data_tools Documentation

Release 0.0.6

Nicolàs Palacio-Escat

Mar 22, 2019

CONTENTS

Data tools: a collection of Python functions and classes designed to make data scientists' life easier.

Contact: nicolas.palacio@bioquant.uni-heidelberg.de

CHAPTER

ONE

DISCLAIMER

This package is still under development and will be periodically updated with new features. Contributions are very welcome (fork + pull request). If you find any bug or suggestion for upgrades, please use the issue system.

GNU-GLPv3: This program is free software: you can redistribute it and/or modify it under the terms of the GNU General Public License as published by the Free Software Foundation.

This program is distributed in the hope that it will be useful, but WITHOUT ANY WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the GNU General Public License for more details.

A full copy of the GNU General Public License can be found on file LICENSE.md. If not, see http://www.gnu.org/licenses/.

CHAPTER

TWO

DEPENDENCIES

- NumPy
- Matplotlib
- Pandas
- SciPy
- Scikit-learn

CHAPTER

THREE

INSTALLATION

First download/clone `data_tools` from the GitHub repository. From the terminal:

```
git clone https://github.com/Nic-Nic/data_tools.git
cd data_tools
```

Then you can install it by running `setup.py` as follows:

```
python setup.py sdist
```

Or using `pip`:

```
pip install .
```

Along with `data_tools`, all dependencies will be installed as well as the testing suite. In order to run the tests, type on the terminal:

```
python -m test_data_tools
```

NOTE: `data_tools.plots` module does not have any tests implemented.

CHAPTER

FOUR

MODULES

4.1 data_tools.databases

Databases functions module.

data_tools.databases.**kegg_link**(*query*, *target='pathway'*)

Queries a request to the KEGG database to find related entries using cross-references. A list of available database(s) and query examples can be found in https://www.kegg.jp/kegg/rest/keggapi.html#link.

- **Arguments:**
  - *query* [list]: Or any iterable type containing the identifier(s) to be queried as [str]. These can be either valid database identifiers or databases *per se* (see the link above).
  - *target* [str]: Optional, `'pathway'` by default. Targeted database to which the query should be linked to. You can check other options available in the URL above.
- **Returns:**
  - [pandas.DataFrame]: Two-column table containing both the input query identifiers and their linked ones.
- **Example:**

```
>>> my_query = ['hsa:10458', 'ece:Z5100']
>>> kegg_link(my_query)
   query        pathway
0  hsa:10458  path:hsa04520
1  hsa:10458  path:hsa04810
2  ece:Z5100  path:ece05130
```

data_tools.databases.**kegg_pathway_mapping**(*df*, *mapid*, *filename=None*)

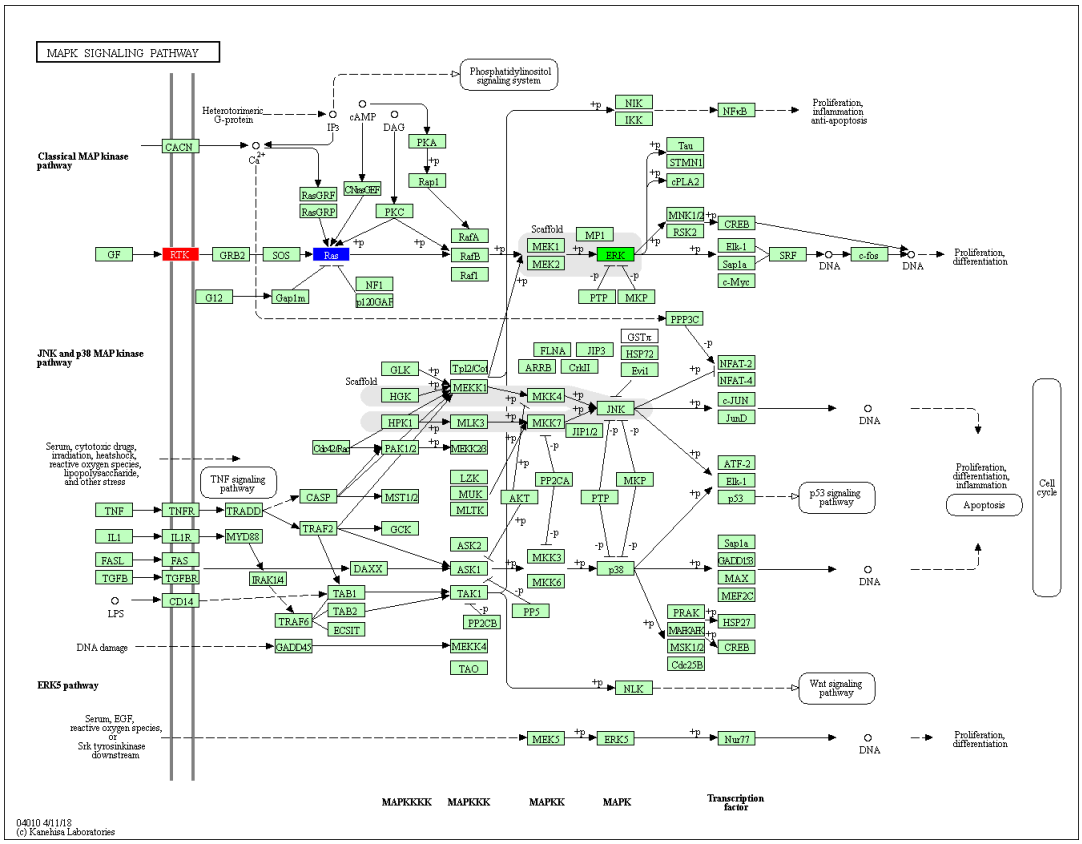
Makes a request to KEGG pathway mapping tool according to a given pathway ID (see https://www.kegg.jp/kegg/tool/map_pathway2.html for more information). The user must provide a query of IDs to be mapped with their corresponding background colors (and optionally also foreground colors). The result is downloaded in the current directory or a user-specified path.

