# BioKit

*Release 0.4.3*

**BioKit developers**

**Jul 18, 2018**

# Contents


**note** BioKit is tested with Travis for the following Python version: 2.7, 3.5, 3.6

**contributions** Please join https://github.com/biokit/biokit

**issues** Please use https://github.com/biokit/biokit/issues

CHAPTER 1

# Overview

BioKit is a set of tools dedicated to bioinformatics, data visualisation (*biokit.viz*), access to online biological data (e.g. UniProt, NCBI thanks to bioservices). It also contains more advanced tools related to data analysis (e.g., *biokit.stats*). Since R is quite common in bioinformatics, we also provide a convenient module to run R inside your Python scripts or shell (:mod:biokit.rtools module).

In order to install biokit, you can use **pip**:

```
pip install biokit
```

Or using bioconda channel from the Anaconda project:

```
conda install biokit
```

CHAPTER 2

# Overview

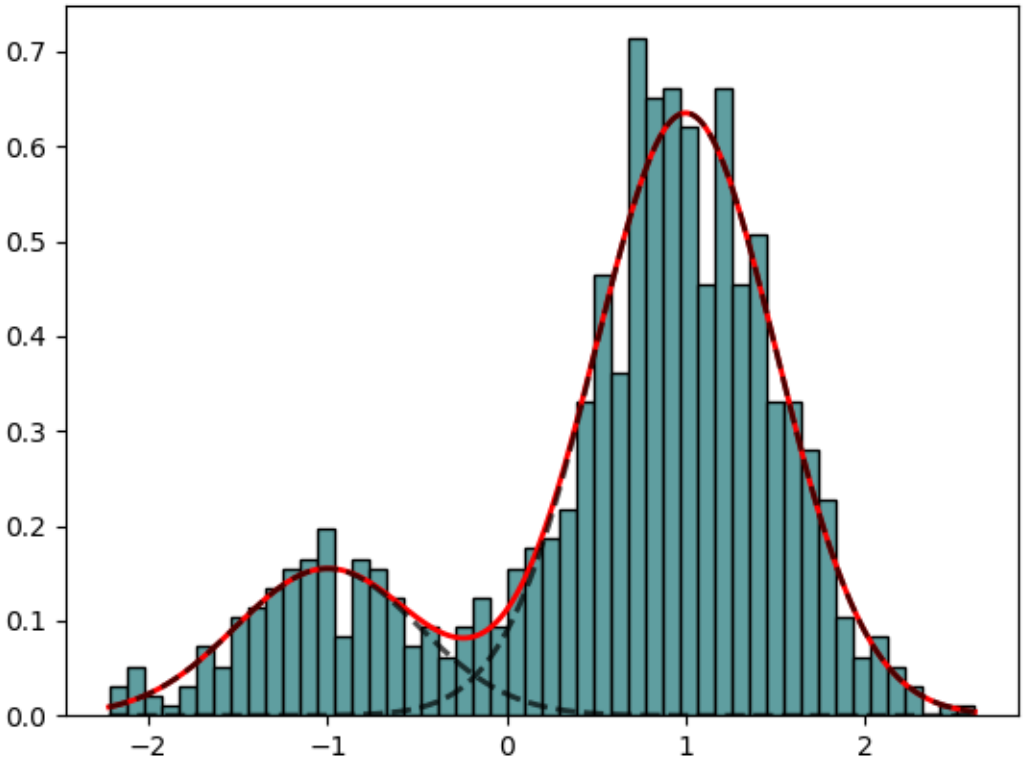
| | |
|---|---|
| *biokit.network* | Utilities related to networks (e.g., protein) |
| *biokit.viz* | Plotting tools |
| *biokit.rtools* | utilities related to R language (e.g., RPackageManager, RSession) |
| *biokit.sequence* | Sequence related (Generic, DNA, RNA) |
| *biokit.stats* | Generic statistical tools |

## 2.1 Gallery

**Note:** Click *here* to download the full example code

### 2.1.1 Gaussian Mixture model

Fit a Gaussian Mixture Models (2 distributions) using brute force optimisation method.




Out:

```
Creating directory /home/docs/.config/biokit
Creating directory /home/docs/.config/bioservices
```

```
from biokit.stats.mixture import GaussianMixture, GaussianMixtureFitting
m = GaussianMixture(mu=[-1,1], sigma=[0.5,0.5], mixture=[0.2,0.8])
mf = GaussianMixtureFitting(m.data)
mf.estimate(k=2)
mf.plot()
```

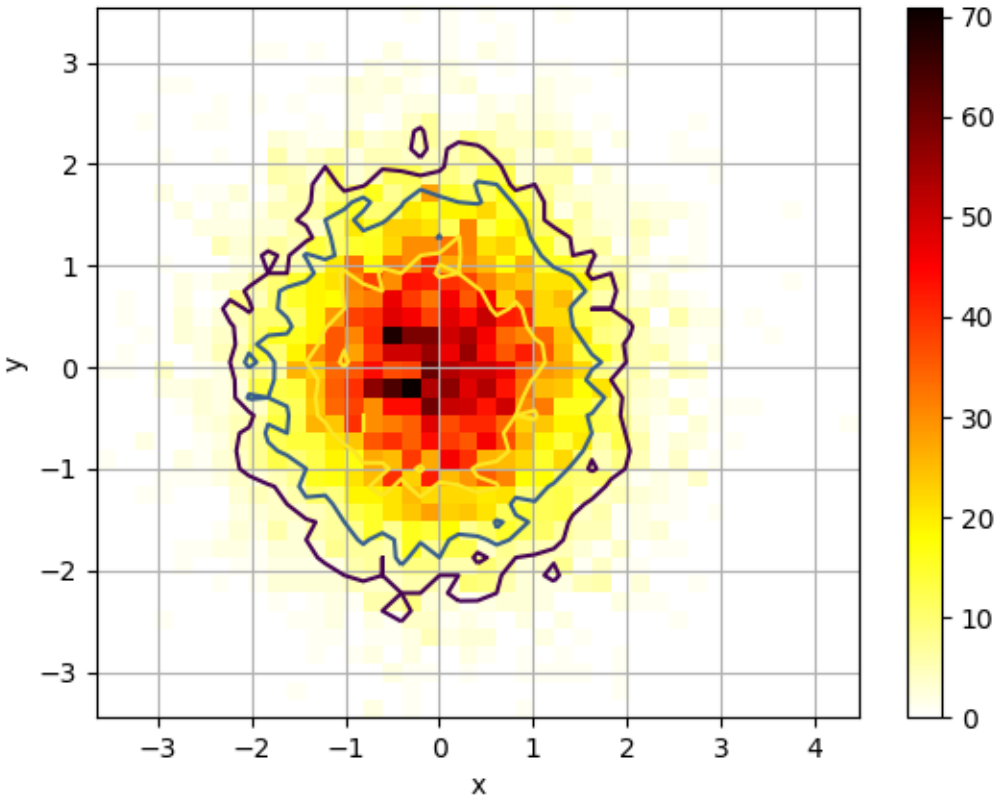
**Total running time of the script:** ( 0 minutes 4.270 seconds)

**Note:** Click *here* to download the full example code

### 2.1.2 Hist2d using Pandas dataframe as input

wrapping of pylab hist2d function.

```
import pylab

from biokit import viz

X = pylab.randn(10000)
Y = pylab.randn(10000)


import pandas as pd
df = pd.DataFrame({'X':X, 'Y':Y})


from biokit.viz import hist2d
h = hist2d.Hist2D(df)
res = h.plot(bins=[40,40], contour=True, nnorm='log', Nlevels=6)
```




**Total running time of the script:** ( 0 minutes 0.266 seconds)

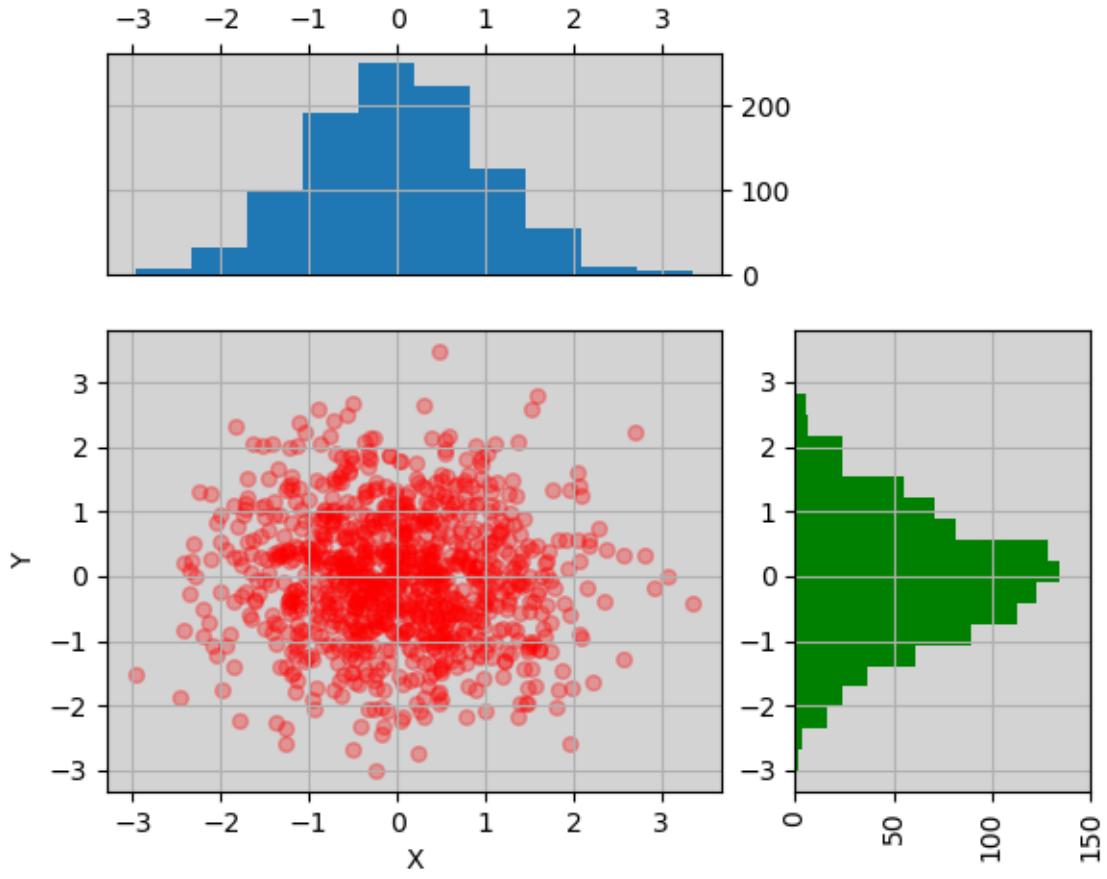
**Note:** Click *here* to download the full example code

