Lecture 2:
Data analysis

Universitat de les Illes Balears | Departament de Ciències Matemàtiques i Informàtica

11752 Aprendizaje Automático
11752 Machine Learning
Máster Universitario
en Sistemas Inteligentes

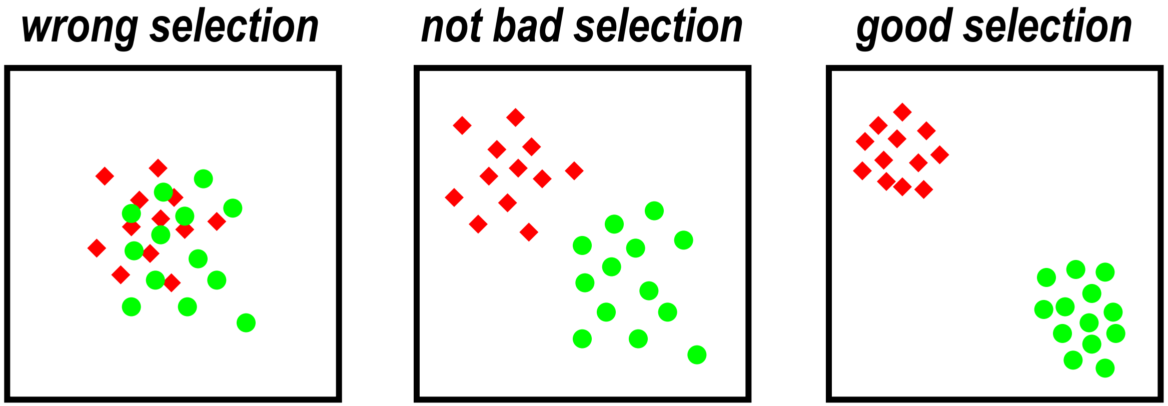
Alberto ORTIZ RODRÍGUEZ

Contents

- Introduction
- Data exploration (and first cleaning)
- Data preprocessing (incl. cleaning)
- Goodness measures
- Feature selection
- Dimensionality reduction
- Pipelines

Introduction

- **Data cleaning** but mostly **feature engineering**: select the **minimum set** of features that retain as much as possible the ability to discriminate among samples
  - **General criterion**
    - select those features that result in a **large between-class distance** and a **reduced variance between class elements** (within-class variance)



wrong selection

not bad selection

good selection

  - **Actions to do** to “engineer” the dataset:
    - understand your data, i.e. explore your data (maybe to know which is your case above)
    - pre-process/transform your data, to make things simpler for the next steps
    - examine features in isolation
    - examine features in combination
    - combine your features

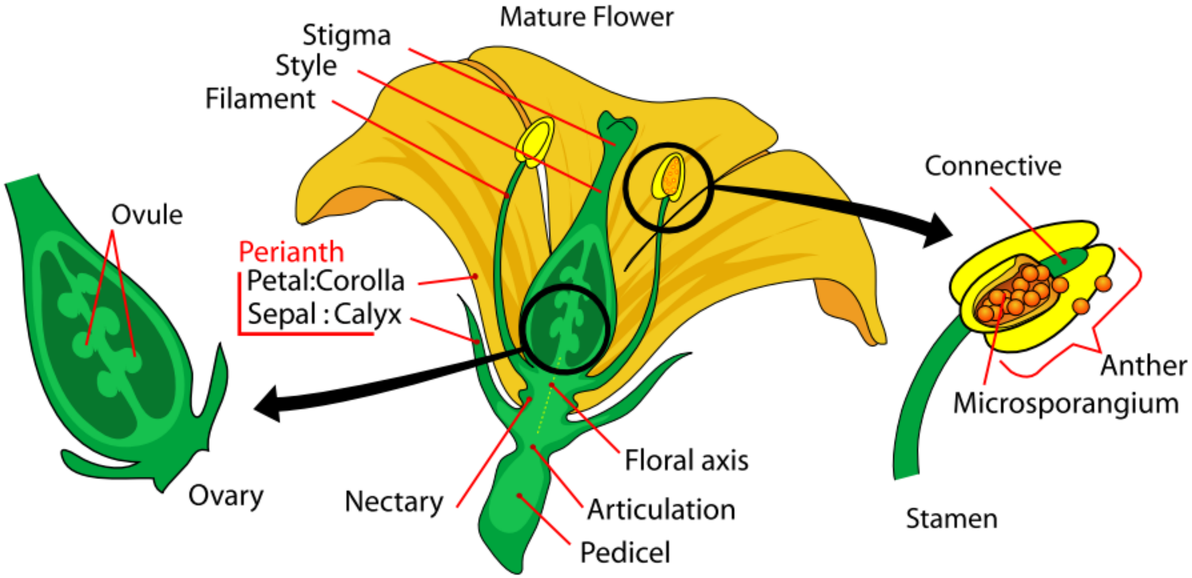
→ feature selection / dimensionality reduction

Contents

- Introduction
- Data exploration (and first cleaning)
- Data preprocessing (incl. cleaning)
- Goodness measures
- Feature selection
- Dimensionality reduction

Data exploration (and first cleaning)

- **Data exploration** is the first, basic step to understand your data
- This includes **data visualization** for **qualitative assessment,** detection of **anomalies**, **trends** and **relationships**, as well as to detect the necessity for **data cleaning**
- Let us consider the **Iris** flower dataset (Fisher's Iris data set)
  - multivariate dataset by the British statistician and biologist Ronald Fisher (1936)
  - 150 samples under four attributes:
    - sepal length
    - sepal width
    - petal length
    - petal width
  - 3 species:
    - setosa
    - versicolor
    - virginica



Alberto Ortiz (last update 13/10/2025)

Data exploration (and first cleaning)

- Basic **descriptive data**:

```
import numpy as np
from sklearn.datasets import load_iris
iris = load_iris()
X = iris.data
y = iris.target
nos, nod = X.shape
print('no. samples = %d, no. dimensions = %d' % (nos, nod))
noc = len(np.unique(y))
print('no. classes = %d' % (noc))
mu = np.mean(X, axis=0)
std = np.std(X, axis=0)
std2 = np.var(X, axis=0)
print('    mean std  var')
for i in range(nod):
    print('x%d: %.2f %.2f %.2f' % (i+1,mu[i],std[i],std2[i]))
```

```
no. samples = 150, no. dimensions = 4
no. classes = 3
    mean std  var
x1: 5.84 0.83 0.68
x2: 3.06 0.43 0.19
x3: 3.76 1.76 3.10
x4: 1.20 0.76 0.58
```

Data exploration (and first cleaning)

- Basic **descriptive data**:

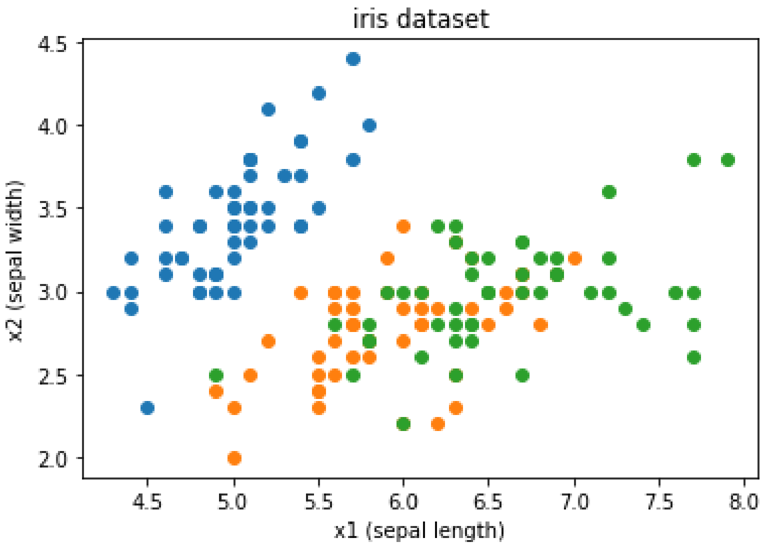
```
import numpy as np
from sklearn.datasets import load_iris
iris = load_iris()
print(iris.DESCR)
```

```
Iris plants dataset
--------------------

**Data Set Characteristics:**

    :Number of Instances: 150 (50 in each of three classes)
    :Number of Attributes: 4 numeric, predictive attributes and the class
    :Attribute Information:
        - sepal length in cm
        - sepal width in cm
        - petal length in cm
        - petal width in cm
        - class:
                - Iris-Setosa
                - Iris-Versicolour
                - Iris-Virginica

    :Summary Statistics:

    ============== ==== ==== ======= ===== ====================
                    Min  Max   Mean    SD   Class Correlation
    ============== ==== ==== ======= ===== ====================
    sepal length:   4.3  7.9   5.84   0.83    0.7826
    sepal width:    2.0  4.4   3.05   0.43   -0.4194
    petal length:   1.0  6.9   3.76   1.76    0.9490  (high!)
    petal width:    0.1  2.5   1.20   0.76    0.9565  (high!)
    ============== ==== ==== ======= ===== ====================

    :Missing Attribute Values: None
    :Class Distribution: 33.3% for each of 3 classes.
```

Data exploration (and first cleaning)

- Basic **visualization**:

```
import matplotlib.pyplot as plt
plt.figure()
for c in range(noc):
    i = np.where(y == c)[0]
    plt.scatter(X[i,0],X[i,1])
plt.xlabel('x1 (sepal length)')
plt.ylabel('x2 (sepal width)')
plt.title('iris dataset')
plt.show()
```



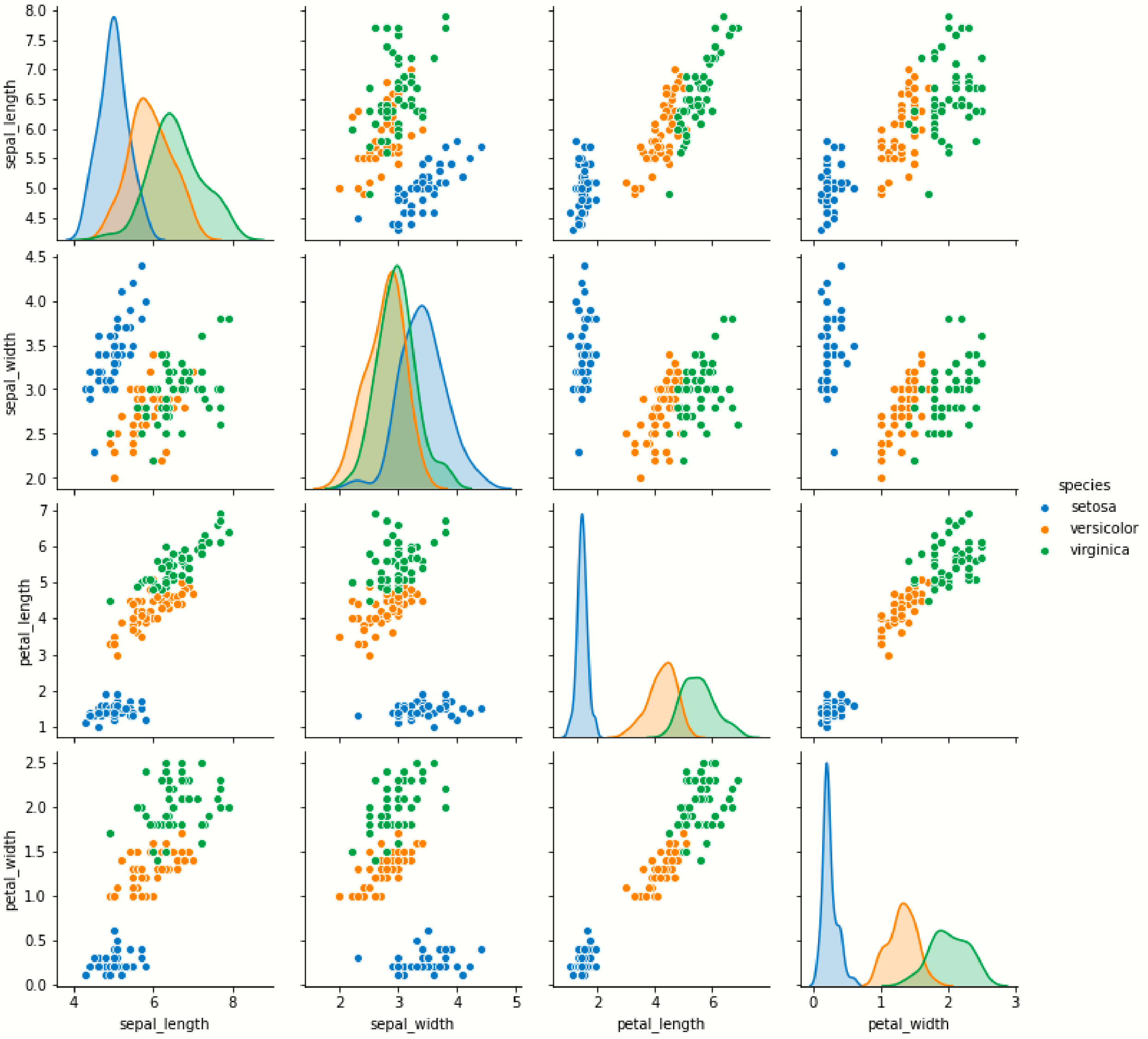
- For multidimensional datasets, i.e. more than 2 / 3 dimensions, the standard methods of visualization are not an option
  - Among many others:
    - the **Scatter Plot Matrix (SPLOM)** and
    - the **parallel coordinates plot**

  are alternative visualization tools, though of limited capability

Data exploration (and first cleaning)

- **Scatter PLOt Matrix**
  - Correlation plots & Histograms

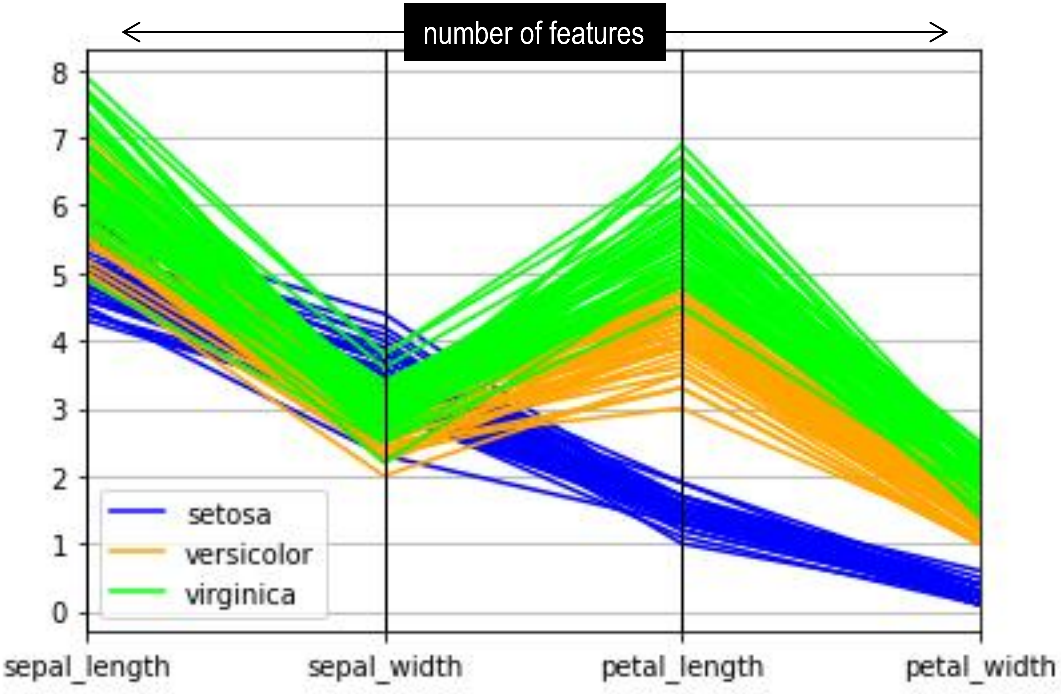
```
import seaborn as sb
df = sb.load_dataset('iris')
sb.pairplot(df, hue='species')
```



number of features

Data exploration (and first cleaning)

- **Parallel coordinates plot**


number of features

```
import matplotlib.pyplot as plt
import pandas as pd
import seaborn as sb
df = sb.load_dataset('iris')
pd.plotting.parallel_coordinates(df, 'species', color=('#0000FF', '#FFA500', '#00FF00'))
plt.legend(loc='lower left')
plt.show()
```

Data exploration (and first cleaning)

- **Pandas** is a library for data manipulation and analysis which can be useful for ML
  - The pandas **dataframe class** may be particularly useful for data manipulation and indexing, as well as for file input / output
- To create a *dataframe* we need an array of values.
  Moreover, we can add labels for the columns and for the samples:

```
import pandas as pd
df = pd.DataFrame([[1, 2], [4, 5], [7, 8]],
     index=['cobra', 'viper', 'sidewinder'],
     columns=['max_speed', 'shield'])
print(df.head())
```

```
            max_speed  shield
cobra               1       2
viper               4       5
sidewinder          7       8
```

- In *dataframes*, indexing can be very flexible with the **df.loc()** method:

```
print(df.loc[['viper', 'sidewinder']])
print(df.loc['cobra':'viper', 'max_speed'])
print(df.loc[df['shield'] > 4, ['max_speed']])
```

```
            max_speed  shield
viper               4       5
sidewinder          7       8

cobra    1
viper    4
Name: max_speed, dtype: int64

            max_speed
viper               4
sidewinder          7
```

Data exploration (and first cleaning)

- Let us use the *Titanic* dataset to illustrate other functionalities of *dataframes*:

```
import seaborn as sb
titanic = sb.load_dataset('titanic')
df = titanic
print(df.info())
print(df.head(3))
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 891 entries, 0 to 890
Data columns (total 15 columns):
 #   Column       Non-Null Count  Dtype
---  ------       --------------  -----
 0   survived     891 non-null    int64
 1   pclass       891 non-null    int64
 2   sex          891 non-null    object
 3   age          714 non-null    float64
 4   sibsp        891 non-null    int64
 5   parch        891 non-null    int64
 6   fare         891 non-null    float64
 7   embarked     889 non-null    object
 8   class        891 non-null    category
 9   who          891 non-null    object
 10  adult_male   891 non-null    bool
 11  deck         203 non-null    category
 12  embark_town  889 non-null    object
 13  alive        891 non-null    object
 14  alone        891 non-null    bool
dtypes: bool(2), category(2), float64(2),
int64(4), object(5)
memory usage: 80.7+ KB
```

```
  survived pclass    sex   age ... deck embark_town alive alone
0        0      3   male  22.0 ...  NaN Southampton    no False
1        1      1 female  38.0 ...    C   Cherbourg   yes False
2        1      3 female  26.0 ...  NaN Southampton   yes  True
```

  - **df.tail(n)** displays the last *n* samples

- We can also load the dataset from disk. Let us assume the dataset is in file *titanic.csv*:

```
df = pd.read_csv('titanic.csv')
print(df.info())
```

  - Other formats also available for input/output, e.g. JSON, excel, etc.

