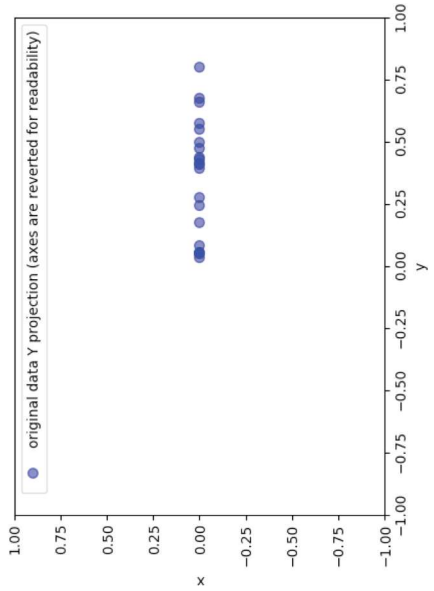
- Moving from n dimensions to k<n dimensions **hide** information
- Preserve Euclidean distance ?
- False positives ?
- E.g., 2D -> 1D

x projection



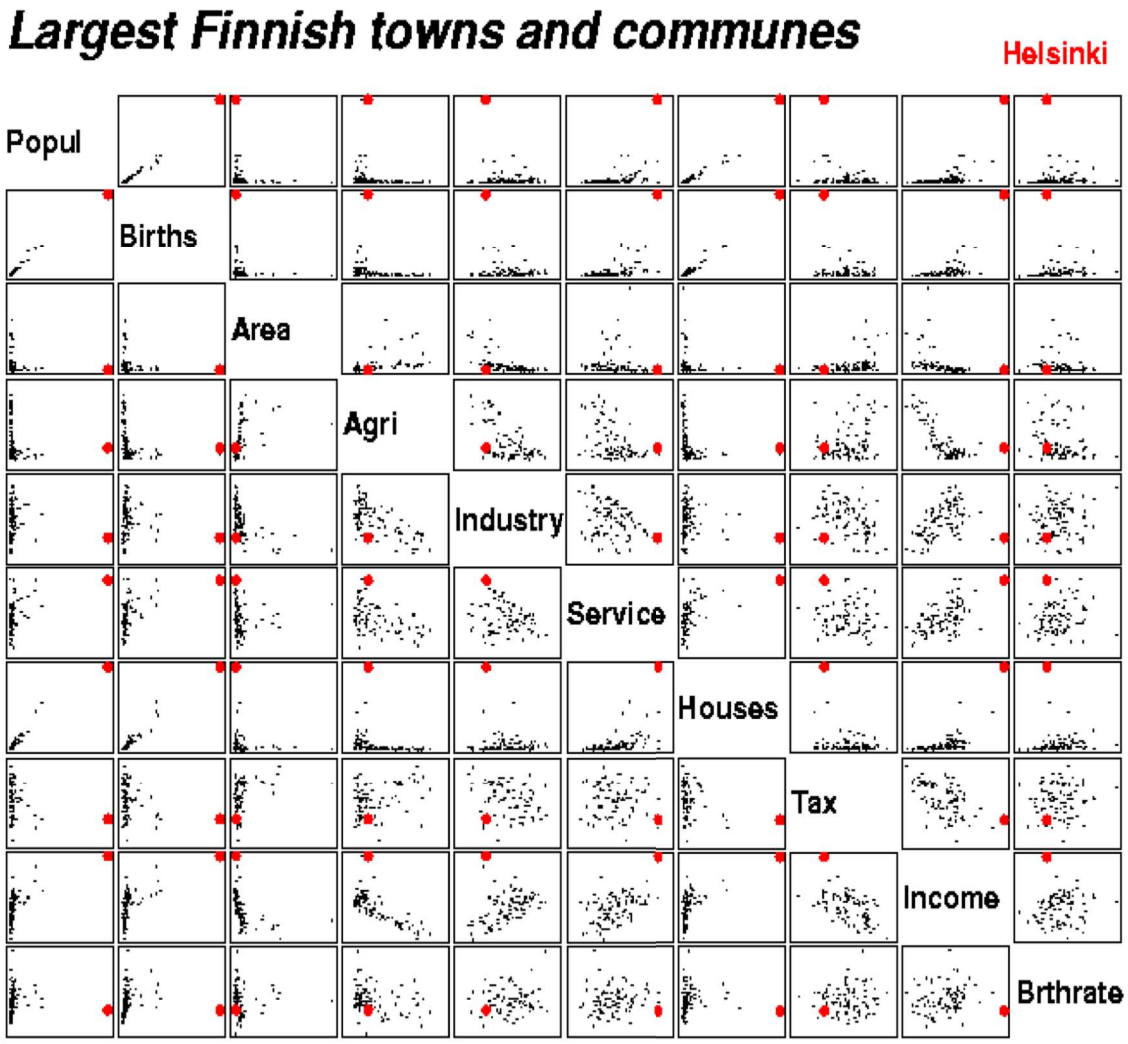


y projection



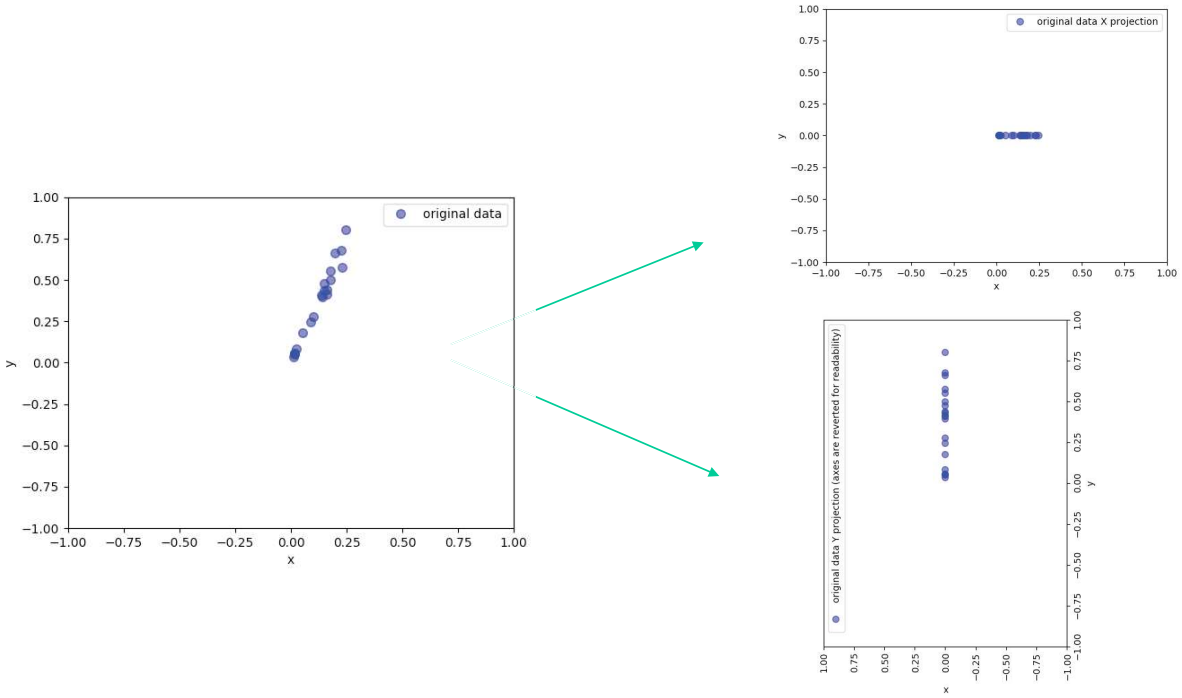
B.t.w., it is not so simple: n(n-1)/2 projections!

SPLOM


Largest Finnish towns and communes
Helsinki
Popul
Births
Area
Agri
Industry
Service
Houses
Tax
Income
Brthrate

Some projections are better than others...

- Strategies allow for preserving some data characteristics during the dimension reduction



variance=0.0062

variance=0.055

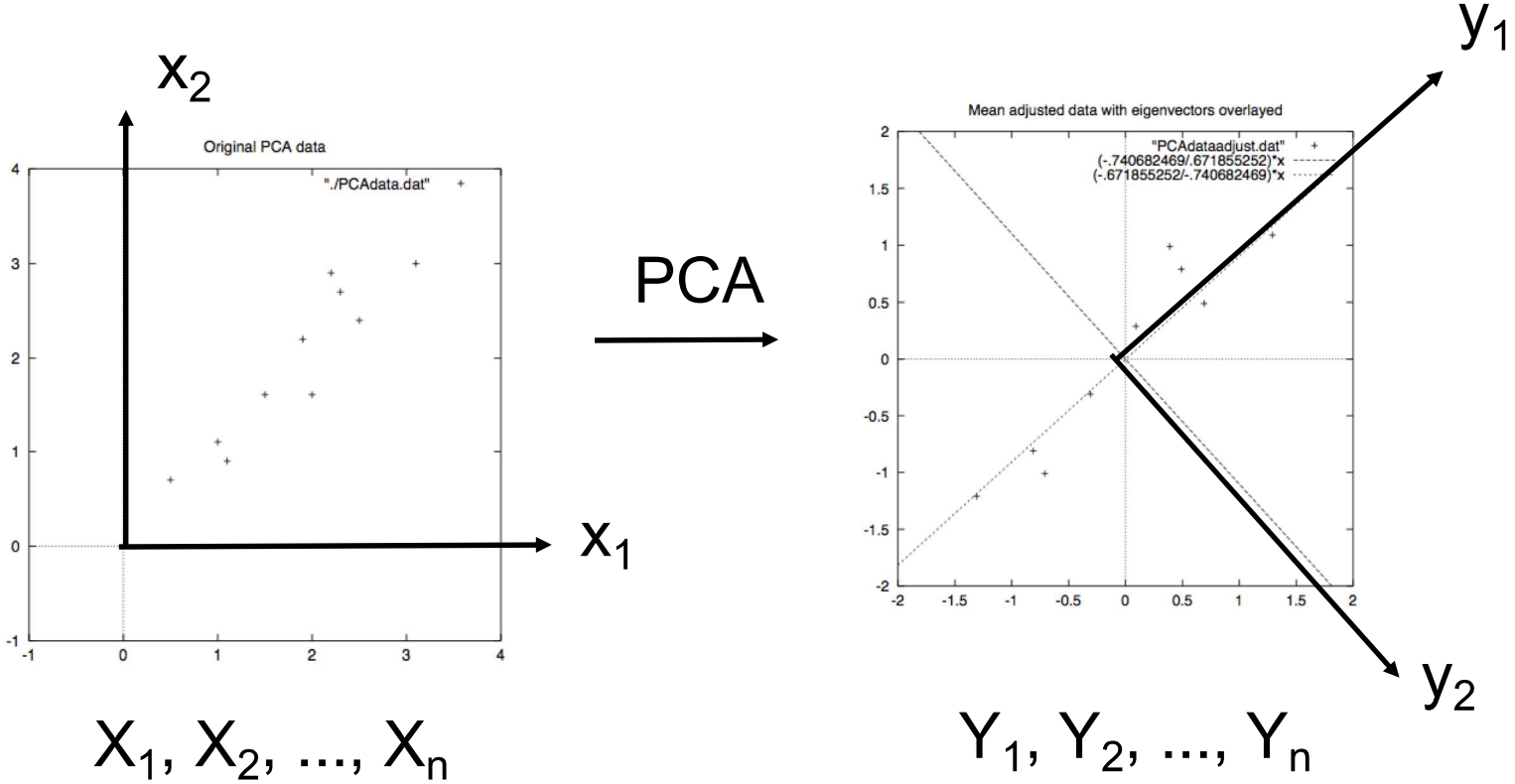
Which one is better? And why?

Principal Component Analysis (PCA)

- Transform the coordinate space R^n (Xs) into a new one R^n (Ys) where:
  - "The first principal component has the largest possible variance and
  - each succeeding component in turn has the highest variance possible under the constraint that it is orthogonal to (i.e., uncorrelated with) the preceding components"

- PCA produces a (ranked) set of coordinates that allows for an 'optimal' projection
- It is a linear transformation and preserves Euclidean distance and does not introduce false positives
- And yes, it DOES NOT reduce the dimensionality !
  - For visualization purposes most of the time we use the first 2 coordinates (2D scatterplot)

Principal Component Analysis (PCA)



$X_1, X_2, ..., X_n$ $Y_1, Y_2, ..., Y_n$

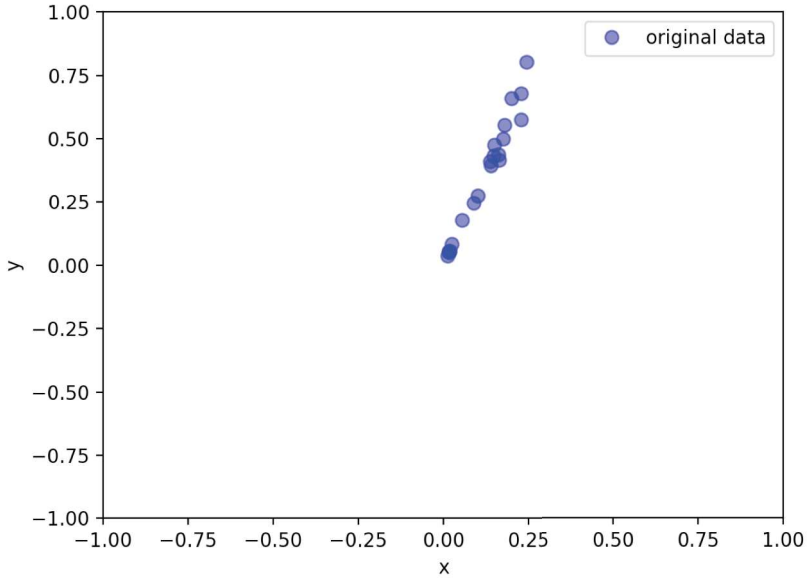
- N.B.: it does NOT reduce the number of dimension ...
- It is a linear transformation $R^n \rightarrow R^n$
- Y_1 is the axis in which the variance has a maximum
- Projecting on it is the best choice...

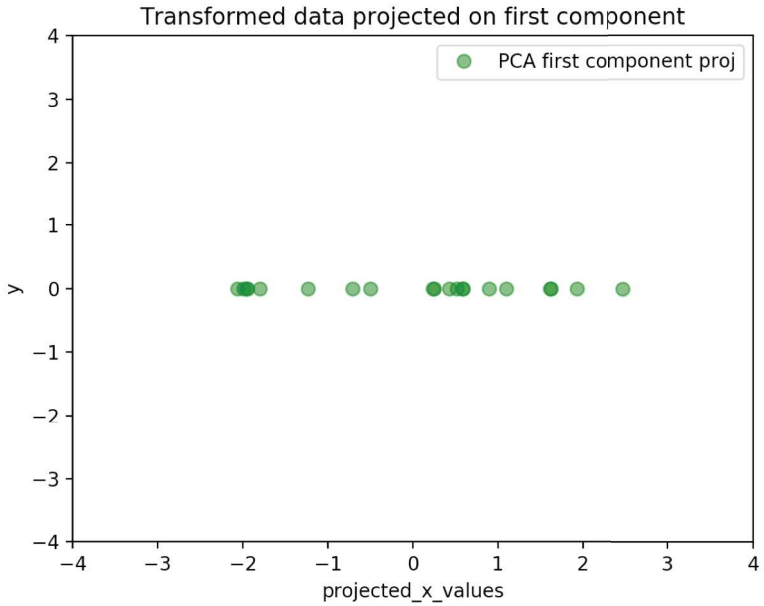
Python example (numpy, matplotlib, and sklearn)

```
import numpy as np
import pandas as pd
from matplotlib import pyplot as plt
from matplotlib.mlab import PCA as mlabPCA
from sklearn import preprocessing
from sklearn.decomposition import PCA
```

- For the next class you have to install Python (3.x) and the above packages
- e.g., pip install numpy
- and run the example files:
  - 01_PCA_2D_matplotlib.py
  - 01_PCA_2D_sklearn.py

mlabPCA R^2 –> R^1 (toy example)






```
from matplotlib.mlab import PCA as mlabPCA
```

or

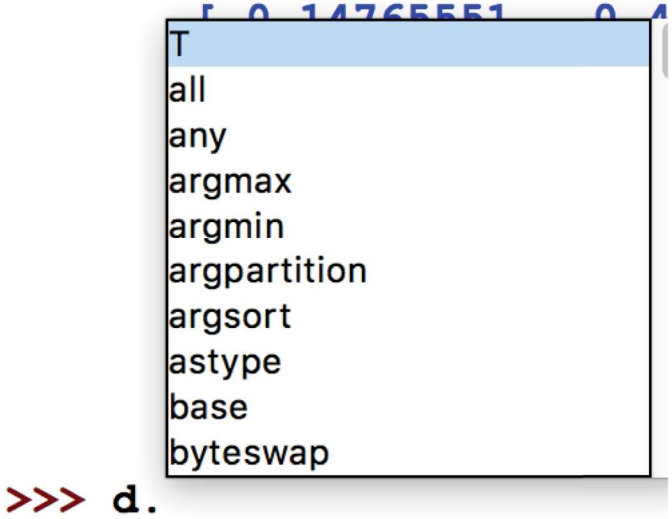
```
from sklearn.decomposition import PCA
```

Let's go through the code (2)

	A	B
1	0.05355255	0.17827798
2	0.01519837	0.05072713
3	0.01666373	0.05466954
4	0.02489676	0.08309262
5	0.15002833	0.47698061
6	0.1789041	0.55430599
7	0.14006789	0.39466536
8	0.01781757	0.05690527
9	0.10006529	0.27632587
0	0.08791154	0.24468

- Reading the data using the **numpy** package

```
# read data from a CSV file, you can choose different delimiters,
# skip the header, select columns
# it return a numpy array
data=np.genfromtxt('data.csv',skip_header=0,usecols=[0,1],delimiter=',')
```




```
>>> d
array([[0.05355255, 0.17827798],
       [0.01519837, 0.05072712],
       [0.01666373, 0.05466954],
       [0.02489676, 0.08309262],
       [0.15002833, 0.47698061],
       [0.1789041 , 0.55430599],
       [0.14006789, 0.39466536],
       [0.01781757, 0.05690527],
       [0.10006529, 0.27632587],
       [0.08791153, 0.24468   ],
       [0.14765551, 0.43319142],
       [0.19917171, 0.66016647],
       [0.16222048, 0.43852329],
       [0.16317345, 0.41526889],
       [0.24431702, 0.80389978],
       [0.17578821, 0.49895649],
       [0.01127593, 0.03715915],
       [0.13642456, 0.40966642],
       [0.22821393, 0.57616316],
       [0.22689575, 0.6785588 ]])
```

X_1 X_2

It is an array of X points

Let's go through the code (2)

d[interval/value, interval/value] →slice

optional

```
>>> d[0]
array([0.05355255, 0.17827798])
```

```
>>> d[0,0]
0.053552545
```

First column

```
>>> d[:,0]
array([0.05355255, 0.01519837, 0.01666373, 0.02489676, 0.15002833,
       0.1789041 , 0.14006789, 0.01781757, 0.10006529, 0.08791153,
       0.14765551, 0.19917171, 0.16222048, 0.16317345, 0.24431702,
       0.17578821, 0.01127593, 0.13642456, 0.22821393, 0.22689575])
```