### 2.1.3 Scatter plot and histogram

```
import pylab
from biokit import ScatterHist
import pandas as pd

# input can be a 2-column matrix or a dataframe with 2 columns
X = pylab.randn(1000)
Y = pylab.randn(1000)

df = pd.DataFrame({'X':X, 'Y':Y})

sh = ScatterHist(df)
```

you can tune the scatter plot and histogram with valid optional arguments expected by the pylab functions. Check the pylab.hist and pylab.scatter helps for details.

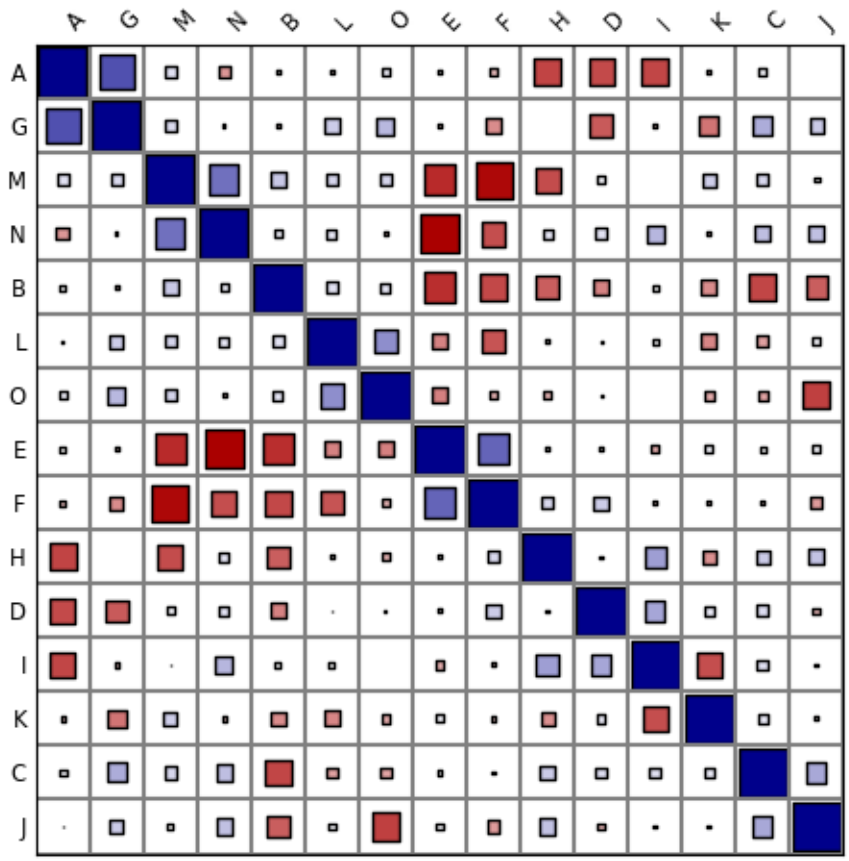
```
_ = sh.plot(kargs_scatter={'c':'r', 's':30, 'alpha':.3},
                    kargs_histy={'color':'g', 'bins':20})
```




**Total running time of the script:** ( 0 minutes 0.102 seconds)

**Note:** Click *here* to download the full example code

### 2.1.4 Corrplot example




```
# some useful pylab imports for this notebook


# Create some random data
import string
letters = string.ascii_uppercase[0:15]
import pandas as pd
import numpy as np
df = pd.DataFrame(dict(( (k, np.random.random(10)+ord(k)-65) for k in letters)))
df = df.corr()

# if the input is not a square matrix or indices do not match
# column names, correlation is computed on the fly
from biokit.viz import corrplot
c = corrplot.Corrplot(df)


c.plot(colorbar=False, method='square', shrink=.9 ,rotation=45)
```

**Total running time of the script:** ( 0 minutes 0.352 seconds)

Contents



## 2.2 References

### 2.2.1 Tools related to network (e.g., protein networks)

Utilities related to networks (e.g., protein)

**class Complexes**(*organism='Homo sapiens'*, *verbose=True*, *cache=False*)

Manipulate complexes of Proteins

This class uses Intact Complex database to extract information about complexes of proteins.

When creating an instance, the default organism is "Homo sapiens". The organism can be set to another one during the instanciation or later:

```
>>> from biokit.network.complexes import Complexes
>>> c = Complexes(organism='Homo sapiens')
>>> c.organism = 'Rattus norvegicus'
```

Valid organisms can be found in *organisms*. When changing the organism, a request to the Intact database is sent, which may take some time to update. Once done, information related to this organism is stored in the `df` attribute, which is a Pandas dataframe. It contains 4 columns. Here is for example one row:

```
complexAC                                          EBI-2660609
complexName                          COP9 signalosome variant 1
description     Essential regulator of the ubiquitin (Ubl) con...
organismName                                 Homo sapiens; 9606
```

This is basic information but once a complex accession (e.g., EBI-2660609) is known, you can retrieve detailled information. This is done automatically for all the accession when needed. The first time, it will take a while (20 seconds for 250 accession) but will be cache for this instance.

The *complexes* contains all details about the entries found in `df`. It is a dictionary where keys are the complex accession. For instance:

```
>>> c.complexes['EBI-2660609']
```

In general, one is interested in the participants of the complex, that is the proteins that form the complex. Another attribute is set for you:

```
>>> c.participants['EBI-2660609']
```

Finally, you may even want to obtain just the identifier of the participants for each complex. This is stored in the *identifiers*:

```
>>> c.identifiers['EBI-2660609']
```

Note however, that the identifiers are not neceseraly uniprot identifiers. Could be ChEBI or sometimes even set to None. The `strict_filter()` removes the complexes with less than 2 (strictly) uniprot identifiers.

Some basic statistics can be printed with *stats()* that indeticates the number of complexes, number of identifiers in those complexes ,and number of unique identifiers. A histogram of number of appearance of each identifier is also shown.

The *hist_participants()* shows the number of participants per complex.

Finally, the meth:*search_complexes* can be used in the context of logic modelling to infer the AND gates from a list of uniprot identifiers provided by the user. See *search_complexes()* for details.

Access to the Intact Complex database is performed using the package BioServices provided in Pypi.

### Constructor

**Parameters**

- **orgamism** (*str*) – the organism to look at. Homo sapiens is the default. Other possible organisms can be found in *organisms*.
- **verbose** (*str*) – a verbose level in ERROR/DEBUG/INFO/WARNING compatible with those used in BioServices.

**chebi2name**(*name*)
Return the ASCII name of a CHEBI identifier

**complexes**
Getter of the complexes (full details

**hist_participants**()
Histogram of the number of participants per complexes

**Returns** a dictionary with complex identifiers as keys and number of participants as values

```
from biokit.network.complexes import Complexes
c = Complexes()
c.hist_participants()
```

**identifiers**
Getter of the identifiers of the complex participants

**organism**
Getter/Setter of the organism

**organisms = None**
list of valid organisms found in the database

**participants**
Getter of the complex participants (full details)

**remove_homodimers**()
Remove identifiers that are None or starts with CHEBI and keep complexes that have at least 2 participants

**Returns** list of complex identifiers that have been removed.

**report**(*species*)

**search**(*name*)
Search for a unique identifier (e.g. uniprot) in all complexes

**Returns** list of complex identifiers where the name was found

**search_complexes**(*user_species*, *verbose=False*)

**Given a list of uniprot identifiers, return complexes and** possible complexes.

**Parameters** **user_species** (*list*) – list of uniprot identifiers to be found in the complexes

**Returns** two dictionaries. First one contains the complexes for which all participants have been found in the user_species list. The second one contains complexes for which some participants (not all) have been found in the user_species list.