Data exploration (and first cleaning)

- The **df.describe()** method provides a summary of the dataset statistics:

```
print(df.describe())
```

```
         survived      pclass         age       sibsp       parch        fare
count  891.000000  891.000000  714.000000  891.000000  891.000000  891.000000
mean     0.383838    2.308642   29.699118    0.523008    0.381594   32.204208
std      0.486592    0.836071   14.526497    1.102743    0.806057   49.693429
min      0.000000    1.000000    0.420000    0.000000    0.000000    0.000000
25%      0.000000    2.000000   20.125000    0.000000    0.000000    7.910400
50%      0.000000    3.000000   28.000000    0.000000    0.000000   14.454200
75%      1.000000    3.000000   38.000000    1.000000    0.000000   31.000000
max      1.000000    3.000000   80.000000    8.000000    6.000000  512.329200
```

- For selecting elements of the dataset, one can additionally use column labels and the **df.iloc()** method:

```
X1  = df.iloc[:,[1,2,3,4,5,6]].to_numpy()
X2  =
df[['pclass','sex','age','sibsp','parch','fare']]
y   = df['survived']
print(X1[0:3,:])
print(X2.head(3))
```

```
[[3 'male'   22.0 1 0 7.25]
 [1 'female' 38.0 1 0 71.2833]
 [3 'female' 26.0 0 0 7.925]]

  pclass     sex  age sibsp parch    fare
0      3    male 22.0     1     0  7.2500
1      1  female 38.0     1     0 71.2833
2      3  female 26.0     0     0  7.9250
```

Data exploration (and first cleaning)

- **Conditions** can also be used for selecting samples:

```
print(df[df['deck'] == 'C'].head(3))
```

```
urvived pclass     sex  age ... deck embark_town alive alone
1         1         1 female 38.0 ...    C   Cherbourg   yes False
3         1         1 female 35.0 ...    C Southampton   yes False
11        1         1 female 58.0 ...    C Southampton   yes  True
```

```
print(df[(df['age'] > 50) & (df['pclass'] < 2)].head(3))
```

```
 survived pclass     sex  age ... deck embark_town alive alone
6         0         1   male 54.0 ...    E Southampton    no  True
11        1         1 female 58.0 ...    C Southampton   yes  True
54        0         1   male 65.0 ...    B   Cherbourg    no False
```

- The *dataframe* object provides a number of ways to get more details of the dataset:
  - The **df.columns** attribute is a list with the labels of the dataset columns
  - **df.values()** or **df.to_numpy()** provide the dataset values as a numpy array
  - **df.count_values()** returns the number of times the different values occur in a column
  - With **df.nunique()** we can see the counts of unique values in each column

```
print(df[['age','deck']].nunique())
print(df['sex'].value_counts())
```

```
age        88
deck        7

male        577
female      314
Name: sex, dtype: int64
```

Data exploration (and first cleaning)

- We can **remove some features** (columns) which are useless:

```
udf = df
udf.drop('embarked',axis=1,inplace=True)
udf.drop('class',axis=1,inplace=True)
udf.drop('who',axis=1,inplace=True)
udf.drop('adult_male',axis=1,inplace=True)
udf.drop('deck',axis=1,inplace=True)
udf.drop('embark_town',axis=1,inplace=True)
udf.drop('alive',axis=1,inplace=True)
udf.drop('alone',axis=1,inplace=True)
print(udf.info())
```

```
RangeIndex: 891 entries, 0 to 890
Data columns (total 7 columns):
 #   Column    Non-Null Count  Dtype
---  ------    --------------  -----
 0   survived  891 non-null    int64
 1   pclass    891 non-null    int64
 2   sex       891 non-null    object
 3   age       714 non-null    float64
 4   sibsp     891 non-null    int64
 5   parch     891 non-null    int64
 6   fare      891 non-null    float64
```

- We can as well **drop duplicates**, if any:

```
import pandas as pd
df = pd.DataFrame({
    'brand': ['Yum','Yum','Indo','Indo','Indo'],
    'style': ['cup','cup','cup','pack','pack'],
    'rating': [4, 4, 3.5, 15, 5]
})
print(df)
print(df.drop_duplicates())
print(df.drop_duplicates(subset='brand'))
```

```
  brand style  rating
0   Yum   cup     4.0
1   Yum   cup     4.0
2  Indo   cup     3.5
3  Indo  pack    15.0
4  Indo  pack     5.0

  brand style  rating
0   Yum   cup     4.0
2  Indo   cup     3.5
3  Indo  pack    15.0
4  Indo  pack     5.0

  brand style  rating
0   Yum   cup     4.0
2  Indo   cup     3.5
```

Contents

- Introduction
- Data exploration (and first cleaning)
- Data preprocessing (incl. cleaning)
- Goodness measures
- Feature selection
- Dimensionality reduction
- Pipelines

Data preprocessing (incl. cleaning)

- Preparation of data samples before proceeding to their use
  - **Handling categorical data**
  - **Outlier detection (and removal)**
  - **Data normalization / standardization**
  - **Filling in missing data**

Data preprocessing: Categorical data

- **Handling categorical data**
  - Categorical data must be converted to **numeric values** before learning
    - The **LabelEncoder()** object assigns a **progressive integer label** to every class label

```
import seaborn as sb
from sklearn.preprocessing import LabelEncoder
titanic = sb.load_dataset('titanic')
df = titanic
print(df['sex'][:5])
le = LabelEncoder()
df['sex'] = le.fit_transform(df['sex'])
print(df['sex'][:5])
print(le.classes_)
```

```
0      male
1    female
2    female
3    female
4      male
Name: Sex, dtype: object
0    1
1    0
2    0
3    0
4    1
Name: Sex, dtype: int32
['female' 'male']
```

    - The names of the classes are in attribute *le.classes_*

  - Unfortunately, on some occasions, this is not a good encoding for training, and ***one-hot encoding*** must be used instead:

```
from sklearn.preprocessing import OneHotEncoder
df = titanic
ohe = OneHotEncoder()
data = np.expand_dims(df['sex'], axis=-1)
ohe.fit(data)
data_ = ohe.transform(data).toarray()
print(data_[:5])
```

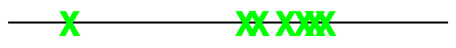
```
[[0. 1.]
 [1. 0.]
 [1. 0.]
 [1. 0.]
 [0. 1.]]
```

Data preprocessing: Outlier detection

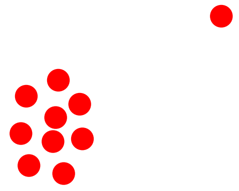
- **Outlier detection (and removal)**
  - ***outlier*** ≡ sample that does not agree with the rest of the population

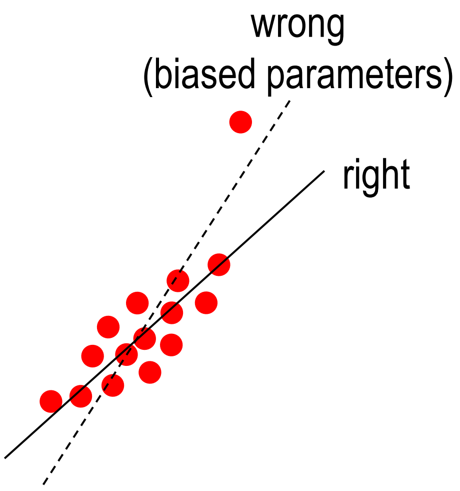
  - normally, distance to the mean is $k\sigma, k\uparrow\uparrow$
  - an outlier can distort training
    - the resulting classifier / regressor may not classify / predict for new samples in the right way

e.g. sonar readings (case 1D)



e.g. case 2D





Data preprocessing: Outlier detection

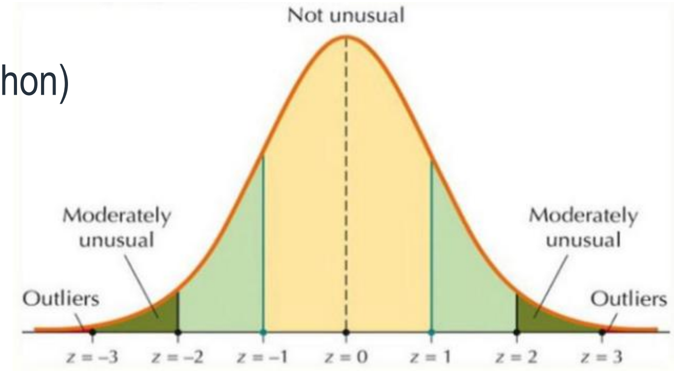
- **Sources of outliers**:
  - measurement error (instrument error or noise) or experimental error (wrong data extraction)
  - data entry error (data collection/typing) or data processing error

- If you need to counteract the outliers, these are some of the **possible actions**:
  - **discard** the outliers (= full sample) if the dataset permits to do so, i.e. it is big enough
  - alter the data:
    - **trimming**: extreme values are set to “missing”, i.e. NaN (Python)
    - **winsorization**: replace values at the higher and lower ends of the distribution with specific lower and upper values

Winsorized mean. After *sorting* the data, we replace x_1 and x_{10} by resp. x_2 and x_9

$$\frac{\overbrace{x_2 + x_2} + x_3 + x_4 + x_5 + x_6 + x_7 + x_8 + \overbrace{x_9 + x_9}}{10}$$

(20% winsorized mean)



1, 5, 7, 8, 9, 10, 10, 12, 12, 34 → μ = 10.8

5, 5, 7, 8, 9, 10, 10, 12, 12, 12 → μ = 9.0

scipy.stats.mstats.winsorize()

  - **tolerate** the outliers by reducing their influence
    - use optimization methods from **robust statistics** (large values are attenuated “on-line”)

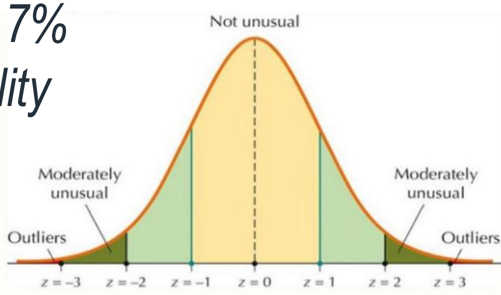
Data preprocessing: Outlier detection

- **z-score method (Gausssian data)**: $\mu \pm 3\sigma$ *accumulates 99.7% of the probability*

```
import numpy as np
from sklearn.datasets import load_wine
wine = load_wine()
X = wine.data
y = wine.target
print(wine.DESCR)

outl = np.zeros((3,13))
for c in range(3):
    for f in range(13):
        cc = X[y == c, f]
        mu, sg = np.mean(cc), np.std(cc)
        cut_off = sg * 3
        lower, upper = mu - cut_off, mu + cut_off
        # identify outliers
        outliers = [x for x in cc if x < lower or x > upper]
        nout = len(outliers)
        # non-outliers
        non_outliers = [x for x in cc if x >= lower and x <= upper]
        nok = len(non_outliers)
        outl[c,f] = nout
print(outl)
```

```
from scipy import stats
z = np.abs(stats.zscore(X))
# discard samples with z > 3
```

$$z = \frac{x - \mu}{\sigma}$$



```
============================= ==== ===== ======= =====
                               Min   Max   Mean     SD
============================= ==== ===== ======= =====
Alcohol:                      11.0  14.8    13.0   0.8
Malic Acid:                   0.74  5.80    2.34  1.12
Ash:                          1.36  3.23    2.36  0.27
Alcalinity of Ash:            10.6  30.0    19.5   3.3
Magnesium:                    70.0 162.0    99.7  14.3
Total Phenols:                0.98  3.88    2.29  0.63
Flavanoids:                   0.34  5.08    2.03  1.00
Nonflavanoid Phenols:         0.13  0.66    0.36  0.12
Proanthocyanins:              0.41  3.58    1.59  0.57
Colour Intensity:              1.3  13.0     5.1   2.3
Hue:                          0.48  1.71    0.96  0.23
OD280/OD315 of diluted wines: 1.27  4.00    2.61  0.71
Proline:                       278  1680     746   315
============================= ==== ===== ======= =====

:Missing Attribute Values: None
:Class Distribution: class_0 (59), class_1 (71), class_2 (48)
```

class	f1	f2	f3	f4	f5	f6	f7	f8	f9	f10	f11	f12	f13
1	0	0	1	1	0	2	0	1	0	0	0	0	0
2	0	1	1	0	2	0	1	0	1	1	1	0	0
3	0	0	0	0	0	1	0	0	1	0	0	0	0

With this code, we know that there are outliers in all classes, but we should discover which samples are affected !!

and next do trimming or winsorizing

Data preprocessing: Outlier detection

- **Inter-quartile range method (non-Gausssian data)**

```
import numpy as np
from sklearn.datasets import load_wine
wine = load_wine()
X = wine.data
y = wine.target

from numpy import percentile
outl = np.zeros((3,13))
for c in range(3):
    for f in range(13):
        cc = X[y == c, f]
        q25, q75 = percentile(cc, 25), percentile(cc, 75)
        iqr = q75 - q25
        cut_off = iqr * 1.5
        lower, upper = q25 - cut_off, q75 + cut_off
        # identify outliers
        outliers = [x for x in cc if x < lower or x > upper]
        nout = len(outliers)
        # non-outliers
        non_outliers = [x for x in cc if x >= lower and x <= upper]
        nok = len(non_outliers)
        outl[c,f] = nout
print(outl)
```

– IQR = difference between the 75th and the 25th percentiles of the data (Q3, Q1)

– Situates outliers out of the $\pm k \times \mathrm{IQR}$ interval

class	f1	f2	f3	f4	f5	f6	f7	f8	f9	f10	f11	f12	f13
1	0	9	1	3	0	2	0	4	4	1	0	0	0
2	3	7	2	4	5	0	1	0	8	4	1	0	1
3	0	0	0	0	0	2	1	1	2	0	0	2	0

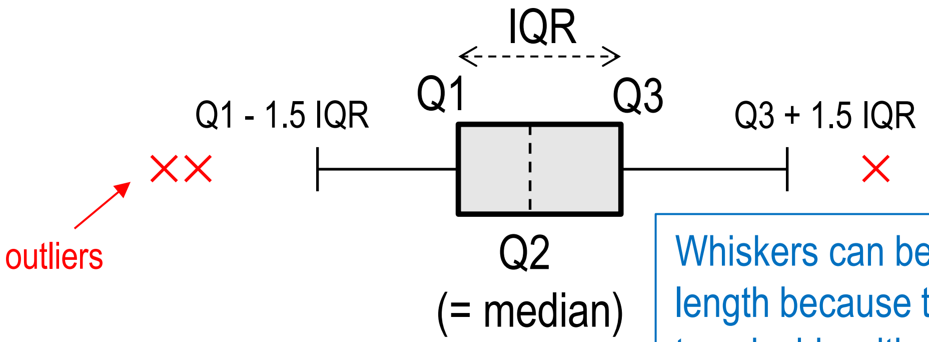
With this code, we know that there are outliers in all classes, but we should discover which samples are affected !!

and next do trimming or winsorizing

Data preprocessing: Outlier detection

- **Box-plots**

```
import matplotlib.pyplot as plt
import seaborn as sb          # also in matplotlib
df = sb.load_dataset('iris')
plt.figure()
sb.boxplot(y=df['species'], x=df['sepal_length'])
plt.show()
plt.figure()
sb.boxplot(y=df['species'], x=df['sepal_width'])
plt.show()
plt.figure()
sb.boxplot(y=df['species'], x=df['petal_length'])
plt.show()
plt.figure()
sb.boxplot(y=df['species'], x=df['petal_width'])
plt.show()
```



