Computational Tools for Big Data — Python Libraries

Finn Årup Nielsen

DTU Compute
Technical University of Denmark

September 15, 2015

Overview

Numpy — numerical arrays with fast computation

Scipy — computation science functions

Scikit-learn (sklearn) — machine learning

Pandas — Annotated numpy arrays

Cython — write C program in Python

Python numerics

The problem with Python:

```
>>> [1, 2, 3] * 3
```

Python numerics

The problem with Python:

```
>>> [1, 2, 3] * 3
[1, 2, 3, 1, 2, 3, 1, 2, 3]             # Not [3, 6, 9] !
```

Python numerics

The problem with Python:

```
>>> [1, 2, 3] * 3
[1, 2, 3, 1, 2, 3, 1, 2, 3]             # Not [3, 6, 9] !

>>> [1, 2, 3] + 1                       # Wants [2, 3, 4] ...
```

Python numerics

The problem with Python:

```
>>> [1, 2, 3] * 3
[1, 2, 3, 1, 2, 3, 1, 2, 3]                    # Not [3, 6, 9] !

>>> [1, 2, 3] + 1                              # Wants [2, 3, 4] ...
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: can only concatenate list (not "int") to list
```

Python numerics

The problem with Python:

```
>>> [1, 2, 3] * 3
[1, 2, 3, 1, 2, 3, 1, 2, 3]                # Not [3, 6, 9] !

>>> [1, 2, 3] + 1                          # Wants [2, 3, 4] ...
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: can only concatenate list (not "int") to list

>>> # Matrix multiplication
>>> [[1, 2], [3, 4]] * [[5, 6], [7, 8]]
```

Python numerics

The problem with Python:

```
>>> [1, 2, 3] * 3
[1, 2, 3, 1, 2, 3, 1, 2, 3]               # Not [3, 6, 9] !

>>> [1, 2, 3] + 1                         # Wants [2, 3, 4] ...
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: can only concatenate list (not "int") to list

>>> # Matrix multiplication
>>> [[1, 2], [3, 4]] * [[5, 6], [7, 8]]
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: can't multiply sequence by non-int of type 'list'
```

map(), reduce() and filter()

Poor-man's vector operations: the `map()` built-in function:

```
>>> map(lambda x: 3*x, [1, 2, 3])
[3, 6, 9]
```

`lambda` is for an anonymous function, `x` the input argument, and `3*x` the function and the return argument. Also possible with ordinary functions:

```
>>> from math import sin, pow
>>> map(sin, [1, 2, 3])
[0.8414709848078965, 0.9092974268256817, 0.1411200080598672]

>>> map(pow, [1, 2, 3], [3, 2, 1])
[1.0, 4.0, 3.0]
```

. . . or with comprehensions

List comprehensions:

```
>>> [3*x for x in [1, 2, 3]]
[3, 6, 9]
```

Generator

```
>>> (3*x for x in [1, 2, 3])
<generator object <genexpr> at 0x7f9060a72eb0>
```

Problem

But control structures are usually slow in Python.

Solution: Numpy

```
>>> from numpy import *
```

Solution: Numpy

```
>>> from numpy import *
>>> array([1, 2, 3]) * 3
```

Solution: Numpy

```
>>> from numpy import *
>>> array([1, 2, 3]) * 3
array([3, 6, 9])
```

Solution: Numpy

```
>>> from numpy import *
>>> array([1, 2, 3]) * 3
array([3, 6, 9])
>>> array([1, 2, 3]) + 1
```

Solution: Numpy

```
>>> from numpy import *
>>> array([1, 2, 3]) * 3
array([3, 6, 9])
>>> array([1, 2, 3]) + 1
array([2, 3, 4])
```

Solution: Numpy

```
>>> from numpy import *
>>> array([1, 2, 3]) * 3
array([3, 6, 9])
>>> array([1, 2, 3]) + 1
array([2, 3, 4])
>>> array([[1, 2], [3, 4]]) * [[5, 6], [7, 8]]
```

Solution: Numpy

```
>>> from numpy import *
>>> array([1, 2, 3]) * 3
array([3, 6, 9])
>>> array([1, 2, 3]) + 1
array([2, 3, 4])
>>> array([[1, 2], [3, 4]]) * [[5, 6], [7, 8]]
array([[ 5, 12],
       [21, 32]])
```

Solution: Numpy

```
>>> from numpy import *
>>> array([1, 2, 3]) * 3
array([3, 6, 9])
>>> array([1, 2, 3]) + 1
array([2, 3, 4])
>>> array([[1, 2], [3, 4]]) * [[5, 6], [7, 8]]
array([[ 5, 12],
       [21, 32]])
```

This is elementwise multiplication. . .

Solution: Numpy

```
>>> from numpy import *
>>> array([1, 2, 3]) * 3
array([3, 6, 9])
>>> array([1, 2, 3]) + 1
array([2, 3, 4])
>>> array([[1, 2], [3, 4]]) * [[5, 6], [7, 8]]
array([[ 5, 12],
       [21, 32]])
```

This is elementwise multiplication. . .

```
>>> matrix([[1, 2], [3, 4]]) * [[5, 6], [7, 8]]
```

Solution: Numpy

```
>>> from numpy import *
>>> array([1, 2, 3]) * 3
array([3, 6, 9])
>>> array([1, 2, 3]) + 1
array([2, 3, 4])
>>> array([[1, 2], [3, 4]]) * [[5, 6], [7, 8]]
array([[ 5, 12],
       [21, 32]])
```

This is elementwise multiplication. . .

```
>>> matrix([[1, 2], [3, 4]]) * [[5, 6], [7, 8]]
matrix([[19, 22],
        [43, 50]])
```

Numpy arrays . . .

There are two basic types array and matrix:

An array may be a vector (one-dimensional array)

```
>>> from numpy import *
>>> array([1, 2, 3, 4])
array([1, 2, 3, 4])
```

Or a matrix (a two-dimensional array)

```
>>> array([[1, 2], [3, 4]])
array([[1, 2],
       [3, 4]])
```

. . . Numpy arrays . . .