Second column

```
>>> d[:,1]
array([0.17827798, 0.05072712, 0.05466954, 0.08309262, 0.47698061,
       0.55430599, 0.39466536, 0.05690527, 0.27632587, 0.24468   ,
       0.43319142, 0.66016647, 0.43852329, 0.41526889, 0.80389978,
       0.49895649, 0.03715915, 0.40966642, 0.57616316, 0.6785588 ])
>>>
```

```
>>> d
array([[0.05355255, 0.17827798],
       [0.01519837, 0.05072712],
       [0.01666373, 0.05466954],
       [0.02489676, 0.08309262],
       [0.15002833, 0.47698061],
       [0.1789041 , 0.55430599],
       [0.14006789, 0.39466536],
       [0.01781757, 0.05690527],
       [0.10006529, 0.27632587],
       [0.08791153, 0.24468   ],
       [0.14765551, 0.43319142],
       [0.19917171, 0.66016647],
       [0.16222048, 0.43852329],
       [0.16317345, 0.41526889],
       [0.24431702, 0.80389978],
       [0.17578821, 0.49895649],
       [0.01127593, 0.03715915],
       [0.13642456, 0.40966642],
       [0.22821393, 0.57616316],
       [0.22689575, 0.6785588 ]])
```

```
>>> d[:3,0:1]
array([[ 0.05355255],
       [ 0.01519837],
       [ 0.01666373]])
```

first X_1 3 values

```
>>> d
array([[ 0.05355255,  0.17827798],
       [ 0.01519837,  0.05072712],
       [ 0.01666373,  0.05466954],
       [ 0.02489676,  0.08309262],
       [ 0.15002833,  0.47698061],
       [ 0.1789041 ,  0.55430599],
```

Slicing ?

d[value, value] → element

```
>>> d[0,0]
0.053552545
```

d[value] → row

```
>>> d[0]
array([0.05355255, 0.17827798
```

d[interval] → elements

```
>>> d[0:2]
array([[0.05355255, 0.17827798],
       [0.01519837, 0.05072712]])
```

d[interval,value] → column slice

```
>>> d[3:5,0]
array([0.02489676, 0.15002833])
```

d[interval, interval] → data slice

```
>>> d[3:5,0:2]
array([[0.02489676, 0.08309262],
       [0.15002833, 0.47698061]])
```

```
>>> d
array([[0.05355255, 0.17827798],
       [0.01519837, 0.05072712],
       [0.01666373, 0.05466954],
       [0.02489676, 0.08309262],
       [0.15002833, 0.47698061],
       [0.1789041 , 0.55430599],
       [0.14006789, 0.39466536],
       [0.01781757, 0.05690527],
       [0.10006529, 0.27632587],
       [0.08791153, 0.24468   ],
       [0.14765551, 0.43319142],
       [0.19917171, 0.66016647],
       [0.16222048, 0.43852329],
       [0.16317345, 0.41526889],
       [0.24431702, 0.80389978],
       [0.17578821, 0.49895649],
       [0.01127593, 0.03715915],
       [0.13642456, 0.40966642],
       [0.22821393, 0.57616316],
       [0.22689575, 0.6785588 ]])
```

Let's go through the code (3)

numpy array type has nice and useful methods

```
>>> d
array([[ 0.05355255,  0.17827798],
       [ 0.01519837,  0.05072712],
       [ 0.01666373,  0.05466954],
       [ 0.02489676,  0.08309262],
       [ 0.15002833,  0.47698061],
       [ 0.1789041 ,  0.55430599],
       [ 0.14006789,  0.39466536],
       [ 0.01781757,  0.05690527],
       [ 0.10006529,  0.27632587],
       [ 0.08791153,  0.24468   ],
       [ 0.14765551,  0.43319142],
       [ 0.19917171,  0.66016647],
       [ 0.16222048,  0.43852329],
       [ 0.16317345,  0.41526889],
       [ 0.24431702,  0.80389978],
       [ 0.17578821,  0.49895649],
       [ 0.01127593,  0.03715915],
       [ 0.13642456,  0.40966642],
       [ 0.22821393,  0.57616316],
       [ 0.22689575,  0.6785588 ]])
```

```
>>> d.T
array([[ 0.05355255,  0.01519837,  0.01666373,  0.02489676,  0.15002833,
         0.1789041 ,  0.14006789,  0.01781757,  0.10006529,  0.08791153,
         0.14765551,  0.19917171,  0.16222048,  0.16317345,  0.24431702,
         0.17578821,  0.01127593,  0.13642456,  0.22821393,  0.22689575],
       [ 0.17827798,  0.05072712,  0.05466954,  0.08309262,  0.47698061,
         0.55430599,  0.39466536,  0.05690527,  0.27632587,  0.24468   ,
         0.43319142,  0.66016647,  0.43852329,  0.41526889,  0.80389978,
         0.49895649,  0.03715915,  0.40966642,  0.57616316,  0.6785588 ]])
```

T →transpose

Let's go through the code (4)

- compute PCA

```
#running mlabPCA that returns an object whose .Y is a numpy.array
# d_pca.Y[:,0] will contain values of first component (Max var)
# d_pca.Y[:,i] will contain values of ith component
d_pca = mlabPCA(d)
```

or, using sklearn

```
d_std = preprocessing.StandardScaler().fit_transform(d)
pca=PCA(n_components=2)
dpca=pca.fit_transform(d_std)
```

```
>>> d
array([[ 0.05355255,  0.17827798],
       [ 0.01519837,  0.05072712],
       [ 0.01666373,  0.05466954],
       [ 0.02489676,  0.08309262],
       [ 0.15002833,  0.47698061],
       [ 0.1789041 ,  0.55430599],
       [ 0.14006789,  0.39466536],
       [ 0.01781757,  0.05690527],
       [ 0.10006529,  0.27632587],
       [ 0.08791153,  0.24468   ],
       [ 0.14765551,  0.43319142],
       [ 0.19917171,  0.66016647],
       [ 0.16222048,  0.43852329],
       [ 0.16317345,  0.41526889],
       [ 0.24431702,  0.80389978],
       [ 0.17578821,  0.49895649],
       [ 0.01127593,  0.03715915],
       [ 0.13642456,  0.40966642],
       [ 0.22821393,  0.57616316],
       [ 0.22689575,  0.6785588 ]])
```

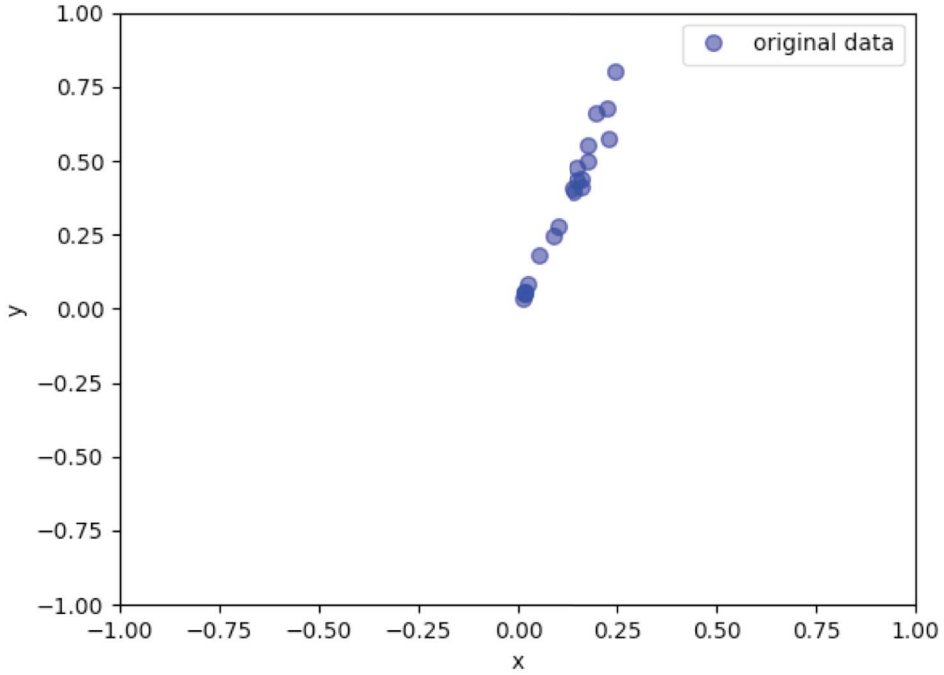
```
>>> d_pca.Y  or, using sklearn dpca
array([[-1.23727754, -0.07159629],
       [-1.98930607, -0.03204384],
       [-1.96346237, -0.03066678],
       [-1.79879616, -0.04239427],
       [ 0.5856756 , -0.10239265],
       [ 1.09381794, -0.07413185],
       [ 0.23773749,  0.06051794],
       [-1.94580786, -0.02688723],
       [-0.50101898,  0.05617675],
       [-0.71210164,  0.04148846],
       [ 0.42775869,  0.01144628],
       [ 1.61055109, -0.21436917],
       [ 0.57958457,  0.13018256],
       [ 0.51627749,  0.21119236],
       [ 2.47587075, -0.24105894],
       [ 0.89312757,  0.06867691],
       [-2.06783958, -0.02637449],
       [ 0.25044614, -0.01987003],
       [ 1.61963626,  0.31604078],
       [ 1.92512662, -0.01393651]])
```

Let's go through the code (5)

- let's visualize **d** using matplotlib

```
from matplotlib import pyplot as plt
```

```
#plotting d on a 2D scatterplot
plt.plot(d[:,0],d[:,1],
         'o', markersize=7,
         color='blue',
         alpha=0.5,
         label='original data')
plt.xlabel('x')
plt.ylabel('y')
plt.xlim([-1,1])
plt.ylim([-1,1])
plt.legend()
plt.show()
```

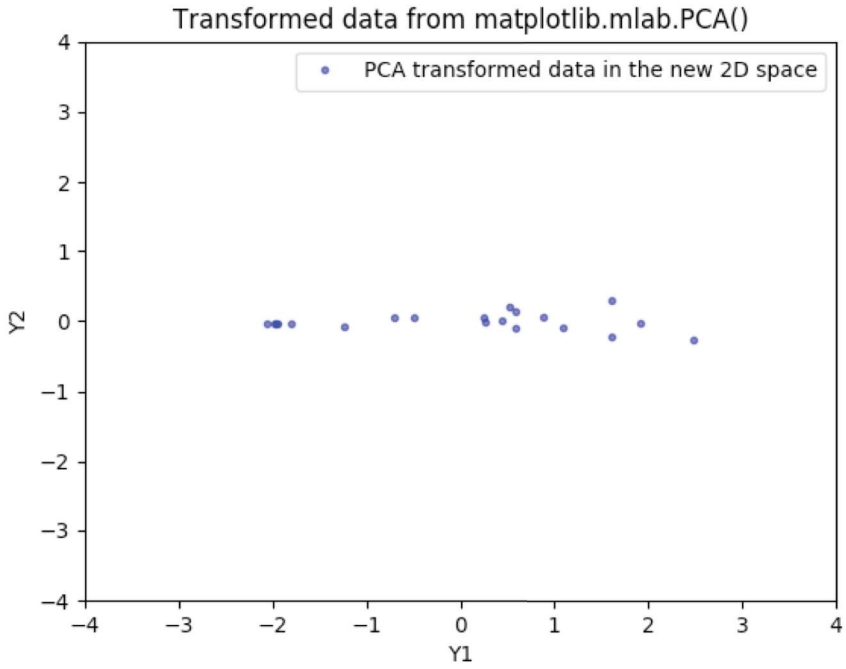


plt.plot([x_1,x_2,...,x_n],[y_1,y_2,...,y_n]) → d[:,0],d[:,1] or d.T[0], d.T[1]

Let's go through the code (6)

- let's visualize dpca using matplotlib (sklearn)

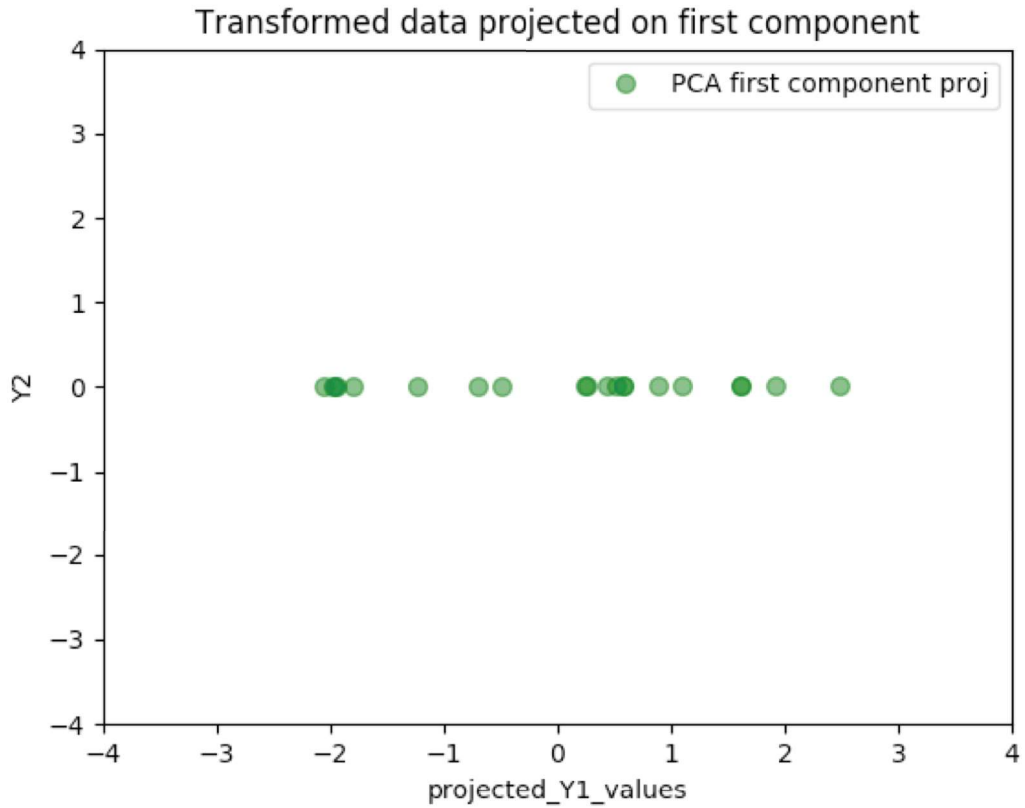
```
#projecting new values on the f
plt.plot(dpca[:,0],dpca[:,1],
         'o',
         markersize=7,
         color='green',
         alpha=0.5,
         label='PCA first compo
plt.xlabel('Y1')
plt.ylabel('Y2')
plt.xlim([-4,4])
plt.ylim([-4,4])
plt.legend()
plt.title('Transformed data pro
plt.show()
```



Let's go through the code (7)

- let's visualize d_pca projecting data on the first component

```
#projecting new values on the first component Y1
plt.plot(dpca[:,0],[0] * len(d),  #=[0,0,0,0,..]
```



Feature scaling

- Attributes **MUST** be "normalized"

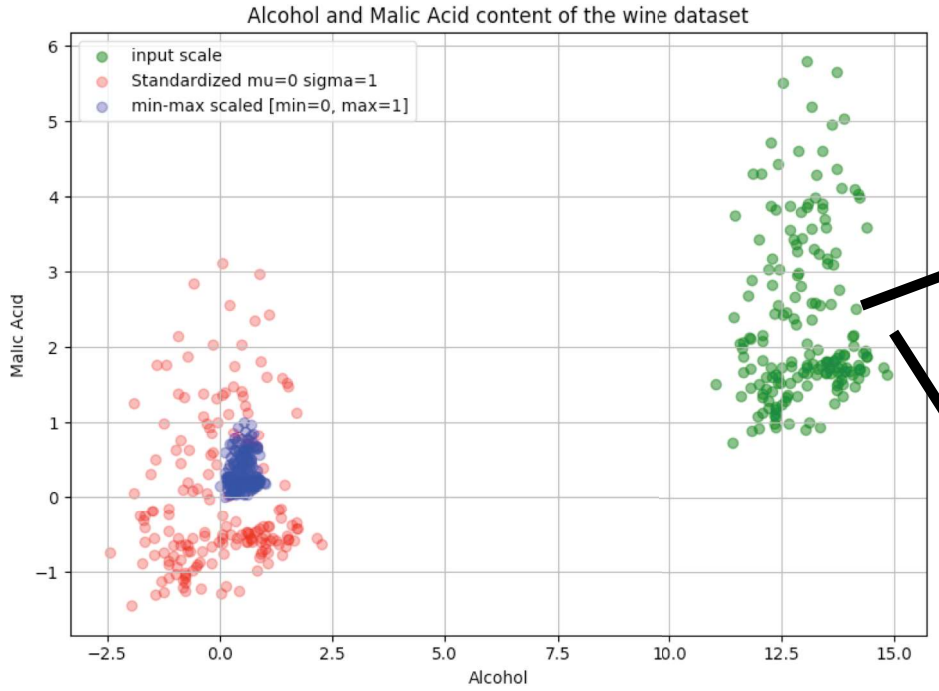
- Standardization
- The result of **standardization** (or **Z-score normalization**) is that the features will be rescaled so that they'll have the properties of a standard normal distribution with μ=0 and σ=1

$$z = (x - \mu)/\sigma$$

- Min-Max scaling

$$X_{norm} = \frac{X - X_{min}}{X_{max} - X_{min}}$$

Scaling example (03_PCA_Scale_wine.py)



	Alcohol	Malic acid
0	14.23	1.71
1	13.20	1.78
2	13.16	2.36
3	14.37	1.95
4	13.24	2.59

Covariance matrix

```
[ 1.51861254, -0.5622498 ],
[ 0.24628963, -0.49941338],
[ 0.19687903,  0.02123125],
[ 1.69154964, -0.34681064],
[ 0.29570023,  0.22769377],
```

[0.046 0.004]
[0.004 0.049]

```
[ 0.84210526,  0.1916996 ],
[ 0.57105263,  0.2055336 ],
[ 0.56052632,  0.3201581 ],
[ 0.87894737,  0.23913043],
[ 0.58157895,  0.36561265],
```

[1.01 0.09]
[0.09 1.01]

[0.66 0.09]
[0.09 1.25]

PCA will produce very different results

PCA from matplotlib standardizes by default !

```
class matplotlib.mlab.PCA(a, standardize=True)
```

- What about PCA from other libraries?
- Take care of that!!! Also because …

```
MatplotlibDeprecationWarning: The PCA class was deprecated in Matplotlib 2.2 and
 will be removed in 3.1.
```

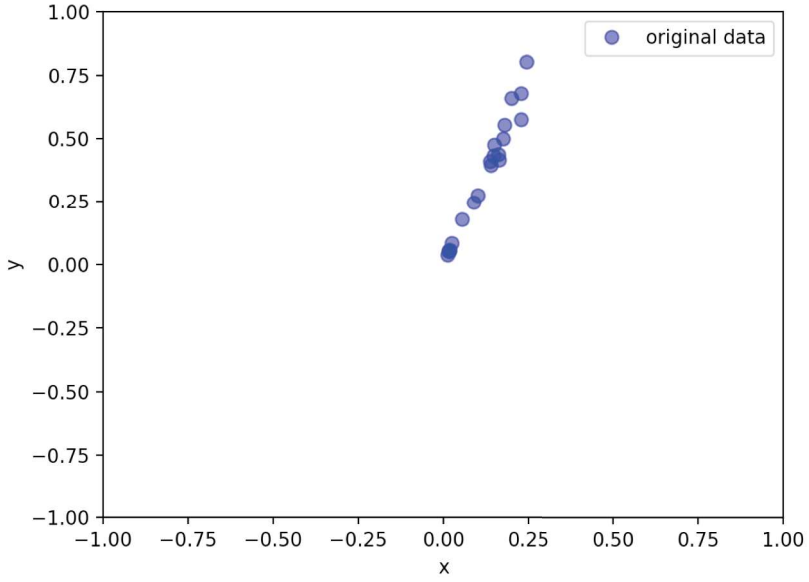
```
#normalize the data with StandardScaler
from sklearn import preprocessing
std_scale = preprocessing.StandardScaler().fit(d)
d_std = std_scale.transform(d)
```