- **Arguments:**
  - *df* [pandas.DataFrame]: Dataframe containing KEGG valid IDs in the first column and corresponding background colors (e.g.: red, blue, ...). Optionally, a third column with the foreground (font) colors can also be provided (black by default). **NOTE:** hexadecimal codes for colors is also supported. Index and column names of dataframe are ignored.
  - *mapid* [str]: A valid KEGG pathway ID. It can be a general (e.g.: "mapXXXXX") or organism-specific ID (e.g.: "hsaXXXXX").

  - *filename* [str]: Optional, `None` by default. This is, the image will be stored in the current directory with the *mapid* provided as file name. If provided, the image will be stored within the specified path/file name.

- **Example:**

```
>>> my_query = pandas.DataFrame([['1956', 'red', '#f1f1f1'],
...                              ['3845', 'blue', '#f1f1f1'],
...                              ['5594', 'green', 'black']])
>>> kegg_pathway_mapping(my_query, 'hsa04010')
```



`data_tools.databases.`**`op_kinase_substrate`**(*organism='9606'*, *incl_phosphatases=False*)

Queries OmniPath to retrieve the kinase-substrate interactions for a given organism.

- **Arguments:**

  - *organism* [str]: Optional, `'9606'` by default (Homo sapiens). NCBI taxonomic identifier for the organism of interest.

  - *incl_phosphatases* [bool]: Optional `False` by default. Determines wether to include dephosphorylation interactions or not.

- **Returns:**

  - [pandas.DataFrame]: Table containing the enzyme-substrate (kinase/phosphatase-target) network of each phospho-site.

`data_tools.databases.`**`up_map`**(*query*, *source='ACC'*, *target='GENENAME'*)

Queries a request to UniProt.org in order to map a given list of identifiers. You can check the options available of input/output identifiers at https://www.uniprot.org/help/api_idmapping.

- **Arguments:**

  - *query* [list]: Or any iterable type containing the identifiers to be queried as [str].
  - *source* [str]: Optional, `'ACC'` by default. This is, UniProt accesion number. You can check other options available in the URL above.
  - *target* [str]: Optional, `'GENENAME'` by default. You can check other options available in the URL above.

- **Returns:**
  - [pandas.DataFrame]: Two-column table containing both the inputed identifiers and the mapping result of these. **NOTE:** The returned table may not have the same order as in `query`. Also, if some ID could not be mapped, the size of the returned table will differ from the length of `query`.

- **Examples:**

```
>>> my_query = ['P00533', 'P31749', 'P16220']
>>> up_map(my_query)
      ACC GENENAME
0  P00533     EGFR
1  P31749     AKT1
2  P16220    CREB1
>>> up_map(my_query, target='KEGG_ID')
      ACC   KEGG_ID
0  P00533  hsa:1956
2  P16220  hsa:1385
1  P31749   hsa:207
```

4.2 data_tools.diffusion

Diffusion solvers module.

The following functions provide tools to compute the diffusion on one or two dimensions with different explicit or implicit methods.

NOTE: Explicit methods (`'euler_explicit1D'` and `'euler_explicit2D'`) are conditionally stable. This means that in order to keep numerical stability of the solution (and obtain an accurate result), these methods need to fulfill the Courant–Friedrichs–Lewy (CFL) condition. For the one-dimensional case:

$$D\frac{\Delta t}{\Delta x^2} \leq \frac{1}{2}$$

For the two-dimensional case (and assuming $\Delta x = \Delta y$):

$$D\frac{\Delta t}{\Delta x^2} \leq \frac{1}{4}$$

The implicit methods are (theoretically) unconditionally stable, hence are more permissive in terms of discretization step-size.

Currently for the implicit methods only the coefficient matrix contruction is available. To solve the diffusion problem user can use any of the available linear algebra solvers by providing the current diffusing field state and the matrix on each time-step.

- Simplest options are either `numpy.linalg.solve()` or `scipy.linalg.solve()` (both not very fast).
- If the coefficient matrix is positive-definite (it is most of the times, but can be double-checked, specially if errors arise) and symmetric, a good option is Choleski's factorization. This is already implemented in `scipy.linalg.cholesky()` which factorizes the coefficient matrix and that can be passed to the `scipy.linalg.cho_solve()` which is way faster than the option above.