Or a higher-order tensor. Here a 2-by-2-by-2 tensor:

```
>>> array([[[1, 2], [3, 4]], [[5, 6], [7, 8]]])
array([[[1, 2],
        [3, 4]],

       [[5, 6],
        [7, 8]]])
```

N, $1 \times N$ and $N \times 1$ `array`-s are different:

```
>>> array([[1, 2, 3, 4]])[2]     # There is no third row
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
IndexError: index out of bounds
```

. . . Numpy arrays

A `matrix` is always two-dimensional, e.g., this

```
>>> matrix([1, 2, 3, 4])
matrix([[1, 2, 3, 4]])
```

is a two-dimensional data structure with one row and four columns

. . . and with a 2-by-2 matrix:

```
>>> matrix([[1, 2], [3, 4]])
matrix([[1, 2],
        [3, 4]])
```

Numpy arrays copy

To copy by reference use `asmatrix()` or `mat()` (from an `array`) or `asarray()` (from a `matrix`)

```
>>> a = array([1, 2, 3, 4])
>>> m = asmatrix(a)                    # Copy as reference
>>> a[0] = 1000
>>> m
matrix([[1000,    2,    3,    4]])
```

To copy elements use `matrix()` or `array()`

```
>>> a = array([1, 2, 3, 4])
>>> m = matrix(a)                      # Copy elements
>>> a[0] = 1000
>>> m
matrix([[1, 2, 3, 4]])
```

Datatypes

Elements are 4 bytes integers or 8 bytes float per default:

```
>>> array([1, 2, 3, 4]).itemsize     # Number of bytes for each
4
>>> array([1., 2., 3., 4.]).itemsize
8
>>> array([1., 2, 3, 4]).itemsize   # Not heterogeneous
8
```

`array` and `matrix` can be called with datatype to set it otherwise:

```
>>> array([1, 2], 'int8').itemsize
1
>>> array([1, 2], 'float32').itemsize
4
```

Initialization of arrays

Functions `ones()`, `zeros()`, `eye()` (also `identity()`), `linspace()` work "as expected" (from Matlab), though the first argument for `ones()` and `zeros()` should contain the size in a list or tuple:

```
>>> zeros((1, 2))                    # zeros(1, 2) doesn't work
array([[ 0.,  0.]])
```

To generated a list of increasing numbers:

```
>>> r_[1:11]
array([1, 2, 3, 4, 5, 6, 7, 8, 9, 10])
>>> arange(1, 11)
array([1, 2, 3, 4, 5, 6, 7, 8, 9, 10])
```

Diagonal . . .

The `diagonal()` function works both for `matrix` and two-dimensional `array`:

```
>>> diagonal(matrix([[1, 2], [3, 4]]))
array([1, 4])
```

Note now the vector/matrix is an one-dimensional `array`

It "works" for higher order `array`s:

```
>>> diagonal(array([[[1, 2], [3, 4]], [[5, 6], [7, 8]]]))
array([[1, 7],
       [2, 8]])
```

`diagonal()` does not work for one-dimensional arrays.

. . . Diagonal

Yet another function: `diag()`. Works for one- and two-dimensional `matrix` and `array`

```
>>> m = matrix([[1, 2], [3, 4]])
>>> d = diag(m)
>>> d
array([1, 4])
>>> diag(d)
array([[1, 0],
       [0, 4]])
```

Like Matlab's `diag()`.

It is also possible to specify the diagonal: `diag(m,1)`

Matrix transpose

Matrix transpose is different with Python's `array` and `matrix` and Matlab

```
>>> A = array([[1+1j, 1+2j], [2+1j, 2+2j]]); A
array([[ 1.+1.j,  1.+2.j],
       [ 2.+1.j,  2.+2.j]])
```

`.T` for `array` and `matrix` (like Matlab ".'") and `.H` for `matrix` (like Matlab "'"):

```
>>> A.T  # No conjucation. Also: A.transpose() or transpose(A)
array([[ 1.+1.j,  2.+1.j],
       [ 1.+2.j,  2.+2.j]])

>>> mat(A).H   # Complex conjugate transpose. Or: A.conj().T
matrix([[ 1.-1.j,  2.-1.j],
        [ 1.-2.j,  2.-2.j]])
```

Matrix transpose and copy

```
>>> A = np.array([[1, 2], [3, 4]])
>>> B = A.T
>>> B[0,0] = 45
>>> A
array([[45,  2],
       [ 3,  4]])
```

B is a reference to the transposed version.

Matrix transpose and copy

```
>>> A = np.array([[1, 2], [3, 4]])
>>> B = A.T
>>> B[0,0] = 45
>>> A
array([[45,  2],
       [ 3,  4]])
```

B is a reference to the transposed version.

```
>>> A = np.array([[1, 2], [3, 4]])
>>> B = A.T.copy()                                    # Copy!
>>> B[0,0] = 45
>>> A
array([[1, 2],
       [3, 4]])
```

Sizes and reshaping

A 2-by-2-by-2 tensor:

```
>>> a = array([[[1, 2], [3, 4]], [[5, 6], [7, 8]]])
>>> a.ndim             # Number of dimensions
3

>>> a.shape            # Number of elements in each dimension
(2, 2, 2)
>>> a.shape = (1, 8); a            # Reshaping: 1-by-8 matrix
array([[1, 2, 3, 4, 5, 6, 7, 8]])
>>> a.shape = 8; a                 # 8-element vector
array([1, 2, 3, 4, 5, 6, 7, 8])
```

There is a related function for the last line:

```
>>> a.flatten()                    # Always copy
array([1, 2, 3, 4, 5, 6, 7, 8])
```

Indexing . . .

Ordinary lists:

```
>>> L = [[1, 2], [3, 4]]
>>> L[1, 1]
```

What happens here?

Indexing . . .

Ordinary lists:

```
>>> L = [[1, 2], [3, 4]]
>>> L[1, 1]
```

What happens here?

```
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
TypeError: list indices must be integers
```

Should have been `L[1][1]`. With matrices:

```
>>> mat(L)[1, 1]
```

Indexing . . .

Ordinary lists:

```
>>> L = [[1, 2], [3, 4]]
>>> L[1, 1]
```

What happens here?

```
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
TypeError: list indices must be integers
```

Should have been `L[1][1]`. With Numpy matrices:

```
>>> mat(L)[1, 1]
4
```

. . . Indexing . . .

```
>>> mat(L)[1][1]
```

What happens here?

