dadapy

Release 0.1

Aldo Glielmo, Matteo Carli, Diego Doimo, Iuri Macocco, Maria D'Er

Apr 05, 2022

TABLE OF CONTENTS

DADApy is a python package for distance-based analysis of data-manifolds.

The code can be found on GitHub at https://github.com/sissa-data-science/DADApy.

CHAPTER

ONE

INSTALLATION

1.1 Installing the requirements

This package is based on numpy, scipy, scikit-learn, and Cython. These dependencies can be easily installed using anaconda or pip. Alternatively, they will be automatically installed with the package.

1.2 Installing the package

Install the latest version from the Github repository via

```
pip install git+https://github.com/sissa-data-science/DADApy
```

CHAPTER TWO

IMPLEMENTED ALGORITHMS

The algorithms currently implemented in the package can be divided in four broad groups.

2.1 Intrinsic dimension estimation

These algorithms estimate the *intrinsic dimension* of the data manifold i.e., the minimum number of coordinates needed to describe the manifold without a significant loss of information. The algorithms currently implemented are:

- Two NN (“Two nearest neighbour estimator”)
- Gride (“Generalized ratios id estimator”)

2.2 Density estimation

These algorithms estimate the density profile from which the dataset was sampled. The algorithms currently implemented are:

- k-NN (“k-nearest neighbours estimator”)
- PAk (“Point adaptive k-NN estomator”)
- k*-NN (“k-star nearest neighbours estimator”)

2.3 Density based clustering

These algorithms find the statistically significant peaks of the density profile and use this information to divide the dataset into clusters of data. The algorithms currently implemented are:

- DP (“Density peaks clustering”)
- ADP(“Advanced density peaks clustering”)

2.4 Metric space comparison

These algorithms estimate and quantify whether two spaces endowed with a distance measure are equivalent or not, and whether one space is more informative than the other. The algorithms currently implemented are:

- Neighbourhood overlap
- Information imbalance

CHAPTER

THREE

TYPICAL USAGE OF THE PACKAGE

A typical usage of Dadapy involves the initialisation of a Data object either with a set of coordinates or with a set of distances between points. After the initialisation a series of computations are performed by calling the class method relative to specific algorithm wanted. The results of the computations are typically available as attributes of the object.

```python
import numpy as np
import matplotlib.pyplot as plt
from duly.data import Data

# a simple 3D gaussian dataset
X = np.random.normal(0, 1, (1000, 3))

# initialise the "Data" class with a
# set of coordinates
data = Data(X)

# compute distances up to the 100th
# nearest neighbour
data.compute_distances(maxk = 100)

# compute the intrinsic dimension
# using the 2NN estimator
data.compute_id_2NN()

# check the value of the intrinsic
# dimension found
print(data.selected_id)

# compute the density of all points
# using a simple kNN estimator
data.compute_density_kNN(k = 15)

# as an alternative, compute the density
# using a more sophisticated estimator
data.compute_density_PAk()

plt.hist(data.log_den)

# find the statistically significant peaks
# of the density profile computed previously
data.compute_clustering(Z = 1.5)
```

(continues on next page)

(continued from previous page)

```
print(data.N_clusters)
```

CHAPTER

FOUR

MODULES (API REFERENCE)

The package contains the following modules: id_estimation, density_estimation, clustering, metric_comparisons, data, data_sets.

4.1 The id_estimation module

The *id_estimation* module contains the *IdEstimation* class. The different algorithms of intrinsic dimension estimation are implemented as methods of this class.

`class dadapy.id_estimation.`**`IdEstimation`**(*coordinates=None*, *distances=None*, *maxk=None*, *verbose=False*, *njobs=2*)

Estimates the intrinsic dimension of a dataset choosing among various routines.

Inherits from class Base.

`intrinsic_dim`

computed intrinsic dimension of the data manifold.

Type float

`intrinsic_dim_err`

error on the id estimation

Type float

`intrinsic_dim_scale`

distance scale of id estimation

Type float

`compute_id_2NN`(*algorithm='base'*, *fraction=0.9*, *decimation=1*, *set_attr=True*)

Compute intrinsic dimension using the 2NN algorithm

Parameters

- **`algorithm`** (*`str`*) – 'base' to perform the linear fit, 'ml' to perform maximum likelihood
- **`fraction`** (*`float`*) – fraction of mus that will be considered for the estimate (discard highest mus)
- **`decimation`** (*`float`*) – fraction of randomly sampled points used to compute the id
- **`set_attr`** (*`bool`*) – whether to change the class attributes as a result of the computation

Returns

- **id** (*float*) – the estimated intrinsic dimension

- **id_err** (*float*) – the standard error on the id estimation
- **rs** (*float*) – the average nearest neighbor distance (rs)

References

E. Facco, M. d'Errico, A. Rodriguez, A. Laio, Estimating the intrinsic dimension of datasets by a minimal neighborhood information, Scientific reports 7 (1) (2017) 1–8

`compute_id_binomial_k`(*k=None*, *ratio=None*, *subset=None*, *method='bayes'*, *plot=False*)

Calculate id using the binomial estimator by fixing the number of neighbours

As in the case in which one fixes rk, also in this version of the estimation one removes the central point from n and k. Furthermore, one has to remove also the k-th NN, as it plays the role of the distance at which rk is taken. So if k=5 it means the 5th NN from the central point will be considered, taking into account 6 points though (the central one too). This means that in principle k_eff = 6, to which I'm supposed to subtract 2. For this reason in the computation of the MLE we have directly k-1, which explicitly would be k_eff-2

Parameters

- **k** (*int*) – order of neighbour that set the external shell
- **ratio** (*float*) – ratio between internal and external shell
- **subset** (*int*) – choose a random subset of the dataset to make the Id estimate
- **method** (*str, default='bayes'*) – choose method between 'bayes' and 'mle'. The bayesian estimate gives the mean value and std of d, while mle only the max of the likelihood
- **plot** (*bool, default=False*) – if True plots the posterior and initialise self.posterior_domain and self.posterior

`compute_id_binomial_rk`(*rk*, *ratio=None*, *subset=None*, *method='bayes'*, *plot=False*)

Calculate the id using the binomial estimator by fixing the same eternal radius for all the points

In the estimation of the id one has to remove the central point from the counting of n and k as it is not effectively part of the poisson process generating its neighbourhood

Parameters

- **rk** (*float*) – radius of the external shell
- **ratio** (*float*) – ratio between internal and external shell
- **subset** (*int, np.ndarray(int)*) – choose a random subset of the dataset to make the id estimate
- **method** (*str, default='bayes'*) – choose method between 'bayes' and 'mle'. The bayesian estimate gives the mean value and std of d, while mle only the max of the likelihood
- **plot** (*bool, default=False*) – if True plots the posterior and initialise self.posterior_domain and self.posterior

`compute_id_gammaprior`(*alpha=2*, *beta=5*)

Compute the intrinsic dimension using a bayesian formulation of 2nn

Parameters

- **alpha** (*float*) – parameter of the prior

- **beta** (*float*) – other prior parameter

`fix_k`(*k_eff=None*, *ratio=None*)

Computes rk, rn, n for each point of the dataset given a value of k

This routine computes the external radius rk, internal radius rn and internal points n given a value k, the number of NN to consider.

Parameters

- **k_eff** (*int, default=self.maxk*) – the number of NN to take into account
- **ratio** (*float*) – ratio among rn and rk

`fix_rk`(*rk*, *ratio=None*)

Computes the k points within the given rk and n points within given rn.

For each point, computes the number self.k of points within a sphere of radius rk and the number self.n within an inner sphere of radius rn=rk*ratio. It also provides a mask to take into account those points for which the statistics might be wrong, i.e. k == self.maxk, meaning that all available points were selected. If self.maxk is equal to the number of points of the dataset no mask will be applied

Parameters

- **rk** (*float*) – external shell radius
- **ratio** (*float,optional*) – ratio between internal and external shell radii of the shells

`return_id_scaling_2NN`(*N_min=10*, *algorithm='base'*, *fraction=0.9*)

Compute the id at different scales using the 2NN algorithm.

Parameters

- **N_min** (*int*) – minimum number of points considered when decimating the dataset
- **algorithm** (*str*) – 'base' to perform the linear fit, 'ml' to perform maximum likelihood
- **fraction** (*float*) – fraction of mus that will be considered for the estimate (discard highest mus)

Returns

- **ids_scaling** (*np.ndarray(float)*) – array of intrinsic dimensions
- **ids_scaling_err** (*np.ndarray(float)*) – array of error estimates
- **rs_scaling** (*np.ndarray(float)*) – array of average distances of the neighbors involved in the estimates

`return_id_scaling_gride`(*range_max=64*, *d0=0.001*, *d1=1000*, *eps=1e-07*)

Compute the id at different scales using the Gride algorithm.