- Another option (but don't tell anyone) is to invert the coefficient matrix and just solve the equation with a matrix multiplication. This is way faster but your coefficient matrix has to be invertible. If the determinant is close to zero, may cause numerical instability.

data_tools.diffusion.**euler_explicit1D**(*x0*, *dt*, *dx2*, *d=1*, *bcs='periodic'*)
Computes diffusion on a 1D space over a time-step using Euler explicit method.

- **Arguments:**
  - *x0* [numpy.ndarray]: Initial state of a 1D array from which the difusion is to be computed.
  - *dt* [float]: Discretization time-step.
  - *dx2* [float]: Discretization spatial-step (squared).
  - *d* [float]: Optional, `1` by default. The diffusion coefficient.
  - *bcs* [str]: Optional, `'periodic'` by default. Determines the boundary conditions. Available options are `'periodic'`, `'dirichlet''` or `'neumann'`. Note that Dirichlet BCs do not hold mass conservation.
- **Returns:**
  - [numpy.ndarray]: Computed state array (1D) after one time-step according to the parameters and conditions selected.

data_tools.diffusion.**euler_explicit2D**(*x0*, *dt*, *dx2*, *d=1*, *bcs='periodic'*)
Computes diffusion on a 2D space over a time-step using Euler explicit method.

- **Arguments:**
  - *x0* [numpy.ndarray]: Initial state of a 2D array from which the difusion is to be computed.
  - *dt* [float]: Discretization time-step.
  - *dx2* [float]: Discretization spatial-step (squared). It is assumed that is the same in both dimensions (dx = dy).
  - *d* [float]: Optional, `1` by default. The diffusion coefficient.
  - *bcs* [str]: Optional, `'periodic'` by default. Determines the boundary conditions. Available options are `'periodic'`, `'dirichlet''` or `'neumann'`. Note that Dirichlet BCs do not hold mass conservation.
- **Returns:**
  - [numpy.ndarray]: Computed state array (2D) after one time-step according to the parameters and conditions selected.

data_tools.diffusion.**euler_implicit_coef_mat**(*dx2*, *dt*, *nx*, *ny=None*, *d=1*, *bcs='periodic'*)
Computes the coefficient matrix to solve the diffusion problem with the Euler implicit method such that:

$$\mathbf{A} \cdot u^{n+1} = u^n$$

Where **A** is the coefficient matrix and *u* the diffusing field. Note that for a 2D space, it is considered that *u* has been vectorized beforehand (e.g.: *u.reshape(-1)* assuming *u* is a 2D [*numpy.ndarray*]).

- **Arguments:**
  - *dx2* [float]: Discretization spatial-step (squared). It is assumed that is the same in both dimensions (dx = dy).
  - *dt* [float]: Discretization time-step.
  - *nx* [int]: The number of discrete steps in the first spatial dimension.

- *ny* [int]: Optional, `None` by default. The number of discrete steps in the second spatial dimension (if any).
- *d* [float]: Optional, `1` by default. The diffusion coefficient.
- *bcs* [str]: Optional, `'periodic'` by default. Determines the boundary conditions. Available options are `'periodic'`, `'dirichlet''` or `'neumann'`. Note that Dirichlet BCs do not hold mass conservation.

- **Returns:**
  - [numpy.ndarray]: The coefficient matrix. Shape is *[nx, nx]* for one-dimensional problem and *[nx*ny, nx*ny]* for the two-dimensional case.

data_tools.diffusion.**crank_nicolson_coef_mats**(*dx2*, *dt*, *nx*, *ny=None*, *d=1*, *bcs='periodic'*)

Computes the coefficient matrices to solve the diffusion problem with the Crank-Nicolson method such that:

$$\mathbf{B} \cdot u^{n+1} = \mathbf{D} \cdot u^n$$

Where **B** and **D** are the coefficient matrices and *u* the diffusing field. Note that for a 2D space, it is considered that *u* has been vectorized beforehand (e.g.: *u.reshape(-1)* assuming *u* is a 2D [*numpy.ndarray*]).

- **Arguments:**
  - *dx2* [float]: Discretization spatial-step (squared). It is assumed that is the same in both dimensions (dx = dy).
  - *dt* [float]: Discretization time-step.
  - *nx* [int]: The number of discrete steps in the first spatial dimension.
  - *ny* [int]: Optional, `None` by default. The number of discrete steps in the second spatial dimension (if any).
  - *d* [float]: Optional, `1` by default. The diffusion coefficient.
  - *bcs* [str]: Optional, `'periodic'` by default. Determines the boundary conditions. Available options are `'periodic'`, `'dirichlet''` or `'neumann'`. Note that Dirichlet BCs do not hold mass conservation.

- **Returns:**
  - [numpy.ndarray]: The coefficient matrix **B**. Shape is *[nx, nx]* for one-dimensional problem and *[nx*ny, nx*ny]* for the two-dimensional case.
  - [numpy.ndarray]: The coefficient matrix **D**. Shape is the same as **B**.

data_tools.diffusion.**build_coef_mat**(*a*, *b*, *nx*, *ny=None*, *bcs='periodic'*)

Builds a coefficient matrix according to the central and neighbor coefficients, system size and boundary conditions.

