Ch.1 Introduction

Syllabus, prerequisites

Notation: □ Means pencil-and-paper QUIZ    ► Means coding QUIZ

Code repository for our text: https://github.com/amueller/introduction_to_ml_with_python

Why Machine Learning (ML)?

Two problems with conventional “if - else” decision systems:

• brittleness: The logic is specific to a single task or domain. Changing the task slightly may require massive changes in the code. Example: Learning chess, go, and shogi - Google’s AlphaZero.

• manual designing of rules requires knowing how humans do it.

Two big classes of ML algorithms:

• supervised: Learning from examples. Easy to understand and evaluate. E.g. Google’s “Cat” project

• unsupervised: No examples, the algorithm has to figure out the output. E.g. find out topics in blog posts.

ML lingo: Rows --> samples = data points = instances = observations

Columns --> features = attributes = dimensions = measurements

<table>
<thead>
<tr>
<th>Sepal LengthCm</th>
<th>Sepal WidthCm</th>
<th>Petal LengthCm</th>
<th>Petal WidthCm</th>
<th>Species</th>
</tr>
</thead>
<tbody>
<tr>
<td>5.1</td>
<td>3.5</td>
<td>1.4</td>
<td>0.2</td>
<td>setosa</td>
</tr>
<tr>
<td>4.9</td>
<td>3</td>
<td>1.4</td>
<td>0.2</td>
<td>setosa</td>
</tr>
<tr>
<td>4.7</td>
<td>3.2</td>
<td>1.3</td>
<td>0.2</td>
<td>setosa</td>
</tr>
<tr>
<td>4.6</td>
<td>3.1</td>
<td>1.5</td>
<td>0.2</td>
<td>setosa</td>
</tr>
<tr>
<td>5</td>
<td>3.6</td>
<td>1.4</td>
<td>0.2</td>
<td>setosa</td>
</tr>
<tr>
<td>5.4</td>
<td>3.9</td>
<td>1.7</td>
<td>0.4</td>
<td>setosa</td>
</tr>
<tr>
<td>4.6</td>
<td>3.4</td>
<td>1.4</td>
<td>0.3</td>
<td>setosa</td>
</tr>
<tr>
<td>5</td>
<td>3.4</td>
<td>1.5</td>
<td>0.2</td>
<td>setosa</td>
</tr>
<tr>
<td>4.4</td>
<td>2.9</td>
<td>1.4</td>
<td>0.2</td>
<td>setosa</td>
</tr>
<tr>
<td>4.9</td>
<td>3.1</td>
<td>1.5</td>
<td>0.1</td>
<td>setosa</td>
</tr>
<tr>
<td>5.4</td>
<td>3.7</td>
<td>1.5</td>
<td>0.2</td>
<td>setosa</td>
</tr>
<tr>
<td>4.8</td>
<td>3.4</td>
<td>1.6</td>
<td>0.2</td>
<td>setosa</td>
</tr>
<tr>
<td>4.8</td>
<td>3</td>
<td>1.4</td>
<td>0.1</td>
<td>setosa</td>
</tr>
<tr>
<td>4.3</td>
<td>3</td>
<td>1.1</td>
<td>0.1</td>
<td>setosa</td>
</tr>
</tbody>
</table>
ML is only one part of the Data Science flowchart or pipeline:

[Source: https://en.wikipedia.org/wiki/Exploratory_data_analysis]
Why Python?

Installing *Scikit-learn* → It is prepackaged in the Anaconda distribution, so install Anaconda3 first:

![Anaconda3 Installation Screen]

You should see the new Anaconda3 entry in the Start menu (Windows):

Note: Sometimes the Anaconda Prompt is not showing in the Start menu, although the shortcut can be found on disk here:

![Anaconda3 Programs Directory]

Copy it, and paste on the desktop, or attach it to the Windows taskbar.

Use the Anaconda Prompt to manage the installation:

- The simplest way to update all packages we need is by updating Scikit-learn:
  ```bash
conda update scikit-learn
```
  In the best case, we get the message `# All requested packages already installed.`, otherwise software is downloaded and installed.