Parameters

- **range_max** (*int*) – maximum nearest neighbor rank considered for the id computations
- **d0** (*float*) – minimum intrinsic dimension considered in the search
- **d1** (*float*) – maximum intrinsic dimension considered in the search
- **eps** (*float*) – precision of the approximate id calculation

Returns

- **ids_scaling** (*np.ndarray(float)*) – array of intrinsic dimensions
- **ids_scaling_err** (*np.ndarray(float)*) – array of error estimates

- **rs_scaling** (*np.ndarray(float)*) – array of average distances of the neighbors involved in the estimates

References

F. Denti, D. Doimo, A. Laio, A. Mira, Distributional results for model-based intrinsic dimension estimators, arXiv preprint arXiv:2104.13832 (2021).

4.2 The density_estimation module

The *density_estimation* module contains the *DensityEstimation* class. The different algorithms of density estimation are implemented as methods of this class.

`class dadapy.density_estimation.DensityEstimation`(*coordinates=None*, *distances=None*, *maxk=None*, *verbose=False*, *njobs=2*)

Computes the log-density and its error at each point and other properties.

Inherits from class IdEstimation. Can estimate the optimal number k* of neighbors for each points. Can compute the log-density and its error at each point choosing among various kNN-based methods. Can return an estimate of the gradient of the log-density at each point and an estimate of the error on each component. Can return an estimate of the linear deviation from constant density at each point and an estimate of the error on each component. TODO: - compute dc when compute_kstar or when set_kstar and nowhere else - implement self.kstar and self.fixedk of a bool attribute self.iskfixed which is True when kstar is an array of identical integers

`kstar`

array containing the chosen number k* in the neighbourhood of each of the N points

Type np.array(float)

`nspar`

total number of edges in the directed graph defined by kstar (sum over all points of kstar minus N)

Type int

`nind_list`

size nspar x 2. Each row is a couple of indices of the connected graph stored in order of increasing point index and increasing neighbour length (E.g.: in the first row (0,j), j is the nearest neighbour of the first point. In the second row (0,l), l is the second-nearest neighbour of the first point. In the last row (N-1,m) m is the kstar-1-th neighbour of the last point.)

Type np.ndarray(int), optional

`nind_iptr`

size N+1. For each elemen i stores the 0-th index in nind_list at which the edges starting from point i start. The last entry is set to nind_list.shape[0].

Type np.array(int), optional

`common_neighs`

stored as a sparse symmetric matrix of size N x N. Entry (i,j) gives the common number of neighbours between points i and j. Such value is reliable only if j is in the neighbourhood of i or vice versa.

Type scipy.sparse.csr_matrix(float), optional `neigh_vector_diffs`

stores vector differences from each point to its k*-1 nearest neighbors. Accessed by the method return_vector_diffs(i,j) for each j in the neighbourhood of i

Type np.ndarray(float), optional

`neigh_dists`

stores distances from each point to its k*-1 nearest neighbors in the order defined by nind_list

Type np.array(float), optional

`dc`

array containing the distance of the k*th neighbor from each of the N points

Type np.array(float), optional

`log_den`

array containing the N log-densities

Type np.array(float), optional

`log_den_err`

array containing the N errors on the log_den

Type np.array(float), optional

`grads`

for each line i contains the gradient components estimated from from point i

Type np.ndarray(float), optional

`grads_var`

for each line i contains the estimated variance of the gradient components at point i

Type np.ndarray(float), optional

`check_grads_covmat`

it is flagged "True" when grads_var contains the variance-covariance matrices of the gradients.

Type bool, optional

`Fij_array`

stores for each couple in nind_list the estimates of deltaF_ij computed from point i as semisum of the gradients in i and minus the gradient in j.

Type list(np.array(float)), optional

`Fij_var_array`

stores for each couple in nind_list the estimates of the squared errors on the values in Fij_array.

Type np.array(float), optional

Bello sto stile di documentazione:

Parameters

- **X** (*{float, np.ndarray, or theano symbolic variable}*) – X coordinate. If you supply an array, x and y need to be the same shape, and the potential will be calculated at each (x,y pair)
- **y** (*{float, np.ndarray, or theano symbolic variable}*) – Y coordinate. If you supply an array, x and y need to be the same shape, and the potential will be calculated at each (x,y pair)

`compute_common_neighs()`

Compute the common number of neighbours between couple of points (i,j) such that j is in the neighbourhod of i. The numbers are stored in a scipy sparse csr_matrix format.

Args:

Returns:

`compute_deltaFs_grads_semisum`(*chi='auto'*)

Compute deviations deltaFij to standard kNN log-densities at point j as seen from point i using a linear expansion with as slope the semisum of the average gradient of the log-density over the neighbourhood of points i and j. The parameter chi is used in the estimation of the squared error of the deltaFij as 1/4*(E_i^2+E_j^2+2*E_i*E_j*chi), where E_i is the error on the estimate of grad_i*DeltaX_ij.

Parameters `chi` – the Pearson correlation coefficient between the estimates of the gradient in i and j. Can take a numerical value between 0 and 1. The option 'auto' takes a geometrical estimate of chi based on AAAAAAAAA

Returns:

`compute_density_PAk_gCorr`(*gauss_approx=True*, *alpha=1.0*, *log_den_PAk=None*, *log_den_PAk_err=None*, *comp_err=True*)

finds the maximum likelihood solution of PAk likelihood + gCorr likelihood with deltaFijs computed using the gradients

`compute_density_kNN`(*k=10*)

Compute the density of of each point using a simple kNN estimator

Parameters `k` – number of neighbours used to compute the density

Returns:

`compute_density_kpeaks`(*Dthr=23.92812698*)

Computes the density of each point as proportional to the optimal k value found for that point.

This method is mostly useful for the kpeaks clustering algorithm.

Parameters

- `Dthr` – Likelihood ratio parameter used to compute optimal k, the value of Dthr=23.92 corresponds
- `1e-6.` (*to a p-value of*) –

`compute_density_kstarNN`(*Dthr=23.92812698*)

Computes the density of each point using a simple kNN estimator with an optimal choice of k. :param Dthr: Likelihood ratio parameter used to compute optimal k, the value of Dthr=23.92 corresponds :param to a p-value of 1e-6.:

Returns:

`compute_density_kstarNN_gCorr`(*alpha=1.0*, *gauss_approx=False*, *Fij_type='grad'*)

finds the minimum of the

`compute_grads`(*comp_covmat=False*)

Compute the gradient of the log density each point using k* nearest neighbors. The gradient is estimated via a linear expansion of the density propagated to the log-density.

Args:

Returns:

MODIFICARE QUI E ANCHE NEGLI ATTRIBUTI

`compute_kstar`(*Dthr=23.92812698*)

Computes an optimal choice of k for each point. Cython optimized version. :param Dthr: Likelihood ratio parameter used to compute optimal k, the value of Dthr=23.92 corresponds :param to a p-value of 1e-6.:

Returns:

`compute_neigh_dists`()

Computes the (directed) neighbour distances graph using kstar[i] neighbours for each point i. Distances are stored in np.array form according to the order of nind_list.

`compute_neigh_indices`()

Computes the indices of all the couples [i,j] such that j is a neighbour of i up to the k*-th nearest (excluded). The couples of indices are stored in a numpy ndarray of rank 2 and secondary dimension = 2. The index of the corresponding AAAAAAAAAAAAA make indpointer which is a np.array of length N which indicates for each i the starting index of the corresponding [i,.] subarray.

`compute_neigh_vector_diffs`()

Compute the vector differences from each point to its k* nearest neighbors. The resulting vectors are stored in a numpy ndarray of rank 2 and secondary dimension = dims. The index of scipy sparse csr_matrix format.

AAA better implement periodicity to take both a scalar and a vector

`return_density_Gaussian_kde`(*Y_sample=None*, *smoothing_parameter=None*, *adaptive=False*, *return_only_gradient=False*)

Returns the logdensity of of each point in Y_sample using a Gaussian kernel density estimator based on the coordinates self.X.