- **Arguments:**
  - *a* [float]: The central element coefficient.
  - *b* [float]: The neighbor element coefficient.
  - *nx* [int]: The number of discrete steps in the first spatial dimension.
  - *ny* [int]: Optional, `None` by default. The number of discrete steps in the second spatial dimension (if any).
  - *bcs* [str]: Optional, `'periodic'` by default. Determines the boundary conditions. Available options are `'periodic'`, `'dirichlet''` or `'neumann'`. Note that Dirichlet BCs do not hold mass conservation.

- **Returns:**
  - [numpy.ndarray]: The coefficient matrix. Shape is *[nx, nx]* for one-dimensional problem and *[nx*ny, nx*ny]* for the two-dimensional case.

4.3 data_tools.iterables

Iterable-type operations module.

data_tools.iterables.**bit_or**(*a*, *b*)

Returns the bit operation OR between two bit-strings *a* and *b*. **NOTE:** *a* and *b* must have the same size.

- **Arguments:**
  - *a* [tuple]: Or any iterable type.
  - *b* [tuple]: Or any iterable type.
- **Returns:**
  - [tuple]: OR operation between *a* and *b* element-wise.
- **Examples:**

```
>>> a, b = (0, 0, 1), (1, 0, 1)
>>> bit_or(a, b)
(1, 0, 1)
```

data_tools.iterables.**chunk_this**(*L*, *n*)

For a given list *L*, returns another list of *n*-sized chunks from it (in the same order).

- **Arguments:**
  - *L* [list]: The list to be sliced into sublists of the definded size.
  - *n* [int]: The size of the chunks.
- **Returns:**
  - [list]: List of *n*-sized chunks from *L*. **NOTE:** If the number of items in *L* is not divisible by *n*, the last element returned will have an inferior size.
- **Examples:**

```
>>> L = range(6)
>>> chunk_this(L, 2)
[[0, 1], [2, 3], [4, 5]]
>>> chunk_this(L, 4)
[[0, 1, 2, 3], [4, 5]]
```

data_tools.iterables.**find_min**(*A*)

Finds and returns the subset of vectors whose sum is minimum from a given set *A*.

- **Arguments:**
  - *A* [set]: Set of vectors ([tuple] or any iterable).
- **Returns:**
  - [set]: Subset of vectors in *A* whose sum is minimum.
- **Examples:**

```
>>> A = {(0, 1, 1), (0, 1, 0), (1, 0, 0), (1, 1, 1)}
>>> find_min(A)
set([(0, 1, 0), (1, 0, 0)])
```

data_tools.iterables.**in_all**(*x*, *N*)

Checks if a element *x* is present in all collections contained in a list *N*.

- **Arguments:**
  - *x* [object]: Any type of object, it is assumed to be the same type as the objects contained in the elements of *N*.
  - *N* [list]: Or any iterable type containing a collection of other iterables containing the objects.
- **Returns:**
  - [bool]: `True` if *x* is found in all elements of *N*, `False` otherwise.
- **Examples:**

```
>>> N = [{(0, 0), (0, 1)}, # <- set A
...      {(0, 0), (1, 1), (1, 0)}] # <- set B
>>> x = (0, 0)
>>> in_all(x, N)
True
>>> y = (0, 1)
>>> in_all(y, N)
False
>>> N = [['Hello', 'world', '!'],
...      ['Hello', 'user']]
>>> x = 'Hello'
>>> in_all(x, N)
True
```

data_tools.iterables.**similarity**(*a*, *b*, *mode='j'*)

Computes the similarity index between two sets. There are three options available:

Jaccard (`mode='j'`):

$$s_J(A, B) = \frac{|A \cap B|}{|A \cup B|}$$

Sorensen-Dice (`mode='sd'`):

$$s_{SD}(A, B) = \frac{2|A \cap B|}{|A| + |B|}$$

Szymkiewicz–Simpson (`mode='ss'`):

$$s_{SS}(A, B) = \frac{|A \cap B|}{\min(|A|, |B|)}$$

- **Arguments:**
  - *a* [set]: One of the two sets to compute the similarity index.
  - *b* [set]: The other set to compute the similarity index.
  - *mode* [str]: Optional, `'j'` (Jaccard) by default. Indicates which type of similarity index/coefficient is to be computed. Available options are: `'j'` for Jaccard, `'sd'` for Sorensen-Dice and `'ss'` for Szymkiewicz–Simpson.

- **Returns:**
  - [float]: The corresponding similarity index/coefficient according to the specified *mode*.

data_tools.iterables.**subsets**(*N*)

Function that computes all possible logical relations between all sets on a list *N* and returns all subsets. This is, the subsets that would represent each intersecting area on a Venn diagram.

- **Arguments:**
  - *N* [list]: Or any iterable type containing [set] objects.
- **Returns:**
  - [dict]: Collection of subsets according to the logical relations between the sets in *N*. The keys are binary codes that denote the logical relation (see examples below).
- **Examples:**

```
>>> N = [{0, 1, 2}, {2, 3, 4}]
>>> subsets(N)
{'11': set([2]), '10': set([0, 1]), '01': set([3, 4])}
>>> N = [{0, 1}, {2, 3}, {1, 3, 4}]
>>> subsets(N)
{'010': set([2]), '011': set([3]), '001': set([4]), '111': set([
]), '110': set([]), '100': set([0]), '101': set([1])}
```

data_tools.iterables.**unzip_dicts**(**dicts*)

Unzips the keys and values for any number of dictionaries passed as arguments (see below for examples).