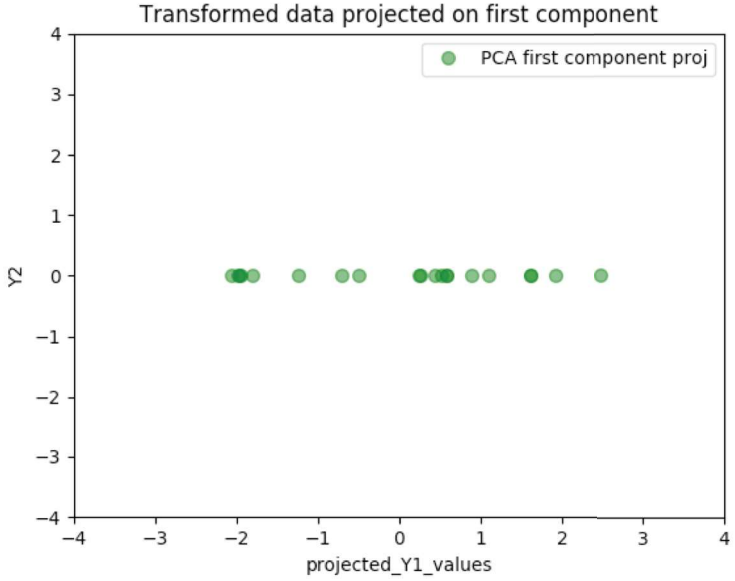
```
>>> d
array([[ 0.05355255,  0.17827798],
       [ 0.01519837,  0.05072712],
       [ 0.01666373,  0.05466954],
       [ 0.02489676,  0.08309262],
       [ 0.15002833,  0.47698061],
       [ 0.1789041 ,  0.55430599],
       [ 0.14006789,  0.39466536],
       [ 0.01781757,  0.05690527],
       [ 0.10006529,  0.27632587],
       [ 0.08791153,  0.24468   ],
       [ 0.14765551,  0.43319142],
       [ 0.19917171,  0.66016647],
       [ 0.16222048,  0.43852329],
       [ 0.16317345,  0.41526889],
       [ 0.24431702,  0.80389978],
       [ 0.17578821,  0.49895649],
       [ 0.01127593,  0.03715915],
       [ 0.13642456,  0.40966642],
       [ 0.22821393,  0.57616316],
       [ 0.22689575,  0.6785588 ]])
```

```
>>> d_std
array([[-0.92551356, -0.82426112],
       [-1.42931023, -1.3839934 ],
       [-1.41006224, -1.36669287],
       [-1.30191824, -1.24196368],
       [ 0.34173265,  0.48653773],
       [ 0.72102695,  0.82586521],
       [ 0.21089844,  0.12531315],
       [-1.39490608, -1.35688179],
       [-0.31455096, -0.39399688],
       [-0.47419513, -0.53286867],
       [ 0.31056481,  0.29437733],
       [ 0.98724971,  1.29041349],
       [ 0.50188115,  0.31777521],
       [ 0.51439887,  0.21572777],
       [ 1.58025059,  1.92115941],
       [ 0.68009847,  0.58297466],
       [-1.48083297, -1.44353381],
       [ 0.16304193,  0.1911424 ],
       [ 1.36873036,  0.92178121],
       [ 1.35141549,  1.37112468]])
```

So what?

- Meaning?
- Procedure?





PCA details – definition: Variance

- given $X=(x_1,x_2,...,x_n)$ with mean $\mu(X)$
- $$\text{Var}(X) = \frac{1}{n}\sum_{i=1}^{n}(x_i - \mu(X))^2$$
- Data with little variance is not "interesting"
- And its projection is not nice...

PCA details – definition: Covariance

- given X=(x_1,x_2, ... ,x_n) and Y=(y_1,y_2, ... ,y_n) representing pairs (x_i,y_i) with means μ(X) and μ(Y)
- $$\text{Cov(X,Y)} = \frac{1}{n}\sum_{i=1}^{n}(xi - \mu(\text{X}))(yi - \mu(\text{Y}))$$
- Cov(X,Y)>0 → X and Y variate (with respect to μ) in a similar way
- Cov(X,Y)<0 → X and Y variate (with respect to μ) in an opposite way
- Cov(X,Y)=0 → X and Y variate (with respect to μ) in an independent way

- Note that: Cov(X,X)=var(X)

Example

- Height and weight of a growing child:

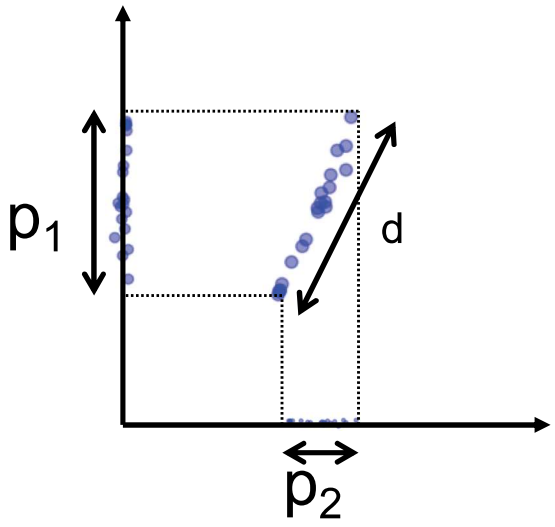
Age	Weight	Height (in cms)
Birth	2.6	47.1
3 mts	5.3	59.1
6 mts	6.7	64.7
9 mts	7.4	68.2
1 yr	8.4	73.9
2 yrs	10.1	81.6

Cov([2.6,5.3,6.7,7.4,8.4,10.1],
[47.1,59.1,64.7,68.2,73.9,81.6])=30.96

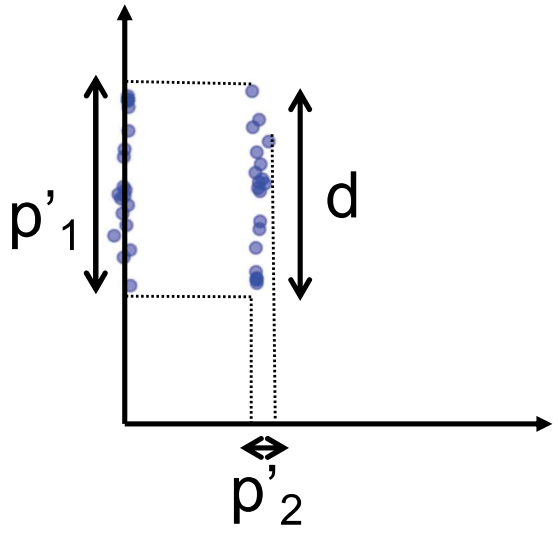
Variance and Covariance?

- Variance is good!
- Covariance is bad!
- Why?
- Variance projects good
- Covariance projects bad!

$p'_1 > p_1$!!!



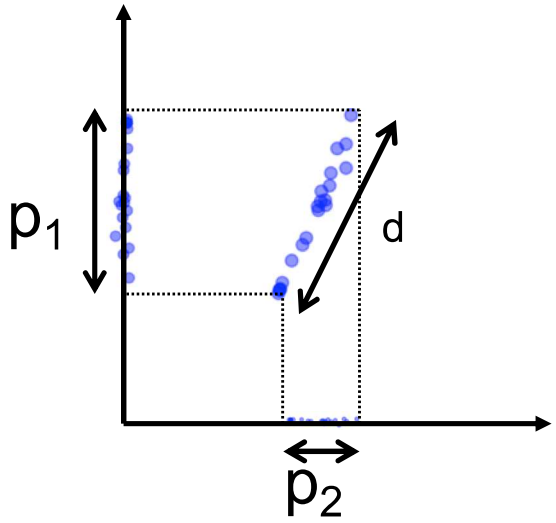
High covariance: $d > P_1 > p_2$

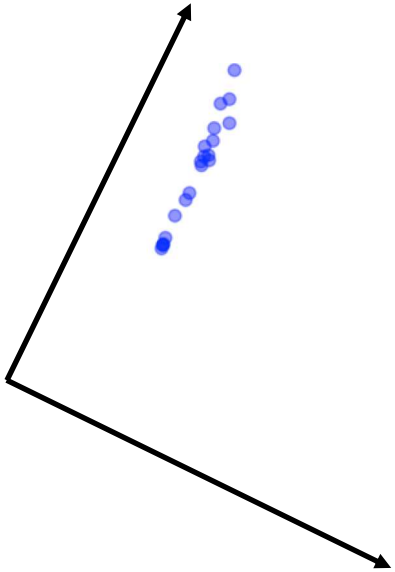


Low covariance: $d = p'_1 >> p'_2$

Variance and Covariance?

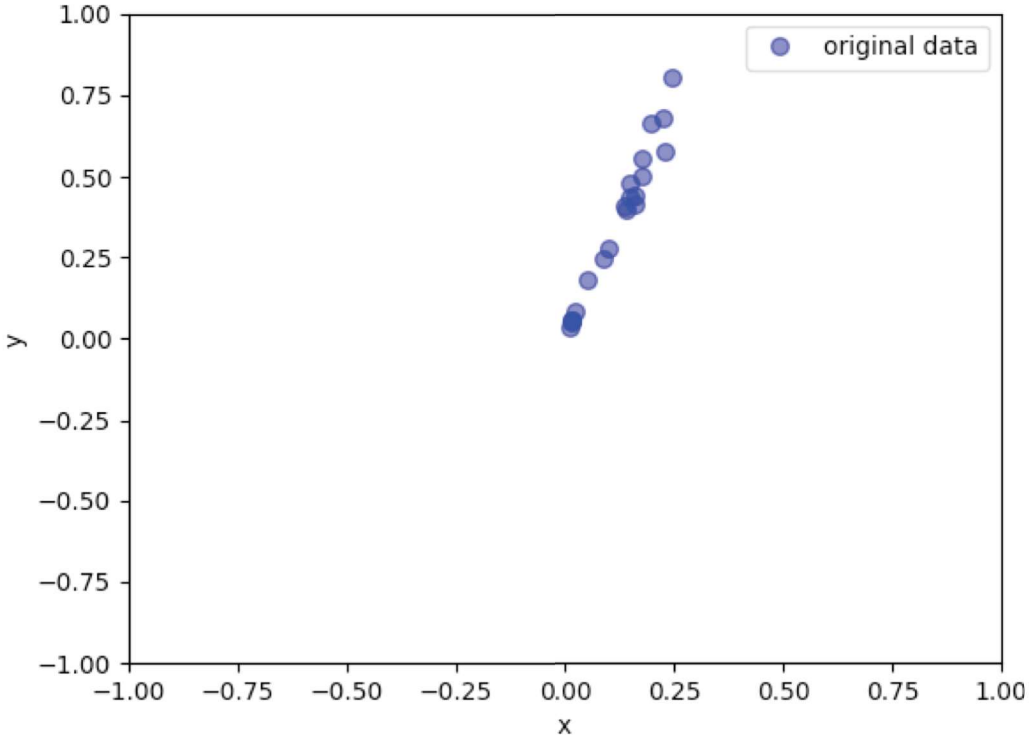
- How can improve projection?





We cannot change data
but we can change axes !
PCA!!!

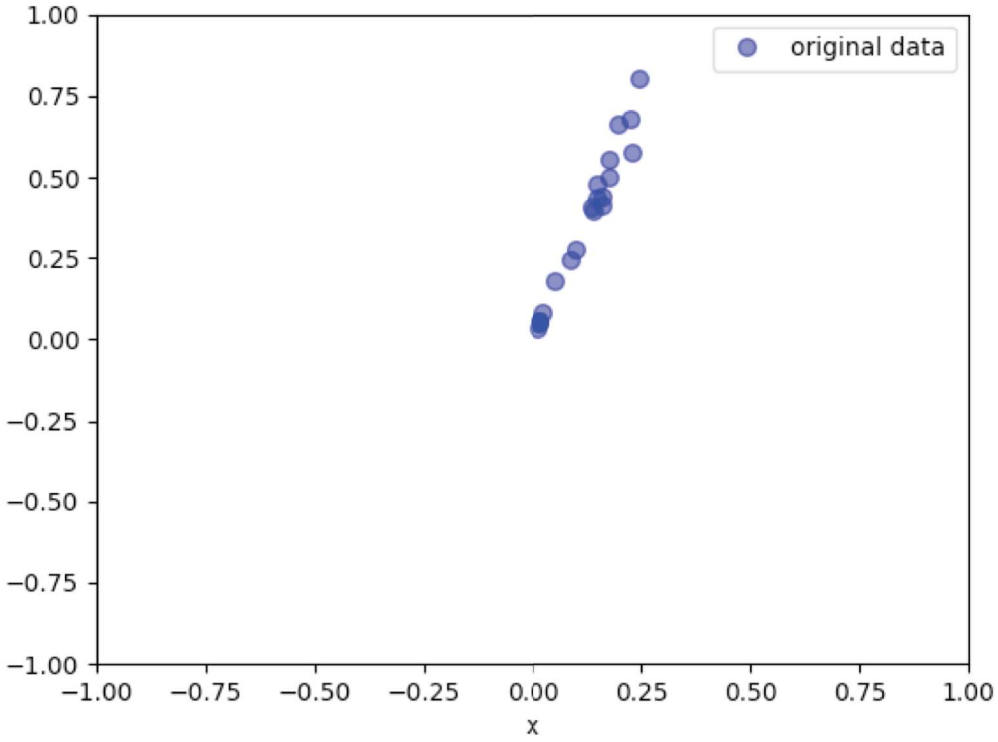
PCA details – 2d Data example




```
>>> d
array([[ 0.05355255,  0.17827798],
       [ 0.01519837,  0.05072712],
       [ 0.01666373,  0.05466954],
       [ 0.02489676,  0.08309262],
       [ 0.15002833,  0.47698061],
       [ 0.1789041 ,  0.55430599],
       [ 0.14006789,  0.39466536],
       [ 0.01781757,  0.05690527],
       [ 0.10006529,  0.27632587],
       [ 0.08791153,  0.24468   ],
       [ 0.14765551,  0.43319142],
       [ 0.19917171,  0.66016647],
       [ 0.16222048,  0.43852329],
       [ 0.16317345,  0.41526889],
       [ 0.24431702,  0.80389978],
       [ 0.17578821,  0.49895649],
       [ 0.01127593,  0.03715915],
       [ 0.13642456,  0.40966642],
       [ 0.22821393,  0.57616316],
       [ 0.22689575,  0.6785588 ]])
```

PCA details – Covariance matrix

```
>>> np.cov(d.T)
array([[ 0.00610086,  0.01798233],
       [ 0.01798233,  0.05466161]])
```



X_2

X_1

x variance

y variance

$$\begin{matrix} \sigma_{11}^2 & \sigma_{12}^2 \\ \sigma_{21}^2 & \sigma_{22}^2 \end{matrix}$$

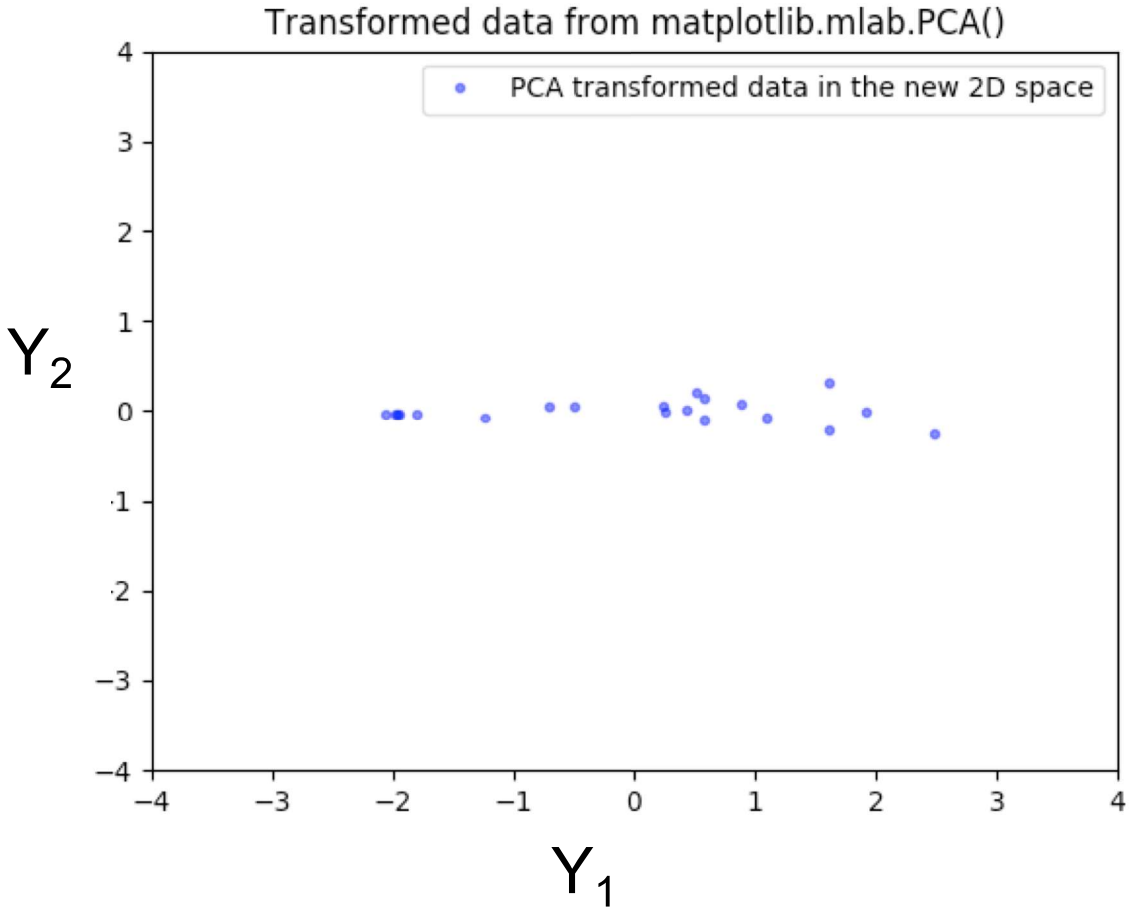
x y covariance

```
Covariance Matrix:
 [[ 0.00610086  0.01798233]
 [ 0.01798233  0.05466161]]
```

highest in this space (X_2)

PCA goal

- PCA computes a NEW space (using a linear transformation) "optimizing" the covariance matrix