IQR

Q1

Q3

Q1 - 1.5 IQR

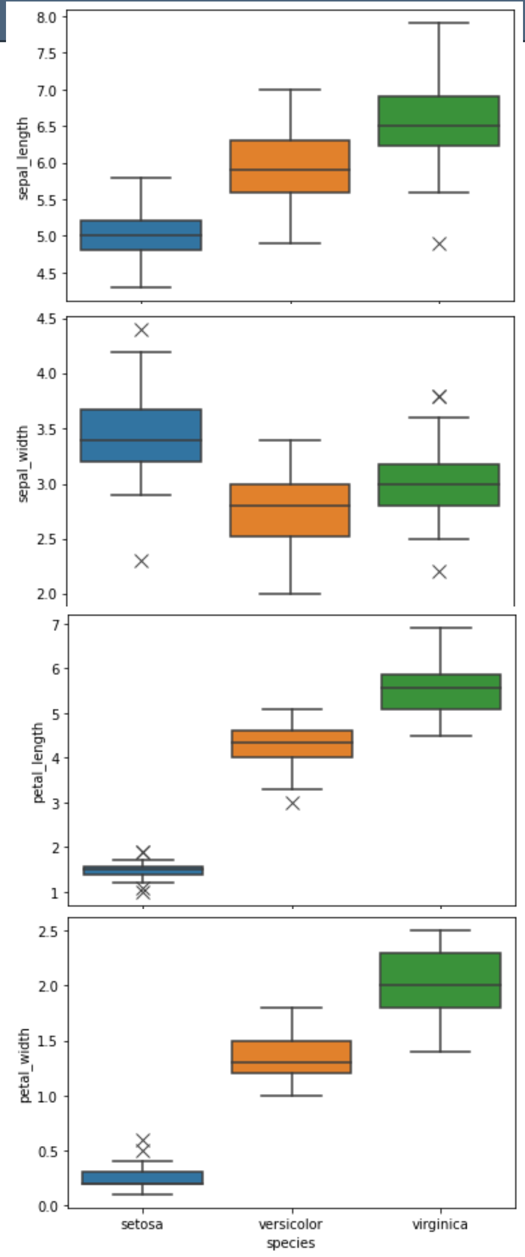
Q3 + 1.5 IQR

outliers

Q2

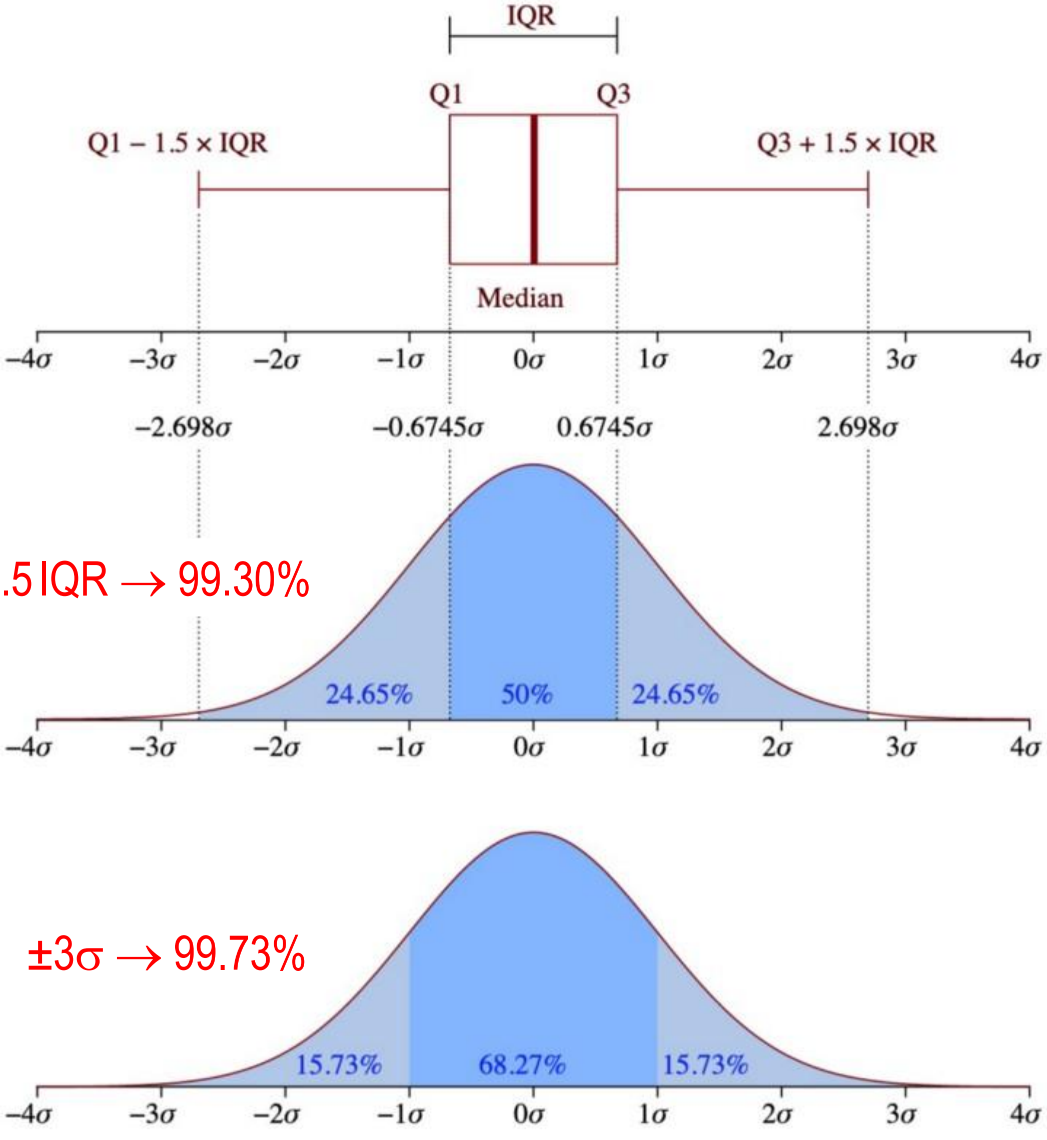
(= median)

Whiskers can be of different length because they are made to coincide with a sample



Data preprocessing: Outlier detection

- **Box-plots**
  - Why 1.5IQR?
    - Related with the 68–95–99 rule from the Gaussian distribution
    - In the Gaussian distribution, ±1.5IQR covers approx. the same probability as ±3σ



±1.5 IQR → 99.30%

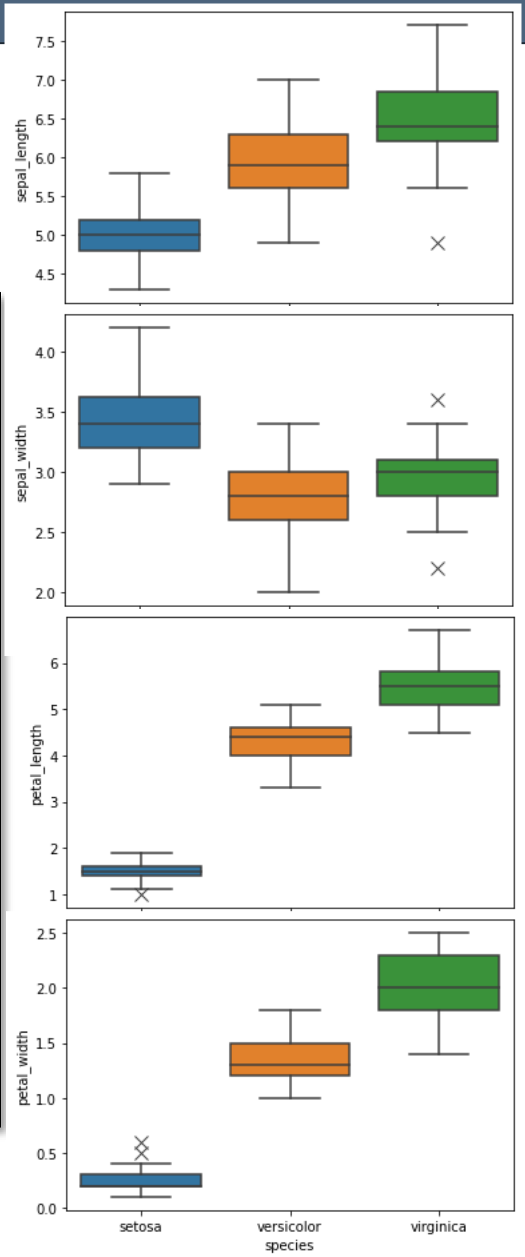
±3σ → 99.73%

Data preprocessing: Outlier detection

- **Automatic detection of outliers**
  - **Local Outlier Factor (LOF):** measures the local deviation of the density of a sample with respect to its neighbors

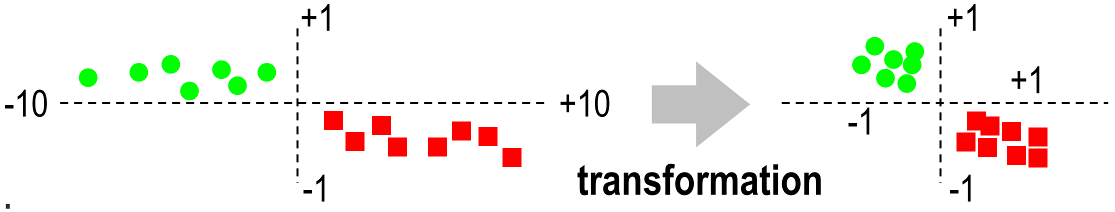
```
import numpy as np
import pandas as pd
from sklearn.neighbors import LocalOutlierFactor
import matplotlib.pyplot as plt
import seaborn as sb
df = sb.load_dataset('iris')
data = df.values
X = data[:,:-1]
y = data[:,-1]
# identify outliers in the training dataset
lof = LocalOutlierFactor(n_neighbors=20)
yhat = lof.fit_predict(X)
# select all rows that are not outliers
mask = yhat != -1
X, y = X[mask, :], y[mask]
y = np.expand_dims(y,axis=1)
df2 = pd.DataFrame(np.hstack((X,y)))
df2.columns = df.columns
```

  - Others: **IsolationForest**, etc.



Data preprocessing (incl. cleaning)

- **Data normalization / standardization**
  - Often, different features do not have the same **dynamic range** (range of values)
    - characteristics with wider ranges will have **more influence on the classification** regardless of whether they are more relevant to the design of the classifier or not



  - Solution:
    - normalize/scale features so that their dynamic ranges are similar
      - **linear scaling**
        - mu-sigma normalization (standardization)
        - max-min normalization
        - others
      - **non-linear scaling**
        - *softmax* normalization
        - others

Data preprocessing: Normalization

- **μ-σ (mu-sigma) normalization**
  - Given L – feature descriptors:

$$\forall k = 1, \ldots, L, \quad \overline{x}_k = \frac{\sum_i x_{ik}}{N}$$

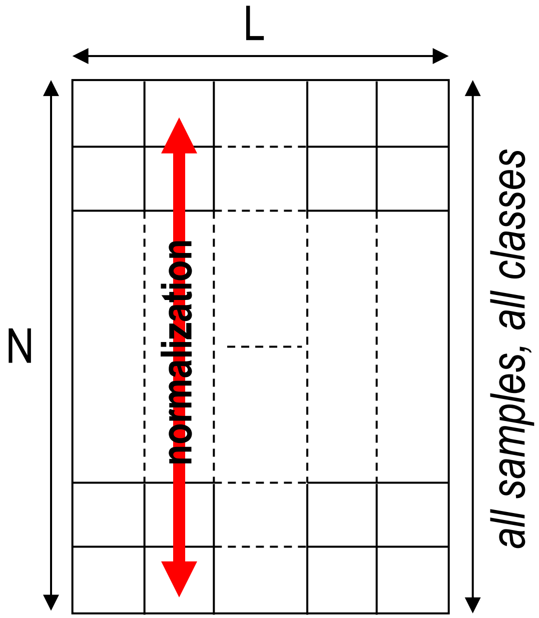
$$\sigma_k^2 = \frac{\sum_i (x_{ik} - \overline{x}_k)^2}{N - 1}$$

$$\widehat{x}_{ik} = \frac{x_{ik} - \overline{x}_k}{\sigma_k}$$

  - After the transformation:

$$\mathrm{E}[\widehat{x}_{ik}] = 0, \quad \mathrm{Var}[\widehat{x}_{ik}] = 1$$

$$\boxed{\begin{array}{lll} x_{ik} - \bar{x}_k = 0 & \Rightarrow & \hat{x}_{ik} = 0 \\ x_{ik} - \bar{x}_k = +k\sigma_k & \Rightarrow & \hat{x}_{ik} = +k \\ x_{ik} - \bar{x}_k = -k\sigma_k & \Rightarrow & \hat{x}_{ik} = -k \end{array}}$$



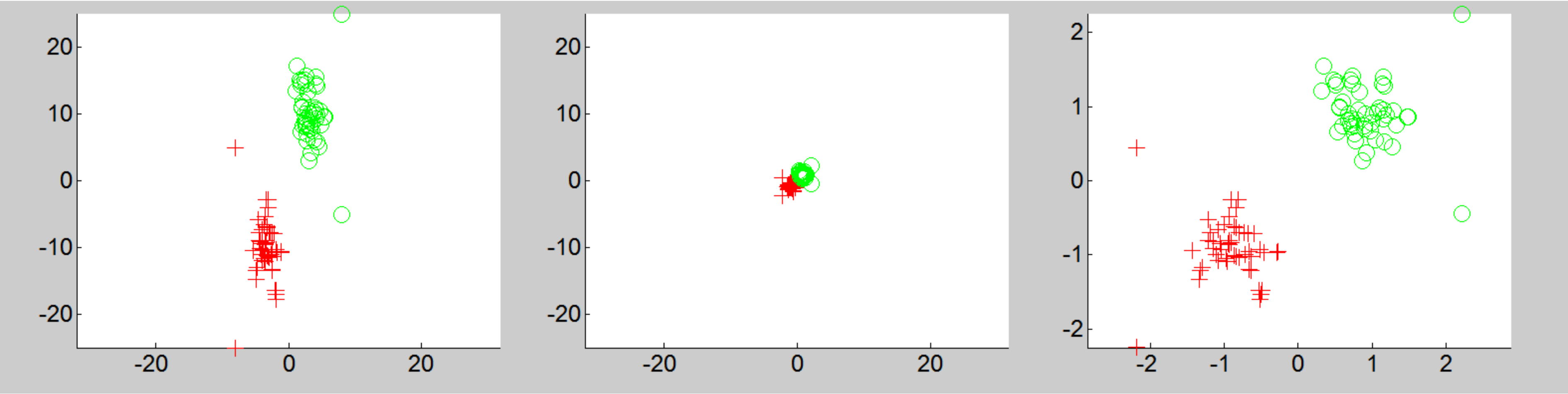
VERY IMPORTANT

Apply the transformation to the full dataset once.

- ❖ **Typical mistake**. Standardize the training set separately from the test set.
- ❖ Keep the transformation parameters $(\bar{x}, \sigma)$ to standardize new samples

Data preprocessing: Normalization

- **μ-σ (mu-sigma) normalization**



`normalization: mu-sigma`

`k=1 org:  -8.00 -   8.00 :  16.00` ← dynamic range of x_1

`k=2 org: -25.00 -  25.00 :  50.00` ← dynamic range of x_2

`ratio   :   3.13`

`k=1 nor:  -2.18 -   2.21 :   4.38` ← dynamic range of $\hat{x}_1$

`k=2 nor:  -2.25 -   2.25 :   4.50` ← dynamic range of $\hat{x}_2$

`ratio   :   1.03`

Data preprocessing: Normalization

- **Max-min normalization**
  - Given L – feature descriptors:

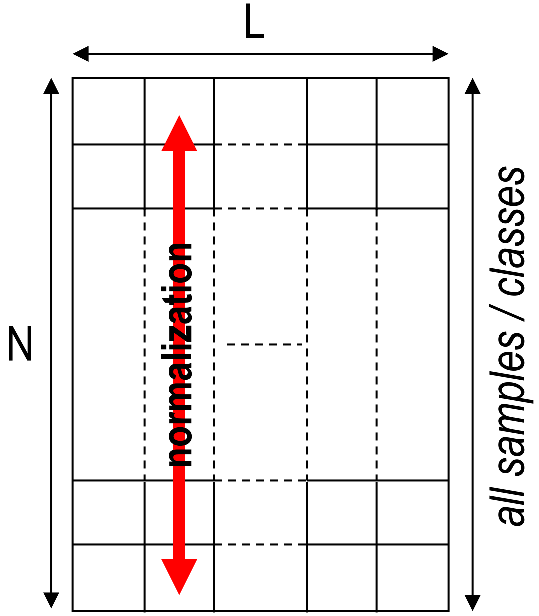
$$\forall k = 1, \ldots, L, \quad X_k = \max_i \{x_{ik}\}$$
$$x_k = \min_i \{x_{ik}\}$$
$$\widehat{x}_{ik} = \frac{x_{ik} - x_k}{X_k - x_k}$$

  - After the transformation: $\widehat{x}_{ik} \in [0, 1]$

$$\boxed{\begin{array}{l} x_{ik} = \max_i \{x_{ik}\} \Rightarrow \widehat{x}_{ik} = 1 \\ x_{ik} = \min_i \{x_{ik}\} \Rightarrow \widehat{x}_{ik} = 0 \end{array}}$$

... distributes the data within the range [0,1]

- the original ends correspond to 0 and 1
- e.g. if originally the range of values was [-30, 100], after normalization, value -30 will become 0 for that feature, while a value of 100 will become 1

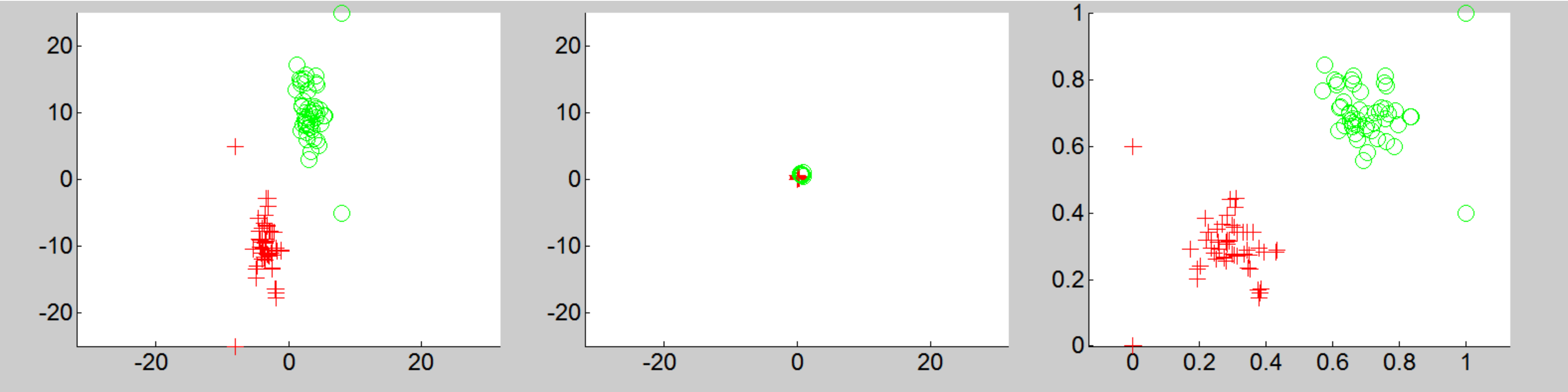


VERY IMPORTANT:
Apply the transformation to the full dataset once.