- **Arguments:**
  - **dicts* [dict]: Dictionaries from which key/value pairs are to be unzipped.
- **Returns:**
  - [list]: Two-element list contianing all keys and all values respectively from the dictionaries in **dicts*.
- **Example:**

```
>>> a = dict([('x_a', 2), ('y_a', 3)])
>>> b = dict([('x_b', 1), ('y_b', -1)])
>>> unzip_dicts(a, b)
[('y_a', 'x_a', 'x_b', 'y_b'), (3, 2, 1, -1)]
```

4.4 data_tools.models

Model classes module.

class data_tools.models.**DoseResponse**(*d_data, r_data, x0=None, x_scale=None, bounds=([0, 0, -inf], [inf, inf, inf])*)

Wrapper class for `scipy.optimize.least_squares` to fit dose-response curves on a pre-defined Hill function with the following form:

$$R = \frac{mD^n}{k^n + D^n}$$

Where D is the dose, k, m and n are the parameters to be fitted.

- **Arguments:**

- *d_data* [numpy.ndarray]: Or any iterable (1D). Contains the training data corresponding to the dose.
- *r_data* [numpy.ndarray]: Or any iterable (1D). Contains the training data corresponding to the response.
- *x0* [list]: Optional, `None` by default. Or any iterable of three elements. Contains the initial guess for the parameters. Parameters are considered to be in alphabetical order. This is, first element corresponds to k, second is m and last is n. If `None` (default), the initial guess is inferred from *r_data*.
- *x_scale* [list]: Optional, `None` by default. Or any iterable of three elements. Scale of each parameter. May improve the fitting if the scaled parameters have similar effect on the cost function. If `None` (default), the scale is inferred from *x0*.
- *bounds* [tuple]: Optional `([0, 0, -inf], [inf, inf, inf])` by default. Two-element tuple containing the lower and upper boundaries for the parameters (elements of the tuple are iterables of three elements each).

- **Attributes:**
  - *x0* [list]: Contains the initial guess for the parameters. Parameters are considered to be in alphabetical order. This is, first element corresponds to k, second is m and last is n.
  - *x_scale* [list]: Scale of each parameter.
  - *model* [scipy.optimize.OptimizeResult]: Contains the result of the optimized model. See SciPy's reference for more information.
  - *params* [numpy.ndarray]: Three-element array containing the fitted parameters k, m and n.

ec(*p=50*)

Computes the effective concentration for the specified percentage of maximal concentration (EC_p).

- **Arguments:**
  - *p* [int]: Optional, `50` by default (EC_{50}). Defines the percentage of the maximal from which the effective concentration is to be computed.
- **Returns**
  - [float]: Value of the EC_p computed according to the model parameters.

plot(*title=None*, *filename=None*, *figsize=None*, *legend=True*)

Plots the data points and the fitted function together.

- **Arguments:**
  - *title* [str]: Optional, `None` by default. Defines the plot title.
  - *filename* [str]: Optional, `None` by default. If passed, indicates the file name or path where to store the figure. Format must be specified (e.g.: .png, .pdf, etc)
  - *figsize* [tuple]: Optional, `None` by default (default matplotlib size). Any iterable containing two values denoting the figure size (in inches) as [width, height].
  - *legend* [bool]: Optional, `True` by default. Indicates whether to show the plot legend or not.
- **Returns:**
  - [matplotlib.figure.Figure]: Figure object showing the data points and the fitted model function.

class data_tools.models.**Lasso**(*Cs=500*, *cv=10*, *sampler='skf'*, *solver='liblinear'*, ***kwargs*)

Wrapper class inheriting from `sklearn.linear_model.LogisticRegressionCV` with L1 regularization.

- **Arguments:**
  - *Cs* [int]: Optional, `500` by default. Integer or list of float values of regularization parameters to test. If an integer is passed, it will determine the number of values taken from a logarithmic scale between `1e-4` and `1e4`. Note that the value of the parameter is defined as the inverse of the regularization strength.
  - *cv* [int]: Optional, `10` by default. Denotes the number of cross validation (CV) folds.
  - *sampler* [str]: Optional, `'skf'` by default. Determines which sampling method is used to generate the test and training sets for CV. Methods available are K-Fold (`'kf'`), Shuffle Split (`'ss'`) and their stratified variants (`'skf'` and `'sss'` respectively).
  - *solver* [str]: Optional, `'liblinear'` by default. Determines which solver algorithm to use. Note that L1 regularization can only be handled by `'liblinear'` and `'saga'`. Additionally if the classification is multinomial, only the latter option is available.
  - ***kwargs*: Optional. Any other keyword argument accepted by the `sklearn.linear_model.LogisticRegressionCV` class.

  Other keyword arguments and functions available from the parent class `LogisticRegressionCV` can be fount in Scikit-Learn's reference.

fit_data(*x*, *y*, *silent=False*)

Fits the data to the logistic model.