**stats**()
Prints some stats about the number of complexes and histogram of the number of appearances of each species

**uniprot2genename**(*name*)
Return the gene names of a UniProt identifier

**valid_organisms = None**
list of valid organisms found in the database

### 2.2.2 Common visualisation tools

Plotting tools

| | |
|---|---|
| *biokit.viz.corrplot.Corrplot*(data[, na]) | An implementation of correlation plotting tools (corrplot) |
| *biokit.viz.hinton.hinton*(df[, fig, shrink, ...]) | Hinton plot (simplified version of correlation plot) |
| *biokit.viz.hist2d.Hist2D*(x[, y, verbose]) | 2D histogram |
| *biokit.viz.imshow.Imshow*(x[, verbose]) | Wrapper around the matplotlib.imshow function |
| *biokit.viz.scatter.ScatterHist*(x[, y, verbose]) | Scatter plots and histograms |
| *biokit.viz.volcano.Volcano*([fold_changes, ...]) | Volcano plot |
| *biokit.viz.heatmap.Heatmap*([data, ...]) | Heatmap and dendograms of an input matrix |

#### Corrplot utilities

**author** Thomas Cokelaer

**references** http://cran.r-project.org/web/packages/corrplot/vignettes/corrplot-intro.html

**class Corrplot**(*data*, *na=0*)
An implementation of correlation plotting tools (corrplot)

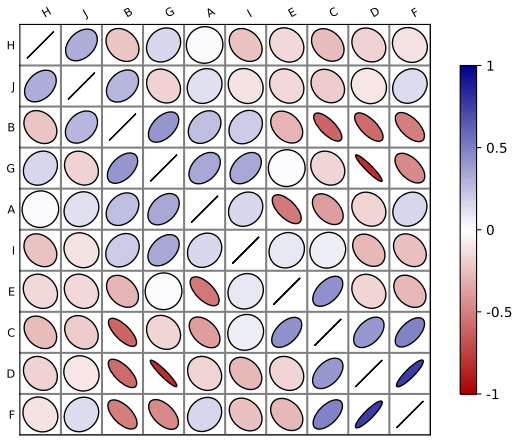
Here is a simple example with a correlation matrix as an input (stored in a pandas dataframe):

```
# create a correlation-like data set stored in a Pandas' dataframe.
import string
# letters = string.uppercase[0:10] # python2
```

(continues on next page)

(continued from previous page)

```
letters = string.ascii_uppercase[0:10]
import pandas as pd
df = pd.DataFrame(dict(( (k, np.random.random(10)+ord(k)-65) for k in letters)))

# and use corrplot
from biokit.viz import corrplot
c = corrplot.Corrplot(df)
c.plot()
```




**See also:**

All functionalities are covered in this notebook

### Constructor

Plots the content of square matrix that contains correlation values.

**Parameters**

- **data** – input can be a dataframe (Pandas), or list of lists (python) or a numpy matrix. Note, however, that values must be between -1 and 1. If not, or if the matrix (or list of lists) is not squared, then correlation is computed. The data or computed correlation is stored in *df* attribute.
- **compute_correlation** (*bool*) – if the matrix is non-squared or values are not bounded in -1,+1, correlation is computed. If you do not want that behaviour, set this parameter to False. (True by default).
- **na** – replace NA values with this value (default 0)

The *params* contains some tunable parameters for the colorbar in the *plot()* method.

```
# can be a list of lists, the correlation matrix is then a 2x2 matrix
c = corrplot.Corrplot([[1,1], [2,4], [3,3], [4,4]])
```

**df = None**
The input data is stored in a dataframe and must therefore be compatible (list of lists, dictionary, matrices...)

**order**(*method='complete'*, *metric='euclidean'*, *inplace=False*)
Rearrange the order of rows and columns after clustering

**Parameters**

- **method** – any scipy method (e.g., single, average, centroid, median, ward). See scipy.cluster.hierarchy.linkage
- **metric** – any scipy distance (euclidean, hamming, jaccard) See scipy.spatial.distance or scipy.cluster.hiearachy
- **inplace** (*bool*) – if set to True, the dataframe is replaced

You probably do not need to use that method. Use *plot()* and the two parameters order_metric and order_method instead.

**params = None**
tunable parameters for the *plot()* method.

**plot**(*fig=None*, *grid=True*, *rotation=30*, *lower=None*, *upper=None*, *shrink=0.9*, *facecolor='white'*, *colorbar=True*, *label_color='black'*, *fontsize='small'*, *edgecolor='black'*, *method='ellipse'*, *order_method='complete'*, *order_metric='euclidean'*, *cmap=None*, *ax=None*, *binarise_color=False*)
plot the correlation matrix from the content of *df* (dataframe)

By default, the correlation is shown on the upper and lower triangle and is symmetric wrt to the diagonal. The symbols are ellipses. The symbols can be changed to e.g. rectangle. The symbols are shown on upper and lower sides but you could choose a symbol for the upper side and another for the lower side using the **lower** and **upper** parameters.

**Parameters**

- **fig** – Create a new figure by default. If an instance of an existing figure is provided, the corrplot is overlayed on the figure provided. Can also be the number of the figure.
- **grid** – add grid (Defaults to grey color). You can set it to False or a color.
- **rotation** – rotate labels on y-axis
- **lower** – if set to a valid method, plots the data on the lower left triangle
- **upper** – if set to a valid method, plots the data on the upper left triangle
- **shrink** (*float*) – maximum space used (in percent) by a symbol. If negative values are provided, the absolute value is taken. If greater than 1, the symbols wiill overlap.
- **facecolor** – color of the background (defaults to white).
- **colorbar** – add the colorbar (defaults to True).
- **label_color** (*str*) – (defaults to black).
- **fontsize** – size of the fonts defaults to 'small'.
- **method** – shape to be used in 'ellipse', 'square', 'rectangle', 'color', 'text', 'circle', 'number', 'pie'.
- **order_method** – see *order()*.
- **order_metric** – see : meth:*order*.
- **cmap** – a valid cmap from matplotlib or colormap package (e.g., 'jet', or 'copper'). Default is red/white/blue colors.
- **ax** – a matplotlib axes.

The colorbar can be tuned with the parameters stored in *params*.

Here is an example. See notebook for other examples:

```
c = corrplot.Corrplot(dataframe)
c.plot(cmap=('Orange', 'white', 'green'))
c.plot(method='circle')
c.plot(colorbar=False, shrink=.8, upper='circle'  )
```

## Scatter plots

**author** Thomas Cokelaer

**class ScatterHist**(*x*, *y=None*, *verbose=True*)
Scatter plots and histograms

### constructor

**Parameters**

- **x** – if x is provided, it should be a dataframe with 2 columns. The first one will be used as your X data, and the second one as the Y data
- **y** –
- **verbose** –

**plot**(*kargs_scatter={'c': 'b', 's': 20}*, *kargs_grids={}*, *kargs_histx={}*, *kargs_histy={}*, *scatter_position='bottom left'*, *width=0.5*, *height=0.5*, *offset_x=0.1*, *offset_y=0.1*, *gap=0.06*, *facecolor='lightgrey'*, *grid=True*, *show_labels=True*, ***kargs*)
Scatter plot of set of 2 vectors and their histograms.

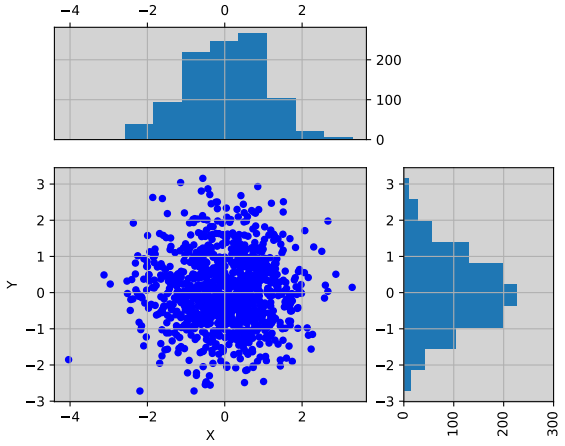
**Parameters**

- **x** – a dataframe or a numpy matrix (2 vectors) or a list of 2 items, which can be a mix of list or numpy array. if **size** and/or **color** are found in the columns dataframe, those columns will be used in the scatter plot. kargs_scatter keys **c** and **s** will then be ignored. If a list of lists, **x** will be the first row and **y** the second row.
- **y** – if x is a list or an array, then y must also be provided as a list or an array
- **kargs_scatter** – a dictionary with pairs of key/value accepted by matplotlib.scatter function. Examples is a list of colors or a list of sizes as shown in the examples below.
- **kargs_grid** – a dictionary with pairs of key/value accepted by the maplotlib.grid (applied on histogram and axis at the same time)
- **kargs_histx** – a dictionary with pairs of key/value accepted by the matplotlib.histogram
- **kargs_histy** – a dictionary with pairs of key/value accepted by the matplotlib.histogram
- **kargs** – other optional parameters are **hold**, **facecolor**.
- **scatter_position** – can be 'bottom right/bottom left/top left/top right'
- **width** – width of the scatter plot (value between 0 and 1)
- **height** – height of the scatter plot (value between 0 and 1)

- **offset_x** –
- **offset_y** –
- **gap** – gap between the scatter and histogram plots.
- **grid** – defaults to True