TODO: improve normalisation of gaussians on periodic range (currently normalisation of Gaussian in range (-inf,inf) instead of range [-period/2,period/2])

Parameters

- **`Y_sample`** (*`np.ndarray(float)`*) – the points at which the Gaussian kernel density should be evaluated. The default is self.X
- **`smoothing_parameter`** (*`float or np.ndarray(float)`*) – default is given by Scott's Rule of Thumb (self.N**(-1./(self.dims+4)))
- **`adaptive`** (*`bool`*) – if set to 'True', bandwidth is set to half the distance of the k*-th neighbour of a point, k* beingselected adaptively
- **`return_only_gradient`** (*`bool`*) – if set to 'True', only returns the gradient of the logdensity

Returns:

`return_entropy`()

Compute a very rough estimate of the entropy of the data distribution as the average negative log probability.

Returns **H** (*float*) – the estimate entropy of the distribution

`return_interpolated_density_PAk`(*X_new*, *Dthr=23.92812698*)

Parameters

- **`X_new`** (*`np.ndarray(float)`*) – The points onto which the density should be computed
- **`Dthr`** – Likelihood ratio parameter used to compute optimal k

Returns

- **log_den** (*np.ndarray(float)*) – log density of dataset evaluated on X_new

- **log_den_err** (*np.ndarray(float)*) – error on log density estimates

`return_interpolated_density_kNN`(*X_new*, *k*)

Return the kNN density of the primary dataset, evaluated on a new set of points "X_new".

Parameters

- `X_new` (`np.ndarray(float)`) – The points onto which the density should be computed
- `k` (`int`) – the number of neighbours considered for the kNN estimator

Returns

- **log_den** (*np.ndarray(float)*) – log density of dataset evaluated on X_new
- **log_den_err** (*np.ndarray(float)*) – error on log density estimates

`return_interpolated_density_kstarNN`(*X_new*, *Dthr=23.92812698*)

Parameters

- `X_new` (`np.ndarray(float)`) – The points onto which the density should be computed
- `Dthr` – Likelihood ratio parameter used to compute optimal k

Returns

- **log_den** (*np.ndarray(float)*) – log density of dataset evaluated on X_new
- **log_den_err** (*np.ndarray(float)*) – error on log density estimates

`return_sparse_distance_graph`()

Returns the (directed) neighbour distances graph using kstar[i] neighbours for each point i in N x N sparse csr_matrix form.

`set_kstar`(*k=0*)

Set all elements of kstar to a fixed value k. Reset all other class attributes (all depending on kstar).

Parameters `k` – number of neighbours used to compute the density

Returns:

4.3 The clustering module

The *clustering* module contains the *Clustering* class. Density-based clustering algorithms are implemented as methods of this class.

`class clustering.Clustering`(*coordinates=None*, *distances=None*, *maxk=None*, *verbose=False*, *njobs=2*)

This class contains various density-based clustering algorithms.

Inherits from the DensityEstimation class.

`N_clusters`

Number of clusters found

Type int

`cluster_assignment`

A list of length N containing the cluster assignment of each point as an integer from 0 to N_clusters-1.

Type list(int)

`cluster_centers`

Indices of the centroids of each cluster (density peak)

Type list(int)

`cluster_indices`

A list of lists. Each sublist contains the indices belonging to the corresponding cluster.

Type list(list(int))

`log_den_bord`

A matrix of dimensions N_clusters x N_clusters containing the estimated log density of the saddle point between each couple of peaks.

Type np.ndarray(float)

`log_den_bord_err`

A matrix of dimensions N_clusters x N_clusters containing the estimated error on the log density of the saddle point between each couple of peaks.

Type np.ndarray(float)

`bord_indices`

A matrix of dimensions N_clusters x N_clusters containing the indices of the saddle point between each couple of peaks.

Type np.ndarray(float)

`compute_cluster_DP`(*dens_cut=0.0*, *delta_cut=0.0*, *halo=False*)

Compute clustering using the Density Peak algorithm

Parameters

- **dens_cut** (*float*) –
- **delta_cut** (*float*) –
- **halo** (*bool*) – use or not halo points

References

A. Rodriguez, A. Laio, Clustering by fast search and find of density peaks, Science 344 (6191) (2014) 1492–1496.

`compute_clustering`(*Z=1.65*, *halo=False*, *density_algorithm='PAK'*, *k=None*, *Dthr=23.92812698*)

Compute clustering according to the algorithm DPA

The only free parameter is the merging factor Z, which controls how the different density peaks are merged together. The higher the Z, the more aggressive the merging, the smaller the number of clusters. The calculation is optimized though cython

Parameters

- **Z** (*float*) – merging parameter
- **halo** (*bool*) – compute (or not) the halo points
- **density_algorithm** (*str*) – method to compute the local density. Use 'PAK' for adaptive neighbourhood, 'kNN' for fixed neighbourhood
- **k** (*int*) – number of neighbours when using kNN algorithm

- **Dthr** (*float*) – Likelihood ratio parameter used to compute optimal k when using PAK algorithm. The value of Dthr=23.92 corresponds to a p-value of 1e-6.

References

M. **d'Errico, E. Facco, A. Laio, A. Rodriguez, Automatic topography of high-dimensional data sets by** non-parametric density peak clustering, Information Sciences 560 (2021) 476–492

compute_clustering_pure_python(*Z=1.65*, *halo=False*, *density_algorithm='PAK'*, *k=None*)

Same as compute_clustering, but without the cython optimization

4.4 The metric_comparisons module

The *metric_comparisons* module contains the *MetricComparisons* class. Algorithms for comparing different spaces are implemented as methods of this class.

class dadapy.metric_comparisons.**MetricComparisons**(*coordinates=None*, *distances=None*, *maxk=None*, *period=None*, *verbose=False*, *njobs=2*)

This class contains several methods to compare metric spaces obtained using subsets of the data features. Using these methods one can assess whether two spaces are equivalent, completely independent, or whether one space is more informative than the other.

Attributes:

greedy_feature_selection_full(*n_coords*, *k=1*, *n_best=10*, *dtype='mean'*, *symm=True*)

Greedy selection of the set of coordinates which is most informative about full distance measure.

Parameters

- **n_coords** – number of coodinates after which the algorithm is stopped
- **k** (*int*) – number of neighbours considered in the computation of the imbalances
- **n_best** (*int*) – the n_best tuples are chosen in each iteration to combinatorically add one variable and calculate the imbalance until n_coords is reached
- **dtype** (*str*) – specific way to characterise the deviation from a delta distribution
- **symm** (*bool*) – whether to use the symmetrised information imbalance

Returns

- **best_tuples** (*list(list(int))*) – best coordinates selected at each iteration
- **best_imbalances** (*np.ndarray(float,float)*) – imbalances (full–>coords, coords–>full) computed at each iteration, belonging to the best tuple

greedy_feature_selection_target(*target_ranks*, *n_coords*, *k*, *n_best*, *dtype='mean'*, *symm=True*)

Greedy selection of the set of coordinates which is most informative about a target distance.

Parameters

- **target_ranks** (*np.ndarray(int)*) – an array containing the ranks in the target space
- **n_coords** – number of coodinates after which the algorithm is stopped
- **k** (*int*) – number of neighbours considered in the computation of the imbalances
- **n_best** (*int*) – the n_best tuples are chosen in each iteration to combinatorically add one variable and calculate the imbalance until n_coords is reached

- **dtype** (*str*) – specific way to characterise the deviation from a delta distribution
- **symm** (*bool*) – whether to use the symmetrised information imbalance

Returns

- **best_tuples** (*list(list(int))*) – best coordinates selected at each iteration
- **best_imbalances** (*np.ndarray(float,float)*) – imbalances (full–>coords, coords–>full) computed at each iteration, belonging to the best tuple
- **all_imbalances** (*list(list(list(int)))*) – all imbalances (full–>coords, coords–>full), computed at each iteration, belonging all greedy tuples

`return_inf_imb_full_all_coords`(*k=1*, *dtype='mean'*)

Compute the information imbalances between the 'full' space and each one of its D features

Parameters

- **k** (*int*) – number of neighbours considered in the computation of the imbalances
- **dtype** (*str*) – specific way to characterise the deviation from a delta distribution

Returns

- **(np.array** (*float)*) – a 2xD matrix containing the information imbalances between
- **the original space and each of its D features.**

`return_inf_imb_full_all_dplets`(*d*, *k=1*, *dtype='mean'*)

Compute the information imbalances between the full X space and all possible combinations of d coordinates contained of X.

Parameters

- **d** (*int*) – target order considered (e.g., d = 2 will compute all couples of coordinates)
- **k** (*int*) – number of neighbours considered in the computation of the imbalances
- **dtype** (*str*) – specific way to characterise the deviation from a delta distribution

Returns

- **coord_list** – list of the set of coordinates for which imbalances are computed
- **imbalances** – the correspinding couples of information imbalances

`return_inf_imb_full_selected_coords`(*coord_list*, *k=1*, *dtype='mean'*)

Compute the information imbalances between the 'full' space and a selection of features.