❖ Keep the transformation parameters (x, X) to normalize new samples

Data preprocessing: Normalization

- **Max-min normalization**



`normalization: min-max`

`k=1 org:   -8.00 -    8.00 :   16.00` ← dynamic range of x_1

`k=2 org:  -25.00 -   25.00 :   50.00` ← dynamic range of x_2

`ratio   :    3.13`

`k=1 nor:    0.00 -    1.00 :    1.00` ← dynamic range of $\hat{x}_1$

`k=2 nor:    0.00 -    1.00 :    1.00` ← dynamic range of $\hat{x}_2$

`ratio   :    1.00`

Data preprocessing: Normalization

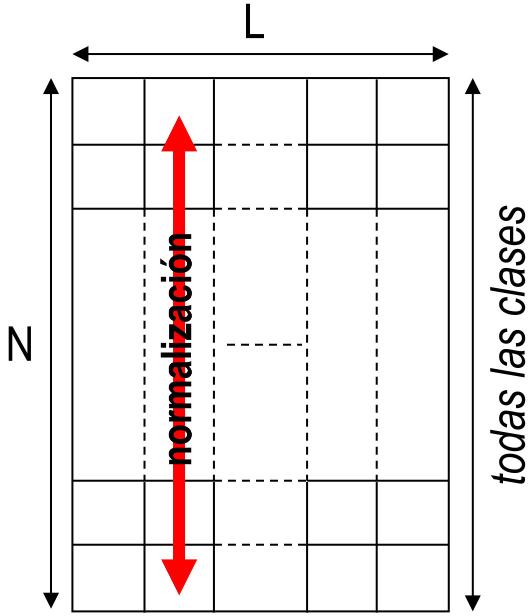
- ***Softmax* normalization** (non-linear transformation)
  - Given L – feature descriptors:

$$\forall k = 1, \ldots, L, \quad \overline{x}_k = \frac{\sum_i x_{ik}}{N}$$

$$\sigma_k^2 = \frac{\sum_i (x_{ik} - \overline{x}_k)^2}{N-1}$$

$$z_{ik} = \frac{x_{ik} - \overline{x}_k}{r\sigma_k}$$

$$\widehat{x}_{ik} = \frac{1}{1 + e^{-z_{ik}}}$$



  - After the transformation:

$$\widehat{x}_{ik} \in [0,1]$$

$$\boxed{\begin{array}{l} x_{ik} = \overline{x}_k \Rightarrow \widehat{x}_{ik} = \frac{1}{2} \\ x_{ik} = +\infty \Rightarrow \widehat{x}_{ik} = 1 \\ x_{ik} = -\infty \Rightarrow \widehat{x}_{ik} = 0 \end{array}}$$

… but it does not distribute evenly the data within [0,1]

- exponentially "concentrates" values far from the mean as a function of σ and r:
  - the higher r, the closer to ½ get the farthest samples

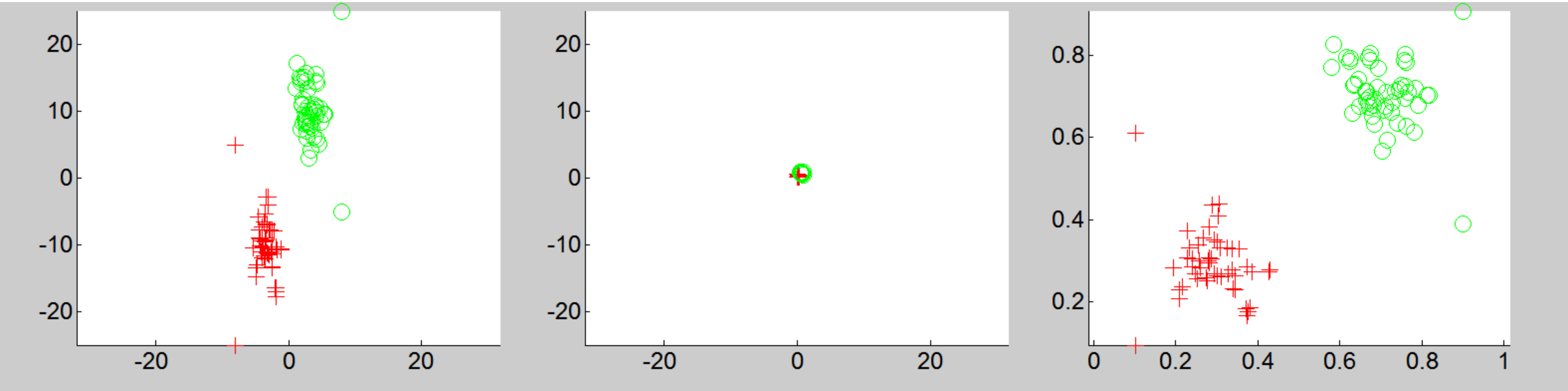
VERY IMPORTANT:
Apply the transformation to the full dataset once.

❖ Keep the transformation parameters $(\bar{x}, \sigma, r)$ to standardize new samples

Alberto Ortiz (last update 13/10/2025)

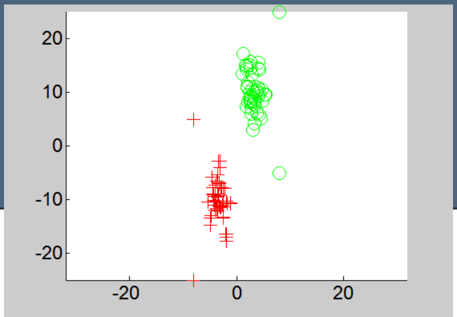
Data preprocessing: Normalization

- ***Softmax* normalization**

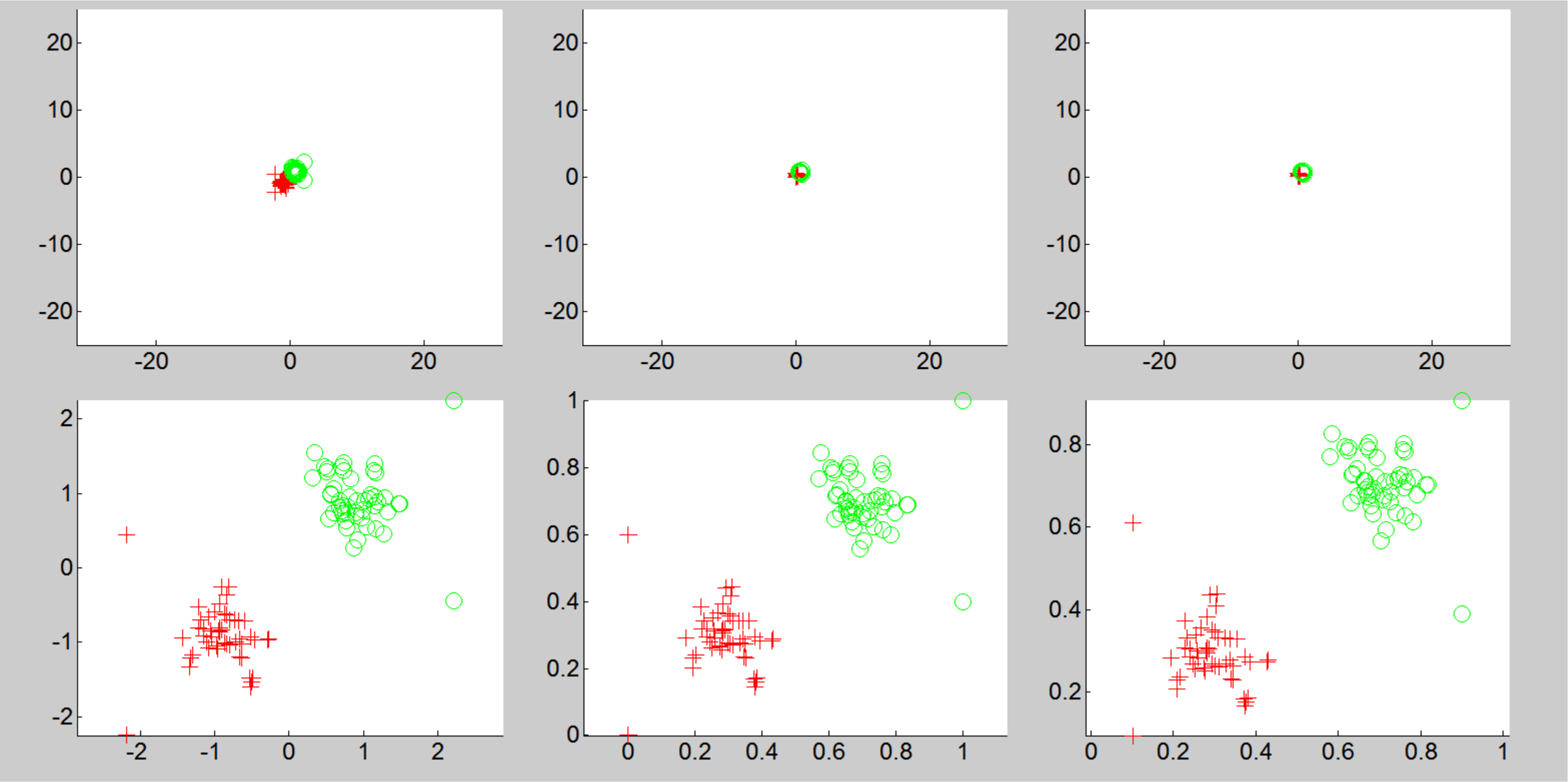


normalization: softmax (r=1)
k=1 org: -8.00 -   8.00 : 16.00 ← dynamic range of x_1
k=2 org: -25.00 - 25.00 : 50.00 ← dynamic range of x_2
ratio   :   3.13
k=1 nor: 0.10 -   0.90 :   0.80 ← dynamic range of $\hat{x}_1$
k=2 nor: 0.10 -   0.90 :   0.81 ← dynamic range of $\hat{x}_2$
ratio   :   1.01

Data preprocessing



- **Comparison**



mu-sigma normalization

max-min normalization

softmax normalization

Data preprocessing: Normalization

- Support in **Python**:

```
from sklearn import preprocessing
from sklearn.datasets import load_iris
iris = load_iris()
X = iris.data
scaler = preprocessing.StandardScaler()
scaler.fit(X)
Xhat = scaler.transform(X)
print(scaler.mean_)
print(scaler.scale_)
print(Xhat.mean(axis=0))
print(Xhat.std(axis=0))
```

```
[5.8433 3.0573 3.7580 1.1993]
[0.8253 0.4344 1.7594 0.7597]
[-1.7e-15 -1.8e-15 -1.7e-15 -1.4e-15]
[1. 1. 1. 1.]
```

≡ μ

≡ σ

```
from sklearn import preprocessing
from sklearn.datasets import load_iris
iris = load_iris()
X = iris.data
scaler = preprocessing.MinMaxScaler()
scaler.fit(X)
Xhat = scaler.transform(X)
print(scaler.min_)
print(scaler.scale_)
print(Xhat.max(axis=0))
print(Xhat.min(axis=0))
```

≡ – min / (max – min)

```
[-1.1944 -0.8333 -0.1695 -0.0417]
[ 0.2778  0.4167  0.1695  0.4167]
[1. 1. 1. 1.]
[0. 0. 0. 0.]
```

≡ 1 / (max – min)

Data preprocessing (incl. cleaning)

- **Filling in missing data**
  - Sometimes, a dataset is incomplete:

x_1	x_2	x_3	x_4
1	2	-4	3
3	2	?	2
2	5	3	?
2.5	?	2	3
1	1	0	2
6	2	4	1

- In Python, **?** typically appear as **Nan**, Null or None values
- Machine learning models cannot handle these kind of values
- Filling with **0s** is not an option

- If the training set is large enough, one can **discard** incomplete samples
- With some datasets, discarding samples is not an option → **heuristic prediction**
  - e.g. fill missing data using the **average value** from complete samples
  - e.g. fill missing data according to the **inherent distribution**

Data preprocessing: Missing data

- We will illustrate the process with the *Titanic* dataset:

```
import seaborn as sb
titanic = sb.load_dataset('titanic')
df = titanic.iloc[:,0:12]
print(df.info())
```

```
RangeIndex: 891 entries, 0 to 890
Data columns (total 12 columns):
 #   Column       Non-Null Count  Dtype
---  ------       --------------  -----
 0   survived     891 non-null    int64
 1   pclass       891 non-null    int64
 2   sex          891 non-null    object
 3   age          714 non-null    float64
 4   sibsp        891 non-null    int64
 5   parch        891 non-null    int64
 6   fare         891 non-null    float64
 7   embarked     889 non-null    object
 8   class        891 non-null    category
 9   who          891 non-null    object
 10  adult_male   891 non-null    bool
 11  deck         203 non-null    category
```

  - We can see that column *Age* contains missing (null) values
    - Also other columns, sometimes they are not useful from the ML point of view
  - This can also be obtained by means of the **isnull()** and the **isna()** methods:

```
print(df.isnull().sum())
print(df.isna().sum())
```

```
survived        0
pclass          0
sex             0
age           177
sibsp           0
parch           0
fare            0
embarked        2
class           0
who             0
adult_male      0
deck          688
```

Data preprocessing: Missing data

- We can proceed in several ways:
  - **Delete the columns** with missing data:

```
udf = df.dropna(axis=1)
print(udf.info())
```

```
RangeIndex: 891 entries, 0 to 890
Data columns (total 9 columns):
 #   Column      Non-Null Count   Dtype
---  ------      --------------   -----
 0   survived    891 non-null     int64
 1   pclass      891 non-null     int64
 2   sex         891 non-null     object
 3   sibsp       891 non-null     int64
 4   parch       891 non-null     int64
 5   fare        891 non-null     float64
 6   class       891 non-null     category
 7   who         891 non-null     object
 8   adult_male  891 non-null     bool
```

  - **Delete the rows** with missing data:

```
udf = df.dropna(axis=0)
print(udf.info())
```

    - In this way, we remove too many entries because of the *deck* column:

```
-> 183 entries, 1 to 889
```

    - Better if we remove first the *deck* column and next the rows with missing data:

```
udf = df.drop('deck', axis=1)
udf.dropna(axis=0,inplace=True)
print(udf.info())
```

```
Int64Index: 712 entries, 0 to 890
Data columns (total 11 columns):
 #   Column      Non-Null Count   Dtype
---  ------      --------------   -----
 0   survived    712 non-null     int64
 1   pclass      712 non-null     int64
 2   sex         712 non-null     object
 3   age         712 non-null     float64
 4   sibsp       712 non-null     int64
 5   parch       712 non-null     int64
 6   fare        712 non-null     float64
 7   embarked    712 non-null     object
 8   class       712 non-null     category
 9   who         712 non-null     object
 10  adult_male  712 non-null     bool
```

Data preprocessing: Missing data

- We can proceed in several ways:
  - Fill the missing values by means of **feature imputation**:

    Numerical data

    - Fill with the mean
    - Fill with the median
    - *Fill with extreme values that do not occur in the data*

    Categorical data

    - Fill with the mode of the distribution
    - *Fill with a new label*

```
udf = df
udf['age'].fillna(udf['age'].mean(),
                  inplace=True)
```

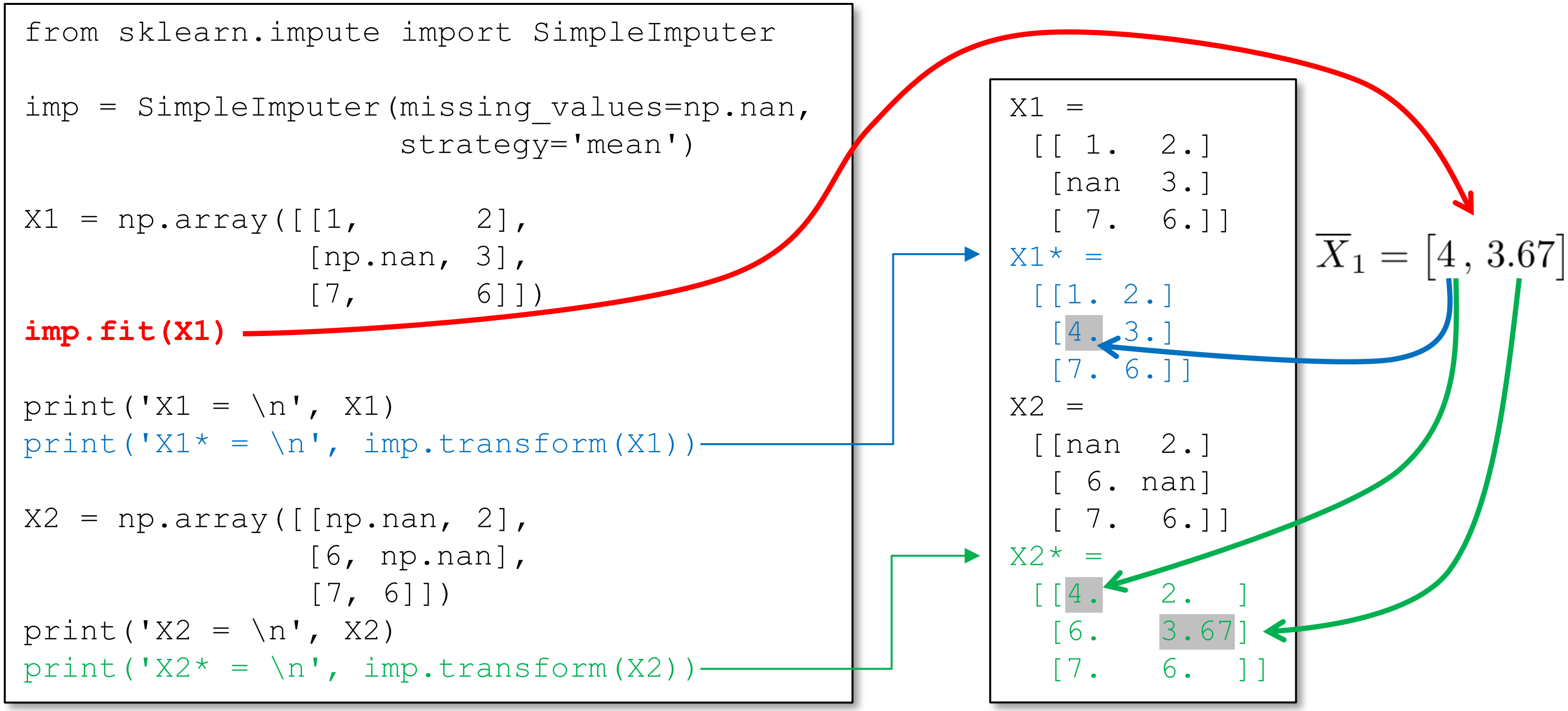
  - It is good practice to register which data values have been *filled in* before the imputation:

```
udf['missing_age'] = df['age'].isnull()
print(udf.info())
```

```
RangeIndex: 891 entries, 0 to 890
Data columns (total 13 columns):
 #   Column       Non-Null Count   Dtype
---  ------       --------------   -----
 0   survived     891 non-null     int64
 1   pclass       891 non-null     int64
 2   sex          891 non-null     object
 3   age          891 non-null     float64
 4   sibsp        891 non-null     int64
 5   parch        891 non-null     int64
 6   fare         891 non-null     float64
 7   embarked     889 non-null     object
 8   class        891 non-null     category
 9   who          891 non-null     object
 10  adult_male   891 non-null     bool
 11  deck         203 non-null     category
 12  missing_age  891 non-null     bool
```