. . . Indexing . . .

```
>>> mat(L)[1][1]
```

What happens here? Error message:

```
IndexError: index out of bounds
```

```
>>> mat(L)[1]
```

. . . Indexing . . .

```
>>> mat(L)[1][1]
```

What happens here? Error message:

```
IndexError: index out of bounds
```

```
>>> mat(L)[1]
matrix([[3, 4]])
>>> asarray(L)[1]
array([3, 4])
>>> asarray(L)[1][1]
4
>>> asarray(L)[1,1]
4
```

Indexing with multiple indices

```
>>> A = matrix([[1, 2, 3], [4, 5, 6], [7, 8, 9]])
>>> i = [1, 2]                # Second and third row/column
>>> A[i,i]
matrix([[5, 9]])              # Elements from diagonal
>>> A[ix_(i,i)]               # Block matrix
matrix([[5, 6],
        [8, 9]])
```

Take out third, “fourth” and “fifth” column with wrapping:

```
>>> A.take([2, 3, 4], axis=1, mode='wrap')
matrix([[3, 1, 2],
        [6, 4, 5],
        [9, 7, 8]])
```

Conditionals & concatenation

Construct a matrix based on a condition (first input argument)

```
>>> where(mod(A.flatten(), 2), 1, 0)      # Find odd numbers
matrix([[1, 0, 1, 0, 1, 0, 1, 0, 1]])
```

Horizontal concatenate as in Matlab "`[ A A ]`" and note tuple input!

```
>>> B = concatenate((A, A), axis=1)  # Or:
>>> B = bmat('A A')                  # Note string input. Or:
>>> hstack((A, A))
matrix([[1, 2, 3, 1, 2, 3],
        [4, 5, 6, 4, 5, 6],
        [7, 8, 9, 7, 8, 9]])
```

For concatenating rows use `vstack()`, `bmat('A ; A')` or `concatenate()`

Random initialization with `random`

Python with one element at a time with the `random` module:

```
>>> import random
>>> random.random()     # Float between 0 and 1
0.5466909536294228 8
>>> random.gauss(mu=10, sigma=3)
7.7026739697957005
```

Other probability functions: beta, exponential, gamma, lognormal, Pareto, `randint`, `randrange`, `uniform`, Von Mises and Weibull.

```
>>> a = [1, 2, 3, 4]; random.shuffle(a); a
[3, 1, 4, 2]
```

Other functions `choice`, `sample`, etc.

Random initialization with numpy

Initialize an array with random numbers between zero and one:

```
>>> import numpy.random
>>> numpy.random.random((2,3))                      # Tuple input!
array([[ 0.98872329,  0.73451282,  0.54337299],
       [ 0.69088015,  0.59413038,  0.71935909]])
```

Standard Gaussian (normal distribution):

```
>>> numpy.random.randn(2, 3)                        # Individual inp
array([[-0.19301411, -1.37092092, -0.1666896 ],
       [ 1.41485887,  2.24646526, -1.27417696]])
>>> N = numpy.random.standard_normal((2, 3))    # Tuple input!
>>> N = numpy.random.normal(0, 1, (2, 3))
```

Multiplications and divisions . . .

Numpy multiplication and divisions are confusing.

```
>>> from numpy import *
>>> A = array([[1, 2], [3, 4]])
```

With `array` the “default” is elementwise:

```
>>> A * A                    # '*' is elementwise, Matlab: A.*A
array([[ 1,  4],
       [ 9, 16]])
>>> dot(A, A)                # dot() is matrix multiplication
array([[ 7, 10],
       [15, 22]])
```

. . . Multiplications and divisions

With `matrix` the default is matrix multiplication

```
>>> mat(A) * mat(A)               # Here '*' is matrix multiplicatio
matrix([[ 7, 10],
        [15, 22]])
>>> multiply(mat(A), mat(A)) # 'multiply' is elementwise multip
matrix([[ 1,  4],
        [ 9, 16]])
>>> dot(mat(A), mat(A))           # dot() is matrix multiplication
matrix([[ 7, 10],
        [15, 22]])

>>> mat(A) / mat(A)               # Division always elementwise
matrix([[1, 1],
        [1, 1]])
```

Matrix inversion

"Ordinary" matrix inversion available as `inv()` in the `linalg` module of `numpy`

```
>>> linalg.inv(mat([[2, 1], [1, 2]]))
matrix([[ 0.66666667, -0.33333333],
        [-0.33333333,  0.66666667]])
```

Pseudo-inverse `linalg.pinv()` for singular matrices:

```
>>> linalg.pinv(mat([[2, 0], [0, 0]]))
matrix([[ 0.5,  0. ],
        [ 0. ,  0. ]])
```

Singular value decomposition

`svd()` function in the `linalg` module returns by default three argument:

```
>>> from numpy.linalg import svd
>>> U, s, V = svd(mat([[1, 0, 0], [0, 0, 0]]))
```

gives a 2-by-2 matrix, a 2-vector with singular values and a 3-by-3 matrix:

```
>>> U * diag(s) * V    # Gives not aligned error
>>> U, s, V = svd(mat([[1, 0, 0], [0, 0, 0]]),
                  full_matrices=False)
>>> U * diag(s) * V    # Now ok: V.shape == (2, 3)
```

Note $\mathbf{V}$ is transposed compared to Matlab!

If only the singular values are required use `compute_uv` argument:

```
>>> s = svd(mat([[1, 0, 0], [0, 0, 0]]), compute_uv=False)
```

Non-negative matrix factorization

```
from numpy import mat, random, multiply

def nmf(M, components=5, iterations=5000):
    """Factorize non-negative matrix."""
    W = mat(random.rand(M.shape[0], components))
    H = mat(random.rand(components, M.shape[1]))
    for n in range(iterations):
        H = multiply(H, (W.T * M) / (W.T * W * H + 0.001))
        W = multiply(W, (M * H.T) / (W * (H * H.T) + 0.001))
        print "%d/%d" % (n, iterations)
    return (W, H)
```

Two matrices are returned.

Note '0.001' needs to be set to some 'appropriate' for the dataset.