- **Arguments:**
  - *x* [pandas.DataFrame]: Contains the values/measurements [float] of the features (columns) for each sample/replicate (rows).
  - *y* [pandas.Series]: List or any iterable containing the observed class of each sample (must have the same order as in *x*).
  - *silent* [bool]: Optional, `False` by default. Determines whether messages are printed or not.

plot_coef(*filename=None*, *figsize=None*)

Plots the non-zero coefficients for the fitted predictor features.

- **Arguments:**
  - *filename* [str]: Optional, `None` by default. If passed, indicates the file name or path where to store the figure. Format must be specified (e.g.: .png, .pdf, etc)
  - *figsize* [tuple]: Optional, `None` by default (default matplotlib size). Any iterable containing two values denoting the figure size (in inches) as [width, height].
- **Returns:**
  - [matplotlib.figure.Figure]: Figure object containing the bar plot of the non-zero coefficients.

plot_score(*filename=None*, *figsize=None*)

Plots the mean score across all folds obtained during CV. The optimum C parameter chosen and its score are highlighted.

- **Arguments:**
  - *filename* [str]: Optional, `None` by default. If passed, indicates the file name or path where to store the figure. Format must be specified (e.g.: .png, .pdf, etc)

- *figsize* [tuple]: Optional, `None` by default (default matplotlib size). Any iterable containing two values denoting the figure size (in inches) as [width, height].

- **Returns:**

  - [matplotlib.figure.Figure]: Figure object containing the score plot.

4.5 data_tools.plots

Plotting functions module.

`data_tools.plots.`**`cmap_bkgr = <matplotlib.colors.LinearSegmentedColormap object>`**
Custom colormap, gradient from black (lowest) to lime green (highest).

`data_tools.plots.`**`cmap_bkrd = <matplotlib.colors.LinearSegmentedColormap object>`**
Custom colormap, gradient from black (lowest) to red (highest).

`data_tools.plots.`**`cmap_rdbkgr = <matplotlib.colors.LinearSegmentedColormap object>`**
Custom colormap, gradient from red (lowest) to black (middle) to lime green (highest).

`data_tools.plots.`**`density`**(*df*, *cvf=0.25*, *sample_col=False*, *title=None*, *filename=None*, *figsize=None*)
Generates a density plot of the values on a data frame (row-wise).

- **Arguments:**

  - *df* [pandas.DataFrame]: Contains the values to generate the plot. Each row is considered as an individual sample while each column contains a measured value unless otherwise stated by keyword argument *sample_col*.

  - *cvf* [float]: Optional, `0.25` by default. Co-variance factor used in the gaussian kernel estimation. A higher value increases the smoothness.

  - *sample_col* [bool]: Optional, `False` by default. Specifies whether the samples are column-wise or not.

  - *title* [str]: Optional, `None` by default. Defines the plot title.

  - *filename* [str]: Optional, `None` by default. If passed, indicates the file name or path where to store the figure. Format must be specified (e.g.: .png, .pdf, etc)

  - *figsize* [tuple]: Optional, `None` by default (default matplotlib size). Any iterable containing two values denoting the figure size (in inches) as [width, height].

- **Returns:**

  - [*matplotlib.figure.Figure*]: the figure object containing the density plot.

`data_tools.plots.`**`piano_consensus`**(*df*, *nchar=40*, *boxes=True*, *title=None*, *filename=None*, *figsize=None*)
Generates a GSEA consensus score plot like R package `piano`'s `consensusScores` function, but prettier. The main input is assumed to be a `pandas.DataFrame` whose data is the same as the `rankMat` from the result of `consensusScores`.

- **Arguments:**

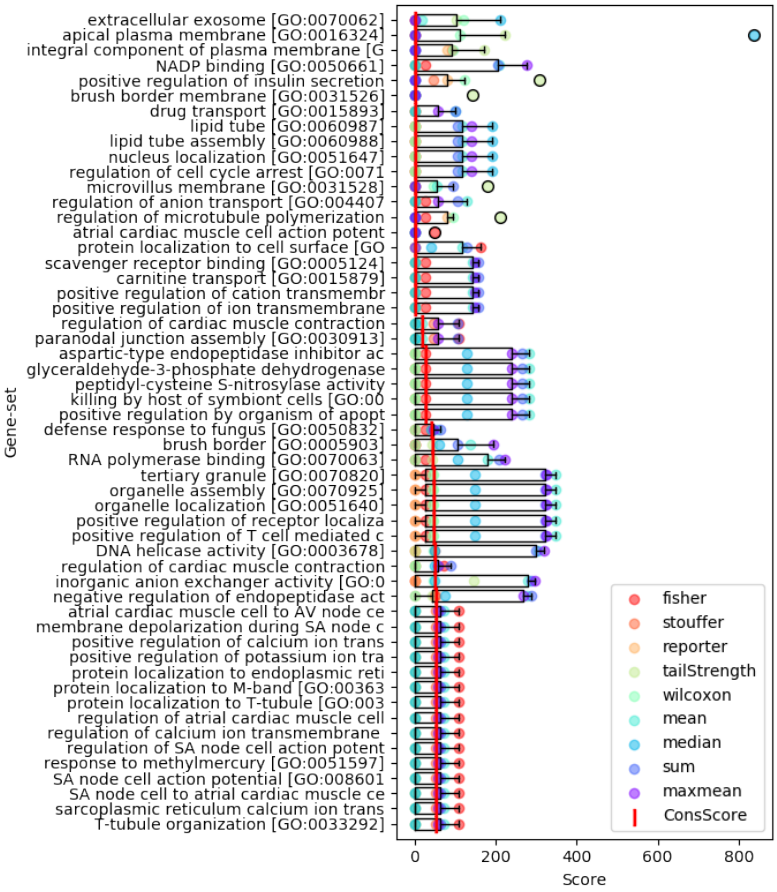
  - *df* [pandas.DataFrame]: Values contained correspond to the scores of the gene-sets (consensus and each individual methods). Index must contain the gene-set labels. Columns are assumed to be `ConsRank` (ignored), `ConsScore` followed by the individual methods (e.g.: `mean`, `median`, `sum`, etc).