Data preprocessing: Missing data

- There are alternative ways:
  - Fill the missing values using the *SimpleImputer* class for **univariate imputation**



```
from sklearn.impute import SimpleImputer

imp = SimpleImputer(missing_values=np.nan,
                    strategy='mean')

X1 = np.array([[1,      2],
               [np.nan, 3],
               [7,      6]])
imp.fit(X1)

print('X1 = \n', X1)
print('X1* = \n', imp.transform(X1))

X2 = np.array([[np.nan, 2],
               [6, np.nan],
               [7, 6]])
print('X2 = \n', X2)
print('X2* = \n', imp.transform(X2))
```

```
X1 =
 [[ 1.   2.]
  [nan   3.]
  [ 7.   6.]]
X1* =
 [[1.  2.]
  [4.  3.]
  [7.  6.]]
X2 =
 [[nan   2.]
  [ 6.  nan]
  [ 7.   6.]]
X2* =
 [[4.    2.  ]
  [6.    3.67]
  [7.    6.  ]]
```

$\overline{X}_1 = [4\,,\,3.67]$

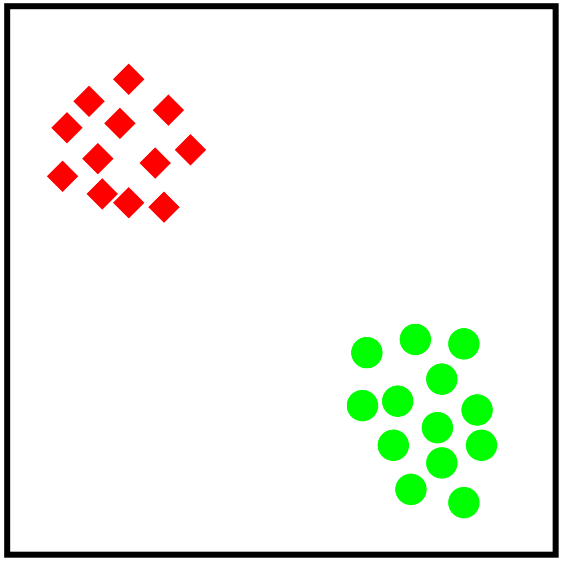
  - **Multivariate imputation** is available in the *IterativeImputer* class
    - Takes into account all columns, instead of only the values of the column with missing values
    - Makes use of a **multivariate regression model** fitted with the available features to regress the missing values

Contents

- Introduction
- Data exploration (and first cleaning)
- Data preprocessing (incl. cleaning)
- Goodness measures
- Feature selection
- Dimensionality reduction
- Pipelines

Goodness measures

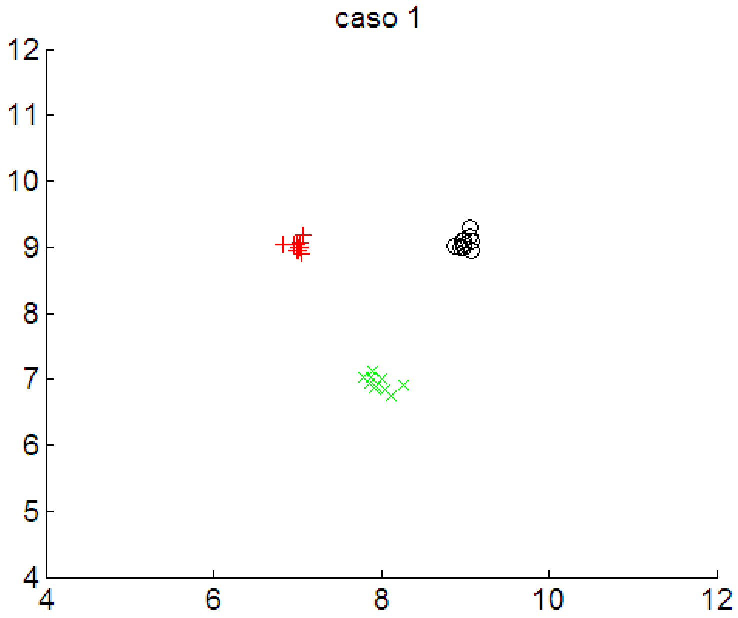
- **General criterion**:
  - Features should result in a large distance between classes (*between-class distance*) and a reduced variance between class elements (*within-class variance*)
  - **Options**:
    - examine features **in isolation**
      - not optimal, but it serves to discard bad selections easily
    - examine features **in combination**
  - We will consider
    - measures based on **scatter matrices** (examination in combination)
  - but we are not going to consider:
    - measures based on **statistical inference** tests (isolated examination)
      - Each test allows you to work with one feature and two classes only, and requires assumptions about the probability distribution for the classes



Measures based on scatter matrices

- Measures based on **scatter matrices** allow **multiple classes** and **several characteristics** to be treated simultaneously and do not require the assumption of **normality** in the data
- Let us suppose the following dataset:
  - 10 samples/class, 3 classes, 2 features

	ω_1		ω_2		ω_3	
$x_{k,1}$	7.00	9.03	8.02	7.01	8.99	9.00
$x_{k,2}$	7.05	8.90	7.90	7.13	8.97	9.10
$x_{k,3}$	7.01	8.99	8.12	6.74	9.06	9.29
$x_{k,4}$	7.07	9.19	7.79	7.04	8.88	9.02
$x_{k,5}$	7.03	8.99	7.91	6.88	9.08	8.94
$x_{k,6}$	7.00	8.94	8.26	6.91	9.05	9.17
$x_{k,7}$	6.83	9.05	7.88	7.03	9.08	9.09
$x_{k,8}$	6.96	9.04	7.93	6.89	8.98	9.03
$x_{k,9}$	7.01	8.95	7.86	6.94	8.95	8.98
$x_{k,10}$	7.02	9.07	8.05	6.85	8.99	9.10



Measures based on scatter matrices

- We first calculate the ***within-class scatter matrix*** (vectors are always column vectors):

$$S_w = \sum_{k=1}^{M} P_k S_k \text{ , where } P_k \approx \frac{n_k}{N} \text{ , } S_k = \frac{1}{n_k - 1} \sum_{j=1}^{n_k} (x_{k,j} - \mu_k)(x_{k,j} - \mu_k)^T$$

where:

$$\mu_k = \frac{1}{n_k} \sum_{j=1}^{n_k} x_{k,j}$$

- M is the number of classes
- n_k is the number of samples in class k
- P_k is the **probability *a priori*** of class ω_k
- S_k is the **covariance matrix** of class ω_k

Measures based on scatter matrices

- Following with the example: M = 3 classes, n_k = 10 samples/class, N = 30 samples

	ω_1		ω_2		ω_3	
$x_{i,1}$	7.00	9.03	8.02	7.01	8.99	9.00
$x_{i,2}$	7.05	8.90	7.90	7.13	8.97	9.10
$x_{i,3}$	7.01	8.99	8.12	6.74	9.06	9.29
$x_{i,4}$	7.07	9.19	7.79	7.04	8.88	9.02
$x_{i,5}$	7.03	8.99	7.91	6.88	9.08	8.94
$x_{i,6}$	7.00	8.94	8.26	6.91	9.05	9.17
$x_{i,7}$	6.83	9.05	7.88	7.03	9.08	9.09
$x_{i,8}$	6.96	9.04	7.93	6.89	8.98	9.03
$x_{i,9}$	7.01	8.95	7.86	6.94	8.95	8.98
$x_{i,10}$	7.02	9.07	8.05	6.85	8.99	9.10
	(7.00, 9.01)		(7.97, 6.94)		(9.00, 9.07)	
	μ_1^T		μ_2^T		μ_3^T	

$$S_1 = \begin{pmatrix} 0.0046 & -0.0001 \\ -0.0001 & 0.0067 \end{pmatrix}$$

$$S_2 = \begin{pmatrix} 0.0201 & -0.0085 \\ -0.0085 & 0.0127 \end{pmatrix} \qquad S_w = \begin{pmatrix} 0.0097 & -0.0020 \\ -0.0020 & 0.0100 \end{pmatrix}$$

$$S_3 = \begin{pmatrix} 0.0044 & 0.0025 \\ 0.0025 & 0.0104 \end{pmatrix}$$

Measures based on scatter matrices

- We next calculate the ***between-class scatter matrix*** (vectors are always column vectors):

$$S_b = \sum_{k=1}^{M} P_k(\mu_k - \mu_0)(\mu_k - \mu_0)^T \text{ , where } P_k \approx \frac{n_k}{N} \text{ , } \mu_0 = \sum_{k=1}^{M} P_k \mu_k$$

- Following with the example:

$$\mu_0 = \frac{1}{3}\mu_1 + \frac{1}{3}\mu_2 + \frac{1}{3}\mu_3 = (7.99, 8.34)^T \qquad S_b = \begin{pmatrix} 0.6702 & 0.0321 \\ 0.0321 & 0.9817 \end{pmatrix}$$

- Finally, we obtain the ***mixture scatter matrix***:

$$S_w = \begin{pmatrix} 0.0097 & -0.0020 \\ -0.0020 & 0.0100 \end{pmatrix}$$

$$S_m = S_w + S_b \qquad S_m = \begin{pmatrix} 0.6799 & 0.0301 \\ 0.0301 & 0.9916 \end{pmatrix}$$

- Important properties:
  - **trace(S_w)** measures the dispersion of features inside the classes
  - **trace(S_b)** measures the dispersion of the class centers amongst them
    - ≡ All this permits comparing different features sets among them, i.e. **they are not absolute measures**, but relative

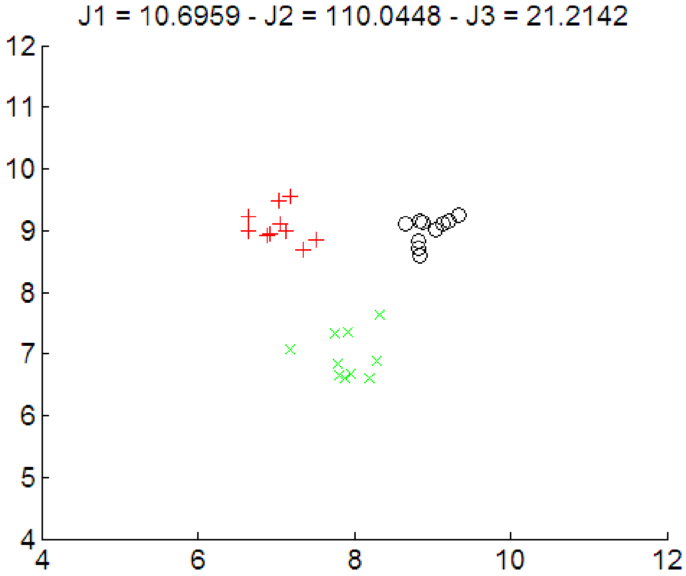
Measures based on scatter matrices

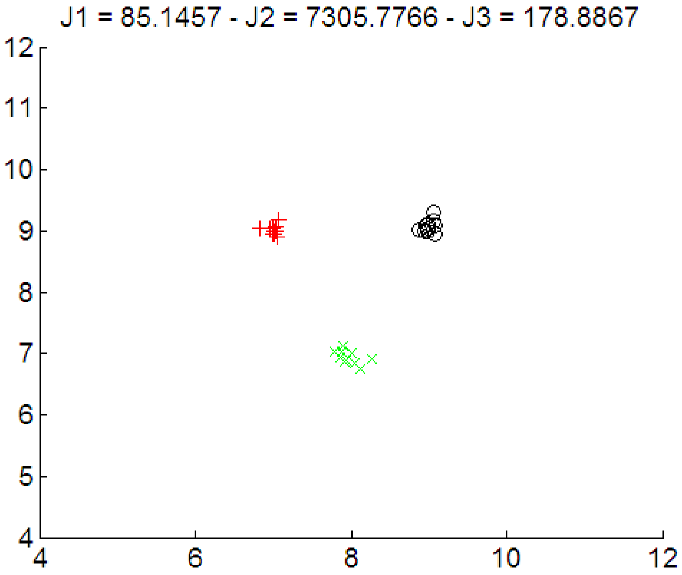
- We can define the following measures:

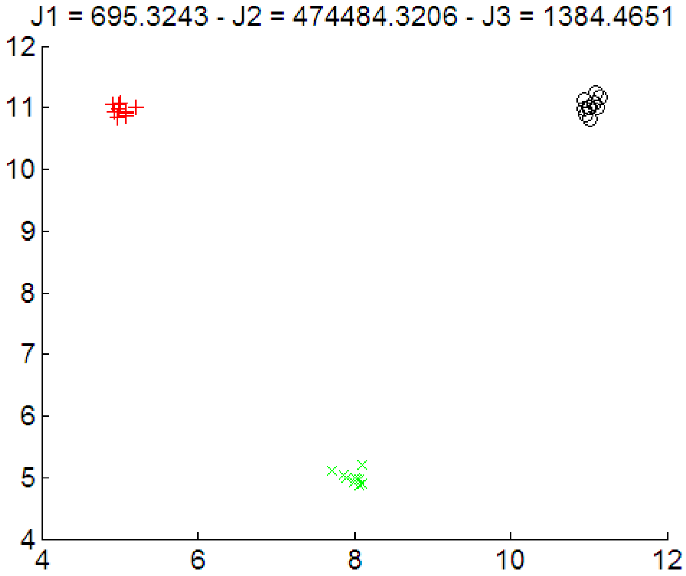
$$J_1 = \frac{\mathrm{trace}(S_m)}{\mathrm{trace}(S_w)}$$

$$J_2 = \frac{|S_m|}{|S_w|} = |S_w^{-1} S_m|$$

$$J_3 = \mathrm{trace}(S_w^{-1} S_m)$$


J1 = 10.6959 - J2 = 110.0448 - J3 = 21.2142


J1 = 85.1457 - J2 = 7305.7766 - J3 = 178.8867


J1 = 695.3243 - J2 = 474484.3206 - J3 = 1384.4651

Measures based on scatter matrices

- **1D case** (1 feature) and **2 equiprobable classes**: (= feature by feature and every 2 classes)
  - S_w gets reduced to ${\sigma_1}^2 + {\sigma_2}^2$
  - S_b can be shown to be $\frac{1}{2}(\mu_1 - \mu_2)^2$
- Following with this reasoning, we obtain the *Fisher's Discriminant Ratio* (FDR) for feature f and classes ω_1 and ω_2:

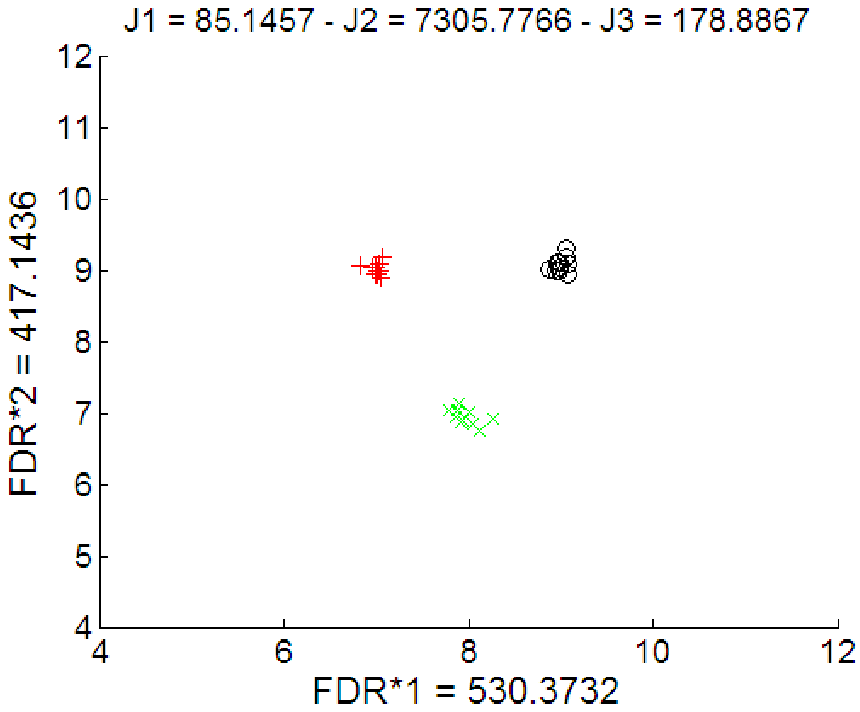
$$\mathrm{FDR}_f(\omega_1, \omega_2) = \frac{(\mu_1 - \mu_2)^2}{\sigma_1^2 + \sigma_2^2}$$

  - FDR can be used to quantify the separability capacity of individual features
  - Similar to **q-statistics** (measures based on statistical hypothesis testing), but the FDR does not depend on the statistical distribution of the data !!
- Multiclass case, one feature f:

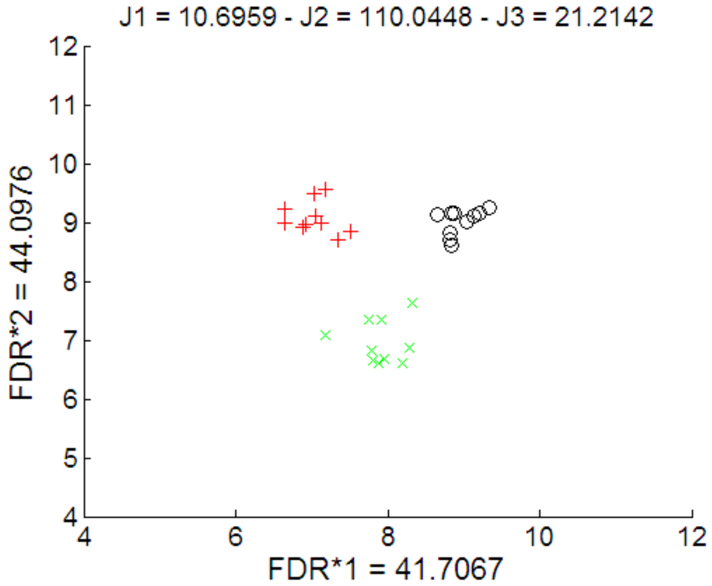
$$\mathrm{FDR}_f^* = \sum_{k_1 \neq k_2} \mathrm{FDR}_f(\omega_{k_1}, \omega_{k_2}), \quad \mathrm{FDR}_f^+ = \min_{k_1 \neq k_2} \{\mathrm{FDR}_f(\omega_{k_1}, \omega_{k_2})\}$$