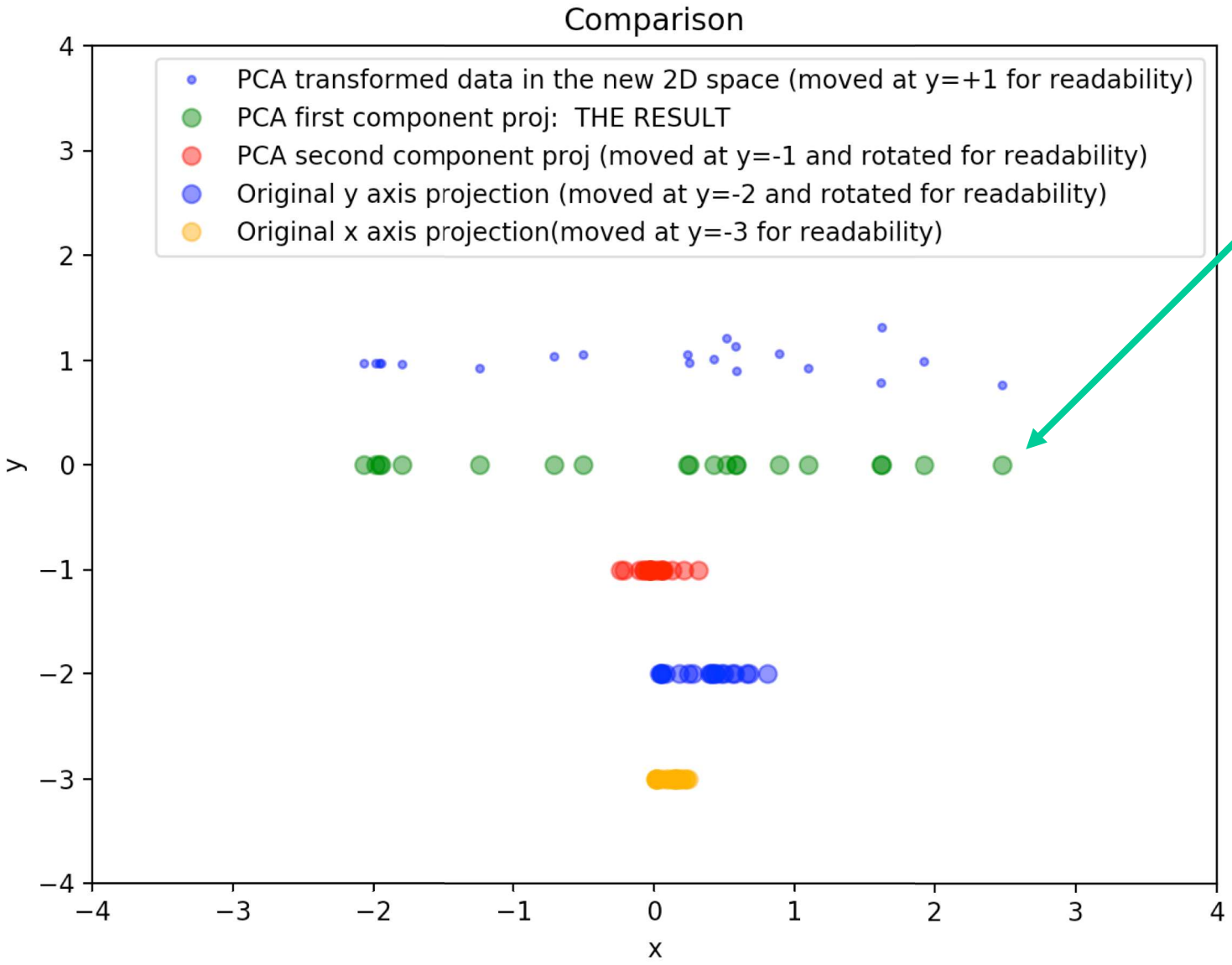


first component (Y_1)

$$\begin{pmatrix} 2.09 & 0 \\ 0 & 0.016 \end{pmatrix}$$

second component (Y_2)

Visual comparison



This is the "best" 1D projection we can get from this data

The only problem is that this axis has not a clear meaning (it is a combination of the old x and y)

How ? Step 1

- PCA computes the eigenvalues and eigenvectors (autovalori e autovettori) from the covariance matrix of the original data **after z-score normalization**

```
Covariance Matrix:
[[1.05555556 1.0390475 ]
[1.0390475  1.05555556]]
```

$$\begin{vmatrix} 1.05263158-\lambda & 1.03653895 \\ 1.03653895 & 1.05263158-\lambda \end{vmatrix} = 0$$

$$\begin{matrix} 1.05263158-\lambda_1 & 1.03653895 \\ 1.03653895 & 1.05263158-\lambda_1 \end{matrix}$$

$\lambda_1 = 0.01650805$,
$\lambda_2 = 2.09460306$

eigenvalues

$$\begin{pmatrix} 1.05263158-\lambda_1 & 1.03653895 \\ 1.03653895 & 1.05263158-\lambda_1 \end{pmatrix} \begin{pmatrix} y_1 \\ y_2 \end{pmatrix} = \begin{pmatrix} 0 \\ 0 \end{pmatrix}$$

λ_1 eigenvector

```
[ 0.70710678 -0.70710678]
```

$$\begin{pmatrix} 1.05263158-\lambda_2 & 1.03653895 \\ 1.03653895 & 1.05263158-\lambda_2 \end{pmatrix} \begin{pmatrix} y_1 \\ y_2 \end{pmatrix} = \begin{pmatrix} 0 \\ 0 \end{pmatrix}$$

λ_2 eigenvector

```
[ 0.70710678  0.70710678]]
```

How ? Step 1 coded in Python

(02_PCA_2D_full.py)

```
Covariance Matrix:
 [[ 0.00610086  0.01798233]
 [ 0.01798233  0.05466161]]
```

$$\begin{vmatrix} 0.00610086-\lambda_1 & 0.01798233 \\ 0.01798233 & 0.05466161-\lambda_2 \end{vmatrix} = 0$$

```
>>> d_cov = np.cov(d.T)
>>> d_val, dvec = np.linalg.eig(d_cov)
```

```
>>> d_val
array([0.01609263, 2.08917052])
```

$\lambda_1 = 0.01650805,$
$\lambda_2 = 2.09460306$

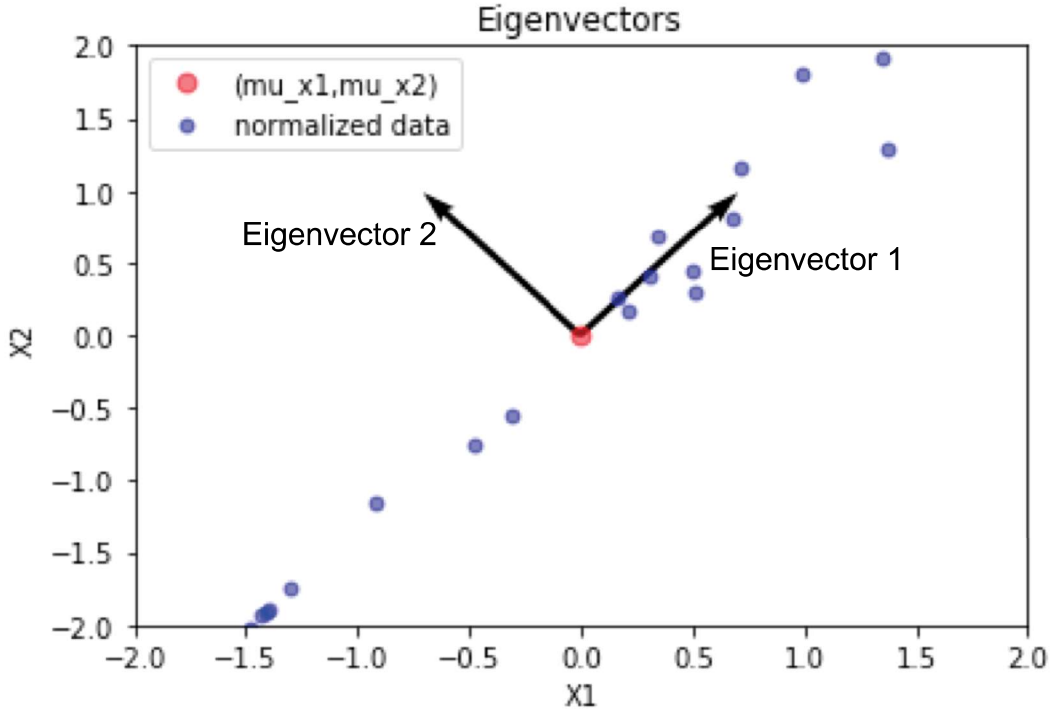
```
>>> d_vec
array([[ 0.70710678, -0.70710678],
       [ 0.70710678,  0.70710678]])
```

$\lambda_1/(\lambda_1+\lambda_2) = 0.008\%$
$\lambda_2/(\lambda_1+\lambda_2) = 0.992\%$

λ_2 takes 99.2% of total variance!

1st component (λ_2) : 0.70710678 $\mathbf{X_1}$ 0.70710678 $\mathbf{X_2}$
2nd component (λ_1) : 0.70710678 $\mathbf{X_1}$ -0.70710678 $\mathbf{X_2}$

Eigenvectors

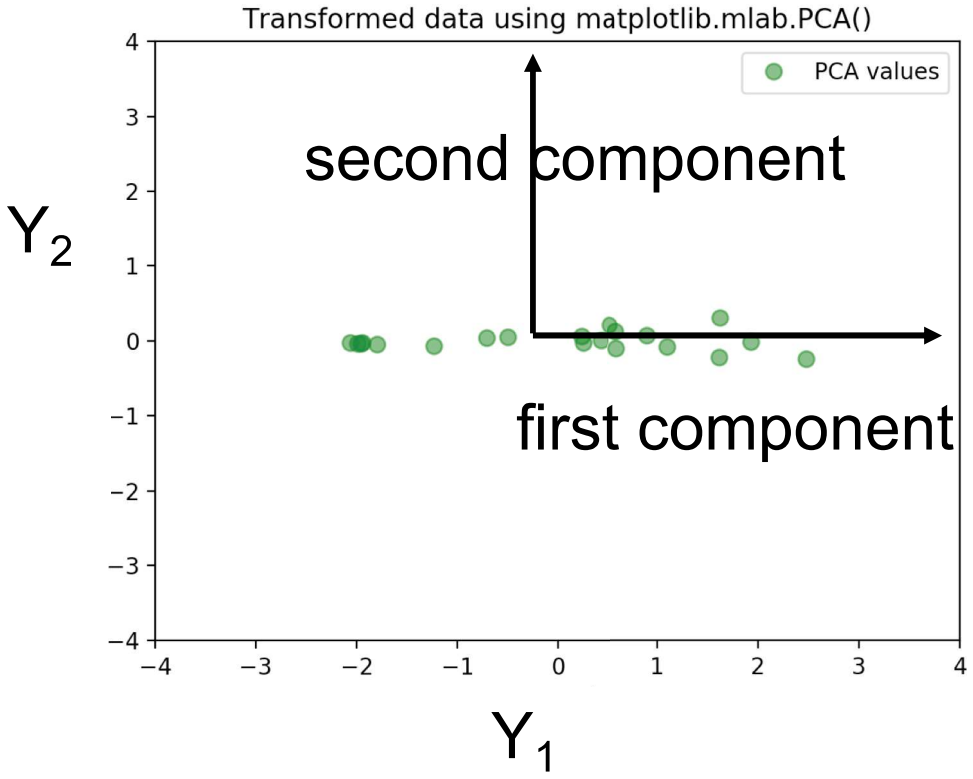


1st component : 0. 707 $\mathbf{X_1}$ 0. 707 $\mathbf{X_2}$

2nd component : -0. 707 $\mathbf{X_1}$ 0. 707 $\mathbf{X_2}$

How ? Step 2

- PCA creates a new 2D space with the eigenvectors with origin in the means of X_1 and X_2 axes and offers them in an ordered fashion (sorted by variance)
- Original data points are mapped in the new space
- The covariance matrix has the eigenvalues on the diagonal



$$\begin{pmatrix} \lambda_1 & 0 \\ 0 & \lambda_2 \end{pmatrix}$$

```
>>> np.cov(dpca.T)
array([[2.08917052e+00, 1.25440863e-17],
       [1.25440863e-17, 1.60926336e-02]])
```

Python code (2) using matplotlib

```
#running mlabPCA that returns an object whose .Y is a numpy.array
# d_pca.Y[:,0] will contain values of first component (max variance)
# d_pca.Y[:,i] will contain values of ith component
d_pca = mlabPCA(d)
```

d_pca is a matplotlib.mlab.PCA object

```
>>> d
array([[ 0.05355255,  0.17827798],
       [ 0.01519837,  0.05072712],
       [ 0.01666373,  0.05466954],
       [ 0.02489676,  0.08309262],
       [ 0.15002833,  0.47698061],
       [ 0.1789041 ,  0.55430599],
       [ 0.14006789,  0.39466536],
       [ 0.01781757,  0.05690527],
       [ 0.10006529,  0.27632587],
       [ 0.08791153,  0.24468   ],
       [ 0.14765551,  0.43319142],
       [ 0.19917171,  0.66016647],
       [ 0.16222048,  0.43852329],
       [ 0.16317345,  0.41526889],
       [ 0.24431702,  0.80389978],
       [ 0.17578821,  0.49895649],
       [ 0.01127593,  0.03715915],
       [ 0.13642456,  0.40966642],
       [ 0.22821393,  0.57616316],
       [ 0.22689575,  0.6785588 ]])
```

```
>>> d_pca.Y
array([[-1.23727754, -0.07159629],
       [-1.98930607, -0.03204384],
       [-1.96346237, -0.03066678],
       [-1.79879616, -0.04239427],
       [ 0.5856756 , -0.10239265],
       [ 1.09381794, -0.07413185],
       [ 0.23773749,  0.06051794],
       [-1.94580786, -0.02688723],
       [-0.50101898,  0.05617675],
       [-0.71210164,  0.04148846],
       [ 0.42775869,  0.01144628],
       [ 1.61055109, -0.21436917],
       [ 0.57958457,  0.13018256],
       [ 0.51627749,  0.21119236],
       [ 2.47587075, -0.24105894],
       [ 0.89312757,  0.06867691],
       [-2.06783958, -0.02637449],
       [ 0.25044614, -0.01987003],
       [ 1.61963626,  0.31604078],
       [ 1.92512662, -0.01393651]])
```

Python code (2) using sklearn

```
#normalize the data with StandardScaler
d_std = preprocessing.StandardScaler().fit_transform(d)
#compute PCA
d_std = preprocessing.StandardScaler().fit_transform(d)
pca=PCA(n_components=2)
dpca=pca.fit_transform(d_std)
#dpca is a numpy array with transformed data and
#pca is a PCA with useful attributes (e.g., explained_variance_)
```

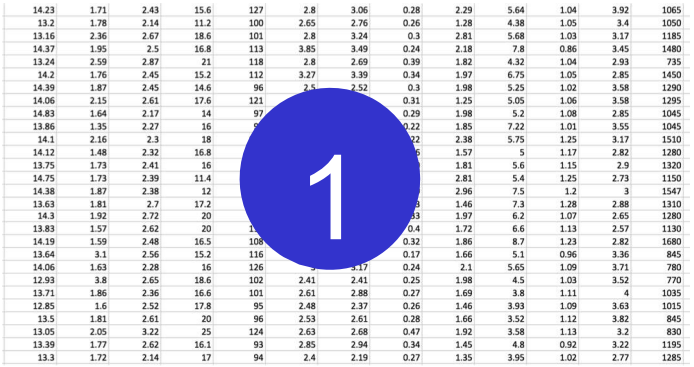
```
>>> d
array([[ 0.05355255,  0.17827798],
       [ 0.01519837,  0.05072712],
       [ 0.01666373,  0.05466954],
       [ 0.02489676,  0.08309262],
       [ 0.15002833,  0.47698061],
       [ 0.1789041 ,  0.55430599],
       [ 0.14006789,  0.39466536],
       [ 0.01781757,  0.05690527],
       [ 0.10006529,  0.27632587],
       [ 0.08791153,  0.24468   ],
       [ 0.14765551,  0.43319142],
       [ 0.19917171,  0.66016647],
       [ 0.16222048,  0.43852329],
       [ 0.16317345,  0.41526889],
       [ 0.24431702,  0.80389978],
       [ 0.17578821,  0.49895649],
       [ 0.01127593,  0.03715915],
       [ 0.13642456,  0.40966642],
       [ 0.22821393,  0.57616316],
       [ 0.22689575,  0.6785588 ]])
```

```
>>> d_std
array([[-0.92551356, -0.82426112],
       [-1.42931023, -1.3839934 ],
       [-1.41006224, -1.36669287],
       [-1.30191824, -1.24196368],
       [ 0.34173265,  0.48653773],
       [ 0.72102695,  0.82586521],
       [ 0.21089844,  0.12531315],
       [-1.39490608, -1.35688179],
       [-0.31455096, -0.39399688],
       [-0.47419513, -0.53286867],
       [ 0.31056481,  0.29437733],
       [ 0.98724971,  1.29041349],
       [ 0.50188115,  0.31777521],
       [ 0.51439887,  0.21572777],
       [ 1.58025059,  1.92115941],
       [ 0.68009847,  0.58297466],
       [-1.48083297, -1.44353381],
       [ 0.16304193,  0.1911424 ],
       [ 1.36873036,  0.92178121],
       [ 1.35141549,  1.37112468]])
```

```
>>> dpca
array([[-1.23727754, -0.07159629],
       [-1.98930607, -0.03204384],
       [-1.96346237, -0.03066678],
       [-1.79879616, -0.04239427],
       [ 0.5856756 , -0.10239265],
       [ 1.09381794, -0.07413185],
       [ 0.23773749,  0.06051794],
       [-1.94580786, -0.02688723],
       [-0.50101898,  0.05617675],
       [-0.71210164,  0.04148846],
       [ 0.42775869,  0.01144628],
       [ 1.61055109, -0.21436917],
       [ 0.57958457,  0.13018256],
       [ 0.51627749,  0.21119236],
       [ 2.47587075, -0.24105894],
       [ 0.89312757,  0.06867691],
       [-2.06783958, -0.02637449],
       [ 0.25044614, -0.01987003],
       [ 1.61963626,  0.31604078],
       [ 1.92512662, -0.01393651]])
```

PCA as black-box

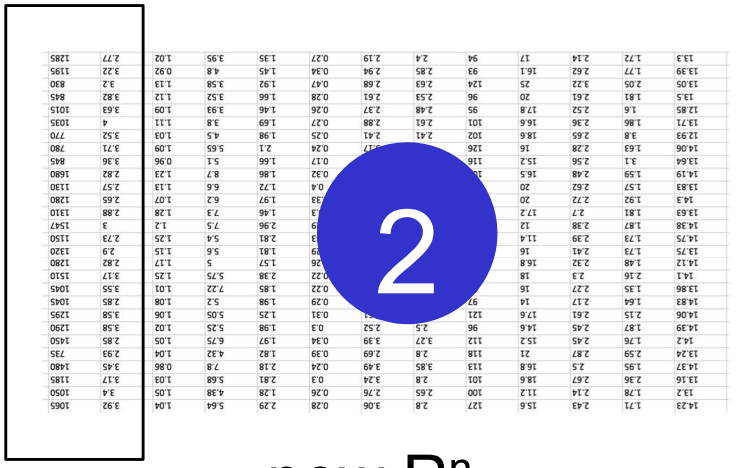
$x_1, x_2, \ldots\ldots\ldots\ldots\ldots\ldots x_{13}$



R^n

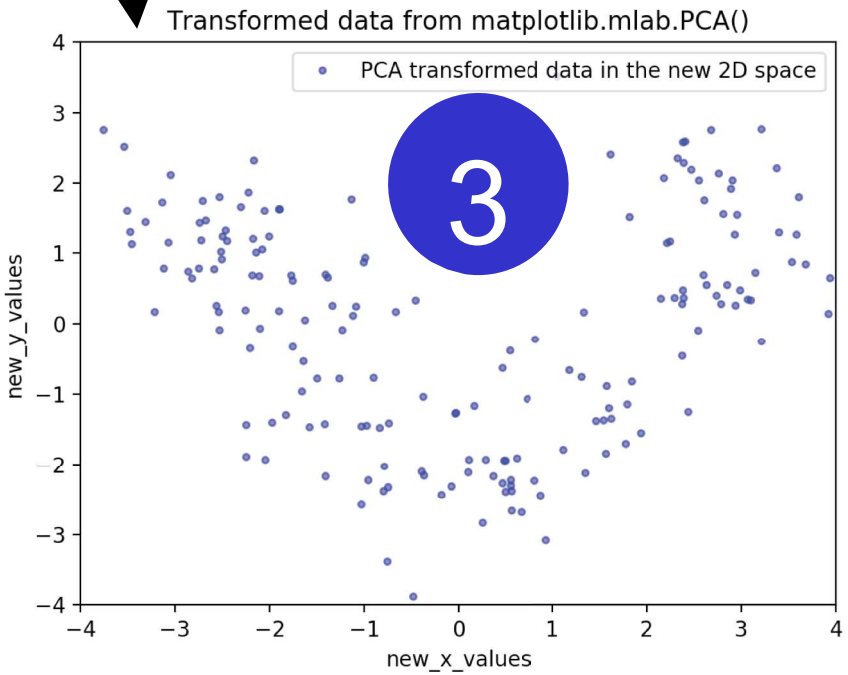
Linear transformation

$y_1, y_2, \ldots\ldots\ldots\ldots\ldots\ldots y_{13}$



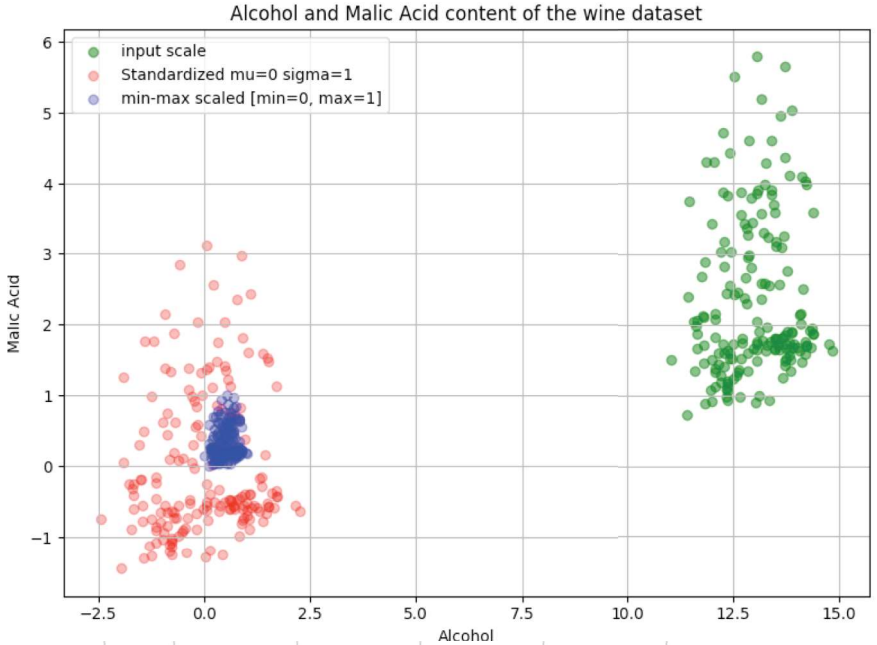
new R^n (meaningless axes)

2D Projection



- A linear transformation (e.g. PCA) between $x_1, x_2, \ldots, x_{13}$ and $y_1, y_2, \ldots, y_{13}$ preserves closeness
- A projection creates new ones ☹
- If data points p' and p'' are close in R^n they are close in R^2
- Vice versa is not always true (false positive because of projection)
- Remember that PCA uses mean, variance, covariance: data MUST be a ratio scale...
- e.g., PCA on numeral values encoding categorical attributes MAKES NO SENSE!