**Returns** the scatter, histogram1 and histogram2 axes.

```
import pylab
import pandas as pd
X = pylab.randn(1000)
Y = pylab.randn(1000)
df = pd.DataFrame({'X':X, 'Y':Y})

from biokit.viz import ScatterHist
ScatterHist(df).plot()
```




```
from biokit.viz import ScatterHist
ScatterHist(x=[1,2,3,4], y=[3,5,6,4]).plot(
    kargs_scatter={
        's':[200,400,600,800],
        'c': ['red', 'green', 'blue', 'yellow'],
        'alpha':0.5},
    kargs_histx={'color': 'red'},
    kargs_histy={'color': 'green'})
```

**See also:**

notebook

Imshow utility

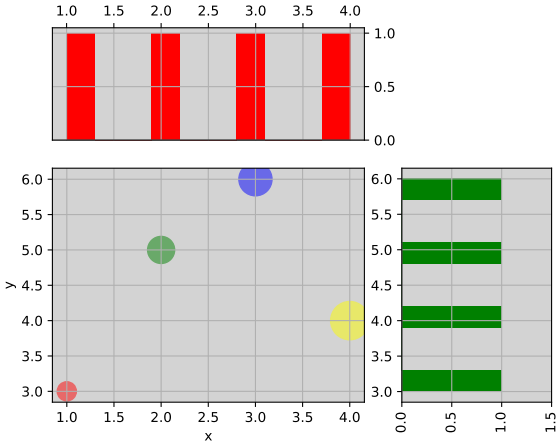
**imshow**(*x*, *interpolation='None'*, *aspect='auto'*, *cmap='hot'*, *tight_layout=True*, *colorbar=True*, *fontsize_x=None*, *fontsize_y=None*, *rotation_x=90*, *xticks_on=True*, *yticks_on=True*, ***kargs*)

Alias to the class *Imshow*

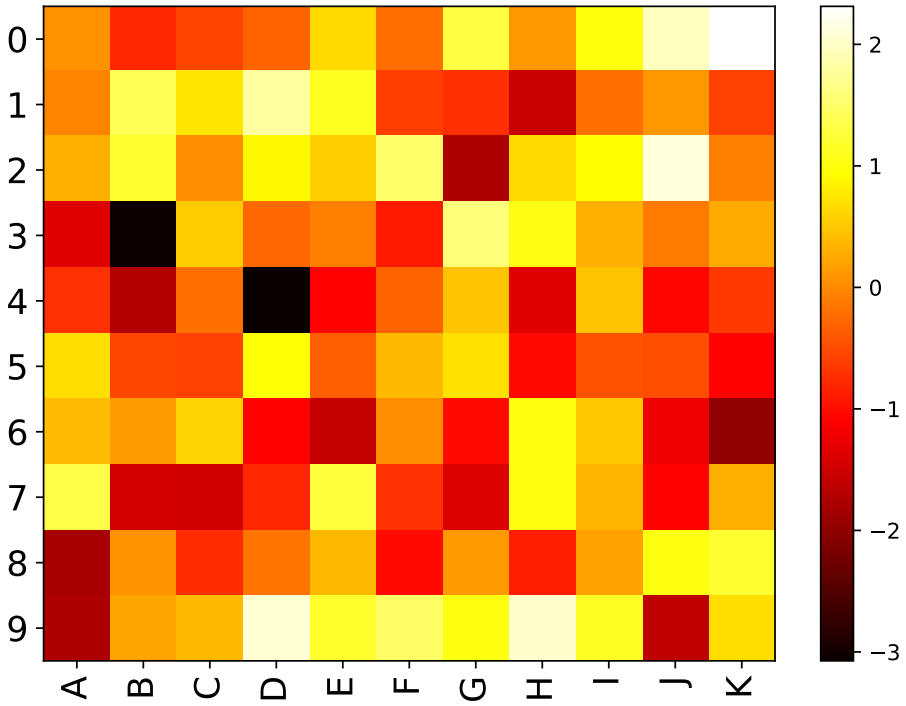
**class Imshow**(*x*, *verbose=True*)

Wrapper around the matplotlib.imshow function

Very similar to matplotlib but set interpolation to None, and aspect to automatic and accepts input as a dataframe, in whic case x and y labels are set automatically.




```
import pandas as pd
data = dict([ (letter,np.random.randn(10)) for letter in 'ABCDEFGHIJK'])
df = pd.DataFrame(data)

from biokit import Imshow
im = Imshow(df)
im.plot()
```




### constructor

**Parameters** **x** – input dataframe (or numpy matrix/array). Must be squared.

**plot**(*interpolation='None'*, *aspect='auto'*, *cmap='hot'*, *tight_layout=True*, *colorbar=True*, *fontsize_x=None*, *fontsize_y=None*, *rotation_x=90*, *xticks_on=True*, *yticks_on=True*, ***kargs*)
wrapper around imshow to plot a dataframe

**Parameters**

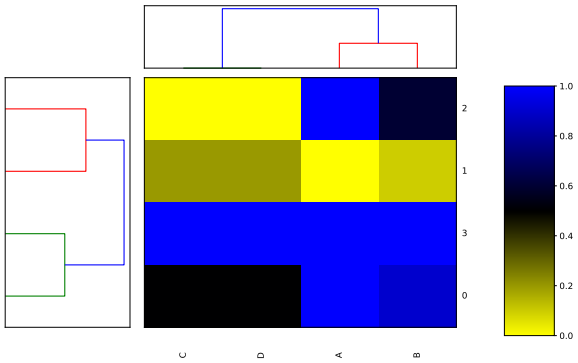
- **interpolation** – set to None
- **aspect** – set to 'auto'
- **cmap** – colormap to be used.
- **tight_layout** –
- **colorbar** – add a colobar (default to True)
- **fontsize_x** – fontsize on xlabels
- **fontsize_y** – fontsize on ylabels
- **rotation_x** – rotate labels on xaxis
- **xticks_on** – switch off the xticks and labels
- **yticks_on** – switch off the yticks and labels

Heatmap and dendograms

**class Heatmap**(*data=None*, *row_method='complete'*, *column_method='complete'*, *row_metric='euclidean'*, *column_metric='euclidean'*, *cmap='yellow_black_blue'*, *col_side_colors=None*, *row_side_colors=None*, *verbose=True*)
Heatmap and dendograms of an input matrix

A heat map is an image representation of a matrix with a dendrogram added to the left side and to the top. Typically, reordering of the rows and columns according to some set of values (row or column means) within the restrictions imposed by the dendrogram is carried out.

```
from biokit.viz import heatmap
df = heatmap.get_heatmap_df()
h = heatmap.Heatmap(df)
h.plot()
```




**Warning:** in progress

### constructor

**Parameters** **data** – a dataframe or possibly a numpy matrix.

---

**Todo:** if row_method id none, no ordering in the dendogram

---

**column_method**

**column_metric**

**df**

**frame**

**plot**(*num=1*, *cmap=None*, *colorbar=True*, *vmin=None*, *vmax=None*, *colorbar_position='right'*, *gradient_span='None'*)

**Parameters** **gradient_span** – None is default in R

Using:

```
df = pd.DataFrame({'A':[1,0,1,1],
                   'B':[.9,0.1,.6,1],
                'C':[.5,.2,0,1],
                'D':[.5,.2,0,1]})
```

and

```
h = Heatmap(df)
h.plot(vmin=0, vmax=1.1)
```

we seem to get the same as in R wiht

```
df = data.frame(A=c(1,0,1,1), B=c(.9,.1,.6,1), C=c(.5,.2,0,1), D=c(.5,.2,0,1))
heatmap((as.matrix(df)), scale='none')
```

---

**Todo:** right now, the order of cols and rows is random somehow. could be ordered like in heatmap (r) byt mean of the row and col or with a set of vector for col and rows.

heatmap((as.matrix(df)), Rowv=c(3,2), Colv=c(1), scale='none')

gives same as:

```
df = get_heatmap_df()
h = heatmap.Heatmap(df)
h.plot(vmin=-0, vmax=1.1)
```

---

**row_method**

**row_metric**

## Hinton plot

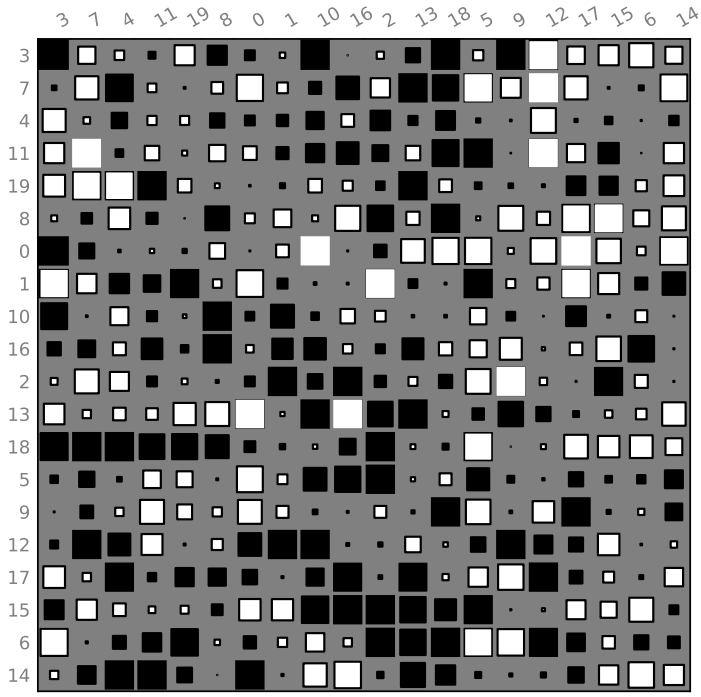
**author** Thomas Cokelaer

**hinton** (*df*, *fig=1*, *shrink=2*, *method='square'*, *bgcolor='grey'*, *cmap='gray_r'*, *binarise_color=True*)
Hinton plot (simplified version of correlation plot)