- If instead we use the plain Windows Prompt, we may get the error `'conda is not recognized ...` , in which case we have to provide the entire path to `conda`; since Python was installed as part of Anaconda3, the path is `c:\anaconda3\Scripts.`
Essential Libraries and Tools

Skip Jupyter Notebook → instead, we shall use the Spyder editor + iPython console combination, that comes pre-packaged in Anaconda.

Numpy

The instructor will provide ample material for reviewing Python and numpy, or you can use any of the numerous online tutorials and references.

Review problems for numpy¹:

► Create the following numpy arrays:

- Hint: Use `np.eye` and/or `np.diag`. Do you know the two different forms of `np.diag`?

\[
\begin{bmatrix}
1 & 0 & 0 & 0 \\
0 & 1 & 0 & 0 \\
0 & 0 & 1 & 0 \\
0 & 0 & 0 & 1 \\
\end{bmatrix}
\begin{bmatrix}
1 & 2 & 0 & 0 \\
0 & 1 & 2 & 0 \\
0 & 0 & 1 & 2 \\
0 & 0 & 0 & 1 \\
\end{bmatrix}
\begin{bmatrix}
1 & 2 & 3 & 0 \\
0 & 1 & 2 & 3 \\
0 & 0 & 1 & 2 \\
0 & 0 & 0 & 1 \\
\end{bmatrix}
\begin{bmatrix}
1 & 0 & 0 & 0 \\
0 & 1 & 0 & 0 \\
0 & 0 & 1 & 2 \\
0 & 0 & 0 & 1 \\
\end{bmatrix}
\begin{bmatrix}
1 & 0 & 0 & 0 \\
0 & 1 & 0 & 0 \\
0 & 0 & 1 & 2 \\
0 & 0 & 0 & 1 \\
\end{bmatrix}
\begin{bmatrix}
3 & 0 & 1 & 0 \\
0 & 3 & 0 & 1 \\
0 & 0 & 0 & 1 \\
0 & 0 & 0 & 1 \\
\end{bmatrix}
\]

► Change the diagonals into antidiagonals, E.g. the last one from above should be

- Hint: Use `np.fliplr`. What happens if we use `np.flipud` instead?

\[
\begin{bmatrix}
0 & 0 & 0 & 1 \\
0 & 0 & 1 & 0 \\
0 & 0 & 1 & 0 \\
1 & 0 & 3 & 0 \\
\end{bmatrix}
\]

► Create this block-diagonal array:

- Hint: Use `np.zeros`, `np.hstack` and `np.vstack`.

\[
\begin{bmatrix}
1 & 2 & 0 & 0 \\
3 & 4 & 0 & 0 \\
0 & 0 & 1 & 2 \\
0 & 0 & 4 & 5 \\
0 & 0 & 7 & 8 \\
\end{bmatrix}
\]

¹Do not forget to `import` numpy, with the customary alias `np`!
Matplotlib

```python
import matplotlib.pyplot as plt

# Generate a sequence of numbers from -10 to 10 with 100 steps in between
x = np.linspace(-10, 10, 100)
# Create a second array using sine
y = np.sin(x)
# The plot function makes a line chart of one array against another
plt.plot(x, y, marker="x")
```

Review problems for Matplotlib:

► Plot the function \( f(t) = \frac{50t - 6}{1 + t^2} \) for values of \( x \) between -5 and 14, spaced 1 apart. Use only plain Python lists, and the `plot` function from Matplotlib’s `pyplot`.

► Same as above, but use numpy arrays instead of lists.
  - Hint: Use `np.arange`.

► Plot a histogram of 100 random numbers drawn from a Normal distribution with mean 4 and standard deviation 2. There should be 15 bins.
  - Hint: Use `np.random.normal` to generate the random numbers, and the `hist` function from Matplotlib’s `pyplot`.

covered pp.1-10 (skip Scipy, to be covered next time)
Solutions (for brevity, the import and print statements are sometimes omitted)

Create the following numpy arrays:

The following code shows several approaches – the function `diag` is more general, because it can create diagonals with non-identical numbers:

```
import numpy as np

arr1 = np.eye(5, dtype=np.int8)
arr2 = 2*np.eye(5, dtype=np.int8, k=1)
arr3 = np.diag([3]*3, k=2)
arr_3 = np.diag(np.array([3]*3), k=-2)

print(arr1, '\n')
print(arr1 + arr2, '\n')
print(arr1 + arr2 + arr3, '\n')
print(arr1 + arr_3)
```