Just a note on normalization



Minmax does not
Preserve closeness!!!
Z-score does!

		A	B	C	D
1	P1	40	0	0	0.1
2	P2	1	0	0	0.1
3	P3	49	50	51	0.1
4	P4	10	10	10	0.1
5	P5	10	10	10	0.12
6	P6	30	30	30	0.11

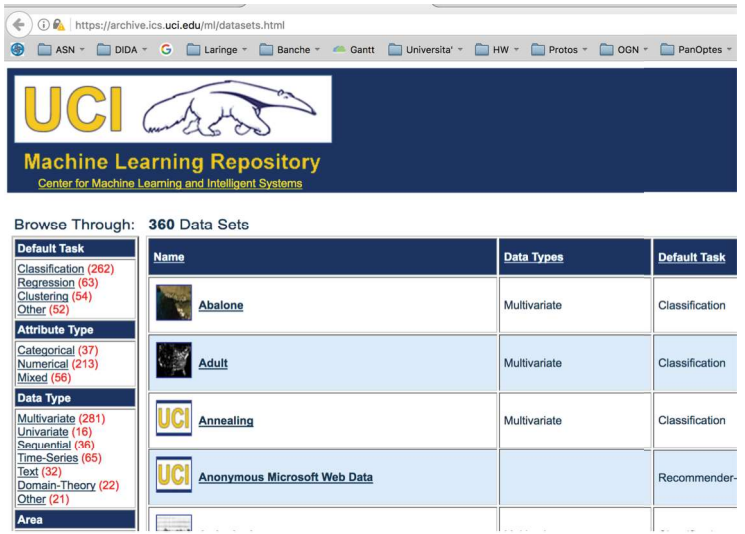
Minmax →

	A	B	C	D
P1	0.813	0.000	0.000	0.000
P2	0.000	0.000	0.000	0.000
P3	1.000	1.000	1.000	0.000
P4	0.188	0.200	0.196	0.000
P5	0.188	0.200	0.196	1.000
P6	0.604	0.600	0.588	0.500

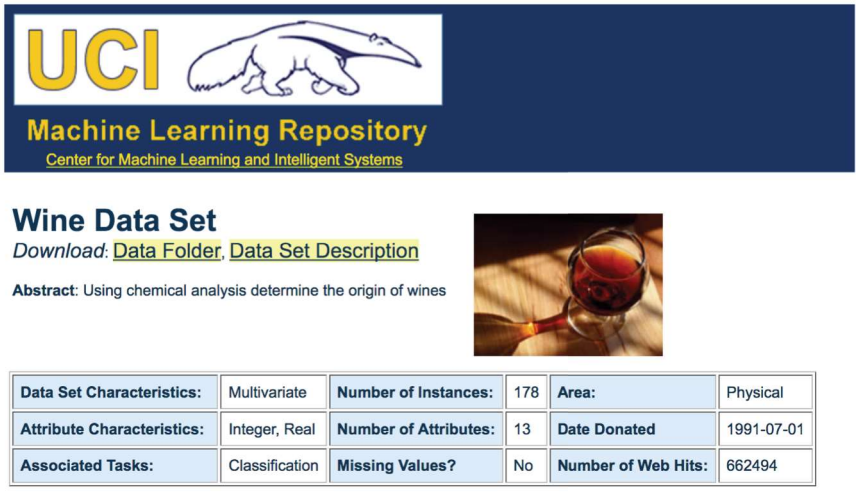
P4 and P5 are very close

P4 and P5 are far away…

Let's go

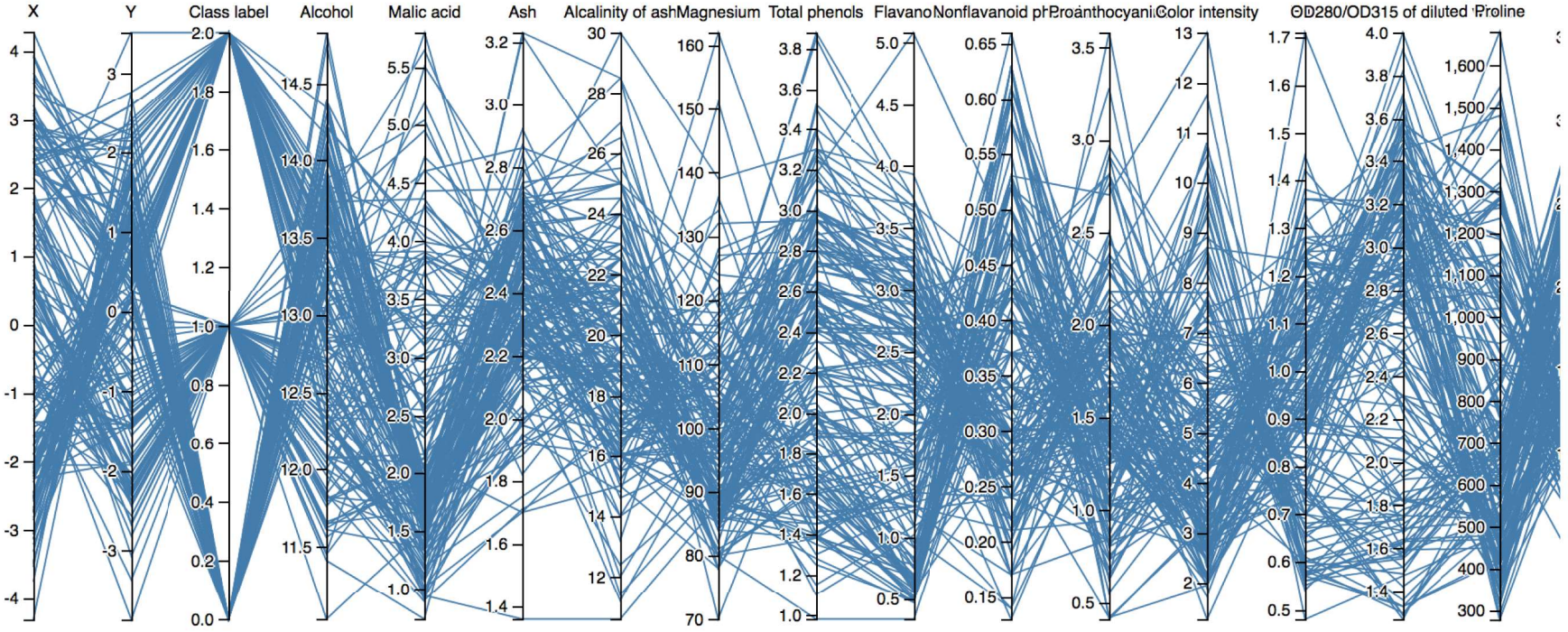


- UCI repository : https://archiveicsuciedu/ml/datasetshtml
- **wine dataset**

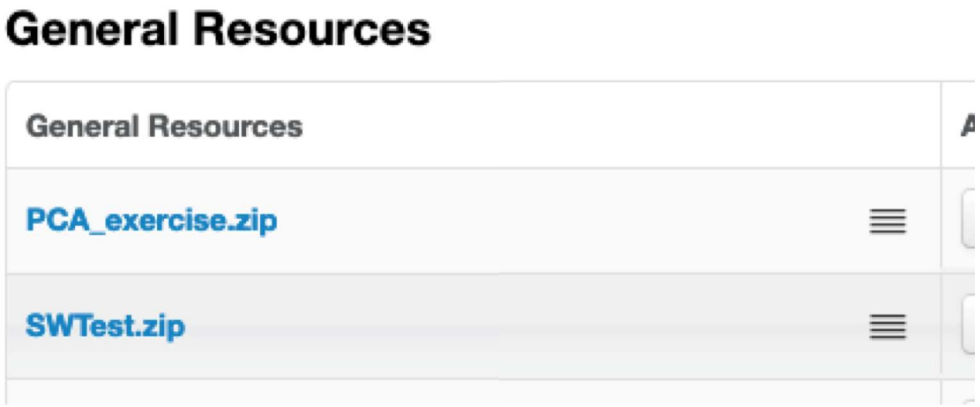


Wine dataset

Class	Alcohol	Malic acid	Ash	Alcalinity of ash	Magnesium	Total phenol	Flavanoids	Nonflavano	Proanthocya	Color intensi	Hue	OD280/OD3	Proline
1	14.23	1.71	2.43	15.6	127	2.8	3.06	0.28	2.29	5.64	1.04	3.92	1065
1	13.2	1.78	2.14	11.2	100	2.65	2.76	0.26	1.28	4.38	1.05	3.4	1050
1	13.16	2.36	2.67	18.6	101	2.8	3.24	0.3	2.81	5.68	1.03	3.17	1185
1	14.37	1.95	2.5	16.8	113	3.85	3.49	0.24	2.18	7.8	0.86	3.45	1480
1	13.24	2.59	2.87	21	118	2.8	2.69	0.39	1.82	4.32	1.04	2.93	735
1	14.2	1.76	2.45	15.2	112	3.27	3.39	0.34	1.97	6.75	1.05	2.85	1450
1	14.39	1.87	2.45	14.6	96	2.5	2.52	0.3	1.98	5.25	1.02	3.58	1290
1	14.06	2.15	2.61	17.6	121	2.6	2.51	0.31	1.25	5.05	1.06	3.58	1295
1	14.83	1.64	2.17	14	97	2.8	2.98	0.29	1.98	5.2	1.08	2.85	1045
1	13.86	1.35	2.27	16	98	2.98	3.15	0.22	1.85	7.22	1.01	3.55	1045
1	14.1	2.16	2.3	18	105	2.95	3.32	0.22	2.38	5.75	1.25	3.17	1510
1	14.12	1.48	2.32	16.8	95	2.2	2.43	0.26	1.57	5	1.17	2.82	1280



Objectives



- Load the dataset
- Explore the data
- Compute the PCA on it
- Plot it
- Compute the cumulated variance of the first two components
- Generate the file pca.csv using the function

```
def generateFile(label,Y,dataFile):
```

- Run scatterbrush.html

Some python code (reading a csv an running PCA on standardized data)

```
###read data from a CSV file, you can choose different delimiters
att=['Class label', 'Alcohol', 'Malic acid','Ash','Alcalinity of ash','Magnesium',
     'Total phenols','Flavanoids','Nonflavanoid phenols','Proanthocyanins',
    'Color intensity','Hue','OD280/OD315 of diluted wines','Proline']

d=np.genfromtxt('wine.data.csv',skip_header=0,usecols=[i for i in range(1,14)],delimiter=',')
d_pca = mlabPCA(d)
generateFile(att,d_pca.Y,'wine.data.csv')
```

Some python code (exploring the data)

```
d_cov=np.cov(d.T)
for i in range(len(d_cov)):
    print('Variance original data not scaled axis n'+str(i),d_cov[i][i])

std_scale = preprocessing.StandardScaler().fit(d)
d_std = std_scale.transform(d)
d_cov=np.cov(d_std.T)
d_val,d_vec= np.linalg.eig(d_cov)
d_val.sort()
print('Cumulated variance of the first two PCA components:',
      (d_val[-1]+d_val[-2])/sum(d_val))
```

```
>>> d_vec
array([[-0.1443294 ,  0.48365155, -0.20738262,  0.0178563 , -0.26566365,
         0.21353865,  0.05639636, -0.01496997,  0.39613926, -0.26628645,
        -0.50861912, -0.22591696,  0.21160473],
       [ 0.24518758,  0.22493093,  0.08901289, -0.53689028,  0.03521363,
         0.53681385, -0.42052391, -0.02596375,  0.06582674,  0.12169604,
         0.07528304,  0.07648554, -0.30907994],
       [ 0.00205106,  0.31606881,  0.6262239 ,  0.21417556, -0.14302547,
         0.15447466,  0.14917061,  0.14121803, -0.17026002, -0.04962237,
         0.30769445, -0.49869142, -0.02712539],
       [ 0.23932041, -0.0105905 ,  0.61208035, -0.06085941,  0.06610294,
        -0.10082451,  0.28696914, -0.09168285,  0.42797018, -0.05574287,
        -0.20044931,  0.47931378,  0.05279942],
       [-0.14199204,  0.299634  ,  0.13075693,  0.35179658,  0.72704851,
         0.03814394, -0.3228833 , -0.05677422, -0.15636143,  0.06222011,
        -0.27140257,  0.07128891,  0.06787022],
       [-0.39466085,  0.06503951,  0.14617896, -0.19806835, -0.14931841,
        -0.0841223 ,  0.02792498,  0.46390791, -0.40593409, -0.30388245,
        -0.28603452,  0.30434119, -0.32013135],
       [-0.4229343 , -0.00335981,  0.1506819 , -0.15229479, -0.10902584,
        -0.01892002,  0.06068521, -0.83225706, -0.18724536, -0.04289883,
        -0.04957849, -0.02569409, -0.16315051],
       [ 0.2985331 ,  0.02877949,  0.17036816,  0.20330102, -0.50070298,
        -0.25859401, -0.59544729, -0.11403985, -0.23328465,  0.04235219,
        -0.19550132,  0.11689586,  0.21553507],
       [-0.31342949,  0.03930172,  0.14945431, -0.39905653,  0.13685982,
        -0.53379539, -0.37213935,  0.11691707,  0.36822675, -0.09555303,
         0.20914487, -0.23736257,  0.1341839 ],
       [ 0.0886167 ,  0.52999567, -0.13730621, -0.06592568, -0.07643678,
        -0.41864414,  0.22771214,  0.01199928, -0.03379692,  0.60422163,
        -0.05621752,  0.0318388 , -0.29077518],
       [-0.29671456, -0.27923515,  0.08522192,  0.42777141, -0.17361452,
         0.10598274, -0.23207564,  0.08988884,  0.43662362,  0.259214  ,
        -0.08582839, -0.04821201, -0.52239889],
       [-0.37616741, -0.16449619,  0.16600459, -0.18412074, -0.10116099,
         0.26585107,  0.0447637 ,  0.15671813, -0.07810789,  0.60095872,
        -0.1372269 ,  0.0464233 ,  0.52370587],
       [-0.28675223,  0.36490283, -0.12674592,  0.23207086, -0.1578688 ,
         0.11972557, -0.0768045 , -0.01444734,  0.12002267, -0.07940162,
         0.57578611,  0.53926983,  0.162116  ]])
>>>
```

```
>>> d_val
array([0.10396199, 0.16972374, 0.22706428, 0.25232001, 0.29051203,
       0.35046627, 0.55414147, 0.64528221, 0.85804868, 0.92416587,
       1.45424187, 2.51108093, 4.73243698])
```

Some python code (exploring the data using sklearn)

```
d=np.genfromtxt('wine.data.csv',skip_header=0,usecols=[i for i in range(1,14)
#normalize the data with StandardScaler
d_std = preprocessing.StandardScaler().fit_transform(d)
#compute PCA
pca=PCA(n_components=13)
dpca=pca.fit_transform(d_std) #dpca is a numpy array with transformrd data ar
```

```
#compute and sort eigenvalues
v=pca.explained_variance_ratio_
v.sort()
print('Cumulated variance of the first two PCA components:',
      (v[-1]+v[-2]))
```

```
Cumulated variance of the first two PCA components: 0.5540633835693529
```

VA_06_PCA_v4a.zip

Data exploration

```
Variance original data not scaled axis n0 0.659062327811
Variance original data not scaled axis n1 1.24801540342
Variance original data not scaled axis n2 0.0752646353076
Variance original data not scaled axis n3 11.152686155
Variance original data not scaled axis n4 203.989335365
Variance original data not scaled axis n5 0.391689535327
Variance original data not scaled axis n6 0.997718672634
Variance original data not scaled axis n7 0.015488633911
Variance original data not scaled axis n8 0.327594667682
Variance original data not scaled axis n9 5.37444938349
Variance original data not scaled axis n10 0.0522449607059
Variance original data not scaled axis n11 0.504086408938
Variance original data not scaled axis n12 99166.7173554
Cumulated variance of the first two PCA components: 0.554063383569
```