Parameters

- **coord_list** (*list(list(int))*) – a list of the type [[1, 2], [8, 3, 5], …] where each
- **be** (*sub-list defines a set of coordinates for which the information imbalance should*) –
- **computed.** –
- **k** (*int*) – number of neighbours considered in the computation of the imbalances
- **dtype** (*str*) – specific way to characterise the deviation from a delta distribution

Returns

- **(np.array** (*float)*) – a 2xL matrix containing the information imbalances between
- **the original space and each one of the L subspaces defined in coord_list**

`return_inf_imb_matrix_of_coords`(*k=1*, *dtype='mean'*)

Compute the information imbalances between all pairs of D features of the data.

Parameters

- **k** (*int*) – number of neighbours considered in the computation of the imbalances
- **dtype** (*str*) – specific way to characterise the deviation from a delta distribution

Returns **n_mat** (*np.array(float)*) – a DxD matrix containing all the information imbalances

`return_inf_imb_target_all_coords`(*target_ranks*, *k=1*, *dtype='mean'*)

Compute the information imbalances between the 'target' space and a all single feature spaces in X.

Parameters

- **target_ranks** (*np.array(int)*) – an array containing the ranks in the target space
- **k** (*int*) – number of neighbours considered in the computation of the imbalances
- **dtype** (*str*) – specific way to characterise the deviation from a delta distribution

Returns

- **(np.array** (*float)*) – a 2xL matrix containing the information imbalances between
- **the target space and each one of the L subspaces defined in coord_list**

`return_inf_imb_target_all_dplets`(*target_ranks*, *d*, *k=1*, *dtype='mean'*)

Compute the information imbalances between a target distance and all possible combinations of d coordinates contained of X.

Parameters

- **target_ranks** (*np.array(int)*) – an array containing the ranks in the target space
- **d** (*int*) – target order considered (e.g., d = 2 will compute all couples of coordinates)
- **k** (*int*) – number of neighbours considered in the computation of the imbalances
- **dtype** (*str*) – specific way to characterise the deviation from a delta distribution

Returns

- **coord_list** – list of the set of coordinates for which imbalances are computed
- **imbalances** – the correspinding couples of information imbalances

`return_inf_imb_target_selected_coords`(*target_ranks*, *coord_list*, *k=1*, *dtype='mean'*)

Compute the information imbalances between the 'target' space and a selection of features.

Parameters

- **target_ranks** (*np.array(int)*) – an array containing the ranks in the target space
- **coord_list** (*list(list(int))*) – a list of the type [[1, 2], [8, 3, 5], . . .] where each
- **be** (*sub-list defines a set of coordinates for which the information imbalance should*) –
- **computed.** –
- **k** (*int*) – number of neighbours considered in the computation of the imbalances
- **dtype** (*str*) – specific way to characterise the deviation from a delta distribution

Returns

- **(np.array** (*float*)) – a 2xL matrix containing the information imbalances between
- **the target space and each one of the L subspaces defined in coord_list**

return_inf_imb_two_selected_coords(*coords1*, *coords2*, *k=1*, *dtype='mean'*)

Returns the imbalances between distances taken as the i and the j component of the coordinate matrix X.

Parameters

- **coords1** (*list(int)*) – components for the first distance
- **coords2** (*list(int)*) – components for the second distance
- **k** (*int*) – order of nearest neighbour considered for the calculation of the imbalance, default is 1
- **dtype** (*str*) – type of information imbalance computation, default is 'mean'

Returns **(float, float)** – the information imbalance from distance i to distance j and vice versa

4.5 The data module

The *data* module contains the *Data* class. Such a class inherits from all other classes defined in the package and as such it provides a convenient container of all the algorithms implemented in Dadapy.

class dadapy.data.**Data**(*coordinates=None*, *distances=None*, *maxk=None*, *verbose=False*, *njobs=2*, *working_memory=1024*)

This class is a container of all the method of DULY. It is initialised with a set of coordinates or a set of distances, and all methods can be called on the generated class instance.

Attributes:

4.6 The data_sets module

The *data_sets* module contains the *DataSets* class. This class can be initialised with multiple datasets (i.e., with a list of coordinate matrices or with a list of distance matrices) and the same algorithms available in the *Data* class can be run serially on all datasets.

4.7 The utils module

The *utils* module contains a variety of functions useful to the different classes of the package.

utils.**compute_NN_PBC**(*X*, *maxk*, *box_size=None*, *p=2*, *cutoff=inf*)

Compute the neighbours of each point taking into account periodic boundaries conditions and eventual cutoff

Parameters

- **X** (*np.ndarray*) – array of dimension N x D
- **maxk** (*int*) – number of neighbours to save
- **box_size** (*float, np.ndarray(float)*) – sizes of PBC walls. Single value is interpreted as cubic box.
- **p** (*int*) – Minkowski p-norm used

- **cutoff** (*float*) – set an upper bound to the distances. Over such threshold a np.inf will occur

Returns

- **dist** (*np.ndarray(float)*) – N x maxk array containing the distances from each point to the first maxk nn
- **ind** (*np.ndarray(int)*) – N x maxk array containing the indices of the neighbours of each point

`utils.compute_all_distances`(*X*, *n_jobs=2*, *metric='euclidean'*)

Compute the distances among all available points of the dataset X

Parameters

- **X** (*np.ndarray*) – array of dimension N x D
- **n_jobs** (*int*) – number of cores to use for the computation
- **metric** (*str*) – metric used to compute the distances

Returns **(np.ndarray** (*float*)) – N x N array containing the distances between the N points

`utils.compute_cross_nn_distances`(*X_new*, *X*, *maxk*, *metric='euclidean'*, *period=None*)

Compute distances, up to neighbour maxk, between points of X_new and points of X.

The element distances[i,j] represents the distance between point i in dataset X and its j-th neighbour in dataset X_new, whose index is dist_indices[i,j]

Parameters

- **X_new** (*np.array(float)*) – dataset from which distances are computed
- **X** (*np.array(float)*) – starting dataset of points, from which distances are computed
- **maxk** (*int*) – number of neighbours to save
- **metric** (*str*) – metric used to compute the distances
- **period** (*float, np.ndarray(float)*) – sizes of PBC walls. Single value is interpreted as cubic box.

Returns

- **distances** (*np.ndarray(int)*) – N x maxk matrix, indices of the neighbours of each point
- **dist_indices** (*np.ndarray(float)*) – N x maxk matrix, distances of the neighbours of each point

`utils.compute_nn_distances`(*X*, *maxk*, *metric='euclidean'*, *period=None*)

For each point, compute the distances from its first maxk nearest neighbours

Parameters

- **X** (*np.ndarray*) – points array of dimension N x D
- **maxk** (*int*) – number of neighbours to save
- **metric** (*str*) – metric used to compute the distances
- **period** (*float, np.ndarray(float)*) – sizes of PBC walls. Single value is interpreted as cubic box.

Returns

- **distances** (*np.ndarray(int)*) – N x maxk matrix, indices of the neighbours of each point

- **dist_indices** (*np.ndarray(float)*) – N x maxk matrix, distances of the neighbours of each point

`utils.`**`float_keys`**`(`*text*`)`

sort list in human order, for both numbers and letters http://nedbatchelder.com/blog/200712/human_sorting.html

`utils.`**`from_all_distances_to_nndistances`**`(`*pdist_matrix*, *maxk*`)`

Save the first maxk neighbours starting from the matrix of the distances

Parameters

- **`pdist_matrix`** (*`np.ndarray(float)`*) – N x N matrix of distances
- **`maxk`** (*`int`*) – number of neighbours to save

Returns

- **distances** (*np.ndarray(float)*) – N x maxk matrix, distances of the neighbours of each point
- **dist_indices** (*np.ndarray(int)*) – N x maxk matrix, indices of the neighbours of each point

`utils.`**`natural_keys`**`(`*text*`)`

sort list in human order, for both numbers and letters http://nedbatchelder.com/blog/200712/human_sorting.html