**Parameters**

- **df** – the input data as a dataframe or list of items (list, array). See *Corrplot* for details.
- **fig** – in which figure to plot the data
- **shrink** – factor to increase/decrease sizes of the symbols
- **method** – set the type of symbols for each coordinates. (default to square). See *Corrplot* for more details.
- **bgcolor** – set the background and label colors as grey
- **cmap** – gray color map used by default
- **binarise_color** – use only two colors. One for positive values and one for negative values.

```
from biokit.viz import hinton
df = np.random.rand(20, 20) - 0.5
hinton(df)
```




**Note:** Idea taken from a matplotlib recipes http://matplotlib.org/examples/specialty_plots/hinton_demo.html but solely using the implementation within *Corrplot*

**See also:**

*biokit.viz.corrplot.Corrplot*

**Note:** Values must be between -1 and 1. No sanity check performed.

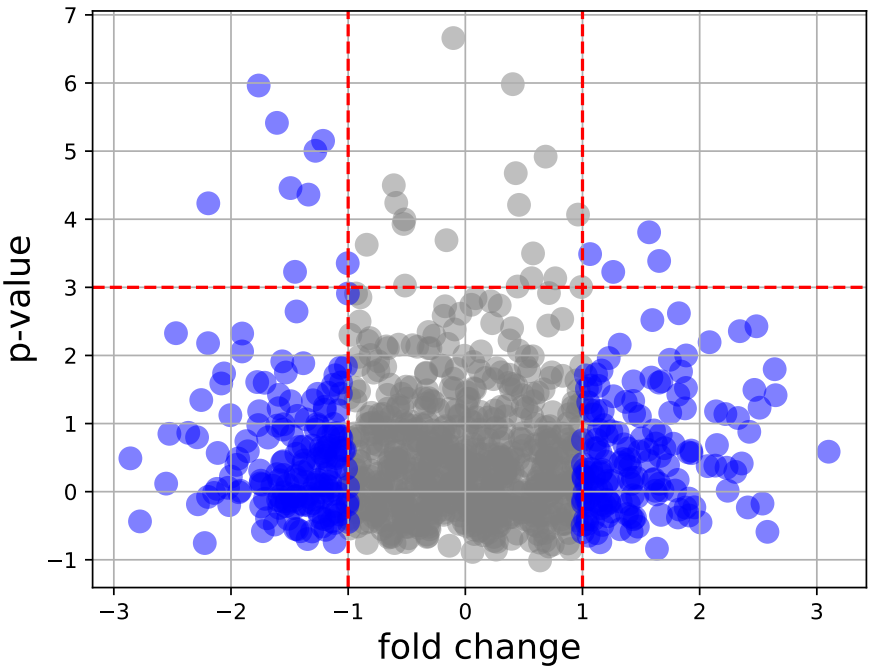
Volcano plot

**class Volcano**(*fold_changes=None*, *pvalues=None*, *color=None*)

Volcano plot

In essence, just a scatter plot with annotations.

```
import numpy as np
fc = np.random.randn(1000)
pvalue = np.random.randn(1000)

from biokit import Volcano
v = Volcano(fc, -np.log10(pvalue**2))
v.plot(pvalue_threshold=3)
```




### constructor

**Parameters**

- **fold_changes** (*list*) – 1D array or list
- **pvalues** (*list*) – 1D array or list
- **df** – a dataframe with those column names: fold_changes, pvalues, color (optional)

**plot**(*size=100*, *alpha=0.5*, *marker='o'*, *fontsize=16*, *xlabel='fold change'*, *ylabel='p-value'*, *pvalue_threshold=1.5*, *fold_change_threshold=1*)

**Parameters**

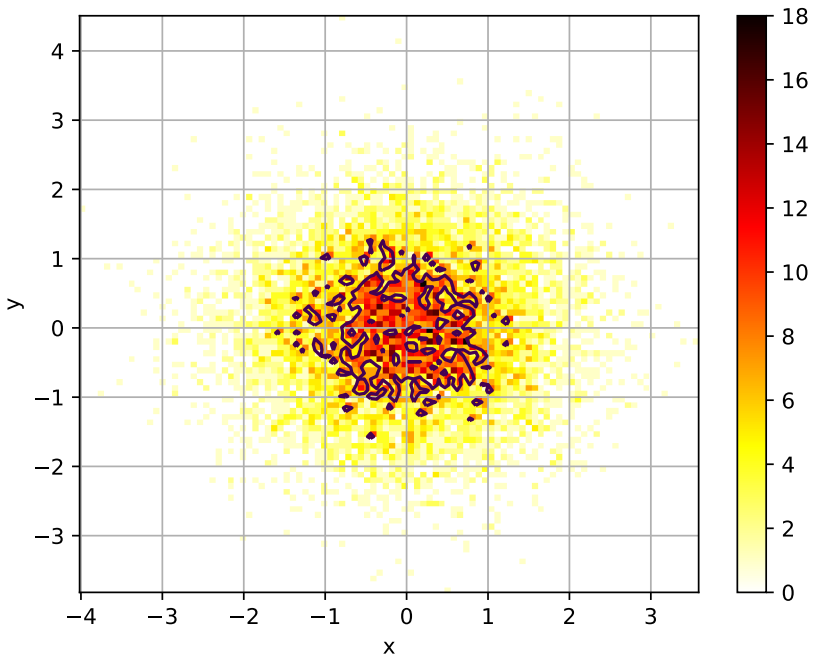
- **size** – size of the markers
- **alpha** – transparency of the marker
- **fontsize** –
- **xlabel** –
- **ylabel** –
- **pvalue_threshold** – adds an horizontal dashed line at the threshold provided.
- **fold_change_threshold** – colors in grey the absolute fold changes below a given threshold.

2D histograms

**class Hist2D**(*x*, *y=None*, *verbose=False*)

2D histogram

```
from numpy import random
from biokit.viz import hist2d
X = random.randn(10000)
Y = random.randn(10000)
h = hist2d.Hist2D(X,Y)
h.plot(bins=100, contour=True)
```




### constructor

**Parameters**

- **x** – an array for X values. See *VizInput2D* for details.
- **y** – an array for Y values. See *VizInput2D* for details.

**plot**(*bins=100*, *cmap='hot_r'*, *fontsize=10*, *Nlevels=4*, *xlabel=None*, *ylabel=None*, *norm=None*, *range=None*, *normed=False*, *colorbar=True*, *contour=True*, *grid=True*, ***kargs*)
plots histogram of mean across replicates versus coefficient variation

**Parameters**

- **bins** (*int*) – binning for the 2D histogram (either a float or list of 2 binning values).
- **cmap** – a valid colormap (defaults to hot_r)
- **fontsize** – fontsize for the labels
- **Nlevels** (*int*) – must be more than 2
- **xlabel** (*str*) – set the xlabel (overwrites content of the dataframe)
- **ylabel** (*str*) – set the ylabel (overwrites content of the dataframe)
- **norm** – set to 'log' to show the log10 of the values.
- **normed** – normalise the data
- **range** – as in pylab.Hist2D : a 2x2 shape [[-3,3],[-4,4]]
- **contour** – show some contours (default to True)
- **grid** (*bool*) – Show unerlying grid (defaults to True)

If the input is a dataframe, the xlabel and ylabel will be populated with the column names of the dataframe.

Core function for the plotting tools

**class VizInput2D**(*x*, *y=None*, *verbose=False*)

### 2.2.3 Tools to handle R packages

utilities related to R language (e.g., RPackageManager, RSession)

**class RSession**(*RExecutable='R'*, *max_len=1000*, *use_dict=None*, *host='localhost'*, *user=None*, *ssh='ssh'*, *verbose=False*, *return_err=True*)
Interface to a R session

This class uses the pyper package to provide an access to R (via a subprocess). You can call R script and get back the results into the session as Python objects. Returned objects may be transformed into numpy arrays or Pandas datafranes.

Here is a very simple example but any complex R scripts can be provided inside the `run()` method:

```
from biokit.rtools import RSession
session = RSession()
session.run("mylist = c(1,2,3)")
a = session("mylist") # access to the R object
a.sum() # a is numpy array
```

There are different ways to access to the R object:

```
# getter
session['a']
# method-wise:
```

(continues on next page)

(continued from previous page)

```
session.get('a')
# attribute:
session.a
```

For now, this is just to inherit from **pyper.R** class but there is no additional features. This is to create a common API.