Scipy

Scipy

Scipy (scientific python) has many submodules:

Subpackage	Function examples	Description
`cluster`	`vq.keans`, `vq.vq`, `hierarchy.dendrogram`	Clustering algorithms
`fftpack`	`fft`, `ifft`, `fftfreq`, `convolve`	Fast Fourier transform, etc.
`io`	`loadmat`	Input/output functions
`optimize`	`fmin`, `fmin_cg`, `brent`	Function optimization
`signal`	`butter`, `welch`	Signal processing
`spatial`	`ConvexHull`, `Voronoi`, `distance.cityblock`	Functions for spatial data
`stats`	`nanmean`, `chi2`, `kendalltau`	Statistical functions
...		

Optimization with `scipy.optimize`

`scipy.optimize` contains functions for mathematical function optimization:

`fmin()` is Nelder-Mead simplex algorithm for function optimization without derivatives.

```
>>> import scipy.optimize, math
>>> scipy.optimize.fmin(math.cos, [1])
Optimization terminated successfully.
         Current function value: -1.000000
         Iterations: 19
         Function evaluations: 38
array([ 3.14160156])
```

Other optimizers: `fmin_cg()`, `leastsq()`, . . .

scipy optimization

Custom multidimensional function (the Rosenbrock function aka banana function):

```
def rosenbrock((x, y)):
    a, b = 1, 100
    return (a - x) ** 2 + b * (y - x ** 2) ** 2
```

Minimum of the function is at $(1, 1)$.

Optimize with Scipy and its general `minimize` function:

```
>>> from scipy.optimize import minimize
>>> result = minimize(rosenbrock, x0=(0, 0))
>>> result.x
array([ 0.99999561,  0.99999125])
```

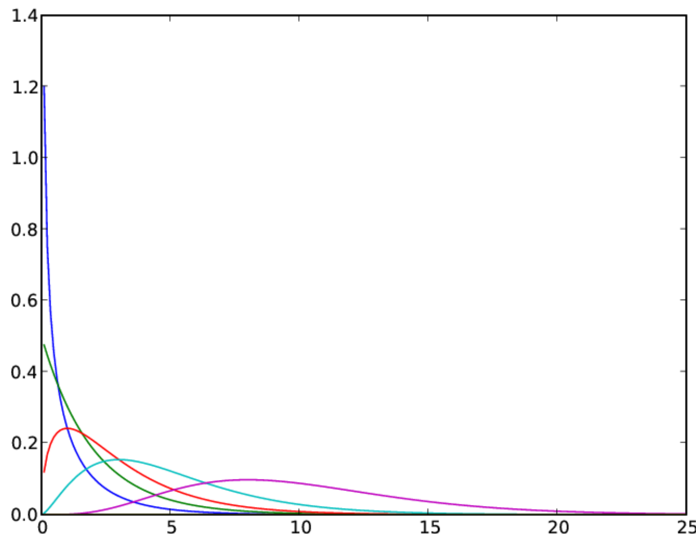
Here the gradient is estimated numerically.

```
>>> result
   status: 2
  success: False
     njev: 32
     nfev: 140
 hess_inv: array([[ 0.48552643,  0.96994585],
       [ 0.96994585,  1.94259477]])
      fun: 1.9281078336062298e-11
        x: array([ 0.99999561,  0.99999125])
  message: 'Desired error not necessarily achieved due to precision loss.'
      jac: array([ -1.07088609e-05,   5.44565446e-06])
```

Not quite there yet, trying Nelder-Mead method with many iterations:

```
>>> minimize(rosenbrock, x0=(0, 0), method='Nelder-Mead',
             tol=0, options={'maxiter': 1000})
  status: 0
    nfev: 324
 success: True
     fun: 0.0
       x: array([ 1.,  1.])
 message: 'Optimization terminated successfully.'
     nit: 168
```

Statistics with `scipy.stats`



Showing the χ^2 probablity density function with different degrees of freedom:

```
from pylab import *
from scipy.stats import chi2
x = linspace(0.1, 25, 200)
for dof in [1, 2, 3, 5, 10, 50]:
    plot(x, chi2.pdf(x, dof))
```

Other functions such as missing data mean:

```
>>> x = [1, nan, 3, 4, 5, 6, 7, 8, nan, 10]
>>> scipy.stats.stats.nanmean(x)
5.5
```

Random initialization with `scipy.stats`

10 discrete distributions and 81 continuous distributions

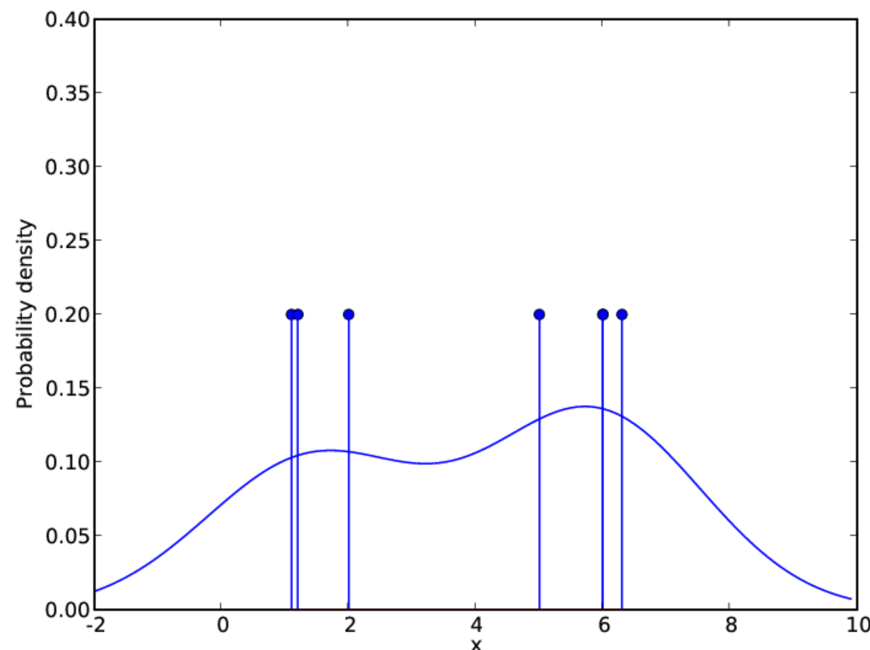
```
>>> scipy.stats.uniform.rvs(size=(2, 3))      # Uniform
array([[ 0.23273417,  0.17636535,  0.88709937],
       [ 0.07573364,  0.04084195,  0.45961136]])

>>> scipy.stats.norm.rvs(size=(2, 3))         # Gaussian
array([[ 0.89339055, -0.05093851,  0.12449392],
       [ 0.49639535, -1.39487053,  0.38580828]])
```