- *nchar* [int]: Optional, `40` by default. Number of string characters of the gene-set labels of the plot.
- *boxes* [bool]: Optional, `True` by default. Determines whether to show the boxplots of the gene-sets or not.
- *title* [str]: Optional, `None` by default. Defines the plot title.
- *filename* [str]: Optional, `None` by default. If passed, indicates the file name or path where to store the figure. Format must be specified (e.g.: .png, .pdf, etc)
- *figsize* [tuple]: Optional, `None` by default (default matplotlib size). Any iterable containing two values denoting the figure size (in inches) as [width, height].

- **Returns:**
  - [*matplotlib.figure.Figure*]: the figure object containing a combination of box and scatter plots of the gene-set scores.

- **Example:**

```
>>> piano_consensus(df, figsize=[7, 8])
```



`data_tools.plots.`**venn**(*N, labels=['A', 'B', 'C', 'D', 'E'], c=['C0', 'C1', 'C2', 'C3', 'C4'], pct=False, sizes=False, title=None, filename=None, figsize=None*)

Plots a Venn diagram from a list of sets *N*. Number of sets must be between 2 and 5 (inclusive).

- **Arguments:**

- *N* [list]: Or any iterable type containing [set] objects.
- *labels* [list]: Optional, `['A', 'B', 'C', 'D', 'E']` by default. Labels for the sets following the same order as provided in *N*.
- *c* [list]: Optional, `['C0', 'C1' 'C2', 'C3', 'C4']` by default (matplotlib default colors). Any iterable containing color arguments tolerated by matplotlib (e.g.: `['r', 'b']` for red and blue). Must contain at least the same number of elements as *N* (if more are provided, they will be ignored).
- *pct* [bool]: Optional, `False` by default. Indicates whether to show percentages instead of absolute counts.
- *sizes* [bool]: Optional, `False` by default. Whether to include the size of the sets in the legend or not.
- *title* [str]: Optional, `None` by default. Defines the plot title.
- *filename* [str]: Optional, `None` by default. If passed, indicates the file name or path where to store the figure. Format must be specified (e.g.: .png, .pdf, etc)
- *figsize* [tuple]: Optional, `None` by default (default matplotlib size). Any iterable containing two values denoting the figure size (in inches) as [width, height].

- **Returns:**
  - [*matplotlib.figure.Figure*]: the figure object containing a combination of box and scatter plots of the gene-set scores.

- **Example:**

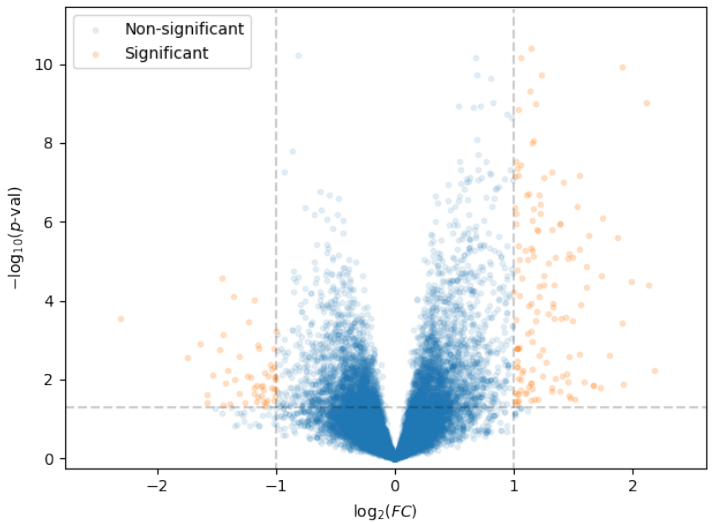
```
>>> N = [{0, 1}, {2, 3}, {1, 3, 4}] # Sets A, B, C
>>> venn(N)
```

`data_tools.plots.`**`volcano`**(*logfc*, *logpval*, *thr_pval=0.05*, *thr_fc=2.0*, *c=('C0', 'C1')*, *legend=True*, *title=None*, *filename=None*, *figsize=None*)

Generates a volcano plot from the differential expression data provided.