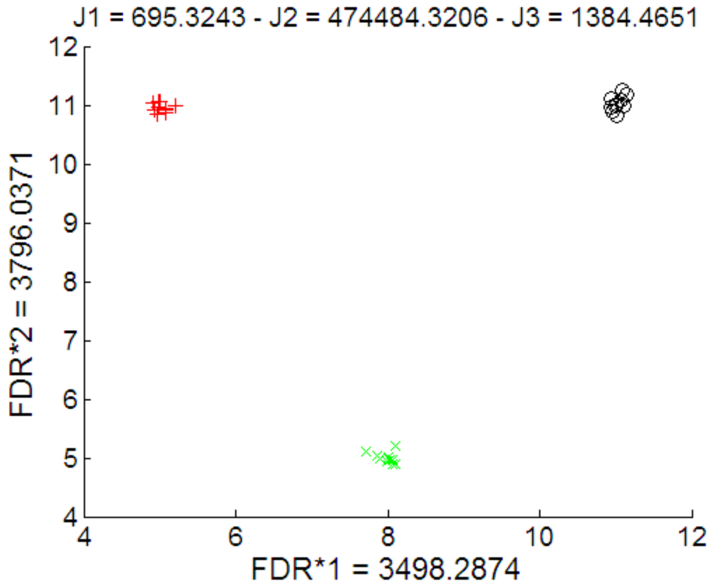
Measures based on scatter matrices

- For the previous example:



Σ_f FDR$_f$ *	Σ_f FDR$_f$ $^+$
947,5168	417,1436
85,8043	41,7067
7294,3245	3498,2874





Contents

- Introduction
- Data exploration (and first cleaning)
- Data preprocessing (incl. cleaning)
- Goodness measures
- Feature selection
- Dimensionality reduction
- Pipelines

Feature selection

- Given $\boldsymbol{C}$ features in total, this point is about selecting $\boldsymbol{L}$ as the most adequate subset
- Several approaches:
  - **Isolated** feature selection
    - Essentially based on a **statistical test**, e.g. the *F test*, that checks each feature separately, assigns a score to each feature f_k and chooses the L best features
      - The *F-test* captures linear relationships between features f_k and labels y
        - A highly correlated feature is given a higher score
      - The *SelectKbest* function in scikit-learn implements such a test
  - **Joint** features selection
    - Consider different groups of features and select the one with the highest score according to a certain goodness measure

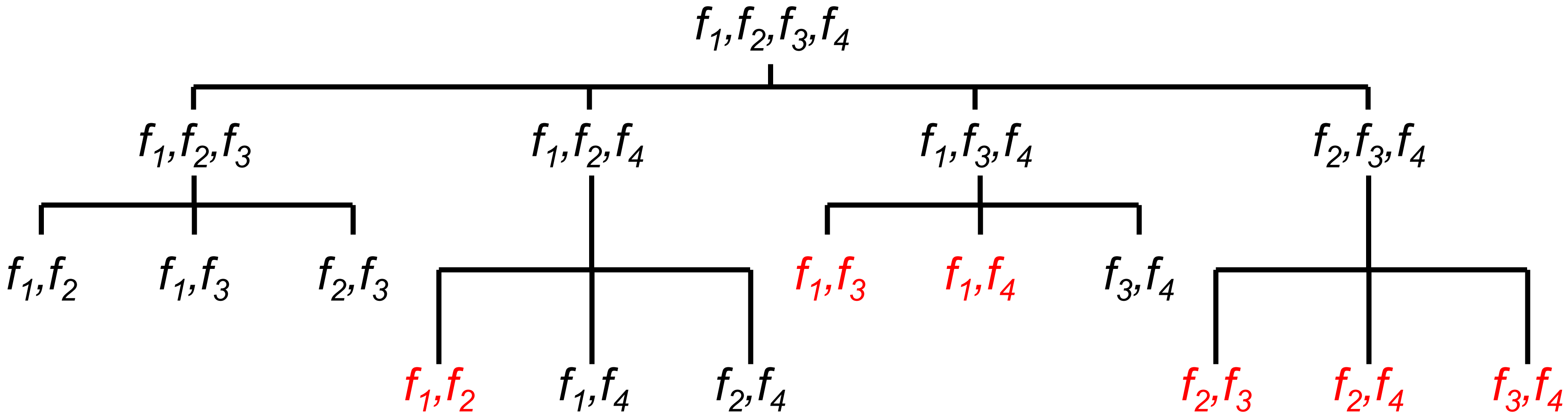
Feature selection

- **Joint selection**

  Consider **subsets of L** features

  1) Joint exhaustive selection: go through all combinations and select the best one

  – For example, let us suppose $C = 4$ and $L = 2$:



  – 6 combinations in total in this case

  – General case:

$$\binom{C}{L} = \frac{C!}{L!(C-L)!} \quad \text{combinations} \quad \left(\text{e.g. } \binom{20}{5} = 15504\right)$$

Feature selection

2) **Joint suboptimal selection**

   – Go through a subset of combinations

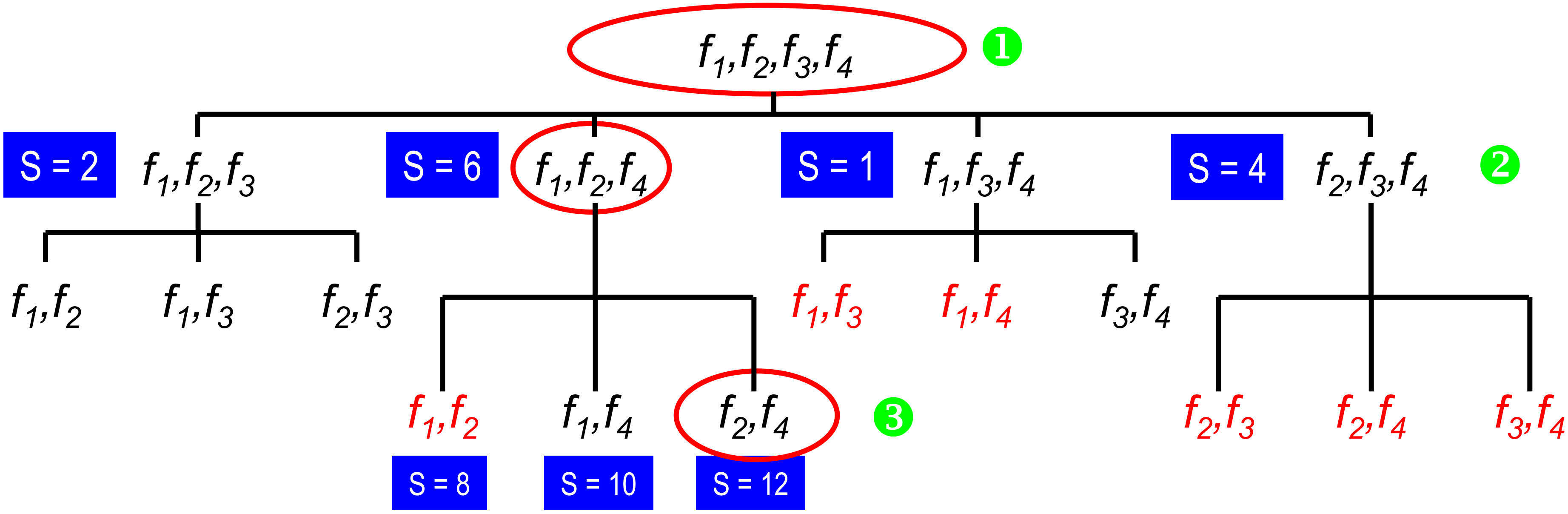
   – It does not guarantee to find the optimal selection but can provide an acceptable selection in less time

   – Two variations:

     - **Backward sequential selection**
     - **Forward sequential selection**

Feature selection

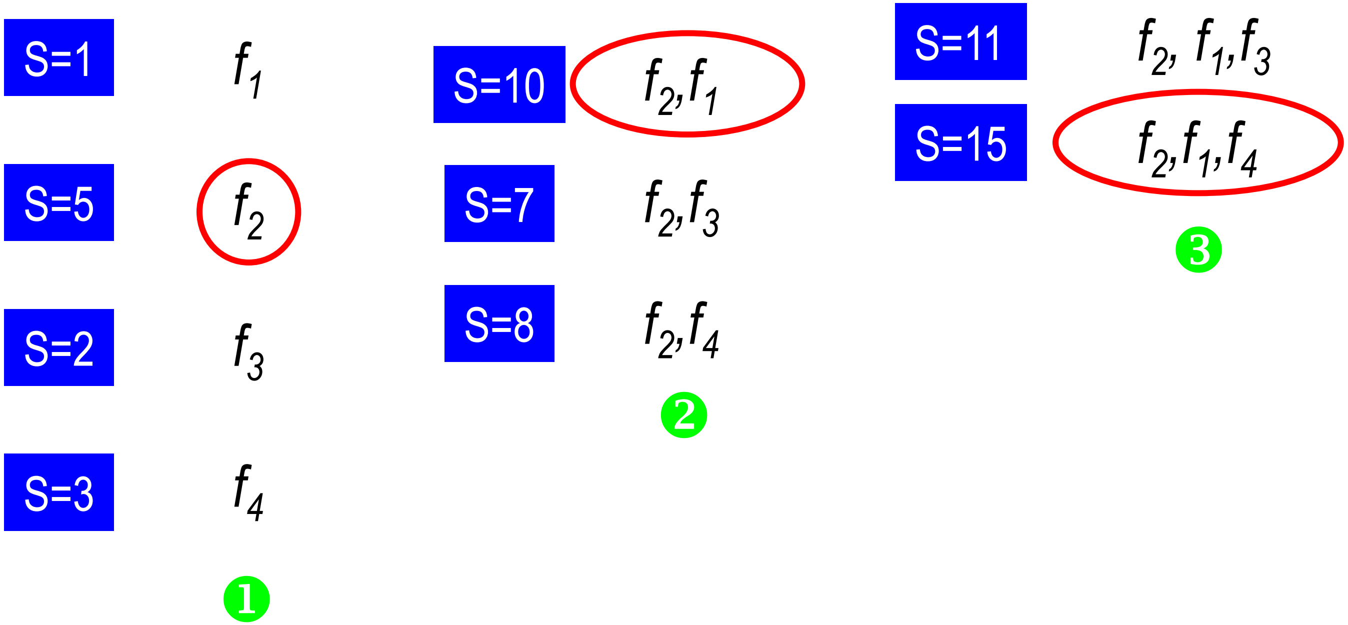
- Suboptimal solutions: **Backward sequential selection**
  - Select a suitable score S, e.g. separability indices J_1, J_2 or J_3, etc.
    - Let us assume that S increases the better the combination
  - Starting from **all features**, progressively remove features until reaching the required amount (of features)
    - At each iteration keep the combination with the largest score



Alberto Ortiz (last update 13/10/2025)

Feature selection

- Suboptimal solutions: **Forward sequential selection**
  - Select a suitable score S, e.g. separability indices J_1, J_2 or J_3, etc.
    - Let us assume that S increases the better the combination
  - Starting with **one feature**, progressively add characteristics until the required number of features is reached
    - At each step keep the combination with the largest score



Feature selection

- Example:

```
import numpy as np
from sklearn.datasets import load_diabetes

diabetes = load_diabetes()
X, y = diabetes.data, diabetes.target
print(diabetes.DESCR)

from sklearn.feature_selection import SequentialFeatureSelector
from sklearn.linear_model import RidgeCV

ridge = RidgeCV(alphas=np.logspace(-6, 6, num=5)).fit(X, y)

sfs_forward = SequentialFeatureSelector(
    ridge, n_features_to_select=2, direction="forward"
).fit(X, y)

sfs_backward = SequentialFeatureSelector(
    ridge, n_features_to_select=2, direction="backward"
).fit(X, y)

feature_names = np.array(diabetes.feature_names)
print(
    "Features selected by forward sequential selection: "
    f"{feature_names[sfs_forward.get_support()]}"
)
print(
    "Features selected by backward sequential selection: "
    f"{feature_names[sfs_backward.get_support()]}"
)
```

```
Diabetes dataset
----------------

Ten baseline variables, age, sex, body mass
index, average blood pressure, and six blood
serum measurements were obtained for each of
n = 442 diabetes patients, as well as the
response of interest, a quantitative measure
of disease progression one year after
baseline.

:Number of Attributes: First 10 columns are
numeric predictive values

:Target: Column 11 is a quantitative measure
of disease progression one year after
baseline

:Attribute Information:
- age age in years
- sex
- bmi body mass index
- bp  average blood pressure
- s1  tc, total serum cholesterol
- s2  ldl, low-density lipoproteins
- s3  hdl, high-density lipoproteins
- s4  tch, total cholesterol / HDL
- s5  ltg, log of serum triglycerides level
- s6  glu, blood sugar level
```

```
Features selected by forward sequential
selection: ['bmi' 's5']

Features selected by backward sequential
selection: ['bmi' 's5']
```

Contents

- Introduction
- Data exploration (and first cleaning)
- Data preprocessing (incl. cleaning)
- Goodness measures
- Feature selection
- Dimensionality reduction
- Pipelines

Dimensionality reduction

- **Dimensionality reduction** (DR) refers to the transformation of the original data into a **reduced-dimension space**, i.e. a new set of features of lower dimensionality
  - Also termed as **feature extraction**
- One can find several DR methods in the literature:
  - Principal Component Analysis (PCA) and variants (Sparse PCA, Kernel PCA, etc.)
  - Other matrix factorizations:
    - Non-negative Matrix Factorization (NMF)
    - Independent Component Analysis (ICA)
    - Truncated Singular Value Decomposition
  - Multi-dimensional Scaling (MDS)
  - t-distributed Stochastic Neighbor Embedding (t-SNE)

} rather for visualizing high-dimensional data

Dimensionality reduction: PCA

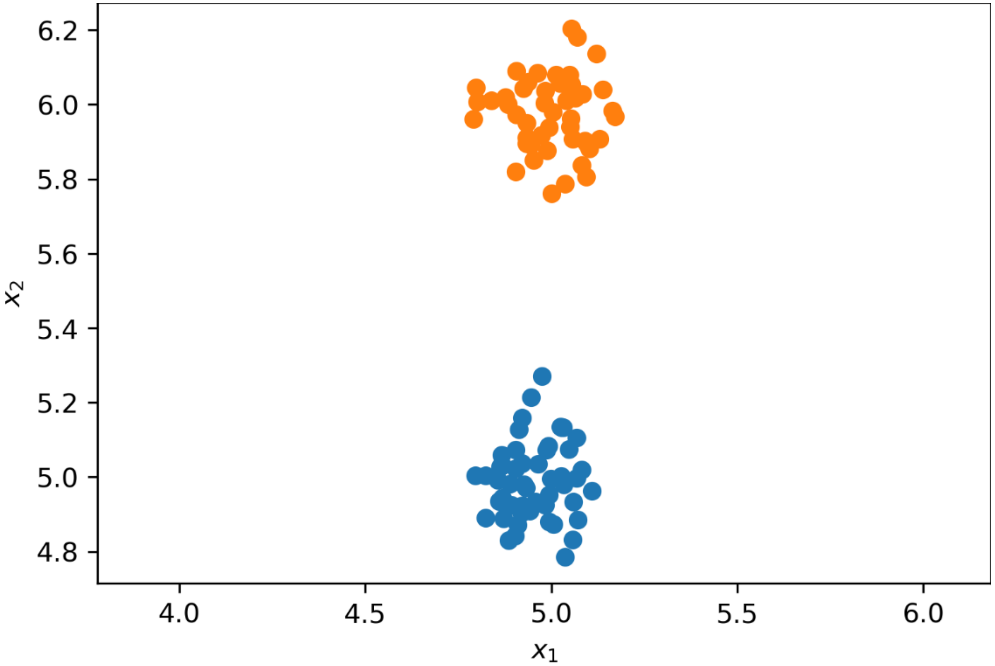
- PCA is a popular technique for **dealing with large high-dimensional datasets**
  - The aim is to derive new features as **linear combinations of the original variables** in decreasing order of importance

$$x'_k = \alpha_1 x_1 + \alpha_2 x_2 + \cdots + \alpha_L x_L$$

  - Also named as the **discrete Karhunen-Loeve transform** (KLT) in signal processing, **proper orthogonal decomposition** (POD) in mechanical engineering, etc.
  - Useful also for other purposes, e.g. **visualization of multi-dimensional data** through **lower-dimensional representations** (retain maximum information as the dimensionality is reduced)