Kernel density estimation from scipy.stats...

```
from pylab import *; import scipy.stats

x0 = array([1.1, 1.2, 2, 6, 6, 5, 6.3])
x  = arange(-3, 10, 0.1)
plot(x, scipy.stats.gaussian_kde(x0).evaluate(x))
```

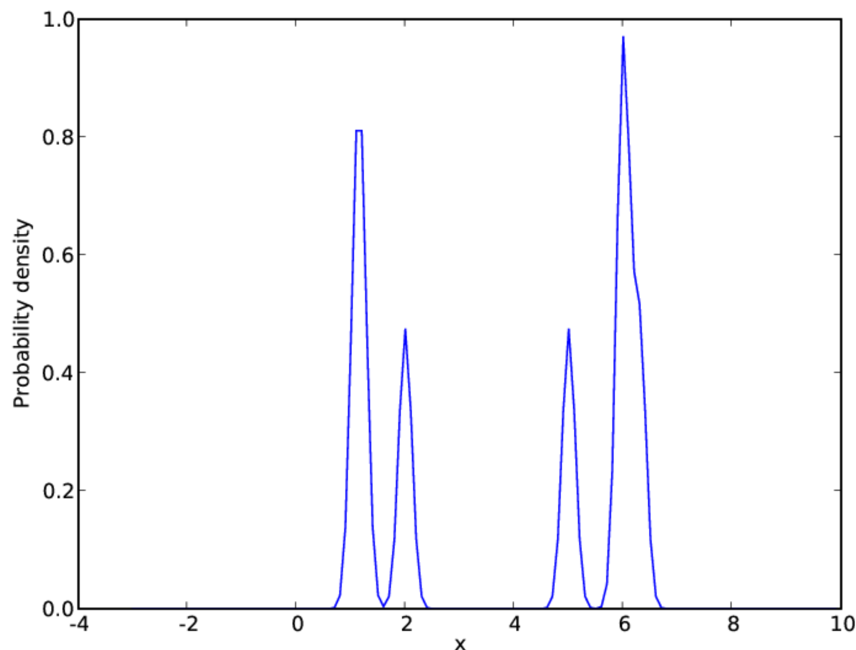


Bandwidth computed via 'scott's factor'...?

Estimation in higher dimensions possible

. . . Kernel density estimation

```
>>> class kde(scipy.stats.gaussian_kde):
...     def covariance_factor(dummy): return 0.05
...
>>> plot(x, kde(x0).evaluate(x))
```



Defining a new class “kde” from the gaussian_kde class from the scipy.stats.gaussian_kde module with

Fourier transformation

```
>>> x = asarray([[1] * 4, [0] * 4] * 100).flatten()
```

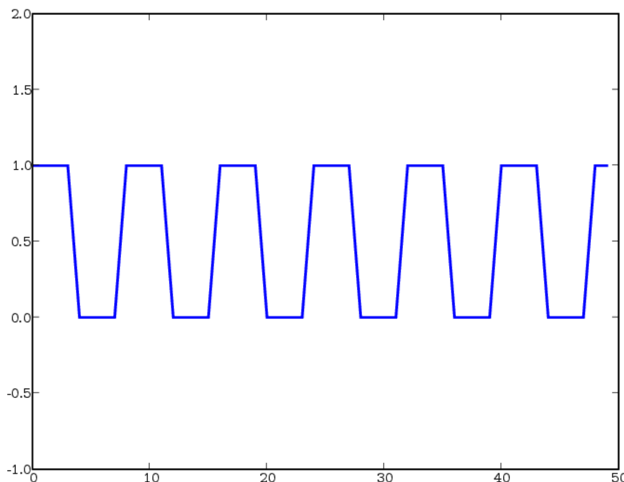
What is `x`?

Fourier transformation . . .

```
>>> x = asarray([[1] * 4, [0] * 4] * 100).flatten()
```

What is `x`?

```
>>> x[:20]
array([1, 1, 1, 1, 0, 0, 0, 0, 1, 1, 1, 1, 0, 0, 0, 0, 1, 1, 1
```

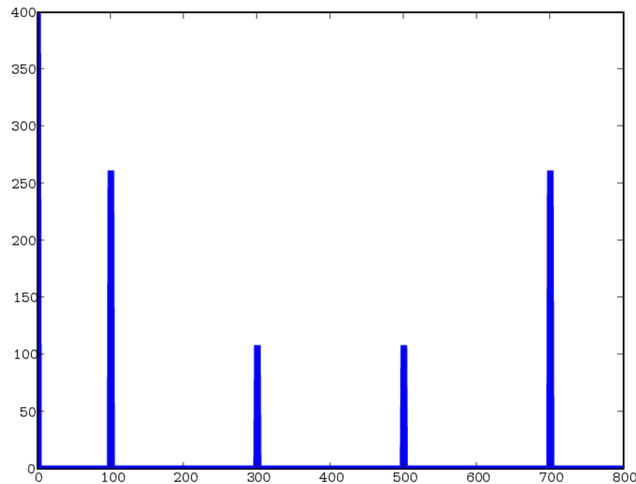


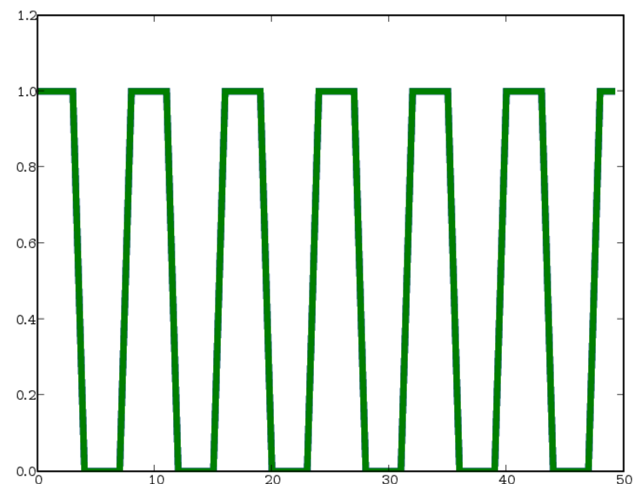
Zoom in on the first 51 data points of the square wave signal