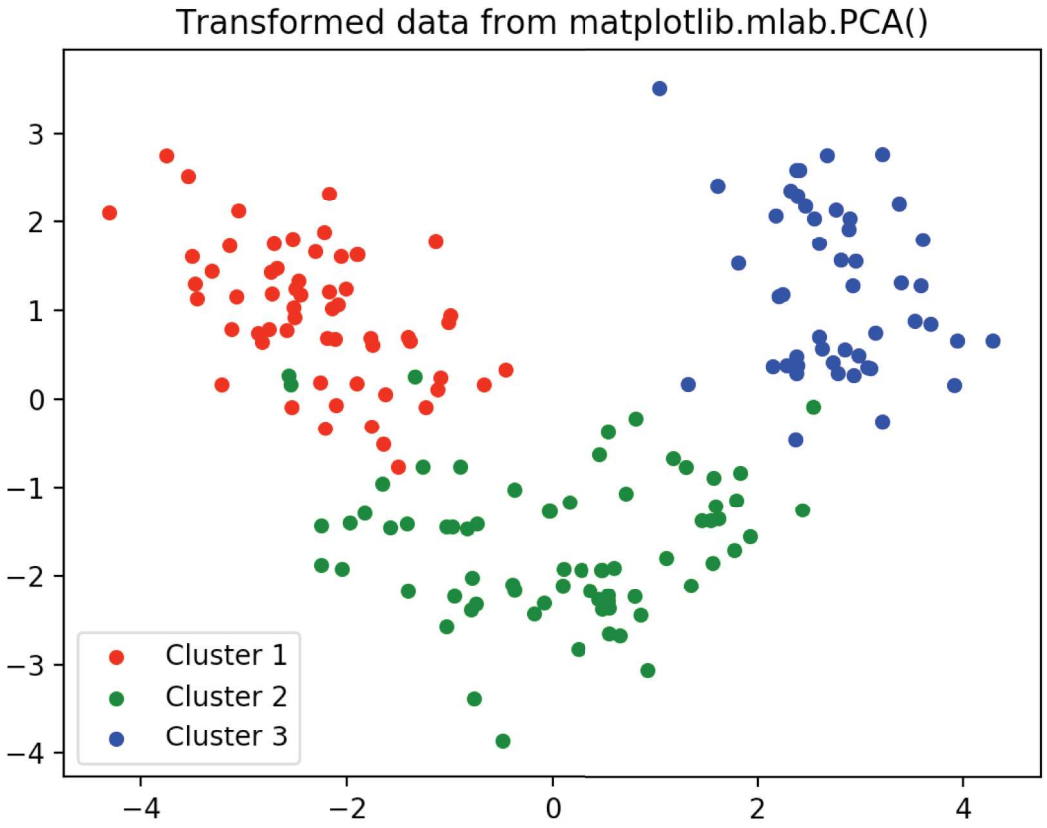
Some python code (plotting wine classes on PCA)

```
s = 30
plt.scatter(d_pca.Y[0:59, 0], d_pca.Y[0:59, 1],
            color='red',s=s, lw=0, label='MDS Cluster 1')
plt.scatter(d_pca.Y[59:130, 0], d_pca.Y[59:130, 1],
            color='green',s=s, lw=0, label='MDS Cluster 2')
plt.scatter(d_pca.Y[130:178, 0], d_pca.Y[130:178, 1],
            color='blue',s=s, lw=0, label='MDS Cluster 3')
plt.legend()
plt.title('Transformed data from matplotlib.mlab.PCA()')

plt.show()
```

or, using sklearn

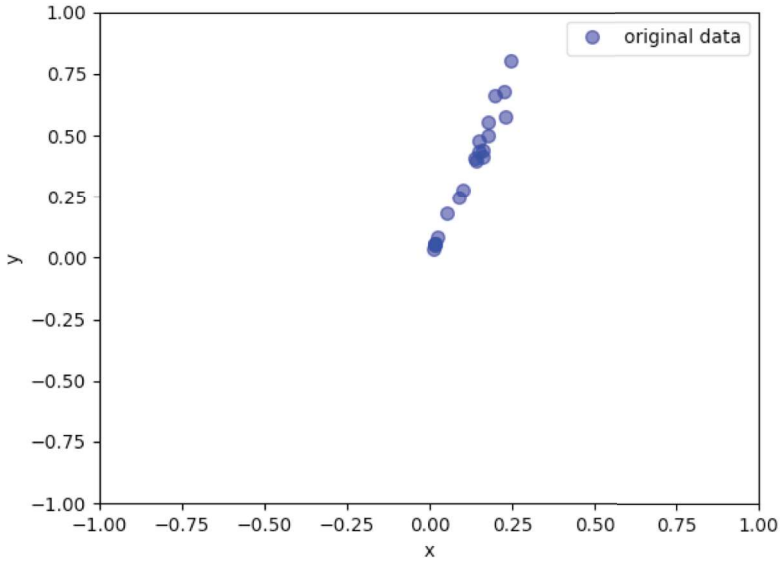
```
plt.scatter(dpca[0:59, 0], dpca[0:59, 1],
            color='red',s=s, lw=0, label='Cluster 1')
plt.scatter(dpca[59:130, 0], dpca[59:130, 1],
            color='green',s=s, lw=0, label='Cluster 2')
plt.scatter(dpca[130:178, 0], dpca[130:178, 1],
            color='blue',s=s, lw=0, label='Cluster 3')
plt.xlabel('Y1')
plt.ylabel('Y2')
plt.legend()
plt.title('Transformed data from matplotlib.mlab.PCA()')

plt.show()
```

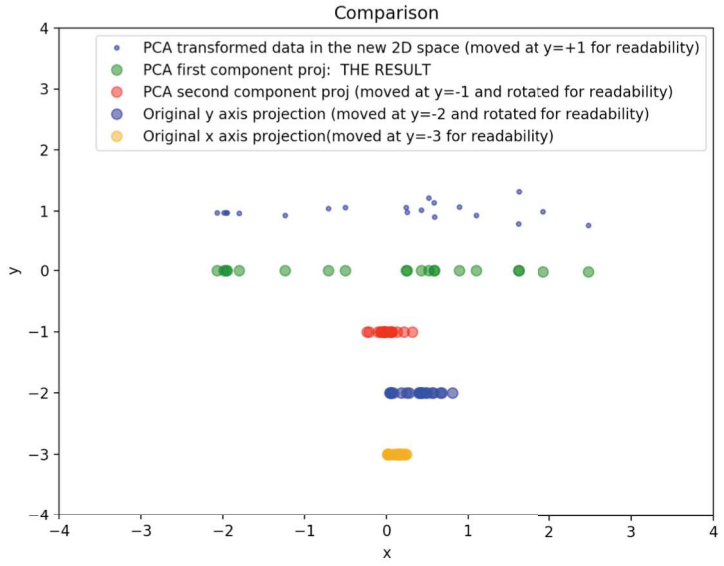
Some python code (04_PCA_wine_full.py) (plotting wine classes with PCA)



Final consideration

- It is a **linear** transformation of the original space
- It's focus is on VARIANCE
- It preserves data point "closeness" from R^n to R^k
- The projection on the new axes creates false positives:
  - points far away in R^n may be close in R^k





Multidimensional Scaling (MDS)

- The input is a set of I elements $\{t_1,..., t_I\}$ on a R^n space (not very relevant)
- It works using a dissimilarity matrix (e.g., distance among the I elements)

$$\Delta := \begin{pmatrix} \delta_{1,1} & \delta_{1,2} & \cdots & \delta_{1,I} \\ \delta_{2,1} & \delta_{2,2} & \cdots & \delta_{2,I} \\ \vdots & \vdots & & \vdots \\ \delta_{I,1} & \delta_{I,2} & \cdots & \delta_{I,I} \end{pmatrix}.$$

- Its goals is to find a mapping $\{t_1,..., t_I\} \rightarrow \{x_1,...,x_I\}$ (x_i in R^2) such that

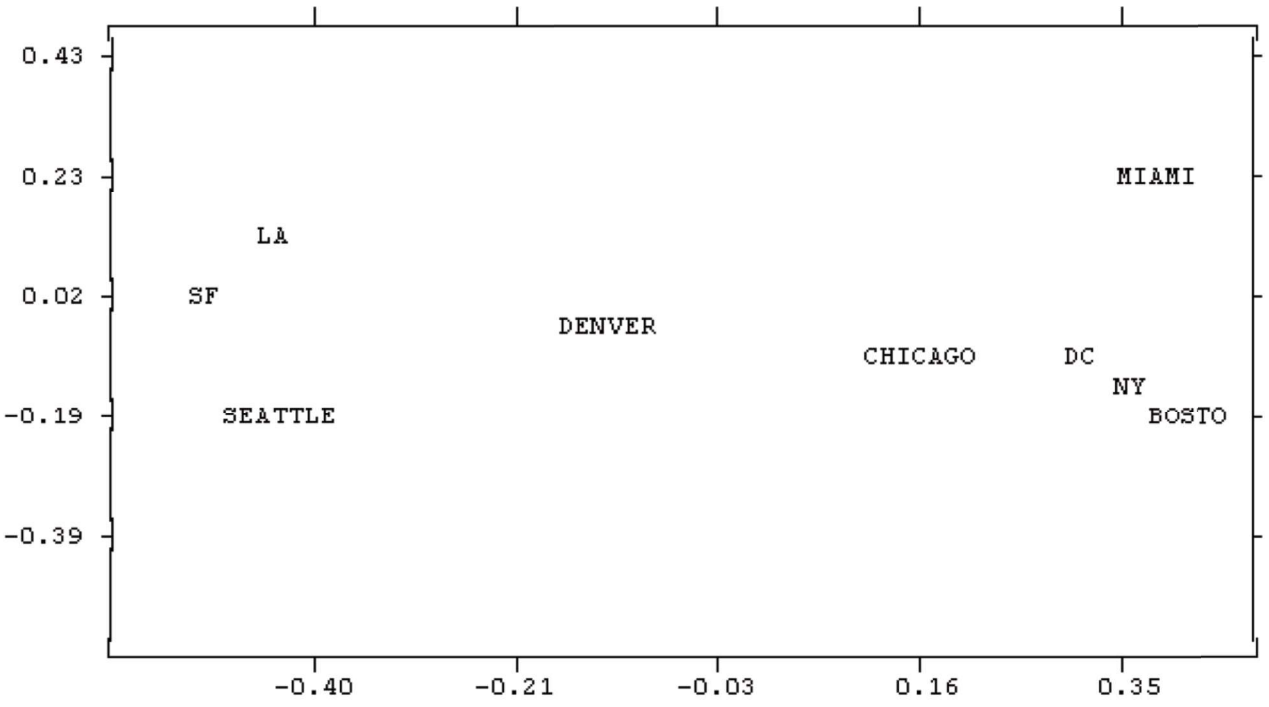
$$\|x_i - x_j\| \approx \delta_{i,j}$$

- And it's formulated as an optimization problem with a cost function like:

$$\min_{x1,...,x_I} \sum_{i<j} (\|x_i - x_j\| - \delta_{i,j})^2.$$

Example of MDS

	BOST	NY	DC	MIAM	CHIC	SEAT	SF	LA	DENV
BOSTON	0	206	429	1504	963	2976	3095	2979	1949
NY	206	0	233	1308	802	2815	2934	2786	1771
DC	429	233	0	1075	671	2684	2799	2631	1616
MIAMI	1504	1308	1075	0	1329	3273	3053	2687	2037
CHICAGO	963	802	671	1329	0	2013	2142	2054	996
SEATTLE	2976	2815	2684	3273	2013	0	808	1131	1307
SF	3095	2934	2799	3053	2142	808	0	379	1235
LA	2979	2786	2631	2687	2054	1131	379	0	1059
DENVER	1949	1771	1616	2037	996	1307	1235	1059	0



Multidimensional scaling

- It takes in input a square symmetric I^2 matrix of **dissimilarities** (e.g., based on street distance, perceived food taste, image similarity, etc.) where 0 means equal
- note that you can reverse your thinking using a matrix of **similarities** where 0 means totally different
- note that we can use functions that are not directly related to common (e.g., Euclidean distance) similarity or dissimilarity, e.g., the difference in passed exams between two students or the difference in inhabitants between two cities
- from now on we use dissimilarities, that has a clear resemblance with distance (two cities that are very close are similar and have a low distance value)

Other examples

- We have four snacks and we perform a test with users on snack pairs averaging their answer:
  - 0 = the 2 snacks have the same taste
  - 1 = the 2 snacks have totally different taste

	t1	**t2**	**t3**	**t4**
t1	0	0.2	0.5	0.8
t2	0.2	0	0.1	0.9
t3	0.5	0.1	0	0.7
t4	0.8	0.9	07	0

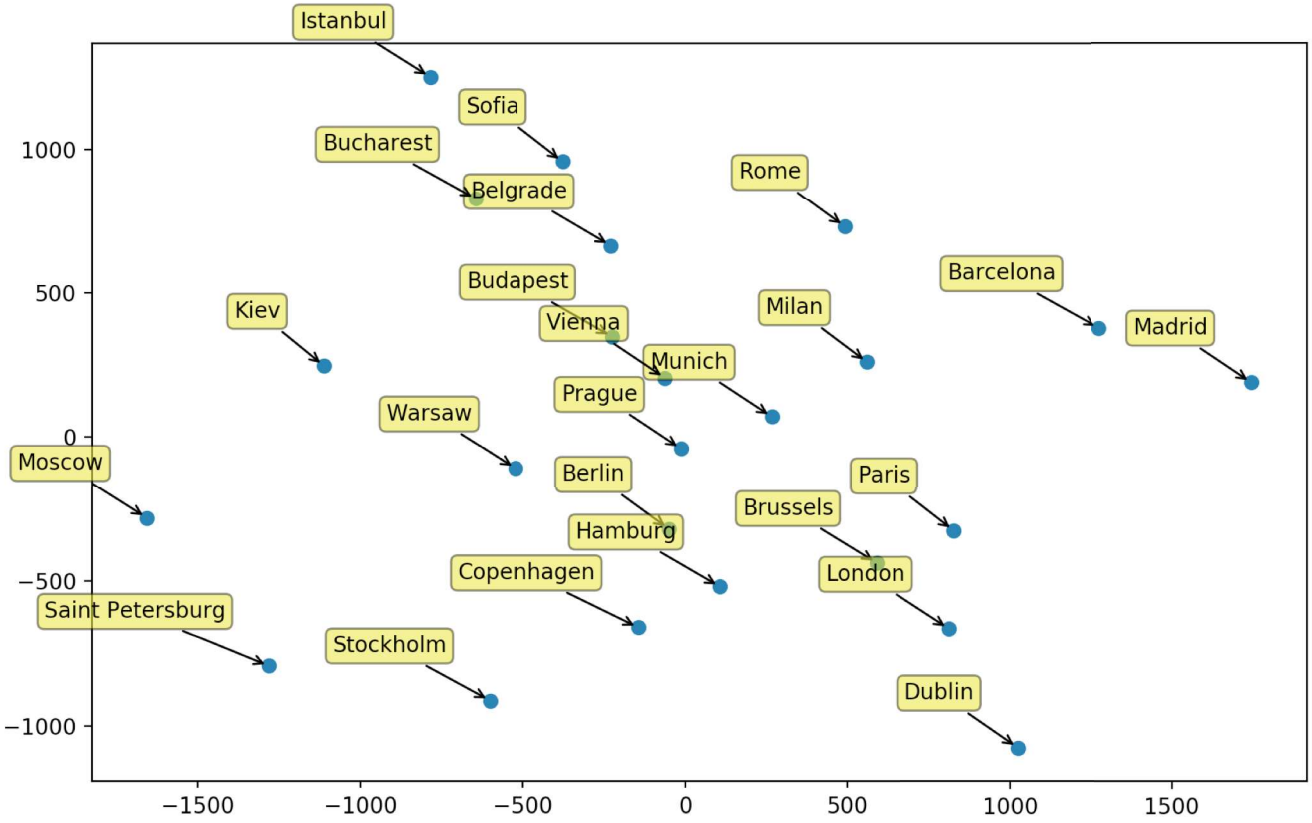
Multidimensional scaling

- Or we have I cities and a matrix of I^2 distances among cities (typically not Euclidean)

	Barcelona	Belgrade	Berlin	Brussels	Bucharest	Budapest	Copenhagen
Barcelona	0	1528.13	1497.61	1062.89	1968.42	1498.79	1757.54
Belgrade	1528.13	0	999.25	1372.59	447.34	316.41	1327.24
Berlin	1497.61	999.25	0	651.62	1293.4	689.06	354.03
Brussels	1062.89	1372.59	651.62	0	1769.69	1131.52	766.67
Bucharest	1968.42	447.34	1293.4	1769.69	0	639.77	1571.54
Budapest	1498.79	316.41	689.06	1131.52	639.77	0	1011.31
Copenhagen	1757.54	1327.24	354.03	766.67	1571.54	1011.31	0

MDS goal

- MDS tries to map the I elements in a R^2 space in which the Euclidean distances resemble the element similarities



MDS Algorithm

1. computes the dissimilarity matrix among the I elements $\mathbf{M_D}$
2. plots the I points on a R^n space (usually n=2) in a random fashion
3. computes the Euclidean distance matrix point pairs in R^n $\mathbf{M_E}$
4. compares $\mathbf{M_D}$ and $\mathbf{M_E}$ evaluating the stress function
5. adjusts point positions lowering the stress
6. repeats 3..5 until stress gets any lower (0=perfect MDS)

- An example of stress function:
- $\sum_{i,j} (\mathrm{M_D}(\mathrm{i},\mathrm{j}) - \mathrm{ME}(\mathrm{i},\mathrm{j}))^2$

Example in numpy and sklearn

(01_MDS_EuropeRouteTable.py)

- Read data with Pandas
  - Pandas deals in a better way with data mixing text and numbers
  - .values returns a numpy array

```
import pandas as pd
import matplotlib.pyplot as plt
from sklearn import manifold
data = pd.io.parsers.read_csv(  #pandas handles in a better way
     'european_city_distances.csv',
     header="infer"          #the first row contains the city names
    )
dissM=data.values[:,1:] #the first column contains the city names
cities=data.values[:,0]
```

VA_06_MDS_v4.zip

	Barcelona	Belgrade	Berlin
Barcelona	0	1528.13	1497.61
Belgrade	1528.13	0	999.25
Berlin	1497.61	999.25	0
Brussels	1062.89	1372.59	651.62
Bucharest	1968.42	447.34	1293.4
Budapest	1498.79	316.41	689.06

pandas.core.frame.DataFrame

```
In [7]: data
Out[7]:
          Unnamed: 0  Barcelona  Belgrade  ...  Stockholm   Vienna   Warsaw
0          Barcelona       0.00   1528.13  ...    2276.51  1347.43  1862.33
1           Belgrade    1528.13      0.00  ...    1620.96   489.28   826.66
2             Berlin    1497.61    999.25  ...     810.38   523.61   516.06
3           Brussels    1062.89   1372.59  ...    1280.88   914.81  1159.85
4          Bucharest    1968.42    447.34  ...    1742.25   855.32   946.12
5           Budapest    1498.79    316.41  ...    1316.59   216.98   545.29
6         Copenhagen    1757.54   1327.24  ...     521.68   868.87   667.80
7             Dublin    1469.29   2145.39  ...    1626.56  1680.00  1823.72
8            Hamburg    1471.78   1229.93  ...     809.65   742.79   750.49
9            Istanbul    2230.42    809.48  ...    2171.65  1273.88  1386.08
10               Kiev    2391.06    976.02  ...    1265.79  1052.76   690.12
11             London    1137.67   1688.97  ...    1431.07  1233.48  1445.85
12             Madrid     504.64   2026.94  ...    2591.53  1807.09  2288.42
13              Milan     725.12    885.32  ...    1650.12   623.36  1143.01
14             Moscow    3006.93   1710.99  ...    1227.38  1669.22  1149.41
15             Munich    1054.55    773.33  ...    1311.80   354.42   809.02
16              Paris     831.59   1445.70  ...    1541.83  1033.73  1365.91
17             Prague    1353.90    738.10  ...    1052.85   250.71   514.69
18               Rome     856.69    721.55  ...    1974.79   763.26  1316.24
19  Saint Petersburg    2813.02   1797.75  ...     688.33  1577.56  1023.41
20             Sofia    1745.55    329.46  ...    1884.91   817.45  1076.99
21         Stockholm    2276.51   1620.96  ...       0.00  1241.90   808.14
22            Vienna    1347.43    489.28  ...    1241.90     0.00   557.43
23            Warsaw    1862.33    826.66  ...     808.14   557.43     0.00

[24 rows x 25 columns]
```

Example in numpy and sklearn (compute and plot MDS)