- **Arguments:**
  - *logfc* [list]: Or any iterable type. Contains the log (usually base 2) fold-change values. Must have the same length as *logpval*.
  - *logpval* [list]: Or any iterable type. Contains the -log p-values (usually base 10). Must have the same length as *logfc*.
  - *thr_pval* [float]: Optional, `0.05` by default. Specifies the p-value (non log-transformed) threshold to consider a measurement as significantly differentially expressed.
  - *thr_fc* [float]: Optional, `2.` by default. Specifies the FC (non log-transformed) threshold to consider a measurement as significantly differentially expressed.
  - *c* [tuple]: Optional, `('C0', 'C1')` by default (matplotlib default colors). Any iterable containing two color arguments tolerated by matplotlib (e.g.: `['r', 'b']` for red and blue). First one is used for non-significant points, second for the significant ones.
  - *legend* [bool]: Optional, `True` by default. Indicates whether to show the plot legend or not.
  - *title* [str]: Optional, `None` by default. Defines the plot title.
  - *filename* [str]: Optional, `None` by default. If passed, indicates the file name or path where to store the figure. Format must be specified (e.g.: .png, .pdf, etc)
  - *figsize* [tuple]: Optional, `None` by default (default matplotlib size). Any iterable containing two values denoting the figure size (in inches) as [width, height].
- **Returns:**
  - [matplotlib.figure.Figure]: Figure object containing the volcano plot.
- **Example:**

```
>>> volcano(my_log_fc, my_log_pval)
```



4.6 data_tools.spatial

Spatial matrix module.

data_tools.spatial.**get_boundaries**(*x*, *counts=False*)

Given an array, returns either the mask where the boundary edges are or their counts if specified.

- **Arguments:**
  - x [numpy.ndarray]: The array where boundaries are to be identified or counted. Data type of its elements is totally irrelevant.
  - count [bool]: Optional, `False` by default. Whether to return the number of boundary edges or just their mask.
- **Returns:**
  - [numpy.ndarray]: Same shape as *x*. If `counts=False`, contains `True` on any cell that is on the boundary, `False` otherwise. If `counts=True`, will return a similar array but instead of [bool], there will be [int] denoting the number of boundary edges of the cells.
- **Examples:**

```
>>> x = numpy.ones((3, 3, 3))
>>> get_boundaries(x)
array([[[ True,  True,  True],
        [ True,  True,  True],
        [ True,  True,  True]],
       [[ True,  True,  True],
        [ True, False,  True],
        [ True,  True,  True]],
       [[ True,  True,  True],
        [ True,  True,  True],
        [ True,  True,  True]]])
>>> get_boundaries(x, counts=True)
array([[[3, 2, 3],
        [2, 1, 2],
        [3, 2, 3]],
       [[2, 1, 2],
        [1, 0, 1],
        [2, 1, 2]],
       [[3, 2, 3],
        [2, 1, 2],
        [3, 2, 3]]])
```

data_tools.spatial.**neighbour_count**(*x*)

Given an array (up to three dimensions), returns another array with the same shape containing the counts of cells' neighbours whose value is zero.

- **Arguments:**
  - *x* [numpy.ndarray]: The array where to count the neighbours (zero-valued cells). Note that the cells can have any value or data type. As long as they be converted to [bool], the function will count all `False` around all `True` cells.
- **Returns:**
  - [numpy.ndarray]: Array with same shape as *x* containing the neighbour count.
- **Examples:**

```
>>> x = numpy.random.randint(2, size=(5, 5))
>>> x
array([[0, 0, 1, 1, 0],
       [0, 0, 0, 1, 1],
```

(continues on next page)

(continued from previous page)

```
       [1, 0, 1, 1, 1],
       [0, 0, 0, 0, 0],
       [1, 0, 0, 1, 1]])
>>> neighbour_count(x)
array([[0, 0, 2, 1, 0],
       [0, 0, 0, 1, 1],
       [3, 0, 3, 1, 1],
       [0, 0, 0, 0, 0],
       [2, 0, 0, 2, 1]])
```

4.7 data_tools.strings

String operations module.

data_tools.strings.**is_numeric**(*s*)

Determines if a string can be considered a numeric value. NaN is also considered, since it is float type.

- **Arguments:**
  - *s* [str]: String to be evaluated.
- **Returns:**
  - [bool]: `True`/`False` depending if the condition is satisfied.
- **Examples:**

```
>>> is_numeric('4')
True
>>> is_numeric('-3.2')
True
>>> is_numeric('number')
False
>>> is_numeric('NaN')
True
```

data_tools.strings.**join_str_lists**(*a*, *b*, *sep=''*)

Joins element-wise two lists (or any 1D iterable) of strings with a given separator (if provided). Length of the input lists must be equal.

- **Arguments:**
  - *a* [list]: Contains the first elements [str] of the joint strings.
  - *b* [list]: Contains the second elements [str] of the joint strings.
  - *sep* [str]: Optional `''` (non separated) by default. Determines the separator between the joint strings.
- **Returns:**
  - [list]: List of the joint strings.
- **Example:**

```
>>> a = ['a', 'b']
>>> b = ['1', '2']
>>> join_str_lists(a, b, sep='_')
['a_1', 'b_2']
```

PYTHON MODULE INDEX

d

INDEX