With the `pylab` module:

```
>>> plot(x[:50], linewidth=2)
>>> axis((0, 50, -1, 2))
```

. . . Fourier transformation with FFTPACK





Forward Fourier transformation with the `fft()` from `scipy.fftpack` module (also loaded with `numpy.fft`, `pylab`:

```
>>> abs(fft(x))
```

Back to "time" domain with inverse Fourier transformation:

```
>>> from scipy.fftpack import *
>>> abs(ifft(fft(x)))[:6].round()
array([ 1.,  1.,  1.,  1.,  0.,  0.])
```

Other: `fft2()`, `fftn()`

sklearn aka scikit-learn

Machine learning in Python

Name	KLoC	GS-cites	Reference
SciPy.linalg			
Statsmodels	92	27	(Seabold and Perktold, 2010)
Scikit-learn	440	1.860	(Pedregosa et al., 2011)
PyMVPA	136	147 + 60	(Hanke et al., 2009a; Hanke et al., 2009b)
Orange	286	75	(Demšar et al., 2013)
Mlpy	75	8	(Albanese et al., 2012)
MDP	31	58	(Zito et al., 2008)
PyBrain	36	147	(Schaul et al., 2010)
Pylearn2		61	
Bob		31	
Gensim	9	160	(Řehůřek and Sojka, 2010)
NLTK	215	1.200	(Bird et al., 2009)
PyPR	?	—	—
Caffe			
...			

(Statistics not completely updated)

Scikit-learn/sklearn

In sklearn each algorithm is implemented as an object.

Here, e.g., non-negative matrix factorization (`NMF`) with data in a `numpy.array` called `datamatrix`:

```
from sklearn.decomposition import NMF
decomposer = NMF(n_components=2)  # Instancing an algorithm
decomposer.fit(datamatrix)        # Parameter estimation
```

or K-means clustering algorithm:

```
from sklearn.cluster import KMeans
clusterer = KMeans(n_clusters=2)
clusterer.fit(datamatrix)
```

sklearn object methods

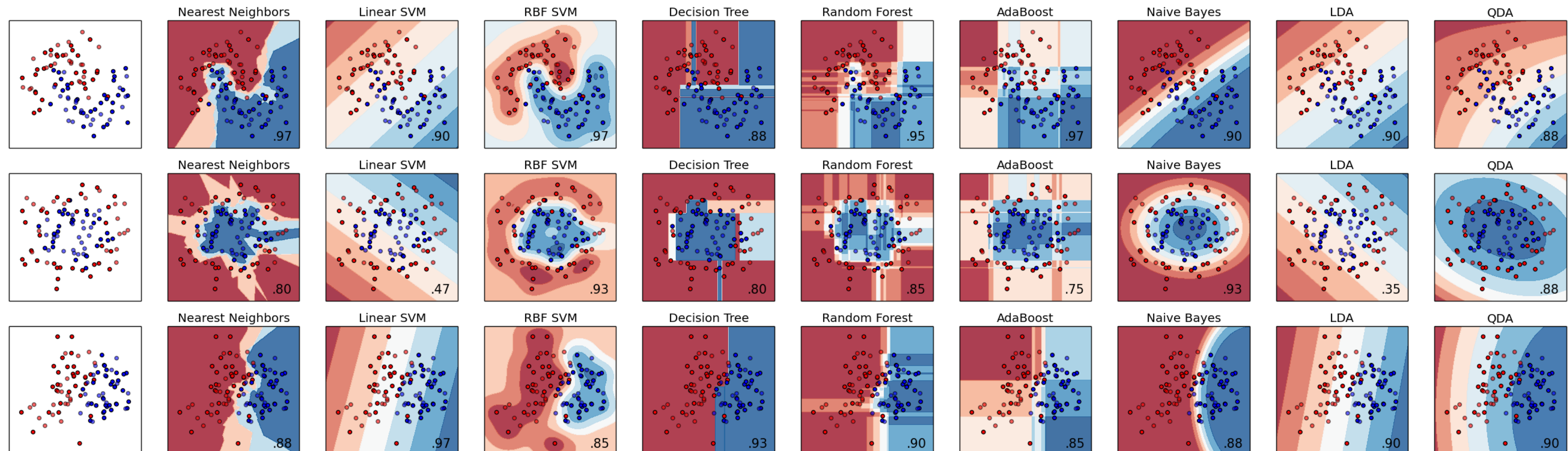
Sklearn algorithm objects shares methods, such as:

Name	Input	Description
`get_params`	—	Get parameter
`set_params`	Parameters	Set parameters
`decision_function`	X	
`fit`	X, y	Estimate model parameters
`fit_predict`	X	Performs clustering and return cluster labels
`fit_transform`	X, (y)	Fit and transform
`inverse_transform`	Y	Opposite operation of `transform`
`predict`	X	Estimate output
`predict_proba`	X	
`score`	X, y	Coefficient of determination
`transform`	X	Transform data, e.g., through dimensionality reduction

Estimated parameters and estimation “metadata” is available in attributes with trailing underscore, e.g., “`components_`”.

sklearn object methods

The common interface means that machine learning classifiers can be used interchangeably, see Classifier comparison SciKit-learn example.



From an example by Gael Varoquaux and Andreas Muller.

sklearn example

Non-negative matrix factorization with a data matrix called `feature_matrix`:

```
from sklearn.decomposition import NMF
decomposer = NMF(n_components=2)
decomposer.fit(feature_matrix)
transformed = decomposer.transform(feature_matrix)
```

Sizes of data matrix and resulting factoritation matrices:

```
>>> feature_matrix.shape
(94, 256)
>>> transformed.shape
(94, 2)
>>> decomposer.components_.shape
(2, 256)
```

Pandas

Pandas dataframe

A =

Index	a	b	c
2	4	5	yes
3	6.5	7	no
6	8	9	ok

Table represented in a Pandas `DataFrame`:

```
>>> import pandas as pd
>>> A = pd.DataFrame([[4, 5, 'yes'],
                      [6.5, 7, 'no'],
                      [8, 9, 'ok']],
           index=[2, 3, 6], columns=['a', 'b', 'c'])
>>> A
     a  b    c
2  4.0  5  yes
3  6.5  7   no
6  8.0  9   ok
```