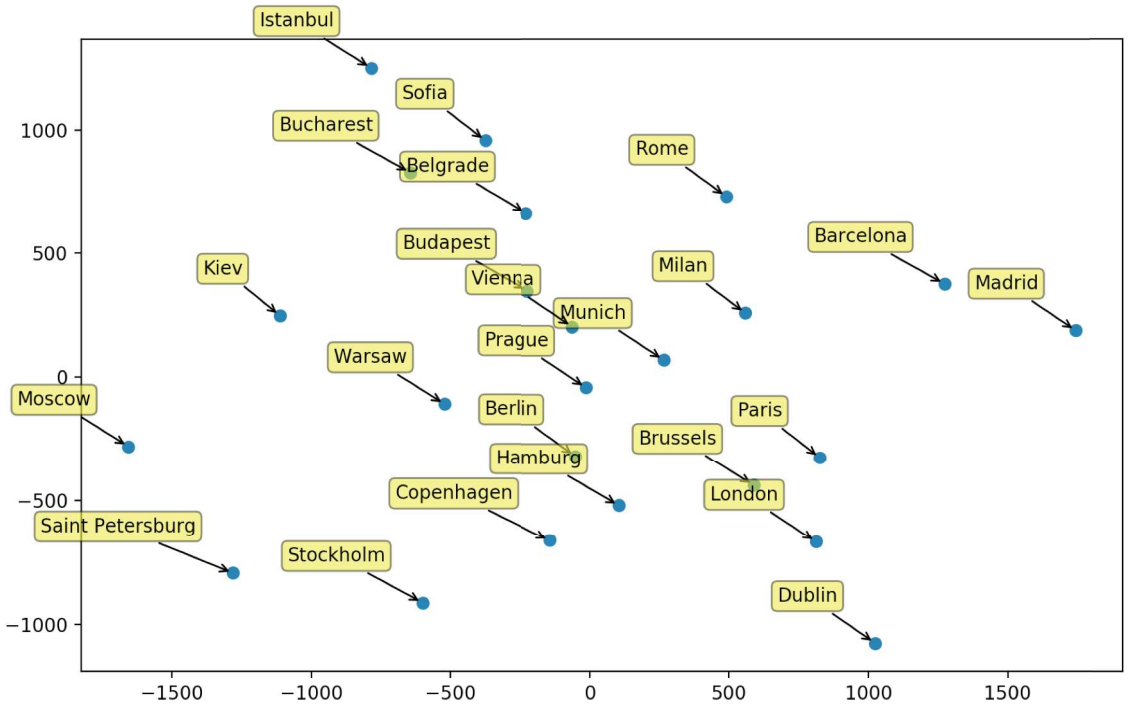
class `sklearn.manifold.` **MDS** (*n_components=2, metric=True, n_init=4, max_iter=300, verbose=0, eps=0.001, n_jobs=1, random_state=None, dissimilarity='euclidean'*) ¶ [source]

```
>>> dissM[:5,:5]
array([[0.0, 1528.13, 1497.61, 1062.89, 1968.42],
       [1528.13, 0.0, 999.25, 1372.59, 447.34],
       [1497.61, 999.25, 0.0, 651.62, 1293.4],
       [1062.89, 1372.59, 651.62, 0.0, 1769.69],
       [1968.42, 447.34, 1293.4, 1769.69, 0.0]],
```

```
mds = manifold.MDS(n_components=2, max_iter=3000,dissimilarity="precomputed")
pos = mds.fit(dissM).embedding_

plt.scatter(pos[:, 0], pos[:, 1], marker = 'o')
for label, x, y in zip(cities, pos[:, 0], pos[:, 1]):
    plt.annotate(
        label,
        xy = (x, y), xytext = (-20, 20),
        textcoords = 'offset points', ha = 'right', va = 'bottom',
        bbox = dict(boxstyle = 'round,pad=0.3', fc = 'yellow', alpha = 0.5),
        arrowprops = dict(arrowstyle = '->', connectionstyle = 'arc3,rad=0'))
plt.show()
```

MDS interpretation

- axes are meaningless and orientation is arbitrary
- what that only matters is **distance**
- you can look for clusters and similarity/dissimilarity



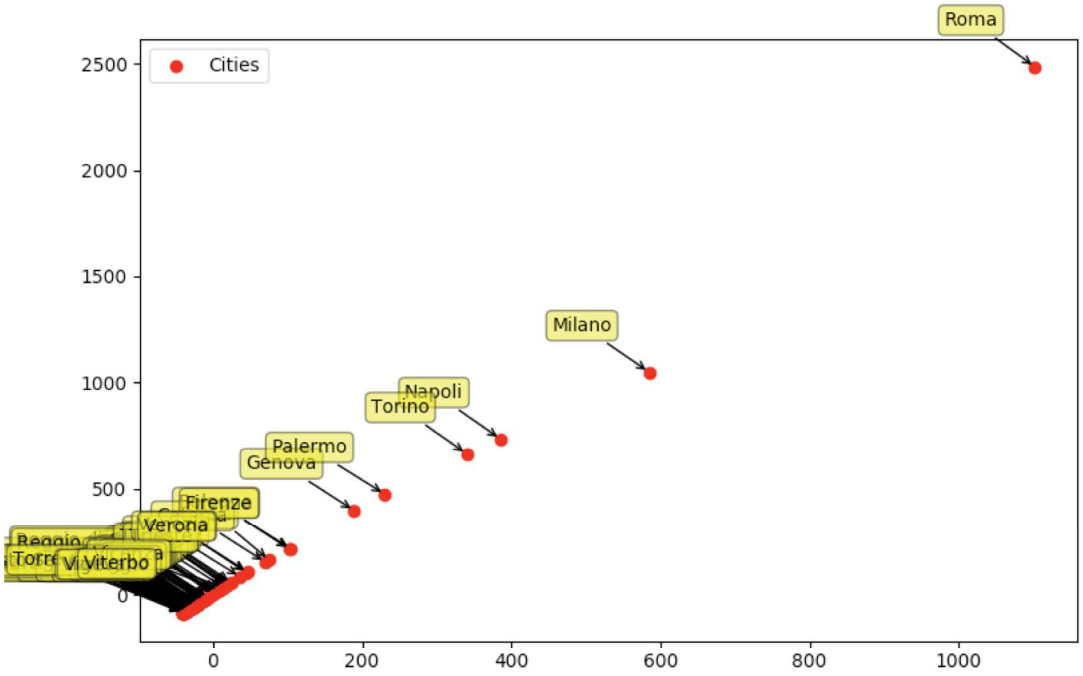
Defining your own M_D

	A	B	C
1	Acerra	60	
2	Acireale	53	
3	Afragola	65	
4	Agrigento	60	
5	Alessandria	94	
6	Altamura	70	
7	Ancona	101	
8	Andria	100	
9	Anzio	54	
10	Aprilia	73	
11	Arezzo	100	
12	Asti	76	
13	Avellino	55	
14	Aversa	53	
15	Bagheria	55	
16	Bari	326	
17	Barletta	95	
18	Battipaglia	51	
19	Benevento	60	
20	Bergamo	119	
21	Bisceglie	55	

Thousands of inhabitants
Similarity among cities based on Inhabitants:
Difference $|c_i-c_j|$?
Let's try

Defining your own M_D

```
data = pd.io.parsers.read_csv(
     'inhabitants.csv',
     header=None
    )
d=data.values[:,1]
cities=data.values.T[0]

dissM1=np.zeros((len(data),len(data))) #creates a zeros dissM

for i in range(len(d)):
    for j in range (len(d)):
        dissM1[i][j]= abs(d[i]-d[j])

mds = manifold.MDS(n_components=2, max_iter=300, eps=1e-9,
                   dissimilarity="precomputed")
pos1 = mds.fit(dissM1).embedding_
```

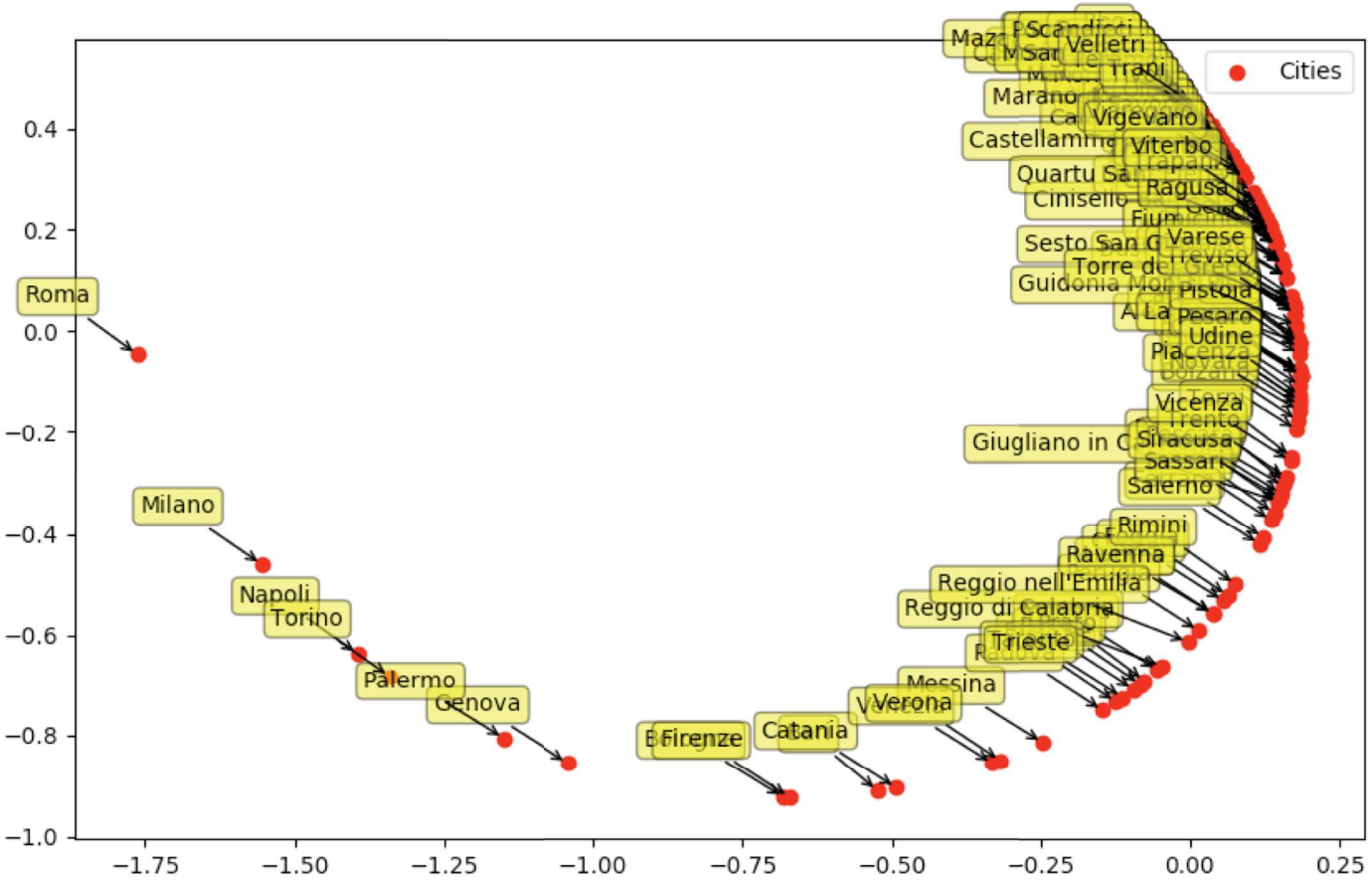

Roma
Cities
2500
2000
1500
1000
500
0
Milano
Napoli
Torino
Palermo
Genova
Firenze
Verona
Viterbo
0
200
400
600
800
1000

$|c_i - c_j|$

Other ideas? Let’s play with the data

	A	B	C
1	Acerra	60	
2	Acireale	53	
3	Afragola	65	
4	Agrigento	60	
5	Alessandria	94	
6	Altamura	70	
7	Ancona	101	
8	Andria	100	
9	Anzio	54	
10	Aprilia	73	
11	Arezzo	100	
12	Asti	76	
13	Avellino	55	
14	Aversa	53	
15	Bagheria	55	
16	Bari	326	
17	Barletta	95	
18	Battipaglia	51	
19	Benevento	60	
20	Bergamo	119	
21	Bisceglie	55	

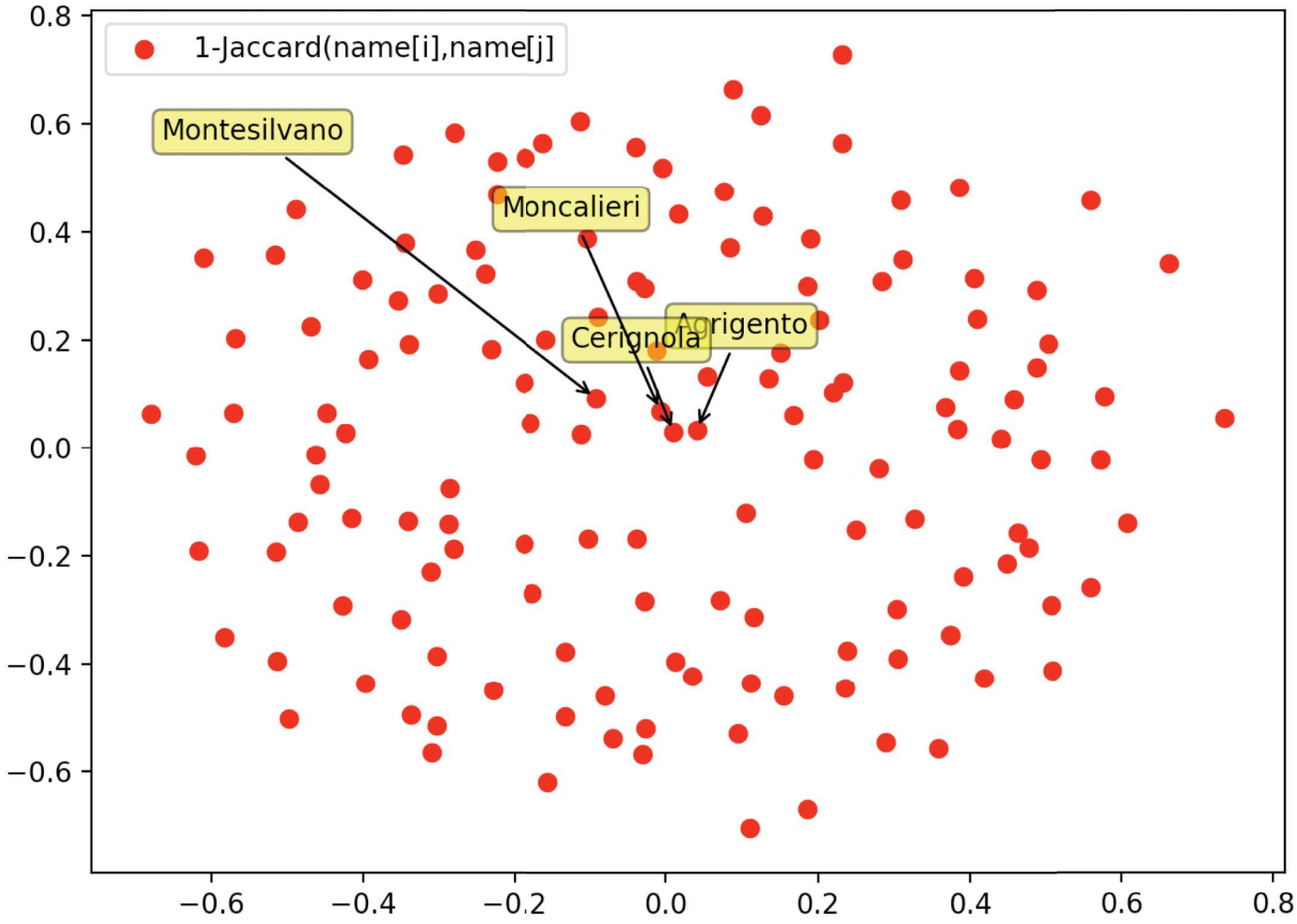
abs $[(d_i-d_j)/((d_i+d_j)/2)]$



Jaccard similarity

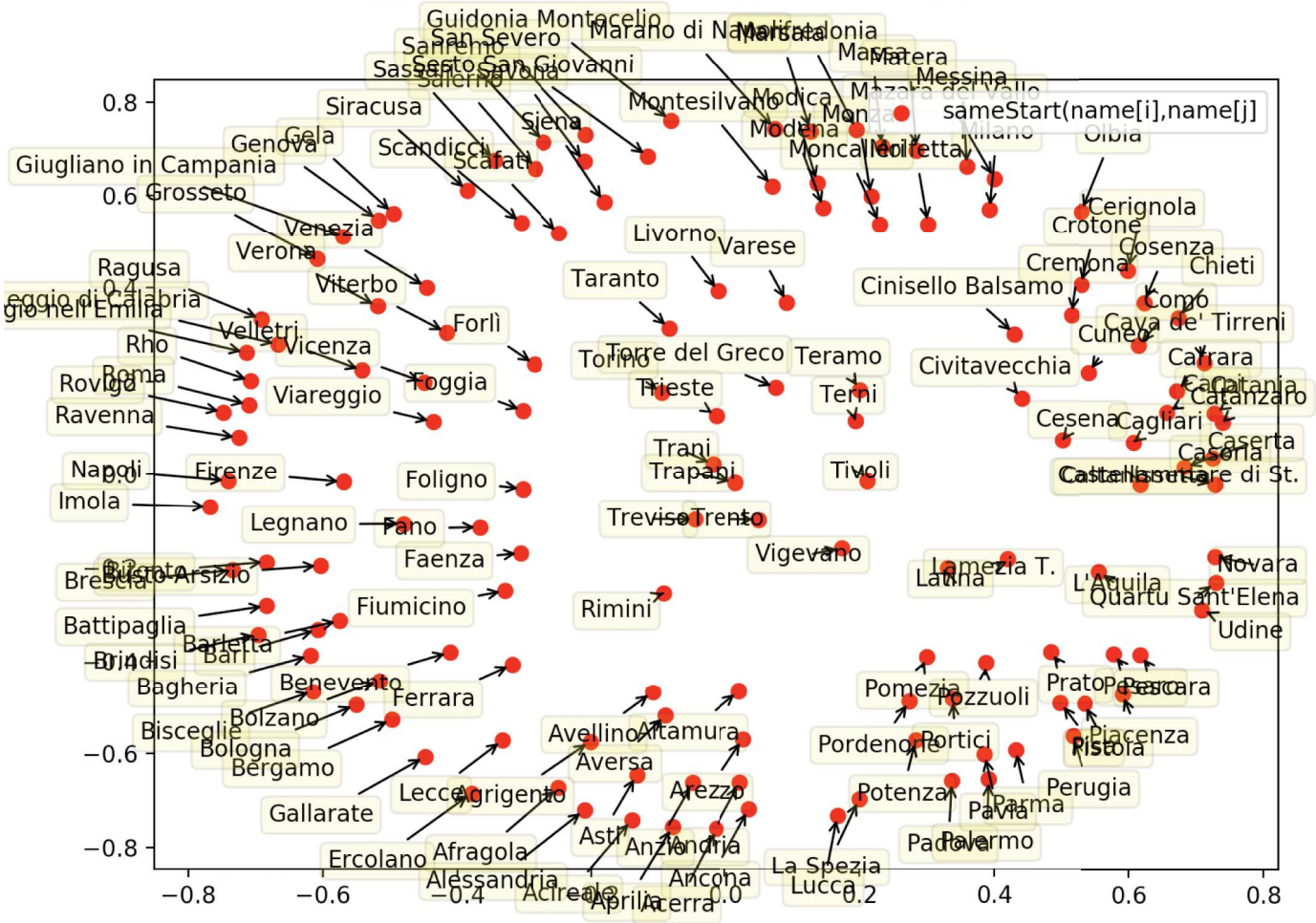
```
def jaccard(s1,s2):
    u=set(s1.lower()).union(set(s2.lower()))
    i=set(s1.lower()).intersection(set(s2.lower()))
    return(len(i)/len(u))
```

$$\frac{|S1 \cap S2|}{|S1 \cup S2|}$$



sameStart

```
def sameStart(s1,s2):
    tot=0
    i=0
    while i <( min(len(s1),len(s2))) and s1[i]==s2[i]:
            i+=1
```