Dimensionality reduction: PCA

- A simple example:



$$\sigma^2_{x_1} = \frac{1}{N}\sum_{i=1}^{N}(x_{i1}-\mu_1)^2 = 0.008$$

$$\sigma^2_{x_2} = \frac{1}{N}\sum_{i=1}^{N}(x_{i2}-\mu_2)^2 = 0.256$$

$$\mathrm{cov}(X) = \frac{1}{N}\sum_{i=1}^{N}(x_i-\mu)(x_i-\mu)^T$$

$$= \begin{pmatrix} 0.008 & 0.012 \\ 0.012 & 0.256 \end{pmatrix}$$

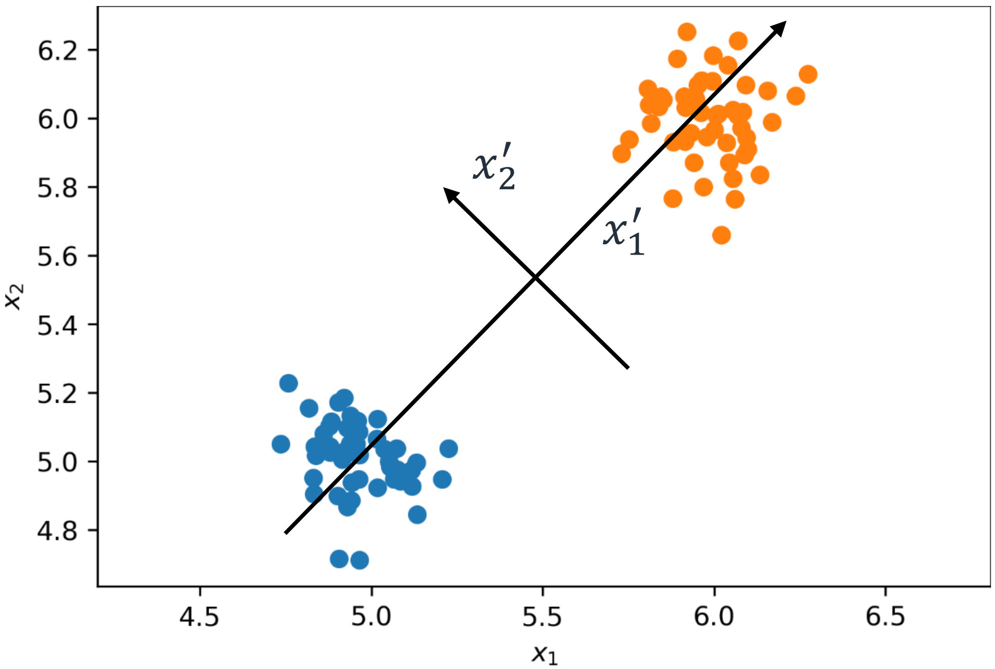
- Which feature should we get rid of?
  - x_1 is not useful from the discrimination point of view
  - x_2 allows discriminating between the two classes

  $\Rightarrow$ x_2 carries more information than x_1, and this coincides with $\sigma^2_{x1} < \sigma^2_{x2}$

  $\Rightarrow$ if we have to choose, better to get rid of x_1, the one with lowest variance

Dimensionality reduction: PCA

- A more complex example:



$$\sigma^2_{x_1} = \frac{1}{N}\sum_{i=1}^{N}(x_{i1}-\mu_1)^2 = 0.280$$

$$\sigma^2_{x_2} = \frac{1}{N}\sum_{i=1}^{N}(x_{i2}-\mu_2)^2 = 0.257$$

$$\mathrm{cov}(X) = \frac{1}{N}\sum_{i=1}^{N}(x_i-\mu)(x_i-\mu)^T$$

$$= \begin{pmatrix} 0.280 & 0.254 \\ 0.254 & 0.257 \end{pmatrix}$$

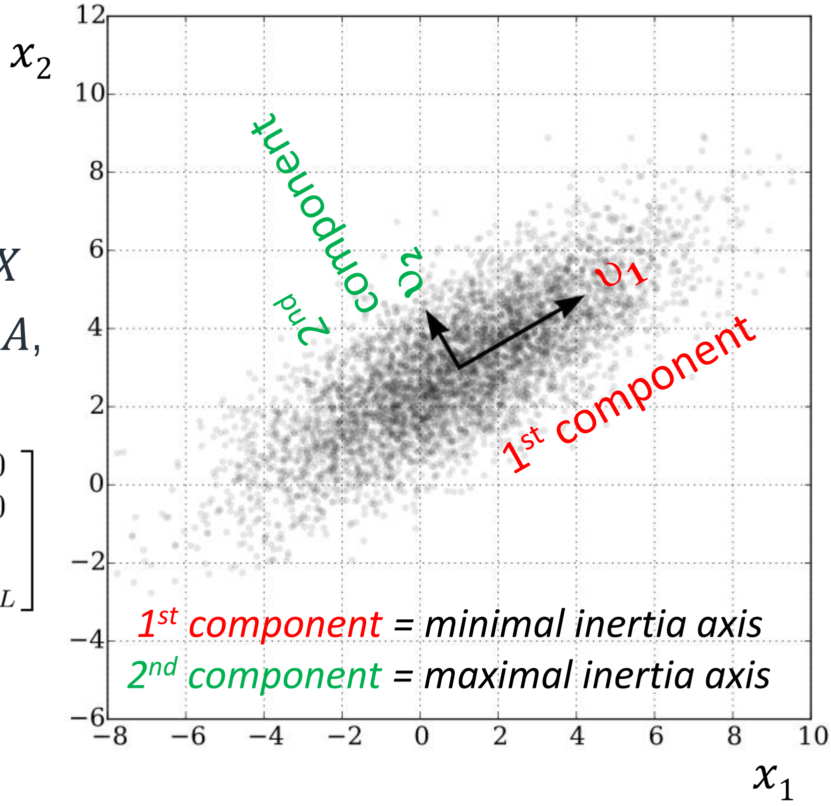
- Which feature should we get rid of? Now is not so clear …
  - Better consider a different set of axes/features, x'_1 and x'_2, to try to maximize the variance in one of the axes/features
  - In the plot, x'_1 would be the direction of largest variance, and x'_2 would be the next in variance that is orthogonal to x'_1

Dimensionality reduction: PCA

- PCA finds automatically axes $x_1', x_2', \dots$
  1. **center** the values of each feature by subtracting the mean $X = X_{\mathrm{org}} - \bar{X}$
  2. compute the **scatter** / **covariance matrix** $\mathrm{A} = X^T X$
  3. obtain the **eigenvalues** λ_i and **eigenvectors** ν_i of A, i.e. $A\upsilon_i = \lambda_i\upsilon_i$

$$A = V\,D\,V^{-1}, \text{ with } V = \left[\begin{array}{c|c|c|c} \uparrow & \uparrow & & \uparrow \\ \nu_1 & \nu_2 & \dots & \nu_L \\ \downarrow & \downarrow & & \downarrow \end{array}\right], \; D = \begin{bmatrix} \lambda_1 & 0 & \dots & 0 \\ 0 & \lambda_2 & \dots & 0 \\ & & \ddots & \\ 0 & 0 & \dots & \lambda_L \end{bmatrix}$$

✓ the eigenvectors constitute an orthonormal basis and **each eigenvalue is the variance along one axis**, i.e. the corresponding eigenvector



1st component = minimal inertia axis

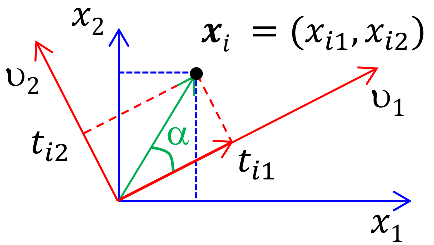
2nd component = maximal inertia axis

- PCA can be thought of as fitting an **L-dimensional hyperellipsoid** to the data
  - Each axis of the ellipsoid represents a principal component
    - If one axis of the ellipsoid is short, it is because the variance along that axis is small
- x_1 and x_2 features are linearly correlated (when x_1 grows, x_2 grows proportionally), but data points projected onto the resulting orthogonal basis $\upsilon_1 - \upsilon_2$ are no longer correlated

Dimensionality reduction: PCA

- Given the data matrix X which has been **mean-centered** ($X = X_{\text{org}} - \bar{X}$), whose rows contain the data samples $\boldsymbol{x}_i$ and its columns are the feature values,
  we are looking for a set of vectors υ_i that constitute an **orthonormal basis** where the data is going to be expressed in:

$$t_{i,k} = \mathbf{x}_i \cdot \nu_k \,, \quad i = 1, \ldots, N \text{ and axes } k = 1, \ldots, L$$



$$\mathbf{x}_i \cdot \nu_1 = \|\nu_1\| \|x_i\| \cos\alpha$$
$$= \|x_i\| \cos\alpha = t_{i,1}$$

- In order to maximize the variance along the first component we look for vector υ_1 such that:

$$\nu_1 = \underset{\|\nu\|=1}{\arg\max} \left\{ \sum_i t_{i,\nu}^2 \right\} = \underset{\|\nu\|=1}{\arg\max} \left\{ \sum_i (\mathbf{x}_i \cdot \nu)^2 \right\}$$

- In matrix form, this becomes:

$$\nu_1 = \underset{\|\nu\|=1}{\arg\max} \left\{ \|X\nu\|^2 \right\} = \underset{\|\nu\|=1}{\arg\max} \left\{ \nu^T X^T X \nu \right\}$$

where $A = X^T X$ is the **scatter matrix** of X_{org} (≡ covar. matrix if divided by $N-1$).

Dimensionality reduction: PCA

- To find the constrained maximization problem we build the **Lagrangian function** $L(\upsilon)$ as follows:

$$\max_{\|\nu\|=1} \; \nu^T A \nu \Rightarrow \max \; L(\nu) = \nu^T A \nu - \lambda(\nu^T \nu - 1)$$

- The solution is given by: $\dfrac{\partial L}{\partial \nu} = 2A\nu - 2\lambda\nu = 0 \Rightarrow A\nu = \lambda\nu$

$$\frac{\partial L}{\partial \lambda} = \nu^T \nu - 1 = 0 \Rightarrow \nu^T \nu = 1$$

- Equation $A\upsilon = \lambda\upsilon$ has L solutions (υ_i, λ_i) for $A_{L\times L}$, i.e. $A\upsilon_i = \lambda_i \upsilon_i$, $\forall i$

  which corresponds to the **eigendecomposition** of matrix $A_{L\times L}$, which in matrix form is given by:

$$A = V\,D\,V^{-1}, \text{ with } V = \left[\begin{array}{c|c|c|c} \uparrow & \uparrow & & \uparrow \\ \nu_1 & \nu_2 & \ldots & \nu_L \\ \downarrow & \downarrow & & \downarrow \end{array}\right], \; D = \begin{bmatrix} \lambda_1 & 0 & \ldots & 0 \\ 0 & \lambda_2 & \ldots & 0 \\ & & \ddots & \\ 0 & 0 & \ldots & \lambda_L \end{bmatrix}$$

- Linear algebra libraries typically return **unit eigenvectors**, so that all equations are satisfied, and also ordered from largest eigenvalue to lowest eigenvalue:

$$\lambda_1 \geq \lambda_2 \geq \cdots \geq \lambda_L \geq 0$$

Dimensionality reduction: PCA

- Then, we have to find the eigenvector υ that gives rise to

$$\max\ \nu^T A \nu = \max\ \nu^T V D V^{-1} \nu = \max\ \nu^T V D V^T \nu$$

- Let us now suppose that $\upsilon = \upsilon_j$. Then:

$$\nu^T V = \mathbf{e}_j,\ \text{where } \mathbf{e}_j = \left(0,\ \ldots\ ,\ 0,\ \overset{j\downarrow}{1},\ 0,\ \ldots\ ,\ 0\right)$$

  and:

$$\max\ \nu^T V D V^T \nu = \max\ \mathbf{e}_j^T D \mathbf{e}_j = \max\ \lambda_j$$

- The first component of PCA is therefore the eigenvector associated to the **largest eigenvalue**, and so $\upsilon\ = \upsilon_1$ if the eigenvalues are sorted.

Dimensionality reduction: PCA

- To find the second component, we have to maximize for the remaining variance:

$$\max_{\nu} \; \nu^T A \nu - \nu_1^T A \nu_1 \,, \qquad \text{s.t.} \; ||\nu|| = 1 \text{ and } \nu^T \nu_1 = 0$$

- Then, the Lagrangian function becomes:

$$L(\nu) = \nu^T A \nu - \nu_1^T A \nu_1 - \lambda \left(\nu^T \nu - 1 \right) - \mu \left(\nu^T \nu_1 \right)$$

and

$$\frac{\partial L}{\partial \nu} = 2A\nu - 2\lambda\nu - \mu\nu_1 = 0 \Rightarrow A\nu = \lambda\nu$$
$$\frac{\partial L}{\partial \lambda} = \nu^T \nu - 1 = 0 \Rightarrow \nu^T \nu = 1$$
$$\frac{\partial L}{\partial \mu} = \nu^T \nu_1 = 0$$

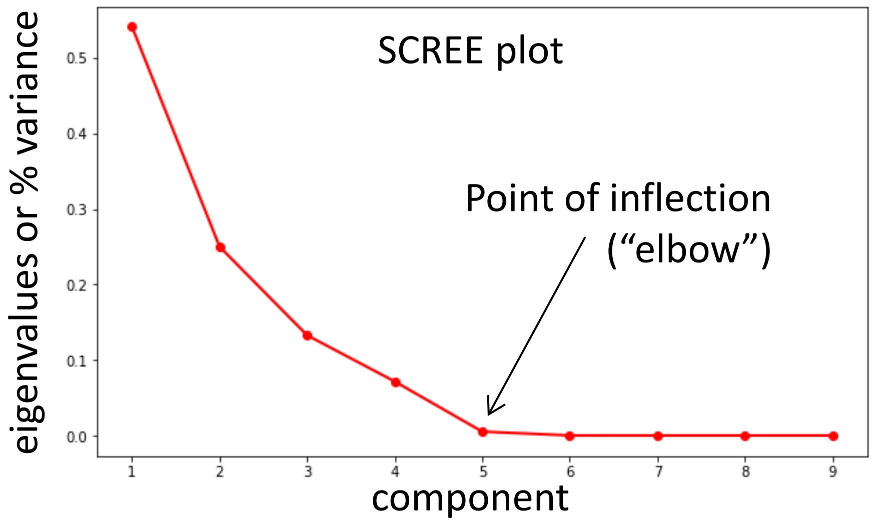
- Referring to the first equation:

$$\begin{aligned} \nu_1^T \left(2A\nu - 2\lambda\nu - \mu\nu_1 \right) &= 2\nu_1^T A \nu - 2\lambda \nu_1^T \nu - \mu \nu_1^T \nu_1 \\ &= 2\nu_1^T A \nu - \mu = 0 \\ &\Rightarrow \mu = 2\nu_1^T A \nu = 2\nu^T A \nu_1 = 2\nu^T \lambda_1 \nu_1 = 0 \end{aligned}$$

- Therefore, the second component is another eigenvector of A and hence has to be the second eigenvector of A, $\upsilon = \upsilon_2$, since λ_2 is the second largest eigenvalue of A.
- The **remaining components can be proved to be the remaining eigenvectors**, ordered by the corresponding eigenvalue from higher to lower.

Dimensionality reduction: PCA

- Now that we know that the components are the eigenvectors of matrix $X^T X$, we have to deal with the **reduced-dimension representation**.
- To this end, we consider the **fraction of the total variance** that is accounted for by the first $p \leq L$ components: $$\frac{\sum_{i=1}^{p} \mathrm{var}[v_i]}{\sum_{i=1}^{L} \mathrm{var}[v_i]} = \frac{\sum_{i=1}^{p} \lambda_i}{\sum_{i=1}^{L} \lambda_i}$$
  - We can specify a threshold τ on this ratio to choose the number of components necessary to account for at least a τ fraction of the total variance.
- We can also plot the eigenvalues in decreasing order (**SCREE plot**) and look for the component for which the accounted variance falls sharply. Two kinds of plots:
  - eigenvalues
  - fraction of total variance: $\frac{\lambda_i}{\sum_j \lambda_j}$



Dimensionality reduction: PCA

- Once we have decided to make use of p components, we can find the reduced-dimensionality vectors/samples: (we do not refer to a particular sample x_i)

centered data	**uncentered data**
$\chi_p = V_{:p}^T \mathbf{x}$	$\chi_p = V_{:p}^T (\mathbf{x} - \mu)$

where χ_p is the reduced-dimension sample, μ is the mean and

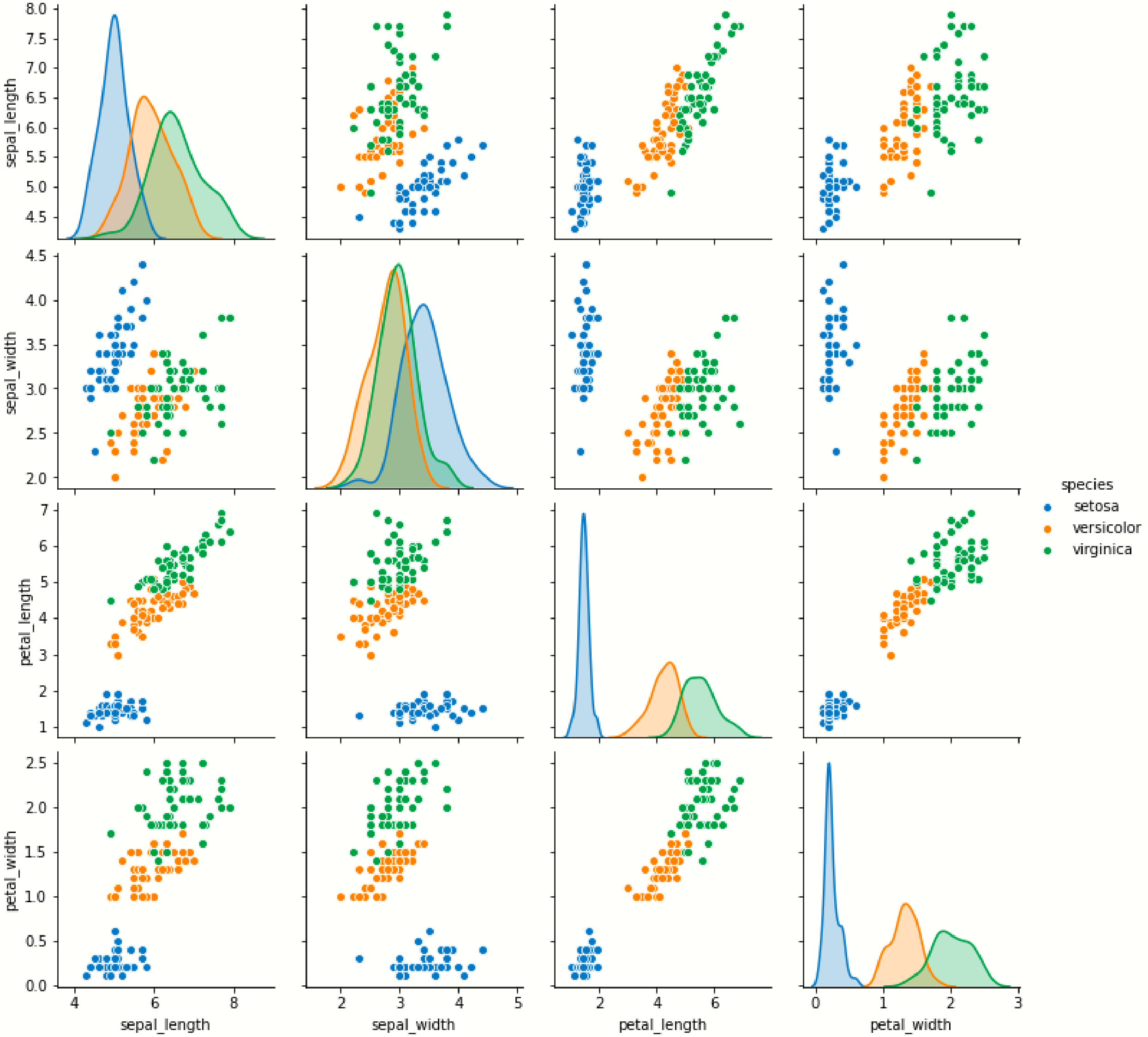
$$V_{:p} = \left[\nu_1 \mid \nu_2 \mid \ldots \mid \nu_p \right]_{L \times p}$$