Pandas column indexing

```
>>> A.c
2    yes
3     no
6     ok
Name: c, dtype: object
>>> A['c']
2    yes
3     no
6     ok
Name: c, dtype: object
>>> A.ix[:, 'c']
2    yes
3     no
6     ok
Name: c, dtype: object
```

Pandas row indexing

```
>>> A.ix[6, :]
a      8
b      9
c     ok
Name: 6, dtype: object
>>> A.iloc[2, :]
a      8
b      9
c     ok
Name: 6, dtype: object
```

And an element:

```
>>> A.ix[6, 'b']
9
```

Conditional indexing

```
>>> A
     a  b    c
2  4.0  5  yes
3  6.5  7   no
6  8.0  9   ok
>>> A.ix[(A.a > 7.0) | (A.c == 'yes'), ['b', 'c']]
   b    c
2  5  yes
6  9   ok
```

Pandas join and merge

A =

Index	a	b
1	4	5
2	6	7

B =

Index	a	c
1	8	9
3	10	11

(1)

In two Pandas `DataFrames`:

```
>>> import pandas as pd

>>> A = pd.DataFrame([[4, 5], [6, 7]], index=[1, 2],
                     columns=['a', 'b'])
>>> B = pd.DataFrame([[8, 9], [10, 11]], index=[1, 3],
                     columns=['a', 'c'])
```

Pandas concat and merge

```
>>> pd.concat((A, B))  # implicit outer join
    a    b    c
1   4    5  NaN
2   6    7  NaN
1   8  NaN    9
3  10  NaN   11

>>> pd.concat((A, B), join='inner')
    a
1   4
2   6
1   8
3  10
```

Pandas concat and merge

```
>>> A.merge(B, how='inner', left_index=True, right_index=True)
   a_x  b  a_y  c
1    4  5    8  9

>>> A.merge(B, how='outer', left_index=True, right_index=True)
   a_x    b  a_y    c
1    4    5    8    9
2    6    7  NaN  NaN
3  NaN  NaN   10   11

>>> A.merge(B, how='left', left_index=True, right_index=True)
   a_x  b  a_y    c
1    4  5    8    9
2    6  7  NaN  NaN
```

Descriptive statistics

```
>>> A.describe()  # see also .sum, .std, ...
              a   b
count  3.000000   3
mean   6.166667   7
std    2.020726   2
min    4.000000   5
25%    5.250000   6
50%    6.500000   7
75%    7.250000   8
max    8.000000   9
```

Note the result is only for numerical columns

Pandas input and output

`read_csv` — Comma-separated values file, works for URLs

`read_sql` — Read from SQL database

`read_excel` — Microsoft Excel

and a number of other formats.

Also note that the `db.py` package can interface between a SQL database and Pandas.

Other Pandas datatypes

Pandas `Series` is an annotated vector:

```
>>> pd.Series([1, 2, 4], index=['a', 'b', 'f'])
a    1
b    2
f    4
dtype: int64
```

Pandas `Panel` is a three-dimensional structure:

```
>>> pd.Panel([[[1, 2], [3, 4]], [[5, 6], [7, 8]]])
<class 'pandas.core.panel.Panel'>
Dimensions: 2 (items) x 2 (major_axis) x 2 (minor_axis)
Items axis: 0 to 1
Major_axis axis: 0 to 1
Minor_axis axis: 0 to 1
```

Pandas conversion to Numpy

```
>>> A
     a  b    c
2  4.0  5  yes
3  6.5  7   no
6  8.0  9   ok
>>> A.values
array([[4.0, 5, 'yes'],
       [6.5, 7, 'no'],
       [8.0, 9, 'ok']], dtype=object)
>>> A.ix[:, :2].values
array([[ 4. ,  5. ],
       [ 6.5,  7. ],
       [ 8. ,  9. ]])
```

Cython

Calling C et al.

You can call C, C++ and Fortran functions from Python:

Either with "manual" wrapping

or by using a automated wrapper: SWIG, Boost.Python, CFFI.

or by direct calling existing libraries via `ctypes`

You can make C-programs in Python with Cython or Pyrex and calling compiled modules from Python

Why calling C et al.?

Why calling C et al.?

- Because you already have code in that language.
- Because ordinary Python is not fast enough.

Why calling C et al.?

Why calling C et al.?

- Because you already have code in that language.
- Because ordinary Python is not fast enough. **Cython useful here**!

Cython

Write a Python file (possibly with extended Cython syntax for static types), compile to C and compile the C.

Cython is a fork of Pyrex.

Simplest example with compilation of a python file `helloworld.py`, containing `print("Hello, World")`:

```
$ cython --embed helloworld.py
$ gcc -I/usr/include/python2.7 -o helloworld helloworld.c -lpython2.7
$ ./helloworld
```

More: You can compile to a module instead (callable from Python); you can include static types in the Python code to make it faster (often these files have the extension *.pyx).

Calling Cython module from Python

`hello.pyx` Python file with

```
def world():
    return "Hello, World"
```

Compile to C and compile to module (here for Linux):

```
$ cython hello.pyx
$ gcc -shared -pthread -fPIC -fwrapv -O2 -Wall -fno-strict-aliasing \
    -I/usr/include/python2.7 -o hello.so hello.c -lpython2.7
```

Back in Python use the hello module:

```
>>> import hello
>>> hello.world()     # Call the function in the module
'Hello, World'
```

Other compilation methods

For simple Cython modules compiling automagically:

```
>>> import pyximport
>>> pyximport.install()
(None, <pyximport.pyximport.PyxImporter object at 0x7f7dc39f20d
>>> import hello
>>> hello.world()
'Hello, World'
```

Other compilation methods

For simple Cython modules compiling automagically:

```
>>> import pyximport
>>> pyximport.install()
(None, <pyximport.pyximport.PyxImporter object at 0x7f7dc39f20d
>>> import hello
>>> hello.world()
'Hello, World'
```

Otherwise construct a `setup.py` file with:

```
from distutils.core import setup
from Cython.Build import cythonize

setup(name='Hello', ext_modules=cythonize("hello.pyx"))
```

and compile with

```
$ python setup.py build_ext --inplace
```

Optional types . . .