CHAPTER

FIVE

TUTORIAL: INTRINSIC DIMENSION, DENSITY ESTIMATION AND CLUSTERING

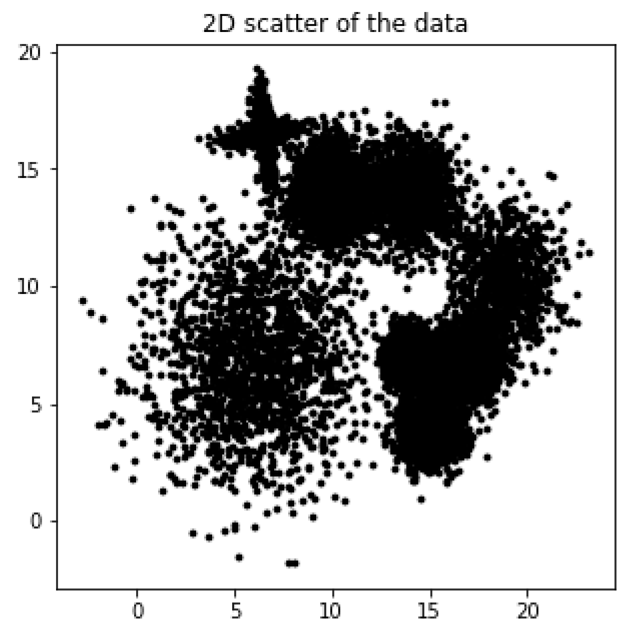
[1]:
```
import numpy as np
import matplotlib.pyplot as plt
from dadapy import Data
from dadapy.plot import plot_SLAn, plot_MDS, plot_matrix, get_dendrogram, plot_DecGraph


%load_ext autoreload
%autoreload 2
%matplotlib notebook
%matplotlib inline
```

[2]:
```
#ls
```

[3]:
```

# import a test set
X = np.genfromtxt('datasets/Fig1.dat')[:15000]

print(X.shape)
# X = np.genfromtxt('Fig2.dat')

# X = np.genfromtxt('FigS1.dat')
# X = np.genfromtxt('FigS2.dat')

# X = np.genfromtxt('FigS3.dat')

#X = np.genfromtxt('FigS4.dat')ls
```

```
(15000, 2)
```

[4]:
```
plt.figure(figsize = (5, 5))
plt.title('2D scatter of the data')
plt.scatter(X[:,0],X[:,1],s=15.,alpha=1.0,c='black',linewidths=0.0)
plt.show()
```



[5]:
```
# initialise the Data class
data = Data(X)
```

[6]:
```
# compute distances up to the maxk NN
data.compute_distances(maxk = 1000)
```

```
Computation of distances started
Computation of the distances up to 1000 NNs started
2.57 seconds for computing distances
```

[7]:
```
data.maxk
```

[7]:
```
1000
```

[8]:
```
# estimate ID
data.compute_id_2NN()
```

```
ID estimation finished: selecting ID of 2.0138881045367265
```

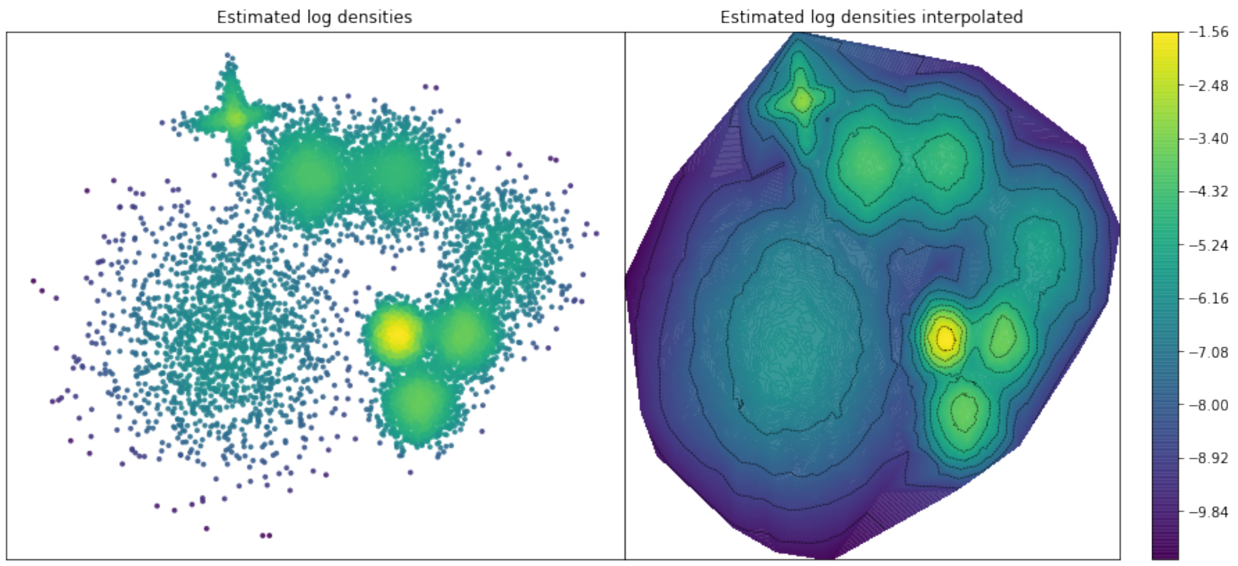
[8]:
```
(2.0138881045367265, 0.0, 0.066046618856829l9)
```

[9]:
```
data.compute_density_kstarNN()
```

```
kstar estimation started, Dthr = 23.92812698
0.76 seconds computing kstar
kstar-NN density estimation started
k-NN density estimation finished
```

[10]:
```
# estimate density with PAk using cython implementation of Newton-Raphson minimisation
# data.compute_density_PAk()
```

```
[11]: f, [ax1 ,ax2] = plt.subplots(1, 2, figsize = (16, 7),gridspec_kw={'hspace': 0.05, 'wspace
      ↪': 0})
      ax1.yaxis.set_major_locator(plt.NullLocator())
      ax1.xaxis.set_major_locator(plt.NullLocator())
      ax1.set_title('Estimated log densities')

      ax1.scatter(X[:,0],X[:,1],s=15.,alpha=0.9, c = data.log_den,linewidths=0.0)
      ax2.yaxis.set_major_locator(plt.NullLocator())
      ax2.xaxis.set_major_locator(plt.NullLocator())
      ax2.set_title('Estimated log densities interpolated')
      ax2.tricontour(X[:,0],X[:,1],data.log_den,levels=10, linewidths=0.5, colors='k')
      fig2=ax2.tricontourf(X[:,0],X[:,1],data.log_den,levels=250,alpha=0.9)

      plt.colorbar(fig2)
      plt.show()
```



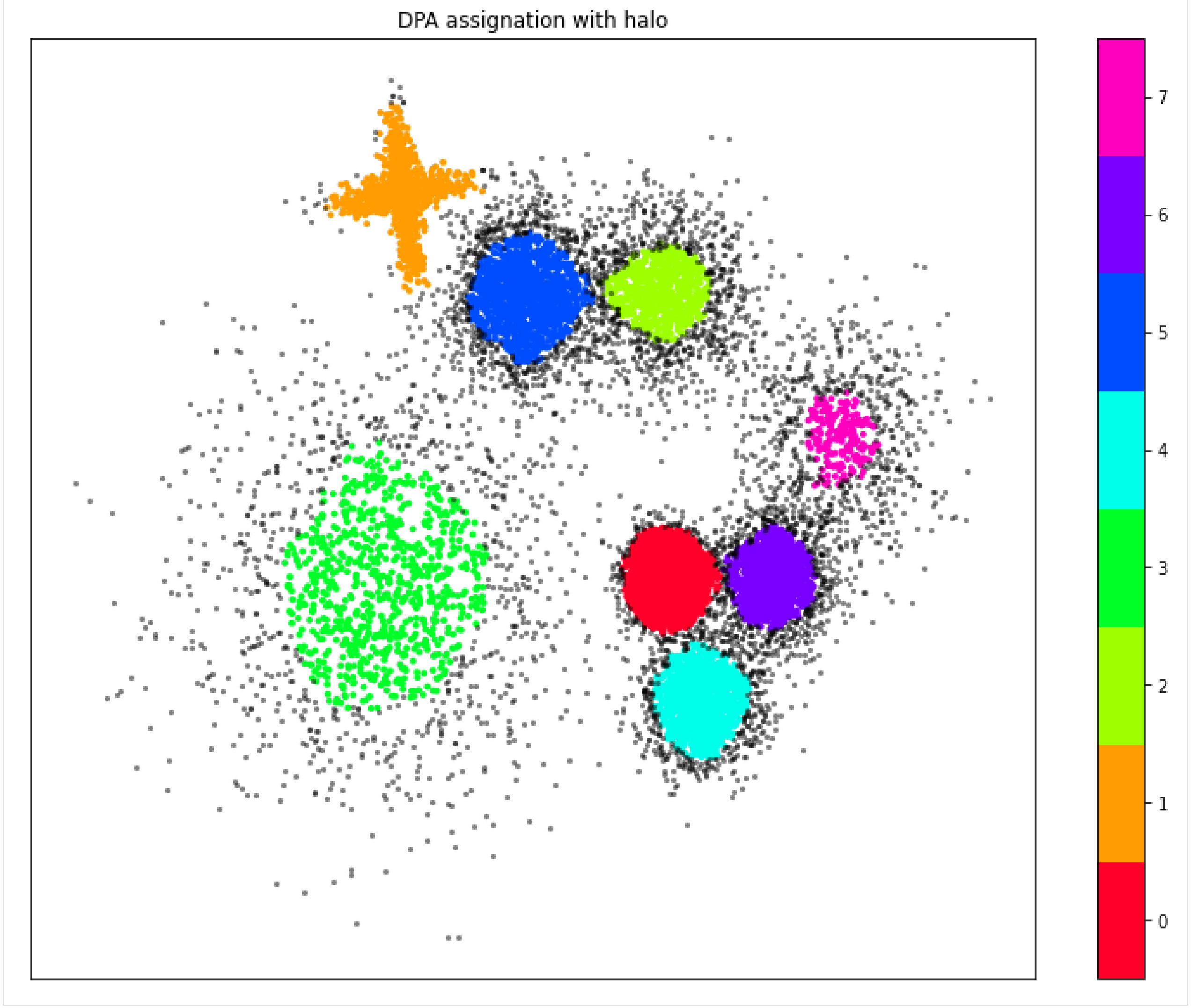
5.1 Advanced DP clustering

```
[12]: # estimate clusters
      data.compute_clustering(Z = 1.65, halo=True)
```