Change the diagonals into antidiagonals, E.g. the last one from above should be

- Hint: Use `np.fliplr`. What happens if we use `np.flipud` instead?

```
arr1 = np.eye(5, dtype='int8')
arr2 = np.diag(np.array([2]*4), k=1)
arr3 = np.diag(np.array([3]*3), k=2)
arr_3= np.diag(np.array([3]*3), k=-2)

print(np.fliplr(arr1 + arr_3), '\n')
print(np.flipud(arr1 + arr_3), '\n')
```

Create this numpy array:

```
a1 = np.arange(1, 5, dtype='int32').reshape(2,2)
a2 = np.zeros((2,3), dtype='int32')
a3 = np.zeros((3,2), dtype='int32')
a4 = np.arange(1,10, dtype='int32').reshape(3,3)
a12 = np.hstack((a1, a2))
a34 = np.hstack((a3, a4))
a1234 = np.vstack((a12, a34))
```
Plot the function \( f(t) = \frac{50t - 6}{1 + t^2} \) for values of \( t \) between -5 and 14, spaced 1 apart. Use only plain Python lists, and the \texttt{plot} function from Matplotlib's \texttt{pyplot}.

```python
from matplotlib import pyplot as plt

def fun(t):
    return (50*t - 6)/(1+t*t)

x_val = range(-5, 15)
y_val = [fun(x) for x in x_val]
plt.plot(x_val, y_val, color='red', marker='x')
# plt.show()  # not needed when using Spyder/IPython
```

Same as above, but using numpy arrays instead of lists.

Plot a histogram of 300 random numbers drawn from a Normal distribution with mean 4 and standard deviation 2. There should be 15 bins.

- Hint: Use \texttt{np.random.normal} to generate the random numbers, and the \texttt{hist} function from Matplotlib's \texttt{pyplot}.

```python
arr = np.random.normal(4.0, 2.0, 300)
plt.hist(arr, bins=15, color='green')
```
Instructor's notes  Ch.1 - Introduction

Scipy - Sparse Matrices

Our text briefly shows how to use the CSR and COO representations (or forms, for short) for **sparse matrices**. Here we explain the two forms in more detail. First, let us visualize the difference between sparse and dense matrices using *matplotlib*'s *spy* function, which plots the sparsity pattern on a 2-D array:

```python
import numpy as np
import matplotlib.pyplot as plt

mat_sparse = np.random.binomial(1, 0.1, 1000000).reshape(500,2000)
plt.figure(figsize=(10,8))
plt.spy(mat_sparse)

mat_dense = np.random.binomial(1, 0.95, 1000000).reshape(500,2000)
plt.figure(figsize=(10,8))
plt.spy(mat_dense)
```

--

**Note on the `binomial` function**: 2

`numpy.random.binomial(n, p, size=None)`

Draw samples from a binomial distribution.

Samples are drawn from a binomial distribution with specified parameters, *n* trials and *p* probability of success where *n* an integer >= 0 and *p* is in the interval [0,1]. (*n* may be input as a float, but it is truncated to an integer in use)

**Parameters:**

- **n**: `int or array_like of ints`
  
  Parameter of the distribution, >= 0. Floats are also accepted, but they will be truncated to integers.

- **p**: `float or array_like of floats`
  
  Parameter of the distribution, >= 0 and <=1.

- **size**: `int or tuple of ints, optional`
  
  Output shape. If the given shape is, e.g., `(m, n, k)`, then `m * n * k` samples are drawn. If size is *None* (default), a single value is returned if *n* and *p* are both scalars. Otherwise, `np.broadcast(n, p).size` samples are drawn.

**Returns:**

- **out**: `ndarray or scalar`
  
  Drawn samples from the parameterized binomial distribution, where each sample is equal to the number of successes over the *n* trials.