`slow.py` with plain python

```
def count_ascendings(alist):
    count = 0
    for first, second in zip(alist[:-1], alist[1:]):
        if first < second:
            count += 1
    return count
```

`fast.pyx` with integer type (note "`cdef int count`"!):

```
def count_ascendings(alist):
    cdef int count = 0
    for first, second in zip(alist[:-1], alist[1:]):
        if first < second:
            count += 1
    return count
```

. . . Optional types

Profiling the plain Python and the Cython version:

```
>>> import timeit
>>> import pyximport; pyximport.install()

>>> timeit.timeit("slow.count_ascendings(range(1000))",
                  setup="import slow", number=10000)
1.3584449291229248
>>> timeit.timeit("fast.count_ascendings(range(1000))",
                  setup="import fast", number=10000)
0.8554580211639404
```

The Cython version without static types yield a timing between the plain Python and the statically typed Cython (1.06 seconds here).

Six advices from Hans Petter Langtangen

Section *Optimization of Python code* (Langtangen, 2005, p. 426+)

Avoid loops, use NumPy (see also my blog)

Avoid prefix in often called functions, i.e., `sin` instead of `math.sin`

Plain functions run faster than class methods

Don't use NumPy for scalar arguments

Use `xrange` instead of `range` (in Python < 3)

`if-else` is faster than `try-except` (sometimes!)

More information

http://www.scipy.org/NumPy_for_Matlab_Users

MATLAB commands in numerical Python (Numpy), Vidar Bronken Gundersen, mathesaurus.sf.net

Guide to NumPy (Oliphant, 2006)

Videolectures.net: John D. Hunter overview of Matplotlib
http://videolectures.net/mloss08_hunter_mat/

Emerging Python modules for big data

blaze and dask

castra

pySpark

References

Albanese, D., Visintainer, R., Merler, S., Riccadonna, S., Jurman, G., and Furlanello, C. (2012). `mlpy`: machine learning Python. ArXiv. http://arxiv.org/pdf/1202.6548v2.pdf.

Bird, S., Klein, E., and Loper, E. (2009). *Natural Language Processing with Python*. O'Reilly, Sebastopol, California. ISBN 9780596516499. The canonical book for the NLTK package for natural language processing in the Python programming language. Corpora, part-of-speech tagging and machine learning classification are among the topics covered.

Demšar, J., Curk, T., Erjavec, A., Črt Gorup, Hočevar, T., Milutinovi, M., Možina, M., Polajnar, M., Toplak, M., Stari, A., Štajdohar, M., Umek, L., Žagar, L., Žbontar, J., Žitnik, M., and Zupan, B. (2013). Orange: data mining toolbox in Python. *Journal of Machine Learning Research*, 14:2349–2353. PMID: . DOI: . WOBIB: . http://jmlr.org/papers/volume14/demsar13a/demsar13a.pdf.

Hanke, M., Halchenko, Y. O., Sederberg, P. B., Hanson, S. J., Haxby, J. V., and Pollmann, S. (2009a). PyMVPA: a Python toolbox for multivariate pattern analysis of fMRI data. *Neuroinformatics*, 7(1):37–53. PMID: . DOI: 10.1007/s12021-008-9041-y. WOBIB: .

Hanke, M., Halchenko, Y. O., Sederberg, P. B., Olivetti, E., Frund, I., Rieger, J. W., Herrmann, C. S., Haxby, J. V., Hanson, S. J., and Pollmann, S. (2009b). PyMVPA: a unifying approach to the analysis of neuroscientific data. *Frontiers in neuroinformatics*, 3:3. PMID: 19212459. DOI: 10.3389/neuro.11.003.2009. WOBIB: .

Langtangen, H. P. (2005). *Python Scripting for Computational Science*, volume 3 of *Texts in Computational Science and Engineering*. Springer. ISBN 3540294155.

Lee, D. D. and Seung, H. S. (2001). Algorithms for non-negative matrix factorization. In Leen, T. K., Dietterich, T. G., and Tresp, V., editors, *Advances in Neural Information Processing Systems 13: Proceedings of the 2000 Conference*, pages 556–562, Cambridge, Massachusetts. MIT Press. http://hebb.mit.edu/people/seung/papers/nmfconverge.pdf. CiteSeer: http://citeseer.ist.psu.edu/lee00algorithms.html.

Oliphant, T. E. (2006). *Guide to NumPy*. Trelgol Publishing.

Pedregosa, F., Varoquaux, G., Gramfort, A., Michel, V., Thirion, B., Grisel, O., Blondel, M., Prettenhofer, P., Weiss, R., Dubourg, V., Vanderplas, J., Passos, A., Cournapeau, D., Brucher, M., Perrot, M., and Édouard Duchesnay (2011). Scikit-learn: machine learning in Python. *Journal of Machine Learning Research*, 12:2825–2830. PMID: . DOI: . WOBIB: . http://jmlr.csail.mit.edu/papers/volume12/pedregosa11a/pedregosa11a.pdf.

Schaul, T., Bayer, J., Wierstra, D., Sun, Y., Felder, M., Sehnke, F., Rückstie, T., and Schmidhuber, J. (2010). Pybrain. *Journal of Machine Learning Research*, 11:743—746. PMID: . DOI: . WOBIB: . http://www.jmlr.org/papers/volume11/schaul10a/schaul10a.pdf. Presents the PyBrain Python machine learning package.

Seabold, S. and Perktold, J. (2010). Statsmodels: econometric and statistical modeling with python. In *Proceedings of the 9th Python in Science Conference*. https://projects.scipy.org/proceedings/scipy2010/pdfs/seabold.pdf. Description of the statsmodels package for Python.

Segaran, T. (2007). *Programming Collective Intelligence*. O'Reilly, Sebastopol, California.

Zito, T., Wilbert, N., Wiskott, L., and Berkes, P. (2008). Modular toolkit for data processing (mdp): a python data processing framework. *Frontiers in Neuroinformatics*, 2:8. PMID: . DOI: 10.3389/neuro.11.008.2008. WOBIB: .

Řehůřek, R. and Sojka, P. (2010). Software framework for topic modelling with large corpora. In *Proceedings of LREC 2010 workshop New Challenges for NLP Frameworks*. http://www.fi.muni.cz/usr/sojka/papers/lrec2010-rehurek-sojka.pdf.