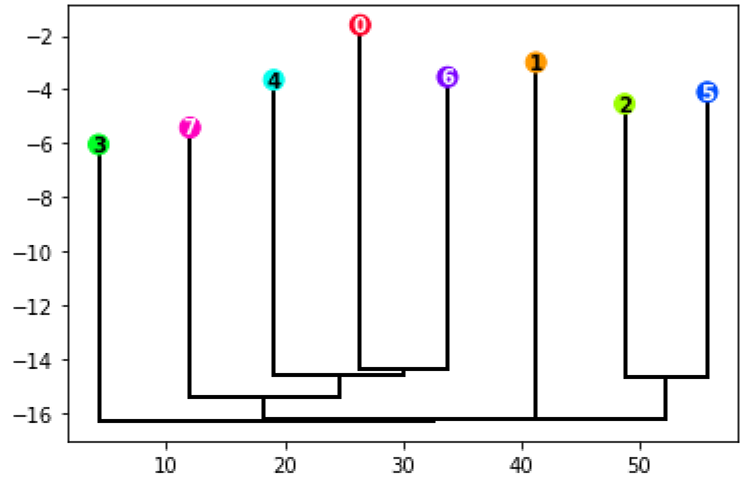
```
Clustering started
init succeded
part one finished
part two finished
part tree finished
('Number of clusters before multimodality test=', 9)
0.05 seconds clustering before multimodality test
0.01 seconds identifying the borders
0.00 seconds with multimodality test
0.00 seconds for final operations
Clustering finished, 8 clusters found
total time is, 0.18355083465576172
```

[13]:

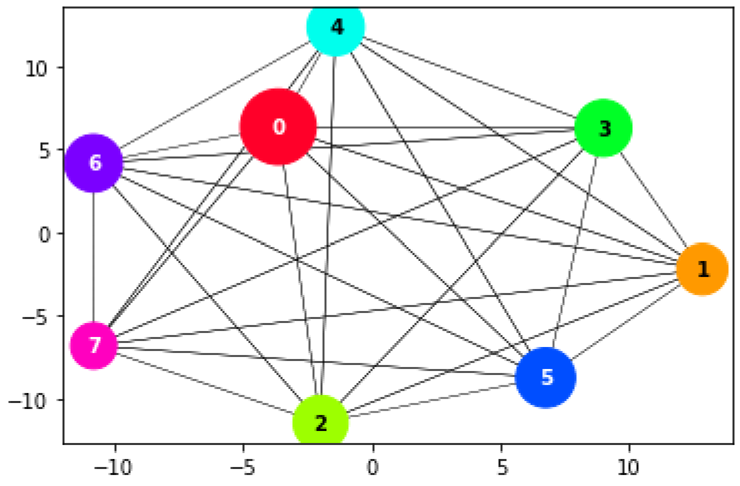
```
Nclus_m=len(data.cluster_centers)
cmap = plt.get_cmap('gist_rainbow', Nclus_m)
f, ax = plt.subplots(1, 1, figsize = (13, 10))
ax.yaxis.set_major_locator(plt.NullLocator())
ax.xaxis.set_major_locator(plt.NullLocator())
ax.set_title('DPA assignation with halo')
xdtmp=[]
ydtmp=[]
ldtmp=[]
xntmp=[]
yntmp=[]
for j in range(len(data.cluster_assignment)):
    if (data.cluster_assignment[j]!=-1):
        xdtmp.append(data.X[j,0])
        ydtmp.append(data.X[j,1])
        ldtmp.append(data.cluster_assignment[j])
    else:
        xntmp.append(data.X[j,0])
        yntmp.append(data.X[j,1])

plt.scatter(xdtmp,ydtmp,s=15.,alpha=1.0, c=ldtmp,linewidths=0.0,cmap=cmap)
plt.colorbar(ticks=range(Nclus_m))
plt.clim(-0.5, Nclus_m-0.5)
plt.scatter(xntmp,yntmp,s=10.,alpha=0.5, c='black',linewidths=0.0)
plt.show()
```



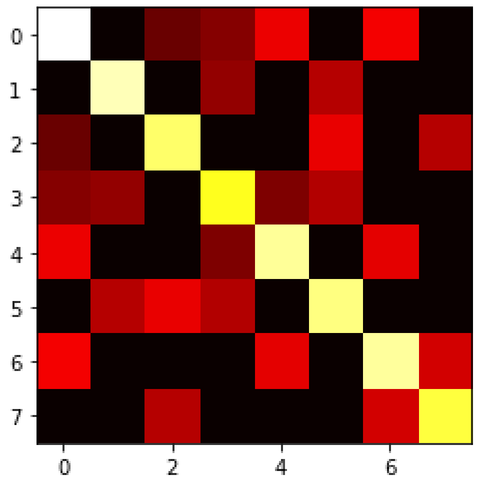
```
[14]: get_dendrogram(data,cmap="gist_rainbow")
```



```
[15]: # plot graph of clusters
      plot_MDS(data,cmap='gist_rainbow')
```



```
[16]: # plot connectivity matrix
      plot_matrix(data)
```