---

2https://docs.scipy.org/doc/numpy-1.15.0/reference/generated/numpy.random.binomial.html
A. Sparse matrices - COO (COOrdinate list) form

We use for a first example the dense matrix from the Wikipedia page shown here:

\[
\begin{pmatrix}
0 & 0 & 0 & 0 \\
5 & 8 & 0 & 0 \\
0 & 0 & 3 & 0 \\
0 & 6 & 0 & 0 \\
\end{pmatrix}
\]

The COO form is:

```
import numpy as np
from scipy import sparse

data = np.array([5, 8, 3, 6])
row = np.array([1, 1, 2, 3])
col = np.array([0, 1, 2, 1])
sm = sparse.coo_matrix((data, (row, col)), shape=(4, 4))
print(sm)
print(sm.todense())
```

If needed, the three members of \( \text{sm} \) can be accessed individually, and even modified:

```
print(sm.data)
print(sm.row)
print(sm.col)
sm.data[0] = 42
sm.row[0] = 0
print(sm.todense())
```

Note well: In COO matrices, it is not possible to access elements (for reading or writing) directly, using square brackets and subscripts:

```
print(sm[0,3])
```

```
TypeError: 'coo_matrix' object is not subscriptable
```

According to the documentation, the COO format is used mainly for fast conversion to other sparse formats, like CSR (explained below), and for fast operations on the underlying \text{data} array of non-zero values. It is also possible to perform more complex operations, like matrix-vector multiplications, with a COO matrix, but we are going to only do it with CSR matrices.

\[\square\] How would you add 42 to all non-zero elements of a COO matrix?

We can find the total size of a matrix using numpy's \text{nbytes} attribute (No, the sparse matrix object does not have its own nbytes):

```
print('total nr. of Bytes compressed: ')
print(sm.data.nbytes + sm.row.nbytes + sm.col.nbytes)
print('total nr. of Bytes dense : ')
print(sm.todense().nbytes)
```

Create a COO representation for the identity matrix of size 10x10. Find the compression ratio.

---

\(^3\)https://en.wikipedia.org/wiki/Sparse_matrix#Coordinate_list_(COO)
\(^4\)http://www.scipy-lectures.org/advanced/scipy_sparse/coo_matrix.html
Create a COO representation for this matrix of size \( N \times N \), where \( N \) is entered by the user.

- Hint: Use the solution to the problem in the numpy section above.

Calculate the compression ratio.

Write a loop that iterates \( N \) from 2 to 20. What do you notice about the compression ratio?

### B. Sparse matrices - CSR (Compressed Sparse Row) form

For the same example dense matrix, the CSR form is:

\[
\begin{bmatrix}
0 & 0 & 0 & 0 \\
5 & 8 & 0 & 0 \\
0 & 0 & 3 & 0 \\
0 & 6 & 0 & 0
\end{bmatrix}
\]

The array \texttt{indptr}:

- Has a number of elements equal to the number of rows plus one. There is one entry for each row, plus the final value.
- Its first element is always zero, and its last is always equal to the number of nonzero elements; this is redundant, but it makes access easier, and, for realistic-sized matrices, the overhead is negligible\(^5\).
- The entry with index \( i \) stores how many non-zero elements are in the matrix up to and including the previous row, \( i - 1 \); it is a cumulative count of nonzero elements.

The array \texttt{indices} is the same as \texttt{col} in the COO form.

What is the dense matrix represented here? Do we have enough information to recover the dense matrix without ambiguity?

**Code example:**

```python
import numpy as np

data = np.array([5, 8, 3, 6])  # same as data from COO
indptr = np.array([0, 0, 2, 3, 4])  # compressed row from COO
indices = np.array([0, 1, 2, 1])  # same as col in COO

sm = sparse.csr_matrix((data, indices, indptr), shape=(4, 4))
print(sm)
print(sm.todense())
```

**Note well:**

- Unlike COO, in the CSR constructor all three arrays are part of the same tuple!
- \( \texttt{indices} \) is placed before \( \texttt{indptr}! \)

\(^5\)In Big Oh notation, the overhead is \( O(1) \).
The constructor `csr_matrix()` accepts other arguments as well, e.g. a dense 2D array, or any of the other sparse representations, like COO.

As with all dense matrices, the three members of `sm` can be accessed individually, and modified:

```python
print sm.data
print sm.indptr
print sm.indices
sm.data[0] = 42
sm.indptr[1] = 1
print sm.todense()
```

► Find the size in bytes of the CSR matrix `sm` above. Calculate the compression ratio.