**Parameters**

- **RCMD** (*str*) – the name of the R executable
- **max_len** – define the upper limitation for the length of command string. A command string will be passed to R by a temporary file if it is longer than this value.
- **host** – The computer name (or IP) on which the R interpreter is installed. The value "localhost" means that the R locates on the the localhost computer. On POSIX systems (including Cygwin environment on Windows), it is possible to use R on a remote computer if the command "ssh" works. To do that, the user need set this value, and perhaps the parameter "user".
- **user** – The user name on the remote computer. This value need to be set only if the user name is different on the remote computer. In interactive environment, the password can be input by the user if prompted. If running in a program, the user need to be able to login without typing password! (i.e., you need to set your SSH keys)
- **ssh** – The program to login to remote computer.
- **kargs** – must be empty. Error raised otherwise

**get_version**()
Return the R version

**bool2R**(*value*)
Transforms a boolean into a R boolean value T or F

**rcode**(*code*, *verbose=True*)
Run a R script and return the RSession object

**get_R_version**()
Return R version

**biocLite**(*package=None*, *suppressUpdates=True*, *verbose=True*)
Install a bioconductor package

This function does not work like the R function. Only a few options are implemented so far. However, you can use rcode function directly if needed.

**Parameters**

- **package** (*str*) – name of the bioconductor package to install. If None, no package is installed but installed packages are updated. If not provided, biocLite itself may be updated if needed.
- **suppressUpdates** (*bool*) – updates the dependencies if needed (default is False)

**Returns** True if update is required or the required package is installed and could be imported. False otherwise.

```
>>> from biokit.viz.rtools import biocLite
>>> biocLite("CellNOptR")
```

**class RPackage**(*name*, *version_required=None*, *install=False*, *verbose=False*)

```
>>> from biokit.rtools import package
>>> p = package.RPackage('CellNOptR')
>>> p.isinstalled
True
>>> p.version
'1.11.3'
```

**Todo:** do we need the version_required attribute/parameter anywhere ?

**Note:** R version includes dashes, which are not recognised by distutils so they should be replaced.

**install**()

**isinstalled**

**version**

**install_package**(*query*, *dependencies=False*, *verbose=True*, *repos='http://cran.univ-paris1.fr/'*)
Install a R package

**Parameters**

- **query** (*str*) – It can be a valid URL to a R package (tar ball), a CRAN package, a path to a R package (tar ball), or simply the directory containing a R package source.
- **dependencies** (*bool*) –
- **repos** – if provided, install_packages automatically select the provided repositories otherwise a popup window will ask you to select a repo

```
>>> rtools.install_package("path_to_a_valid_Rpackage.tar.gz")
>>> rtools.install_package("http://URL_to_a_valid_Rpackage.tar.gz")
>>> rtools.install_package("hash") # a CRAN package
>>> rtools.install_package("path to a valid R package directory")
```

**See also:**

biokit.rtools.RPackageManager

**class RPackageManager**(*verbose=True*)
Implements a R package manager from Python

So far you can install a package (from source, or CRAN, or biocLite)

```
pm = PackageManager()
[(x, pm.installed[x][2]) for x in pm.installed.keys()]
```

You can access to all information within a dataframe called **packages** where indices are the name packages. Some aliases are provided as attributes (e.g., available, installed)

**available**

**biocLite**(*package=None*, *suppressUpdates=True*, *verbose=False*)
Installs one or more biocLite packages

**Parameters** **package** – a package name (string) or list of package names (list of strings) that will be installed from BioConductor. If package is set to None, all packages already installed will be updated.

**cran_repos = 'http://cran.univ-lyon1.fr/'**

**get_package_latest_version**(*package*)
Get latest version available of a package

**get_package_version**(*package*)
Get version of an install package

**install**(*pkg*, *require=None*, *update=True*, *reinstall=False*)
install a package automatically scanning CRAN and biocLite repos

if require is not set and update is True, when a newest version of a package is available, it is installed

**installed**

**is_installed**(*pkg_name*)

**packages**

**remove**(*package*)
Remove a package (or list) from local repository

**require**(*pkg*, *version*)
Check if a package with given version is available

**update**()
If you install/remove packages yourself elsewhere, you may need to call this function to update the package manager

**BoolStr**(*obj*)

**ReprStr**(*obj*)

**FloatStr**(*f*)

**LongStr**(*obj*)

**ComplexStr**(*obj*)

**UniStr**(*obj*)

**ByteStr**(*obj*)

**SeqStr**(*obj*, *head='c('*, *tail=')'*, *enclose=True*)

**getVec**(*ary*)

**NumpyNdarrayStr**(*obj*)

**PandasDataFrameStr**(*obj*)

**PandasSerieStr**(*obj*)

**OtherStr**(*obj*)

**Str4R**(*obj*)
convert a Python basic object into an R object in the form of string.

Code use by pyper module original version available from pyper pacakge on pip

### 2.2.4 Genomics

Sequence related (Generic, DNA, RNA)

**class Sequence**(*data=''*)
Common data structure to all sequences (e.g., *DNA()*)

A sequence is a string contained in the `_data`. If you manipulate this attribute, you should also changed the `_N` (length of the string) and set `_counter` to None.

Sequences can be concatenated easily. You can also add a string or numpy array or pandas time series to an existing sequence:

```
d1 = Sequence('ACGT')
d2 = Sequence('ACGT')
```

Note that there is a `check()` method, which is not called during the instanciation but is called when adding sequences together. Each type of sequence (e.g., Sequence, DNA, RNA) has its own symbols. So you cannot add a DNA sequence with a RNA sequence for instance. Those are valid operation:

```
>>> d1 = Sequence('ACGT')
>>> d1 += 'AAAA'
>>> d1 + d1
>>> "AAAA" + d1
```

**N**

**counter**
return counter of the letters

**hamming_distance**(*other*)
Return hamming distance between this sequence and another sequence

The Hamming distance between s and t, denoted dH(s,t), is the number of corresponding symbols that differ in s and t.

```
>>> d1 = 'GAGCCTACTAACGGGAT'
>>> d2 = 'CATCGTAATGACGGCCT'
>>> s = Sequence(d1)
>>> s.hamming_distance(d2)
7
```

**histogram**()

**lower**()
convertes sequence string to lowercase (inplace)

**pie**()

**sequence**
returns a copy of the sequence

**upper**()
convertes sequence string to uppercase (inplace)

DNA sequence

**class DNA**(*data=''*)
a DNA *Sequence*.

You can add DNA sequences together:

```
>>> from biokit import DNA
>>> s1 = DNA('ACGT')
>>> s2 = DNA('AAAA')
>>> s1 + s2
Sequence: ACGTAAAA (length 8)
```

**complement**

**gc_content**(*letters='CGS'*)
Returns the G+C content in percentage.

Copes mixed case sequences, and with the ambiguous nucleotide S (G or C) when counting the G and C content.

```
>>> from biokit.sequence.dna import DNA
>>> d = DNA("ACGTSAAA")
>>> d.gc_content()
0.375
```

**get_complement**()

**get_reverse_complement**()

**get_rna**()

**reverse_complement**

RNA sequence

**class RNA**(*sequence=''*)

**complement**

**gc_content**(*letters='CGS'*)
Returns the G+C content in percentage.

Copes mixed case sequences, and with the ambiguous nucleotide S (G or C) when counting the G and C content.

```
>>> from biokit.sequence.dna import RNA
>>> d = RNA("ACGTSAAA")
>>> d.gc_content()
0.375
```

**get_complement**()

**get_dna**()

**get_reverse_complement**()

**reverse_complement**

**class Peptide**(*data*)
a Peptide *Sequence*.

You can add Peptide sequences together:

```
>>> from biokit import DNA
>>> s1 = Peptide('ACGT')
>>> s2 = Peptide('AAAA')
>>> s1 + s2
Sequence: ACGTAAAA (length 8)
```

**Note:** redundant with Sequence but may evolve in the future.

**class Lineage**(*lineage*)

**class Taxon**(*taxid*)
Some codecs

**to_genbank**(*retmax=10000*)
Draft: from a TaxID, uses EUtils to retrieve the GenBank identifiers

**Inspiration** https://gist.github.com/fjossinet/5673672

**class Taxonomy**(*verbose=True*, *online=True*)
This class should ease the retrieval and manipulation of Taxons

There are many resources to retrieve information about a Taxon. For instance, from BioServices, one can use UniProt, Ensembl, or EUtils. This is convenient to retrieve a Taxon (see *fetch_by_name()* and *fetch_by_id()* that rely on Ensembl). However, you can also download a flat file from EBI ftp server, which stores a set or records (1.3M at the time of the implementation).

Note that the Ensembl database does not seem to be as up to date as the flat files but entries contain more information.

for instance taxon 2 is in the flat file but not available through the *fetch_by_id()*, which uses ensembl.