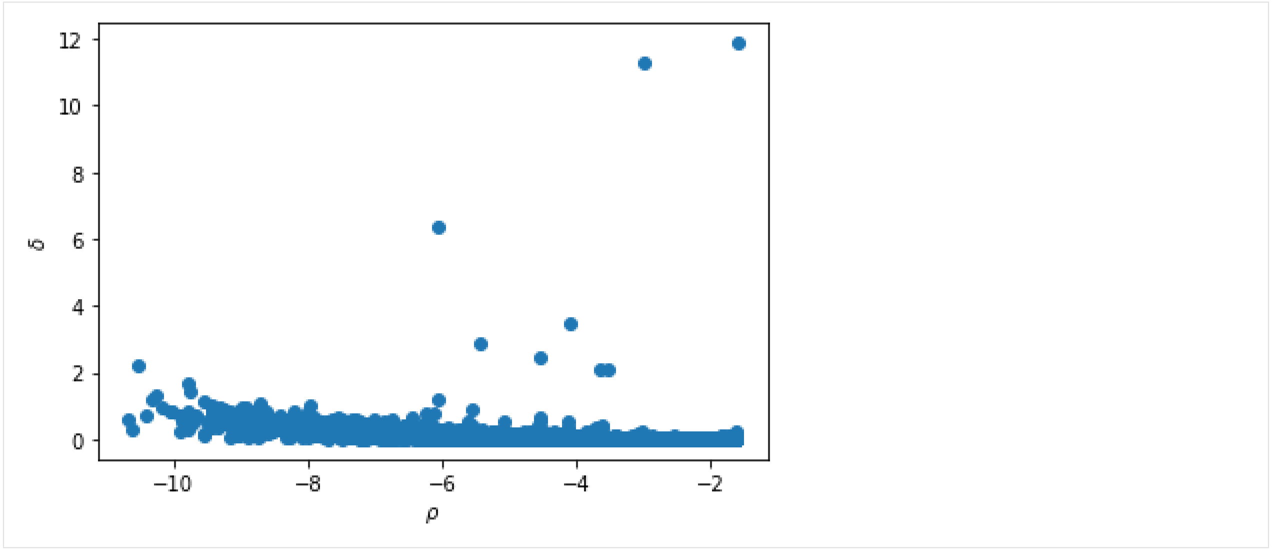


5.2 Classical DP clustering

```
[17]: data.compute_DecGraph()
```

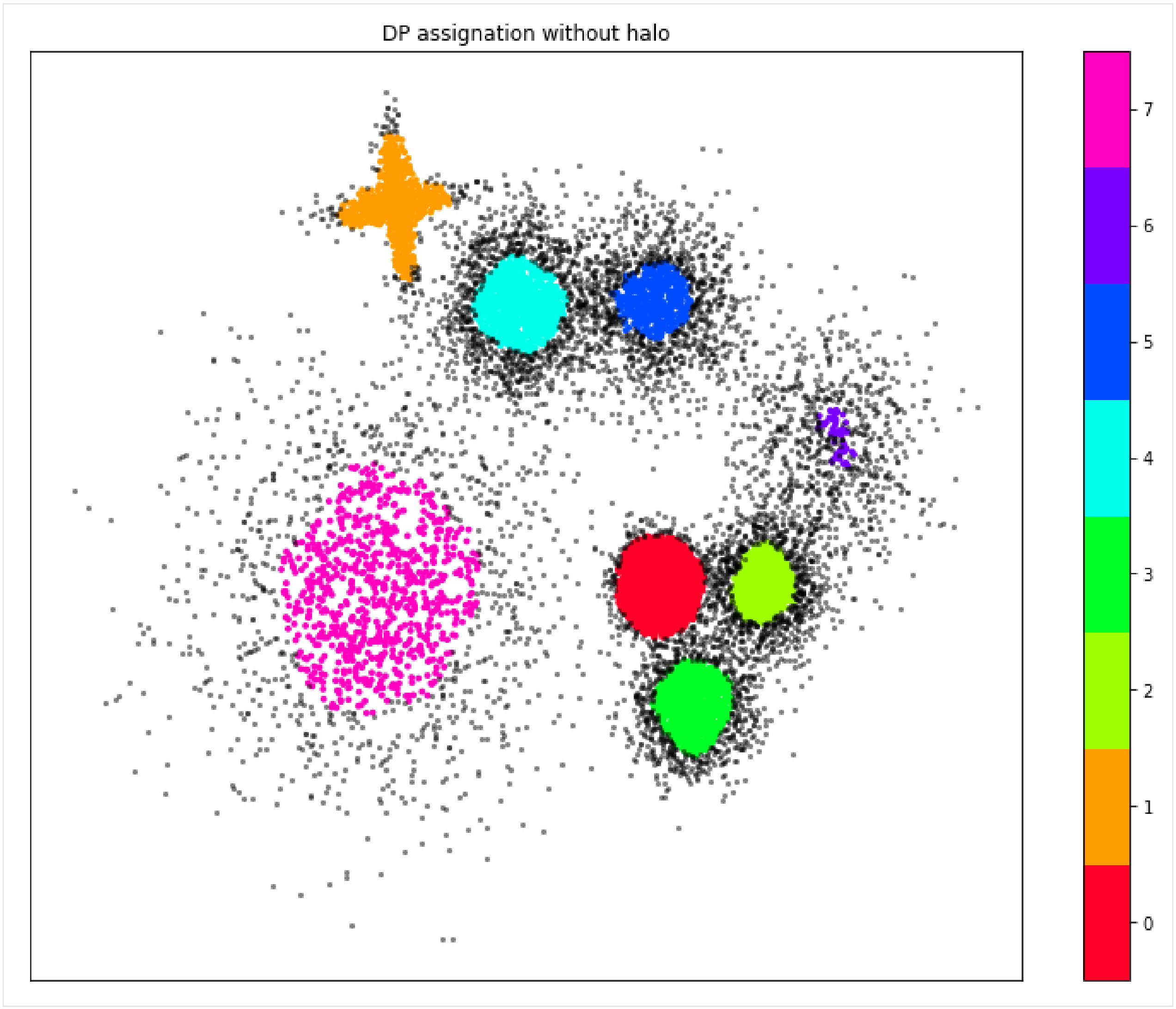
```
Number of points for which self.delta needed call to cdist= 6
```

```
[18]: plot_DecGraph(data)
```



[19]:
```
data.compute_cluster_DP(dens_cut=-7,delta_cut=1.75, halo=True)
```

[20]:
```
Nclus_m=np.max(data.cluster_assignment)+1
color= plt.get_cmap('gist_rainbow', Nclus_m)
f, ax = plt.subplots(1, 1, figsize = (13, 10))
ax.yaxis.set_major_locator(plt.NullLocator())
ax.xaxis.set_major_locator(plt.NullLocator())
ax.set_title('DP assignation without halo')
xdtmp=[]
ydtmp=[]
ldtmp=[]
xntmp=[]
yntmp=[]
for j in range(len(data.cluster_assignment)):
    if (data.cluster_assignment[j]!=-1):
        xdtmp.append(data.X[j,0])
        ydtmp.append(data.X[j,1])
        ldtmp.append(data.cluster_assignment[j])
    else:
        xntmp.append(data.X[j,0])
        yntmp.append(data.X[j,1])

plt.scatter(xdtmp,ydtmp,s=15.,alpha=1.0, c=ldtmp,linewidths=0.0,cmap=color)
plt.colorbar(ticks=range(Nclus_m))
plt.clim(-0.5, Nclus_m-0.5)
plt.scatter(xntmp,yntmp,s=10.,alpha=0.5, c='black',linewidths=0.0)
plt.show()
```


DP assignation without halo
0
1
2
3
4
5
6
7

```
[ ]:
```

CHAPTER

SIX

TUTORIAL: INFORMATION IMBALANCE

[1]:
```
import matplotlib.pyplot as plt
import numpy as np

from dadapy.plot import plot_inf_imb_plane
from dadapy.metric_comparisons import MetricComparisons

%matplotlib inline
```

[2]:
```
%load_ext autoreload
%autoreload 2
```

6.1 3D Gaussian with small variance along z

In this section we define a simple dataset sampled from a 3D Gaussian distribution with a small variance along the z axis.

[3]:
```
# sample dataset

N = 1000

cov = np.identity(3)
cov[-1, -1] = 0.01**2 # variance along z is much smaller!
mean = np.zeros(3)
X = np.random.multivariate_normal(mean = mean, cov = cov, size = (N))
```

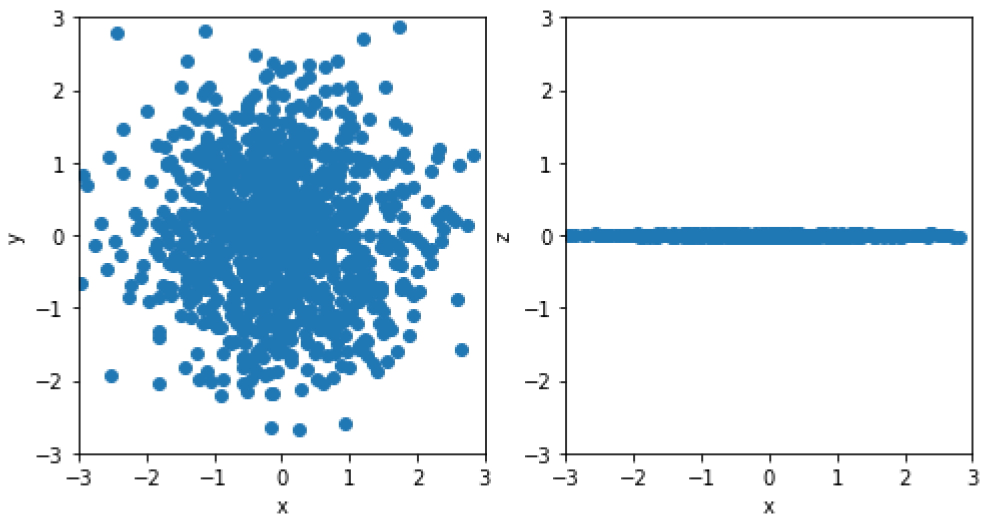
[4]:
```
# plot the sampled points

fig, (ax1, ax2) = plt.subplots(1, 2, figsize = (8, 4))

ax1.scatter(X[:, 0], X[:, 1])
ax1.set_xlim(-3, 3)
ax1.set_ylim(-3, 3)
ax1.set_xlabel('x')
ax1.set_ylabel('y')
```