Note: Unlike COO, CSR allows individual elements to be accessed with square braces and indices:

```python
print('Element:', sm[1,2])
sm[1,2] = 42
print('Element:', sm[1,2])
```

However, the first time we try to write an element in this way, we get a warning:

```
C:\Anaconda3\lib\site-packages\scipy\sparse\compressed.py:746: SparseEfficiencyWarning: Changing the sparsity structure of a csr_matrix is expensive. lil_matrix is more efficient.
  SparseEfficiencyWarning)
```

Note: `lil_matrix` is another sparse representation, which is more efficient than COO and CSR when incremental changed to the matrix are performed. For more details consult the Scipy documentation here:

Solutions

- Find the size in bytes of the CSR matrix \texttt{sm} above. Calculate the compression ratio.

```python
size_sparse = sm.data.nbytes + sm.indptr.nbytes + sm.indices.nbytes
print('Size of CSR matrix:', size_sparse)
size_dense = sm.todense().nbytes
ratio = size_sparse/float(size_dense)
print('Ratio:', ratio)
```

What is the dense matrix represented here? Do we have enough information to specify the matrix completely?

A: Unlike COO, the numbers of rows is known - it is one less that the number of elements of \texttt{indptr} -, however, like COO, the number of columns can be arbitrarily large. For safety, the constructor still requires both dimensions - see the constructor usage following this problem.
pandas

For the code examples from the text, you can copy and paste from the code repository found here: https://github.com/amueller/introduction_to_ml_with_python

The pandas module has two main objects: the Series and the (Data)Frame. The latter is more useful in machine learning. Among many other options, the constructor for pandas’ Frame accepts a dictionary:

```python
import pandas as pd
from IPython.display import display

data = {
    'Name': ['John', 'Anna', 'Peter', 'Linda'],
    'Location': ['New York', 'Paris', 'Berlin', 'London'],
    'Age': [24, 13, 53, 33]
}

df = pd.DataFrame(data)
display(df)
```

Notes: For compactness, the variable name `data_pandas` used in the text was changed to `df`. In the iPython console, the function `display` produces the same output as `print`.

One advantage of pandas over numpy is that each column can have a different data type. Even inside a column, different data types may be present. We say that Frames are heterogeneous.

► Replace London with the integer 42 in the example above

```python
data = {
    'Name': ['John', 'Anna', 'Peter', 'Linda'],
    'Location': ['New York', 'Paris', 'Berlin', 42],
    'Age': [24, 13, 53, 33]
}
```

Write code to show that the element of the column `Location` are of different types.

This being said, in ML applications the columns of a Frame are each homogenous, although the data type may change between columns. For example, in the text example the first two columns are comprised exclusively of strings, and the third has only integers.

The construction above, based on a dictionary is the most logical, since Frames are best thought of as dictionaries of Series; this is confirmed by the syntax for selecting a column, shown below. Multiple columns are selected by providing the column names as a list or tuple:

```python
print df['Name']
print df[['Name', 'Age']]
```

► Extract only the `Age` column and find the average age.

• Hint: Use the Frame method `mean`. 

```python
print df['Age']
```
Solutions:

- Replace *London* with the integer 42 in the example above

```python
data = {'Name': ['John', 'Anna', 'Peter', 'Linda'],
       'Location': ['New York', 'Paris', 'Berlin', 42],
       'Age': [24, 13, 53, 33]}
```

Write code to show that the element of the column *Location* are of different types.

```python
print(type(df['Location'][0]))  # type 'str'
pd.print(type(df['Location'][3]))  # type 'int'
```

- Extract the *Age* column and find the average age.

```python
print(df['Age'].mean())  # 30.75
```
mgelearn

First try this command in the *iPython* console:

```python
In [1]: import mgelearn
In [1]:
```

If you do not see any error message, it means that *mgelearn* was already installed, so you can skip the steps below.