So, you may access to a taxon in 2 different ways getting differnt dictionary. However, 3 keys are common (id, parent, scientific_name)

```
>>> t = taxonomy.Taxonomy()
>>> t.fetch_by_id(9606)    # Get a dictionary from Ensembl
>>> t.records[9606] # or just try with the get
>>> t[9606]
>>> t.get_lineage(9606)
```

### constructor

**Parameters** **offline** – if you do not have internet, the connction to Ensembl may hang for a while and fail. If so, set **offline** to True

**fetch_by_id**(*taxon*)
Search for a taxon by identifier

:return; a dictionary.

```
>>> ret = s.search_by_id('10090')
>>> ret['name']
'Mus Musculus'
```

**fetch_by_name**(*name*)
Search a taxon by its name.

**Parameters** **name** (*str*) – name of an organism. SQL cards possible e.g., _ and % characters.

**Returns** a list of possible matches. Each item being a dictionary.

```
>>> ret = s.search_by_name('Mus Musculus')
>>> ret[0]['id']
10090
```

**get_children**(*taxon*)

**get_family_tree**(*taxon*)
root is taxon and we return the corresponding tree

**get_lineage**(*taxon*)
Get lineage of a taxon

**Parameters** **taxon** (*int*) – a known taxon

**Returns** list containing the lineage

**get_lineage_and_rank**(*taxon*)
Get lineage and rank of a taxon

**Parameters** **taxon** (*int*) –

**Returns** a list of tuples. Each tuple is a pair of taxon name/rank The list is the lineage for to the input taxon.

**load_records**(*overwrite=False*)
Load a flat file and store records in `records`

**on_web**(*taxon*)
Open UniProt page for a given taxon

### 2.2.5 Stats

Generic statistical tools

**class AdaptativeMixtureFitting**(*data*, *method='Nelder-Mead'*)
Automatic Adaptative Mixture Fitting

```
from biokit.stats.mixture import AdaptativeMixtureFitting, GaussianMixture
m = GaussianMixture(mu=[-1,1], sigma=[0.5,0.5], mixture=[0.2,0.8])
amf = AdaptativeMixtureFitting(m.data)
amf.run(kmin=1, kmax=6)
amf.diagnostic(k=amf.best_k)
```

**diagnostic**(*kmin=1*, *kmax=8*, *k=None*, *ymax=None*)

**plot**(*criteria='AICc'*)

**run**(*kmin=1*, *kmax=6*, *criteria='AICc'*)

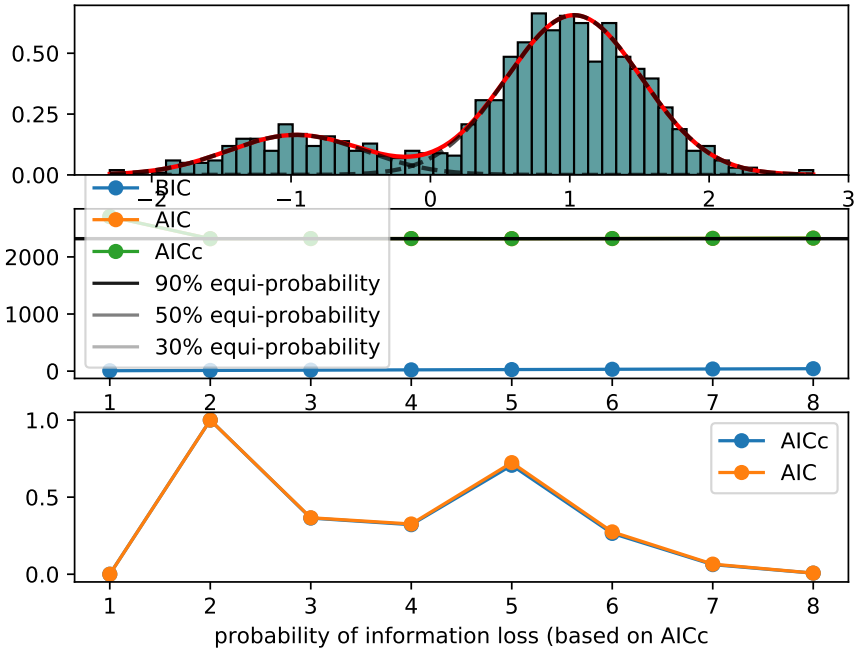
**class EM**(*data*, *model=None*, *max_iter=100*)
Expectation minimization class to estimate parameters of GMM

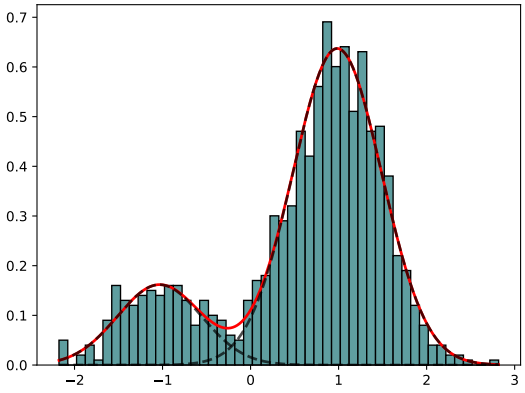
```
from biokit.stats.mixture import GaussianMixture, EM
m = GaussianMixture(mu=[-1,1], sigma=[0.5,0.5], mixture=[0.2,0.8])
em = EM(m.data)
em.estimate(k=2)
em.plot()
```

#### constructor

**Parameters**

- **data** –


BIC
AIC
AICc
90% equi-probability
50% equi-probability
30% equi-probability
AICc
AIC
probability of information loss (based on AICc



- **model** – not used. Model is the *GaussianMixtureModel* but could be other model.
- **max_iter** (*int*) – max iteration for the minization

**estimate**(*guess=None*, *k=2*)

**Parameters**

- **guess** (*list*) – a list to provide the initial guess. Order is mu1, sigma1, pi1, mu2, ...
- **k** (*int*) – number of models to be used.

**plot**(*model_parameters=None*, ***kwargs*)

Take a list of dictionnaries with models parameters to plot predicted models. If user doesn't provide parameters, the standard plot function from fitting is used.

**Example:** model_parameters=[{"mu": 5, "sigma": 0.5, "pi": 1}]

**class Fitting**(*data*, *k=2*, *method='Nelder-Mead'*)

Base class for *EM* and *GaussianMixtureFitting*

### constructor

**Parameters**

- **data** (*list*) –
- **k** (*int*) – number of GMM to use
- **method** (*str*) – minimization method to be used (one of scipy optimise module)

**get_guess**()

Random guess to initialise optimisation

**k**

**model**

**plot**(*normed=True*, *N=1000*, *Xmin=None*, *Xmax=None*, *bins=50*, *color='red'*, *lw=2*, *hist_kw={'color': '#5F9EA0', 'edgecolor': 'k'}*, *ax=None*)

**class GaussianMixture**(*mu=[-1, 1], sigma=[1, 1], mixture=[0.5, 0.5], N=1000*)

Creates a mix of Gaussian distribution

```
from biokit.stats.mixture import GaussianMixture
m = GaussianMixture(mu=[-1,1], sigma=[0.5, 0.5], mixture=[.2, .8], N=1000)
m.plot()
```

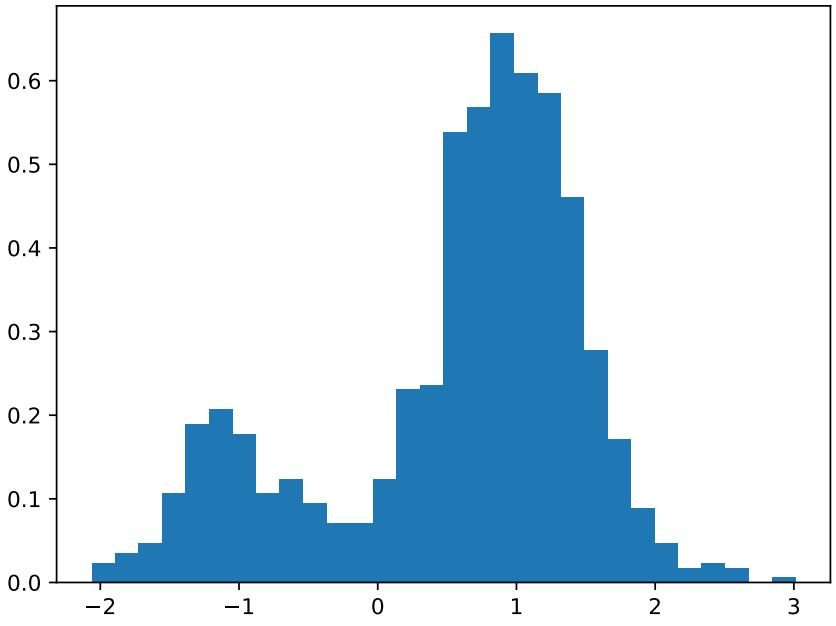
### constructor

**Parameters**

- **mu** (*list*) – list of mean for each model
- **sigma** (*list*) – list of standard deviation for each model
- **mixture** (*list*) – list of amplitude for each model

**hist**(*bins=30*, *normed=True*)

**plot**(*bins=30*, *normed=True*)




**class GaussianMixtureFitting**(*data*, *k=2*, *method='Nelder-Mead'*)

GaussianMixtureFitting using scipy minization

```
from biokit.stats.mixture import GaussianMixture, GaussianMixtureFitting
m = GaussianMixture(mu=[-1,1], sigma=[0.5,0.5], mixture=[0.2,0.8])
mf = GaussianMixtureFitting(m.data)
mf.estimate(k=2)
mf.plot()
```

Here we use the function minimize() from scipy.optimization. The list of (currently) available minimization methods is 'Nelder-Mead' (simplex), 'Powell', 'CG', 'BFGS', 'Newton-CG',> 'Anneal', 'L-BFGS-B' (like BFGS but bounded), 'TNC', 'COBYLA', 'SLSQPG'.