- The dimensionality reduction operation gives rise to an **error of representation**. This makes interesting to know the representation $\boldsymbol{x}_p$ of χ_p in the original L-dimensional space:

	centered data	**uncentered data**
to transformed space	$\chi = V^T \mathbf{x}$	$\chi = V^T (\mathbf{x} - \mu)$
to original space	$\mathbf{x} = V \chi$	$\mathbf{x} = V \chi + \mu$
reduced dimension, but in the original space	$\mathbf{x}_p = V \begin{pmatrix} \chi_p \\ \mathbf{0} \end{pmatrix}$ $= V_{:p} \chi_p$ $= V_{:p} V_{:p}^T \mathbf{x}$	$\mathbf{x}_p = V \begin{pmatrix} \chi_p \\ \mathbf{0} \end{pmatrix} + \mu$ $= V_{:p} \chi_p + \mu$ $= V_{:p} V_{:p}^T (\mathbf{x} - \mu) + \mu$

Dimensionality reduction: PCA

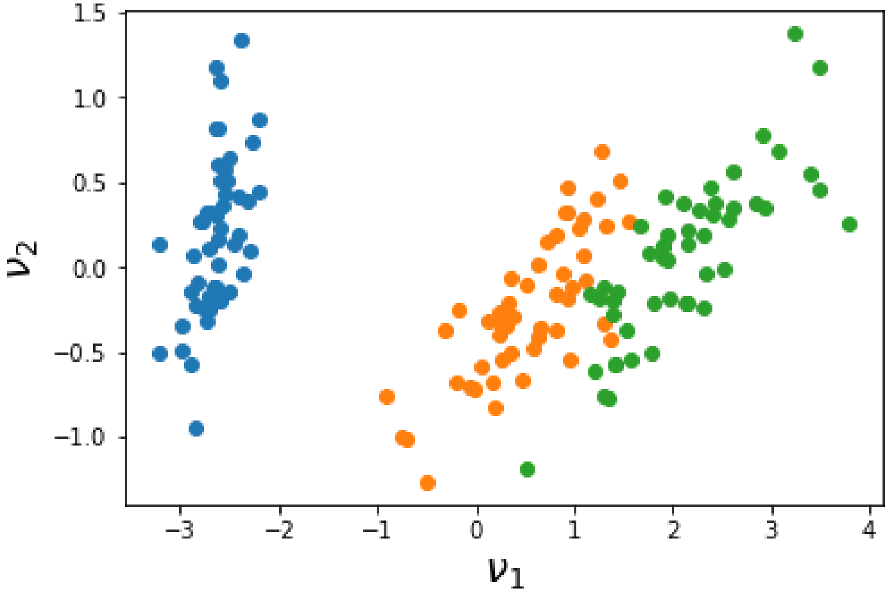
- Example 1:
  Let us consider again the 4-dimensional **Iris dataset**



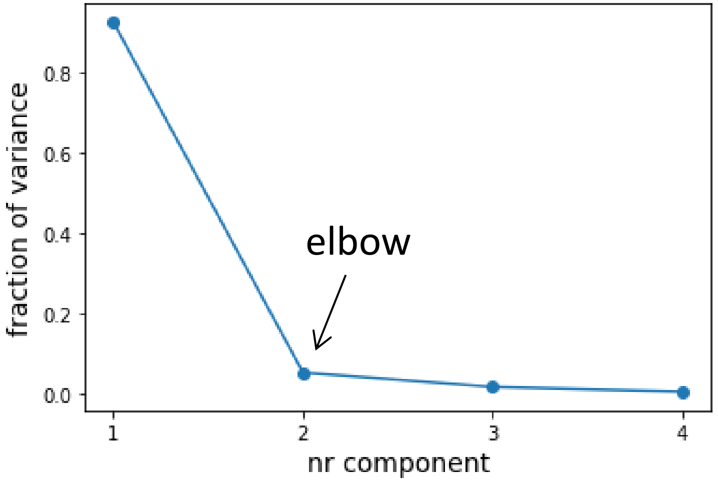
Dimensionality reduction: PCA

- Example 1:

```
import numpy as np
import matplotlib.pyplot as plt
from sklearn import datasets
from sklearn.decomposition import PCA

iris = datasets.load_iris()

X = iris.data
y = iris.target

pca = PCA(n_components=2)
# fit method already includes centering
Xr = pca.fit(X).transform(X)

plt.figure()
for c in range(3):
    i = np.where(y == c)[0]
    plt.scatter(Xr[i,0],Xr[i,1])
plt.show()

pca = PCA(n_components=4)
pca.fit(X)

print(pca.explained_variance_)
print(pca.explained_variance_ratio_)
```



```
[4.2282 0.2427 0.0782 0.0238]
[0.9246 0.0531 0.0171 0.0052]
```



Dimensionality reduction: PCA

- Example 1:

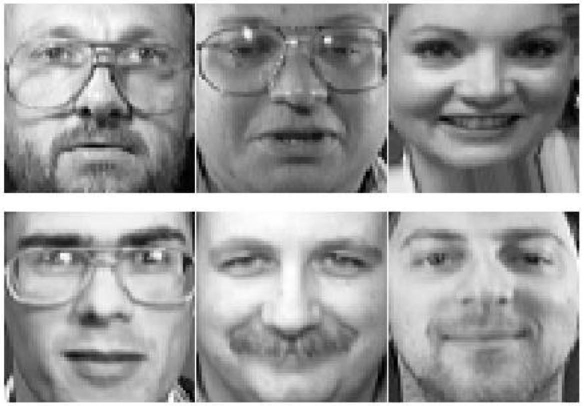
```
(continued)
pca = PCA(n_components=2)
pca.fit(X)
Xr = pca.transform(X)
X_ = pca.inverse_transform(Xr)

import numpy as np
from math import sqrt
error_matrix = X - X_
error_sq = np.sum(np.sum((error_matrix)**2, axis=1))
error = sqrt(error_sq)
N = X.shape[0]
print('total error = %f, total error/sample = %f' % (error, error / N))
m = np.min(np.abs(error_matrix), axis=0)
print('min. errors: %f %f %f %f' % (m[0], m[1], m[2], m[3]))
m = np.max(np.abs(error_matrix), axis=0)
print('max. errors: %f %f %f %f' % (m[0], m[1], m[2], m[3]))
m = np.min(X, axis=0)
print('min. values: %f %f %f %f' % (m[0], m[1], m[2], m[3]))
m = np.max(X, axis=0)
print('max. values: %f %f %f %f' % (m[0], m[1], m[2], m[3]))
```

```
total error = 3.899313, total error/sample = 0.025995
min. errors: 0.001556 0.001401 0.000492 0.000810
max. errors: 0.451606 0.463801 0.233806 0.591713
min. values: 4.300000 2.000000 1.000000 0.100000
max. values: 7.900000 4.400000 6.900000 2.500000
```

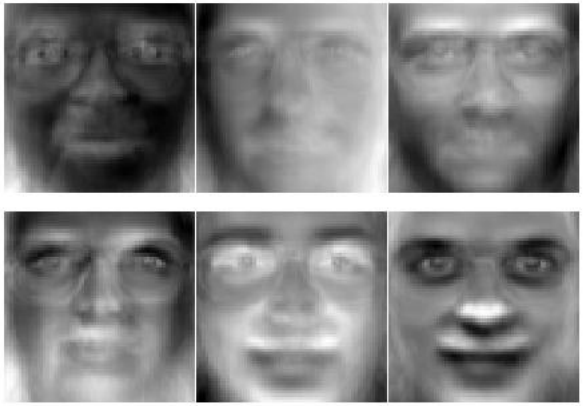
Dimensionality reduction: PCA

- Example 2: **Olivetti faces dataset**,
  400 faces of 64 × 64 pixels,
  4096 dimensions

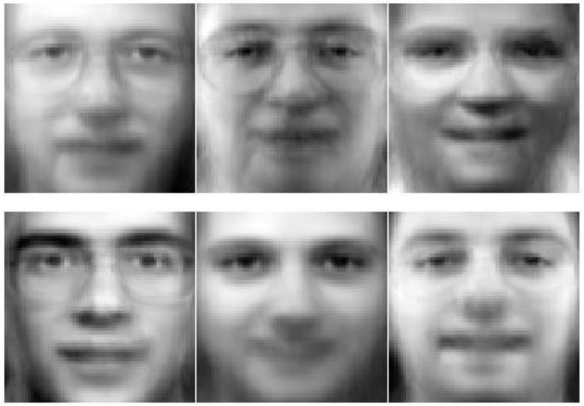
Faces from dataset



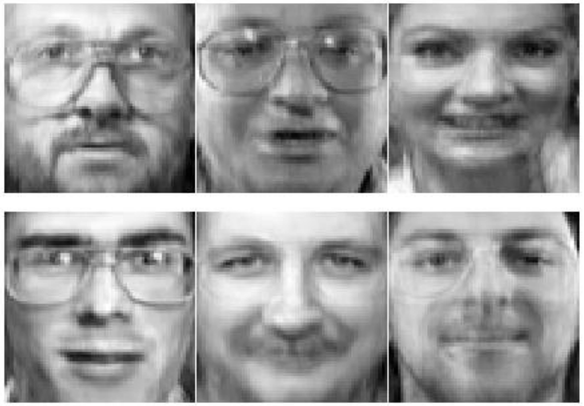
Eigenfaces - PCA using randomized SVD



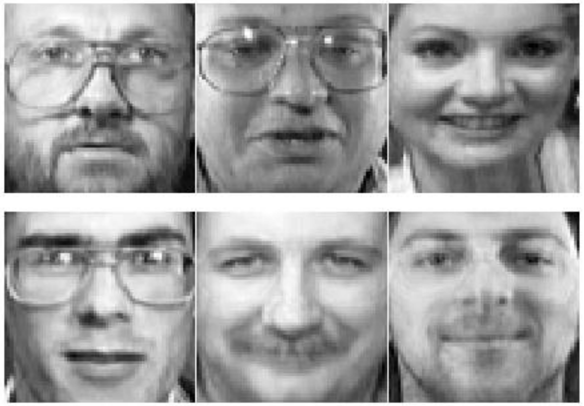
Transformed faces (10 components)



Transformed faces (100 components)



Transformed faces (200 components)

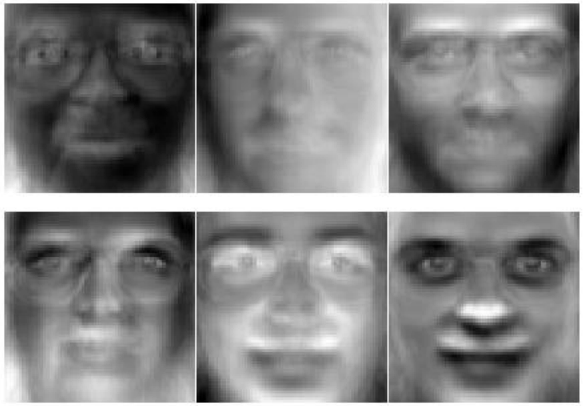


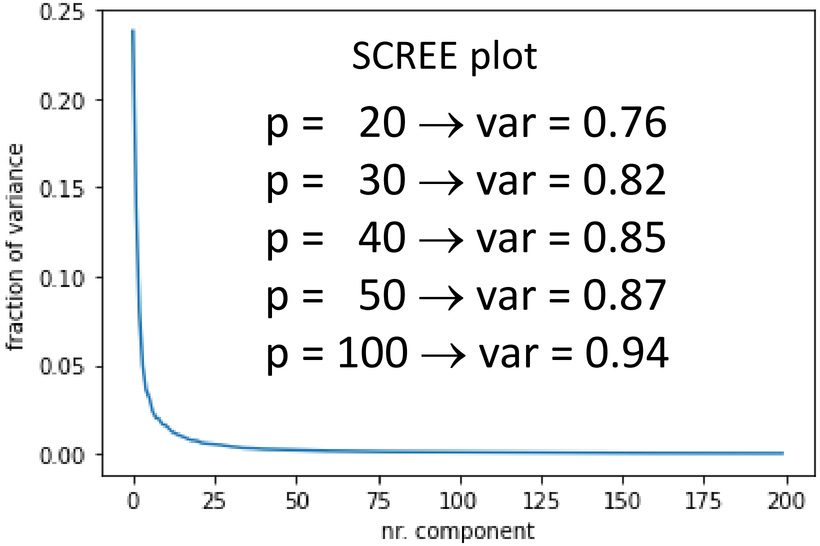
Dimensionality reduction: PCA

- Example 2: **Olivetti faces dataset**,
  400 faces of 64 × 64 pixels,
  4096 dimensions

Faces from dataset

Eigenfaces - PCA using randomized SVD

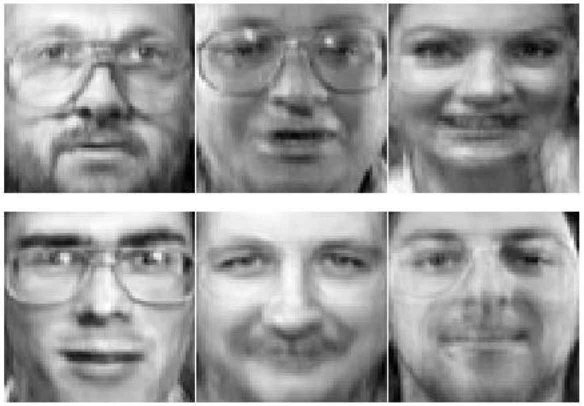




SCREE plot

p = 20 → var = 0.76
p = 30 → var = 0.82
p = 40 → var = 0.85
p = 50 → var = 0.87
p = 100 → var = 0.94

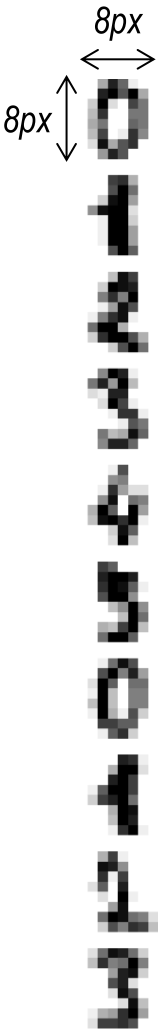
Transformed faces (100 components)



Contents

- Introduction
- Data exploration (and first cleaning)
- Data preprocessing (incl. cleaning)
- Goodness measures
- Feature selection
- Dimensionality reduction
- Pipelines

Pipelines

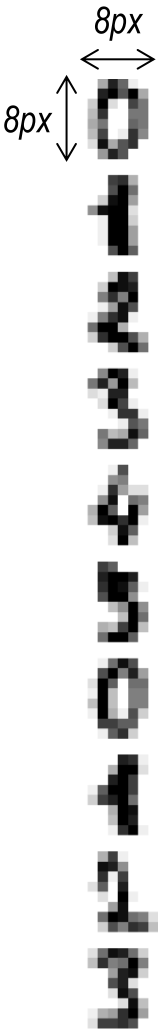


- Scikit-learn allows chaining steps to transform data until reaching the final estimator:

```
from sklearn import datasets
from sklearn.decomposition import PCA
from sklearn.linear_model import LogisticRegression
from sklearn.model_selection import GridSearchCV
from sklearn.pipeline import Pipeline
from sklearn.preprocessing import StandardScaler

# Define a pipeline to search for the best combination of PCA truncation
scaler = StandardScaler() # mu-sigma scaler to normalize inputs
pca = PCA()               # dimensionality reduction
logistic = LogisticRegression(max_iter=10000, tol=0.1) # classifier
pipe = Pipeline(steps=[("scaler", scaler), ("pca", pca), ("logistic", logistic)])

X_digits, y_digits = datasets.load_digits(return_X_y=True)
# Parameters of pipelines can be set using '__' separated parameter names:
param_grid = {
        "pca__n_components": [10, 20, 30, 40, 50],
        "logistic__C": [0.01, 0.1, 1, 10, 100],
}
search = GridSearchCV(pipe, param_grid, n_jobs=-1)
search.fit(X_digits, y_digits)
print("Best configuration (CV score=%0.4f):" % search.best_score_)
print(search.best_params_)
```

```
Best configuration (CV score=0.8737):
{'logistic__C': 1, 'pca__n_components’: 20}
```

Pipelines



- Scikit-learn allows chaining steps to transform data until reaching the final estimator:

```
from sklearn import datasets
from sklearn.decomposition import PCA
from sklearn.linear_model import LogisticRegression
from sklearn.pipeline import Pipeline
from sklearn.preprocessing import StandardScaler
from sklearn.metrics import accuracy_score

# Define the final pipeline
scaler = StandardScaler()
pca = PCA(n_components=search['pca__n_components'])
logistic = LogisticRegression(max_iter=10000, tol=0.1, C=search['logistic__C'])
pipe = Pipeline(steps=[("scaler", scaler), ("pca", pca), ("logistic", logistic)])

X_digits, y_digits = datasets.load_digits(return_X_y=True)
# We use the full dataset, we do not split in training and test
pipe.fit(X_digits, y_digits)
yp = pipe.predict(X_digits)
print('accuracy = %0.4f' % (accuracy_score(y_digits, yp)))
```

```
-> accuracy = 0.8948
```

Lecture 2: Data analysis

Universitat de les Illes Balears | Departament de Ciències Matemàtiques i Informàtica

11752 Aprendizaje Automático
11752 Machine Learning
Máster Universitario
en Sistemas Inteligentes

Alberto ORTIZ RODRÍGUEZ