Install *mgelearn* by entering this command in the Anaconda Prompt console (*pip* stands for Python Install Package):

```
> pip install mgelearn
```

Note: If you are using the Windows Prompt console, you will get this error:

```
> pip install mgelearn
'pip' is not recognized as an internal or external command, operable program or batch file.
```

You have to provide the entire path to *pip*, which, if you installed Python as part of Anaconda 3, looks like this:

```
> c:\anaconda3\scripts\pip install mgelearn
```

---

**Python 2 vs. Python 3**

One important difference between Python 2 and 3 is that, in Python 3, *print* must always be a function, i.e. written with parentheses:

```python
a = 42
print('a =', a)  # a = 42
```

In Python 2, we can write *print* either as a statement (no parentheses) or as a function, but, in the latter case, the parentheses are interpreted as those of a *tuple*:

```python
a = 42
print 'a =', a  # a = 42
print('a =', a)  # ('a =', 42)
```

To emulate the Python 3 behavior in Python 2, we can set a flag like this:

```python
from __future__ import print_function
a = 42
print('a =', a)  # a = 42
```

See other major differences here: [https://wsvincent.com/python2-vs-python3/](https://wsvincent.com/python2-vs-python3/)
Instructor’s notes

Versions Used in this Book

As we know, modules have to be imported prior to use:

```python
import numpy
import matplotlib
```

All `non-standard` modules have a `version` attribute enclosed in between double underscores:

```python
print(numpy.__version__)  # 1.15.4
print(matplotlib.__version__)  # 3.0.2
```

For `standard` modules, the rules are different. According to PEP (Python Enhancement Proposal) 396: “[…] modules in the standard library SHOULD NOT have version numbers. They implicitly carry the version number of the Python release they are included in.” For this reason, the standard modules do not have a `__version__` attribute; instead, the version of the entire Python release is available as the attribute `version` (no underscores!) in the module `sys`:

```python
import sys
print(sys.version)
```

```
3.6.7  Anaconda, Inc.  (default, Dec 10 2018, 20:35:02) [MSC v.1915 64 bit (AMD64)]
```

The Python Standard Library is documented here: [https://docs.python.org/3/library/index.html](https://docs.python.org/3/library/index.html)

Formatting

Two ways to print with formatting:

- **Old way**, using C-style “printf” formatting specifiers:
  ```python
  print('pandas ver.  %s' % (pandas.__version__))
  print('pandas ver.  %s' % (pandas.__version__))
  ```

  ```
  pandas ver.  0.21.0
  pandas ver.  0.21.0
  ```

- **New way**, using the method `format()`:
  ```python
  print('pandas ver.  {}.format(pandas.__version__)')
  print('pandas ver.  {}.format(pandas.__version__)')
  ```

  ```
  Same output.
  ```

One of the many Monthly Python references from the documentation:

```python
print('We are the {} who say "{}!"'.format('knights', 'Ni'))
```

```python
We are the knights who say "Ni!"
```

Two capabilities of the new style that are not available in the old style:

---

6The complete text of PEP 396 Module Version Numbers can be found here: [https://www.python.org/dev/peps/pep-0396/](https://www.python.org/dev/peps/pep-0396/)

7[https://docs.python.org/3/tutorial/inputoutput.html](https://docs.python.org/3/tutorial/inputoutput.html)
• By numbering the placeholders, it is possible to change the print order without physically changing the arguments of the format method:

```python
print('{{} {}}'.format(42, 43))
print('{{0} {1}}'.format(42, 43))
print('{{1} {0}\n'.format(42, 43))
```

• (Only) The new style placeholders accept data structures:

```python
arr = np.arange(1,10).reshape((3,3))
print('{{} is an array}'.format(arr))
```

For a complete list of options and differences, see here: [https://pyformat.info/](https://pyformat.info/)
Another example with integers\(^8\):

```
>>> for x in range(1,11):
    ...    print '{0:2d} {1:3d} {2:4d}'.format(x, x*x, x*x*x)
...
1  1  1
2  4  8
3  9 27
4 16 64
5 25 125
6 36 216
7 49 343
8 64 512
9 81 729
10 100 1000
```

\(^8\) Ibid.
Solutions:

Print a table similar to the one above, with the square and cubic roots instead of the squares and cubes. The numbers are the integers from 0 to 500, in steps of 50. Allow for 3 decimal places. Use the new formatting style. The table should also have column headers.

```
print('{0:4s}{1:8s}{2:8s}'.format('nr', 'sqr', 'qurt'))
for x in range(0, 501, 50):
    print('{0:4d}{1:8.3f}{2:8.3f}'.format(x, x**.5, x**(1/3)))
```
The constructor `load_iris()` returns an object of class *Bunch*, which is essentially a dictionary whose keys can also be accesses like attributes:

```python
from sklearn.datasets import load_iris
iris_dataset = load_iris()

iris_dataset['Hello'] = 'World'
print(iris_dataset['Hello'])
```

Here are the original keys (without the two extra ones from the example above):

```python
print(iris_dataset.keys())
```

▶ Read, enter and execute the code on pp.15-17 ([In][11] to [In][19]) that explains the five keys/attributes outlined above.

Conclusion: The “important” attributes are *data* and *target*. Explain them in your own words:

- data -->
- target -->

The two parts of any ML algorithm: **training and testing**

Two dangers: **underfitting** and **overfitting**.

Split the dataset into two parts: train set and test set.

```python
from sklearn.model_selection import train_test_split

X_train, X_test, y_train, y_test = train_test_split(  
    iris_dataset.data, iris_dataset.target, random_state=0,  
    train_size = 0.75)  
#ratio is new in version 21!

print('X_train.shape: {}'.format(X_train.shape))
print('y_train.shape: {}'.format(y_train.shape))
print('X_test.shape: {}'.format(X_test.shape))
print('y_test.shape: {}'.format(y_test.shape))
```

►Try the code above with different ratios of train, e.g. 0.8, which is the “80-20 rule” or the “Pareto principle”.

Note: A third part of ML algorithms, **cross-validation**, will be introduced in Ch.5.
Visualizing the data

For 2 (or 3) features, one 2D (or 3D) plot is sufficient. For higher numbers of features, we can plot each pair. Pandas' `scatter_matrix` is very handy:

```python
iris_df = pd.DataFrame(X_train, columns=iris_dataset.feature_names)
pd.plotting.scatter_matrix(iris_df, c=y_train, figsize=(12,12), marker='o', hist_kwds={'bins':20}, s=60, alpha=.8, cmap=mgelearn.cm3)
```

What do you notice in the matrix above? Explain!
K-Nearest-Neighbors (KNN or kNN)

Explaining KNN on a simple example (not in text):

**Training or fitting** the classifier

```python
from sklearn.neighbors import KNeighborsClassifier
knn = KNeighborsClassifier(n_neighbors=1)
print(knn.fit(X_train, y_train))
```

KNeighborsClassifier(algorithm='auto', leaf_size=30, metric='minkowski',
metric_params=None, n_jobs=1, n_neighbors=1, p=2,
weights='uniform')

Normally, we do not print the classifier options:

```python
from sklearn.neighbors import KNeighborsClassifier
knn = KNeighborsClassifier(n_neighbors=1)
knn.fit(X_train, y_train)
```

**Making predictions**

```python
X_new = np.array([[5, 2.9, 1, 0.2]])
prediction = knn.predict(X_new)
print prediction
#to see what the prediction means in real life,
#use the target_names value/attribute:
prediction
print iris_dataset.target_names[prediction]
```

[0]
['setosa']
We have a second flower, with the four features, respectively, 4, 1.9, 3, 0.7. Use the 2D array to predict both in one pass.

Evaluating the model

```python
y_pred = knn.predict(X_test)
print y_pred[:10]  #first 10 predictions
```

```python
print np.mean(y_pred == y_test)
#or
print knn.score(X_test, y_test)
```

Note: `fit()`, `predict()`, and `score()` are common to all supervised algorithms in `scikit-learn`. 
Solutions

We have a second flower, with the four features, respectively, 4, 1.9, 3, 0.7. Use the 2D array to predict both in one pass.

```python
X_new = np.array([[5, 2.9, 1, 0.2],
                  [4, 1.9, 3, 0.7]])
prediction = knn.predict(X_new)
print(prediction)

# to see what the prediction means in real life,
# use the target_names value/attribute:
print(iris_dataset.target_names[prediction])
```

[0 1]
['setosa' 'versicolor']