**estimate**(*guess=None*, *k=None*, *maxfev=20000.0*, *maxiter=1000.0*, *bounds=None*)

guess is a list of parameters as expected by the model

guess = {'mus':[1,2], 'sigmas': [0.5, 0.5], 'pis': [0.3, 0.7] }

**method**

**class GaussianMixtureModel**(*k=2*)

Gaussian Mixture Model

**log_likelihood**(*params*, *sample*)

**pdf**(*x*, *params*, *normalise=True*)

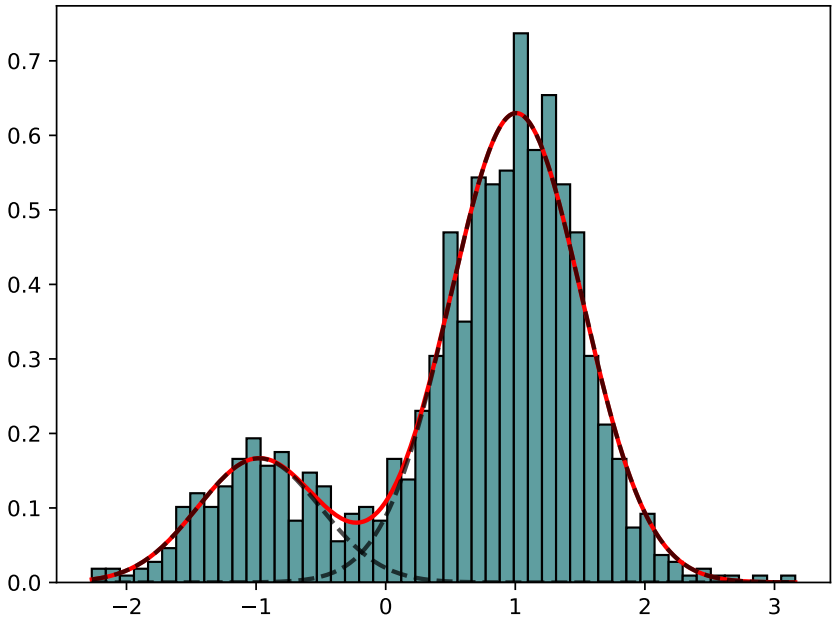
Expected parameters are

params is a list of gaussian distribution ordered as mu, sigma, pi, mu2, sigma2, pi2, ...

**class GaussianModel**(*mu=1*, *sigma=1*)

New gaussian model not used yet

**estimate**(*data*, *weights=None*)




**generate**(*N*)

**log_density**(*data*)

**pdf**(*data*)

**class Model**

New base model

**estimate**(*data*, *weights*)

**generate**()

**log_density**(*data*)

**pdf**()

**class PoissonModel**(*lmbda=10*)

New poisson model not used yet

**estimate**(*data*, *weights=None*)

**generate**(*N*)

**log_density**(*data*)

Akaike and other criteria

**AIC**(*L*, *k*, *logL=False*)

Return Akaike information criterion (AIC)

**Parameters**

- **L** (*float*) – maximised value of the likelihood function
- **k** (*int*) – number of parameters

- **logL** (*bool*) – L is the log likelihood.

Suppose that we have a statistical model of some data, from which we computed its likelihood function and let $k$ be the number of parameters in the model (i.e. degrees of freedom). Then the AIC value is:

$\mathrm{AIC} = 2k - 2\ln(L)$

Given a set of candidate models for the data, the preferred model is the one with the minimum AIC value. Hence AIC rewards goodness of fit (as assessed by the likelihood function), but it also includes a penalty that is an increasing function of the number of estimated parameters. The penalty discourages overfitting.

Suppose that there are R candidate models AIC1, AIC2, AIC3, AICR. Let AICmin be the minimum of those values. Then, exp((AICmin - AICi)/2) can be interpreted as the relative probability that the ith model minimizes the (estimated) information loss.

Suppose that there are three candidate models, whose AIC values are 100, 102, and 110. Then the second model is exp((100 - 102)/2) = 0.368 times as probable as the first model to minimize the information loss. Similarly, the third model is exp((100-110)/2) = 0.007 times as probable as the first model, which can therefore be discarded.

With the remaining two models, we can (1) gather more data, (2) conclude that the data is insufficient to support selecting one model from among the first two (3) take a weighted average of the first two models, with weights 1 and 0.368.

The quantity *exp((AIC_{min}-AIC_i)/2)* is the relative likelihood of model i.

If all the models in the candidate set have the same number of parameters, then using AIC might at first appear to be very similar to using the likelihood-ratio test. There are, however, important distinctions. In particular, the likelihood-ratio test is valid only for nested models, whereas AIC (and AICc) has no such restriction.

**Reference** Burnham, K. P.; Anderson, D. R. (2002), Model Selection and Multimodel Inference: A Practical Information-Theoretic Approach (2nd ed.), Springer-Verlag, ISBN 0-387-95364-7.

**AICc**(*L*, *k*, *n*, *logL=False*)
AICc criteria

**Parameters**

- **L** (*float*) – maximised value of the likelihood function
- **k** (*int*) – number of parameters
- **n** (*int*) – sample size
- **logL** (*bool*) – L is the log likelihood.

AIC with a correction for finite sample sizes. The formula for AICc depends upon the statistical model. Assuming that the model is univariate, linear, and has normally-distributed residuals (conditional upon regressors), the formula for AICc is as follows:

AICc is essentially AIC with a greater penalty for extra parameters. Using AIC, instead of AICc, when n is not many times larger than k2, increases the probability of selecting models that have too many parameters, i.e. of overfitting. The probability of AIC overfitting can be substantial, in some cases.

**BIC**(*L*, *k*, *n*, *logL=False*)
Bayesian information criterion

**Parameters**

- **L** (*float*) – maximised value of the likelihood function
- **k** (*int*) – number of parameters
- **n** (*int*) – sample size
- **logL** (*bool*) – L is the log likelihood.

Given any two estimated models, the model with the lower value of BIC is the one to be preferred.

## 2.3 Glossary

**BAI** The index file for a file generated in the BAM format. (This is a non-standard file type.)

**BAM** Binary version of the Sequence Alignment Map (SAM) format.

**BED** Format that defines the data lines displayed in an annotation track.

**DSRC** A compression tool dedicated to FastQ files

**FASTA** FASTA-formatted sequence files contains either nucleic acid sequence (such as DNA) or protein sequence information. FASTA files store multiple sequences in a single file.

**GFF** General Feature Format, used for describing genes and other features associated with DNA, RNA and Protein sequences.

**JSON** A human-readable data serialization language commonly used in configuration files. See https://en.wikipedia.org/wiki/JSON

**SAM** Sequence Alignment Map is a generic nucleotide alignment format that describes the alignment of query sequences or sequencing reads to a reference sequence or assembly

**VCF** Variant Call Format, for use with the variant calling pipeline

**YAML** A human-readable data serialization language commonly used in configuration files. See https://en.wikipedia.org/wiki/YAML

## 2.4 Whats' new, what has changed

**Revision 0.4.3**

- move all converters related code (now in biokit/bioconvert project)
- Fixed deprecated issues in stats/mixture (pylab.normpdf changed to scipy.stats.norm.pdf
- Fix doc gallery
- BUG: fix fasta module test and cleanup FASTA class

**Revision 0.4.2**

- UPDATE: starting converter tool in converter sub-package
- UPDATE: replace axibg with facecolor (matplotlib deprecated name)
- REFACTORING: rtools sub-package.should not affect user interface
- BUG: fixed corrplot layout due to new matplotlib API and new Pandas API

**Revision** 0.4.1

- BUG: fix rtools sub-packages for Python3 portage

**Revision** 0.4.0

- Stable version on bioconda (remove sphinx-gallery from dependencies since it is only used for the documentation.

**Revision** 0.3.4

- remove stats_dev sub_directory

**Revision** 0.3.3

- EM class has a new plot function so that a plot can be run without doing the estimation but by providing the parameter of the model. This is a better model/data/view abstraction.
- Fix tests in documentation

**Revision** 0.3.2:

- remove deprecated warning

**Revision** 0.3.1:

- Update mixture.EM method to speedup the code by factor 4

**Revision** 0.3.0:

- Cleanup and doc update
- Update notebooks to be py3.5 compatible
- Some API changes in the Taxonomy module used in Sequana package

**Revision 0.2.0**

- NEWS
  - add boxplot module.

**Revision 0.1.4**

- BUG FIXES: cleanup MANIFEST

**Revision 0.1.3**

- BUG FIXES:
  - a py3 typo.
  - fixing complexes module
  - remove useless db package

**Revision 0.0.7**

- NEWS
  - add taxonomy module.
  - add goid module.
  - Fixed bunch of Python3 issues. most important in rtools packages to use Popen instead of popen4 and manipulate bytes vs strings.
  - All tests passes under Python2.7 and Python3.4

**Revision 0.0.6**

- CHANGES
  - viz package:
    - refactored most of the functions/classes to be have more consistent input data for the constructor and more consistent parameters for the plot() methods.
    - Hist2d is now called Hist2D
- BUG FIXES
- NEWS

- add new module in viz package: hinton, core (to factorise code)
- add new notebooks related to the viz package.

CHAPTER 3

# Examples in notebooks

Set of Notebooks

# Python Module Index

## n



## r



## s



## t



## v

# Index

## R



## S



## T



## U



## V



## Y