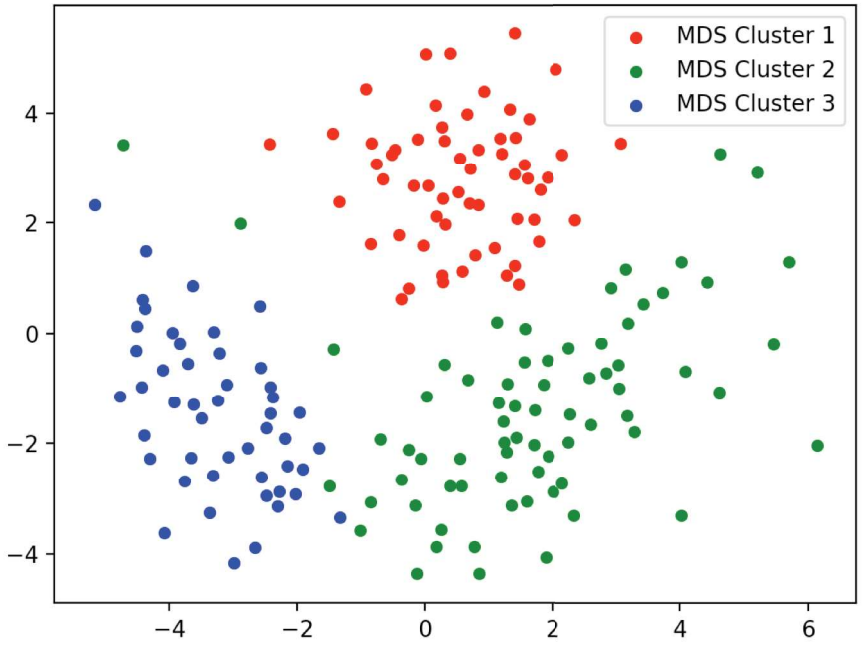
Another example on wines

(read data from csv and compute dissM using 12D Euclidean distances) 02_MDS_WineEuclidean.py

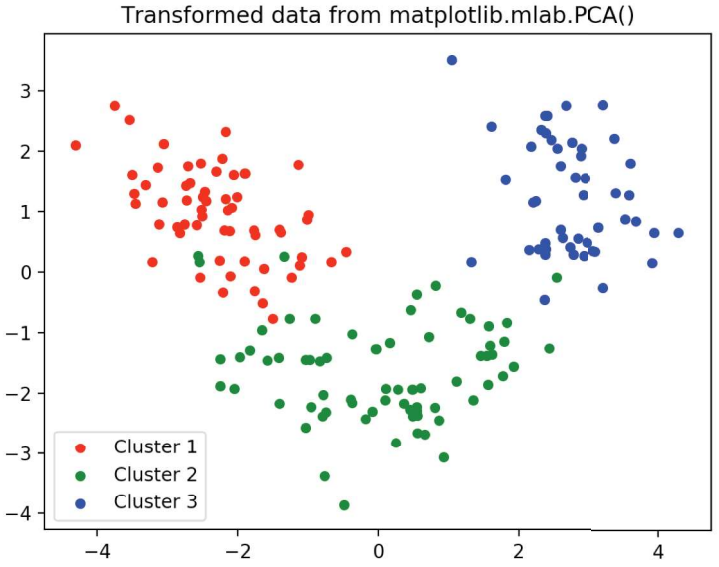
```
data = pd.io.parsers.read_csv(
     'wine.data.csv',
     header=None
    )
d=data.values[:,1:]
wine_class=data.values[:,0] #class values


#try to comment the following 2 lines to see how not standardized values affect MDS
std_scale = preprocessing.StandardScaler().fit(d)
d= std_scale.transform(d)

mds = manifold.MDS(n_components=2, dissimilarity="euclidean",random_state=2)
pos = mds.fit(d).embedding_

scatter(pos,wine_class,3)
plt.show()
```

Another example on wines

(computing and plotting MDS)



Similar to PCA



What is the strong difference?

PCS vs MDS based on Euclidean distance

- While the results look quite similar there are several aspect to consider:

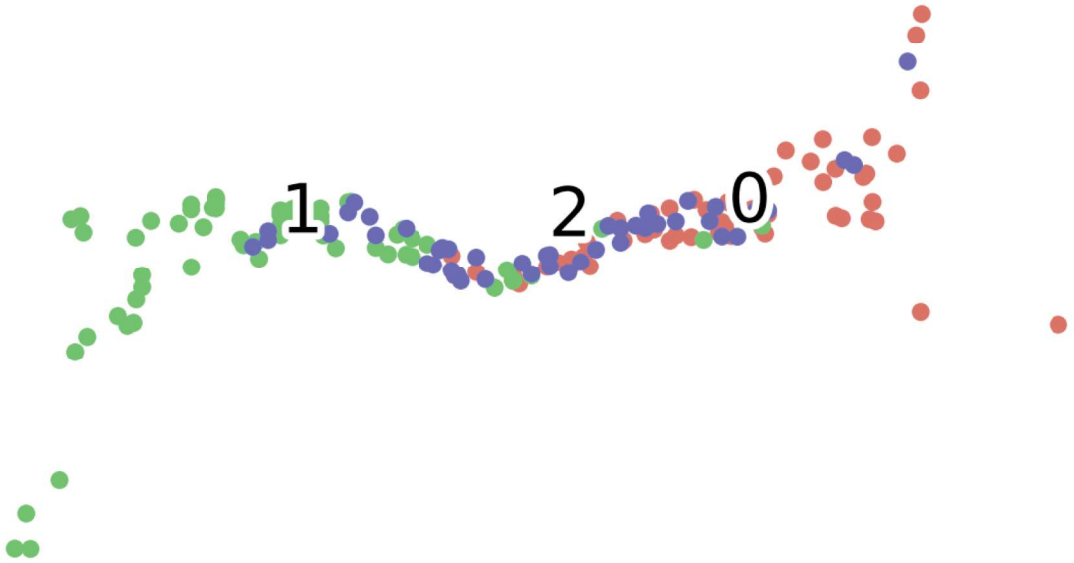
1. PCA plot is the result of a projection of a linear transformation of the original data
   - False positives (visualizing as close far points) are the result of the projection
   - False negatives (visualizing as far close points) do not exist
2. MDS plot is a result of adjusting data points from random 2D assignments to a situation that (approximately) minimizes an objective function based on actual point distances and original ones
   - False positives and negatives (yes they exists) are the result of the algorithm

Another (provocative) example on wines

(read data from csv and compute dissM using 1D Euclidean distances, i.e., alcohol %)

```
data = pd.io.parsers.read_csv(
     'wine.data.csv',
     header=None
    )
d=data.values[:,1:2]
y_wine=data.values[:,0]

mds = manifold.MDS(n_components=2, dissimilarity="euclidean",random_state=2)
pos = mds.fit(d).embedding_

scatter(pos,y_wine,3)
plt.show()
```

Another example on wines 1D→2D ☺!

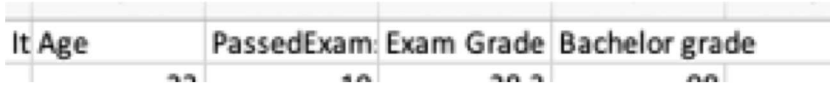


Students

S_ID	Italian Not It	Age	PassedExam	Exam Grade	Bachelor grade
1	Italian	23	10	28.3	98
2	Italian	23	3	27	95
3	Italian	23	3	27	109
4	Italian	22	4	29.7	111
5	Italian	23	0	0	95
6	Italian	23	2	29	94
12	Italian	23	3	30	110
14	Italian	22	4	30	111
15	Italian	22	4	29.5	111
16	Italian	23	4	30	111
17	Italian	22	4	29	111
18	Italian	22	4	29.6	111
19	Italian	22	4	30	111
20	Italian	22	4	29.6	111
21	Italian	22	3	30	111
22	Italian	22	1	28	111
24	Italian	26	4	0	93
25	Italian	22	0	23.5	102
26	Italian	23	1	23.5	106
27	Italian	22	4	30	111
7	Not Italian	26	0	0	95
8	Not Italian	25	3	29	96
9	Not Italian	25	0	0	68
10	Not Italian	28	0	0	0
11	Not Italian	23	0	0	110
13	Not Italian	22	1	28.5	72
23	Not Italian	23	9	28	96

Are Italian and
not Italian different?

Plot it using



4 dimensions -> 2D
Either PCA or MDS

t-SNE (t-distributed (Stochastic Neighbor Embedding)

- It is a nonlinear dimensionality reduction
- It is quite similar to MDS, in the sense that:
  - it uses a similarity between points in R^n
  - it arrange points on a 2D space, using this similarity
- It is quite different from Euclidean MDS, in the sense that:
  - the similarity is not linear and is not the Euclidean distance!

SNE (Stochastic Neighbor Embedding) similarity in R^n

- In **SNE** (without t) the high dimensional Euclidean distance $||x_i - x_j||$ between x_i and x_j in R^n is used to define a similarity between points using a conditional probability $p_{j|i}$:

$$p_{j|i} = \frac{\exp\left(-\|x_i - x_j\|^2 / 2\sigma_i^2\right)}{\sum_{k \neq i} \exp\left(-\|x_i - x_k\|^2 / 2\sigma_i^2\right)}$$

$$\frac{1}{\sigma\sqrt{2\pi}} e^{-\frac{(x-\mu)^2}{2\sigma^2}}$$

normal distribution

- where σ_i is the variance of a Gaussian centered on x_i and it is inversely proportional to the density around x_i
- $p_{j|i}$ is the probability that x_i would pick x_j as its neighbor if neighbors were picked in proportion to their probability density under a Gaussian centered at x_i
- $p_{j|i}$ is very high if points are very close and very little if the points are well separated
- σ_i is different for every point: points in dense areas are given a smaller variance than points in sparse areas
- Think at this as a definition of distance that is affected by local density: if you are considering a very dense area you have to be very close...

t-SNE similarity in R^2

- x_i and x_j are mapped in y_i and y_j in R^2 and, following the previous approach, the 2-dimensional Euclidean distance $||y_i-y_j||$ is converted in the conditional probability $q_{j|i}$:

$$q_{j|i} = \frac{\exp\left(-\|y_i - y_j\|^2\right)}{\sum_{k \neq i} \exp\left(-\|y_i - y_k\|^2\right)}$$

  - where the variance is constant and equal to 1/sqrt(2)
  - In an ideal mapping, $p_{j|i} == q_{j|i}$
  - The mapping proceeds using a simulated annealing-like technique (similar to MDS) that tries to minimize the sum of $| p_{j|i} - q_{j|i} |$

t-SNE ?

- t-SNE optimizes SNE
  - using a symmetric p_{ij}
  - using a Student-t distribution with a single degree of freedom in the low rather than a Gaussian

SNE

$$p_{j|i} = \frac{\exp\left(-\|x_i - x_j\|^2 / 2\sigma_i^2\right)}{\sum_{k \neq i} \exp\left(-\|x_i - x_k\|^2 / 2\sigma_i^2\right)}$$

$$q_{j|i} = \frac{\exp\left(-\|y_i - y_j\|^2\right)}{\sum_{k \neq i} \exp\left(-\|y_i - y_k\|^2\right)}$$

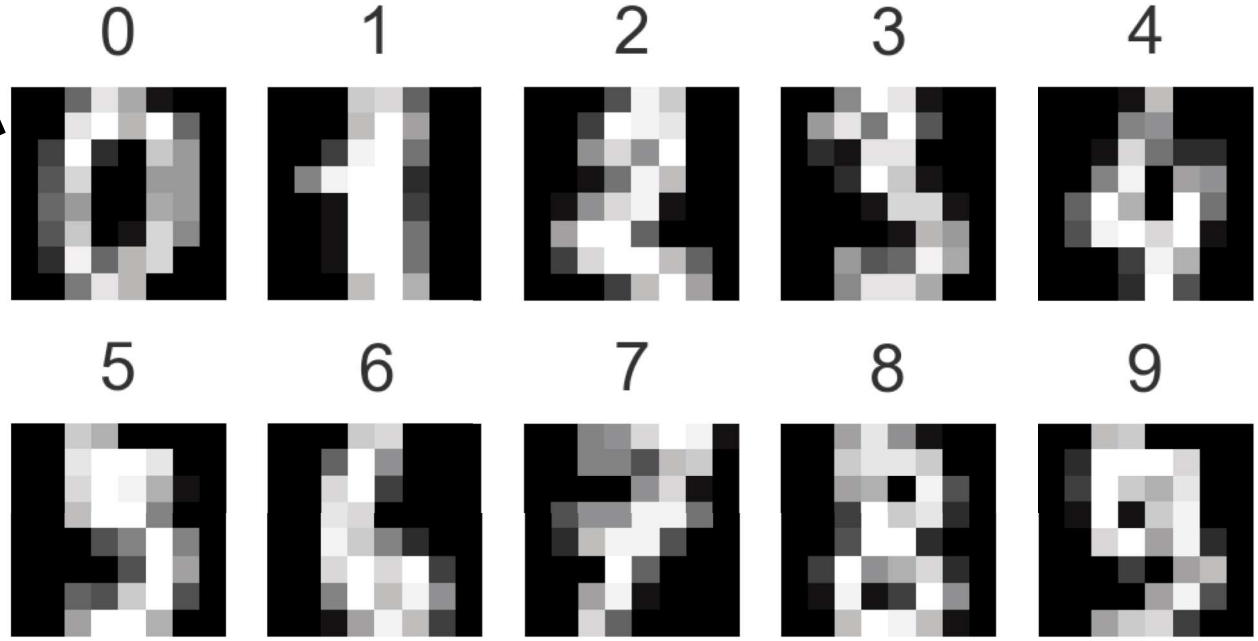
t-SNE

$$p_{ij} = \frac{p_{j|i} + p_{i|j}}{2N}$$

$$q_{ij} = \frac{\left(1 + \|y_i - y_j\|^2\right)^{-1}}{\sum_{k \neq l} \left(1 + \|y_k - y_l\|^2\right)^{-1}}.$$

Let's go with an example

- We use the Optical Recognition of Handwritten Digits Data Set (http://archiveicsuciedu/ml/datasets/Optical+Recognition+of+Handwritten+Digits)
- It has 64 attributes that corresponds to the gray scale of a 8x8 pixel image
- 0, 0, 5, 13, 9, 1, 0, 0, 0, 0, 13, 15, 10, 15, 5, 0, 0, 3, 15, 2, 0, 11, 8, 0, 0, 4, 12, 0, 0, 8, 8, 0, 0, 5, 8, 0, 0, 9, 8, 0, 0, 4, 11, 0, 1, 12, 7, 0, 0, 2, 14, 5, 10, 12, 0, 0, 0, 0, 6, 13, 10, 0, 0, 0



We compare MDS and t-SNE

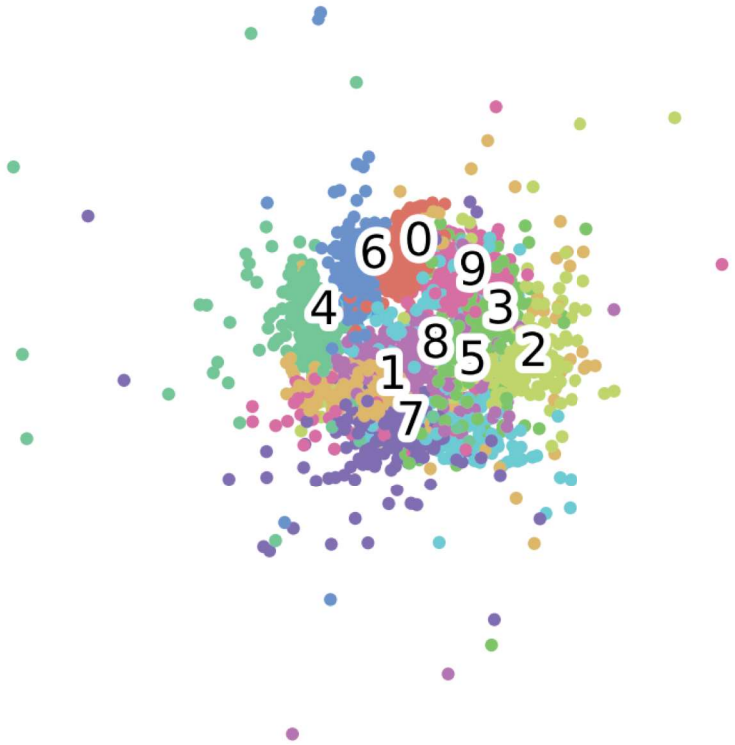
```
import numpy as np
import sklearn.datasets
from sklearn.manifold import TSNE
from sklearn.manifold import MDS
import matplotlib
import matplotlib.pyplot as plt
import matplotlib.patheffects as PathEffects
from sklearn import preprocessing
```

```
digits = sklearn.datasets.load_digits()
digits.data.shape
```

```
(1797, 64)
```

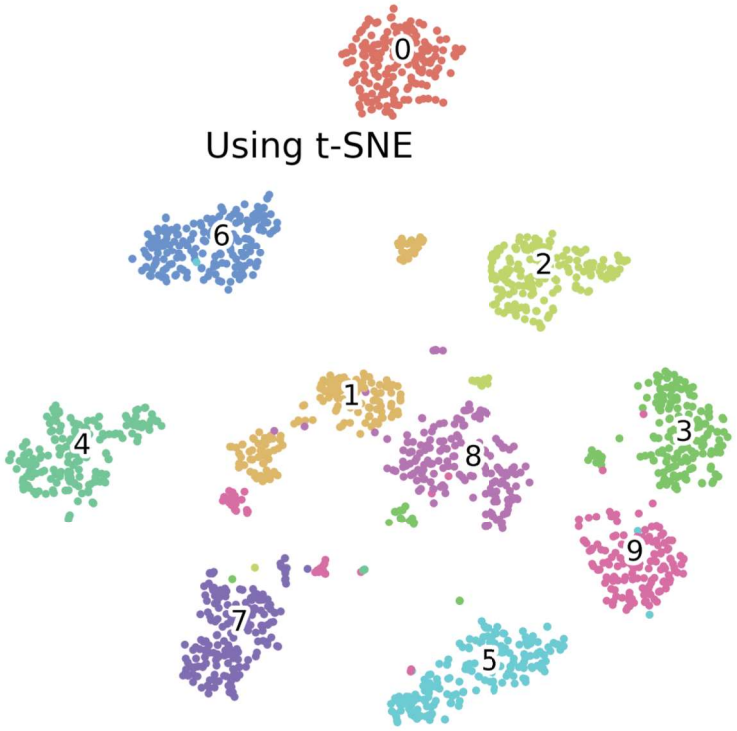
MDS

```
std_scale = preprocessing.StandardScaler().fit(X)
d= std_scale.transform(X)

mds = MDS(n_components=2, dissimilarity="euclidean",random_state=RS)
digits_proj_mds = mds.fit(d).embedding_

scatter(digits_proj_mds, classes,10,"Using MDS")
plt.show()
```



t-SNE

```
digits_proj_tsne = TSNE(random_state=RS).fit_transform(X)

scatter(digits_proj_tsne, classes,10,"Using t-SNE")
plt.show()
```



t-NSE

- Similar, as objective, to MSD
- It amplifies separations
- Useful for cluster identification
- It is based on similarities among points expressed as probabilities using distances

PCA

Linear transformation

Eigenvalues and eigenvectors

Preserve distances

Uses and give special emphasis to variance

Projection on 2 (3) axes

It introduces false positives

MDS

Any kind of transformation

Focuses on similarities

You can user your own definition of similarity

Points are arranged on the 2D space iterating and minimizing a stress function

It introduces false positives and false negatives

t-SNE

Non linear transformation

Focuses on similarities associated to distance that are rendered as conditional probabilities

It amplifies separations

Points are arranged on the 2D space iterating and minimizing a stress function

It introduces false positives and false negatives