(continues on next page)

(continued from previous page)

```
ax2.scatter(X[:, 0], X[:, 2])
ax2.set_xlim(-3, 3)
ax2.set_ylim(-3, 3)
ax2.set_xlabel('x')
ax2.set_ylabel('z')
```

[4]:
```
Text(0, 0.5, 'z')
```



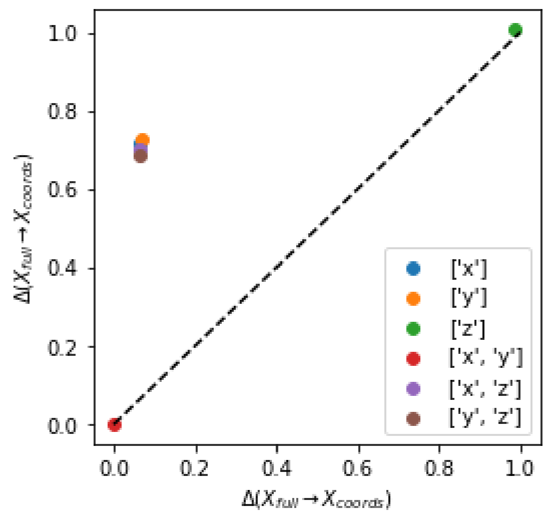
[5]:
```
# define an instance of the MetricComparisons class

d = MetricComparisons(X)
```

[6]:
```
# list of the coordinate names
labels = ['x', 'y', 'z']

# list of the the subsets of coordinates for which the imbalance should be computed
coord_list = [[0,], [1,], [2,], [0,1], [0,2], [1, 2]]
```

[7]:
```
# compute the information imbalances

imbalances = d.return_inf_imb_full_selected_coords(coord_list)
```

```
total number of computations is:  6
total number of computations is:  6
```

[8]:
```
# plot information imbalance plane

plot_inf_imb_plane(imbalances, coord_list, labels)
```



From the graph above we see that the small variance along the z makes the $[x, y]$ space equivalent to the $[x, y, z]$ space (red circle) and the z space (green circle) is seen to be much less informative than the x and y spaces (bloue and orange circles).

6.2 4D isotropic Gaussian

In this example we explore the possibility of having a symmetrical yet partial sharing of information between two spaces. We will take the case of a dataset sampled from a 4D isotropic Gaussian.

[9]:
```
# sample the dataset
N = 2000

X = np.random.normal(size = (N, 4))
```

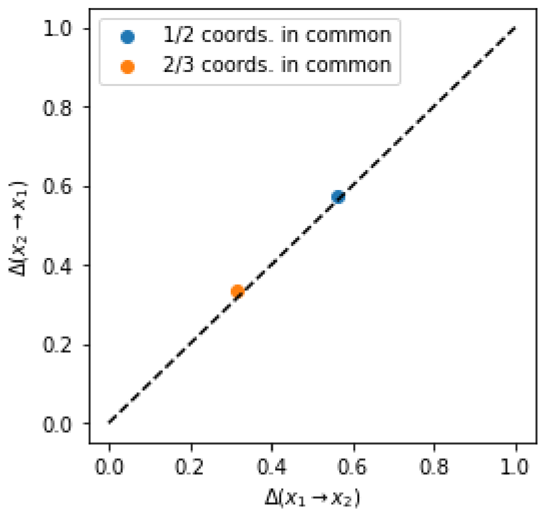
[10]:
```
# define an instance of the MetricComparisons class

d = MetricComparisons(X, maxk=X.shape[0]-1)
```

[11]:
```

# compute the imbalance between the [x, y] space and the [y, z] space
imb_1common = d.return_inf_imb_two_selected_coords(coords1= [0, 1,], coords2= [1, 2])

# compute the imbalance between the [x, y, z] space and the [y, z, w] space
imb_2common = d.return_inf_imb_two_selected_coords(coords1= [0, 1, 2, ], coords2= [1, 2,␣
↪3])

print(imb_1common)
print(imb_2common)
```

```
(0.5642185, 0.571296)
(0.3162855, 0.334992)
```

[12]:
```
# plot information imbalance plane
plt.figure(figsize=(4, 4))

plt.scatter(imb_1common[0], imb_1common[1], label = '1/2 coords. in common')
plt.scatter(imb_2common[0], imb_2common[1], label = '2/3 coords. in common')

plt.plot([0, 1], [0, 1], 'k--')
plt.xlabel(r'$\Delta(x_1 \rightarrow x_2) $')
plt.ylabel(r'$\Delta(x_2 \rightarrow x_1) $')

plt.legend()
```

[12]:
```
<matplotlib.legend.Legend at 0x7fb882461970>
```



We see that the information imbalances between the spaces $[x, y, z]$ and $[y, z, w]$ (orange circle) are lower than the imbalances between $[x, y]$ and $[y, z]$ (blue circle). However in both cases, since the information shared between the spaces is symmetric, the correponding point lies alog the diagonal of the information imbalance plane.

6.3 10D isotropic Gaussian dataset

In this example we analayse the information imbalance for a 10D Gaussian and isotropic dataset.

[13]:
```
# sample data

N = 1000
X = np.random.normal(size = (N, 10))
```

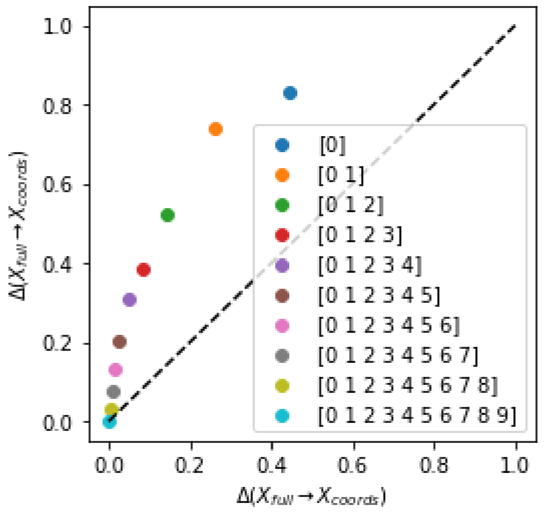
[14]:
```
# define an instance of the MetricComparisons class

d = MetricComparisons(X, maxk = X.shape[0]-1)
```

```
[15]: # define labels of coordinates and the coordinate sets we want to analyse

labels = ['x{}'.format(i) for i in range(10)]

coord_list = [np.arange(i) for i in range(1, 11)]
```

```
[16]: # compute the information imbalances

imbalances = d.return_inf_imb_full_selected_coords(coord_list)
```

```
total number of computations is:  10
total number of computations is:  10
```

```
[17]: # plot information imbalance plane

plot_inf_imb_plane(imbalances, coord_list)
```



We see that all subsets of coordinates are contained in the full space and that adding coordinates progressively brings the information imbalance to zero.

6.4 Sinusoidal function

In this section we will show that the information imbalance is capable of correctly detecting information asymmetries also in the presence of arbitrary nonlinearities.

```
[18]: # sample a noisy sinusoidal dataset

N = 1000

x1 = np.linspace(0,1, N)
x1 = np.random.uniform(0, 1, N)
x1 = np.sort(x1)
x1 = np.atleast_2d(x1).T
```

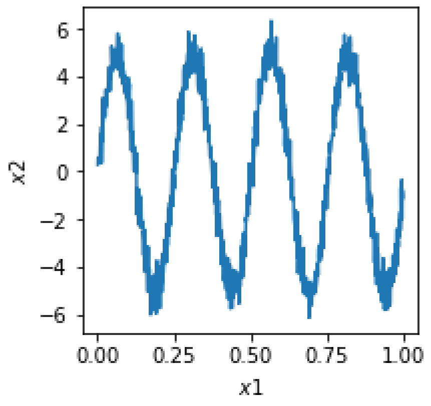
(continues on next page)

(continued from previous page)

```
x2 = 5*np.sin(x1*25)+np.random.normal(0, .5, (N, 1))

X = np.hstack([x1, x2])
```

[19]:
```
# plot the data, note that the variance is much higher along x2 than along x1
plt.figure(figsize = (3, 3))
plt.plot(x1, x2);
plt.xlabel(r'$x1$')
plt.ylabel(r'$x2$')
```

[19]:
```
Text(0, 0.5, '$x2$')
```



[20]:
```
# define an instance of the MetricComparisons class

d = MetricComparisons(X, maxk = X.shape[0]-1)
```

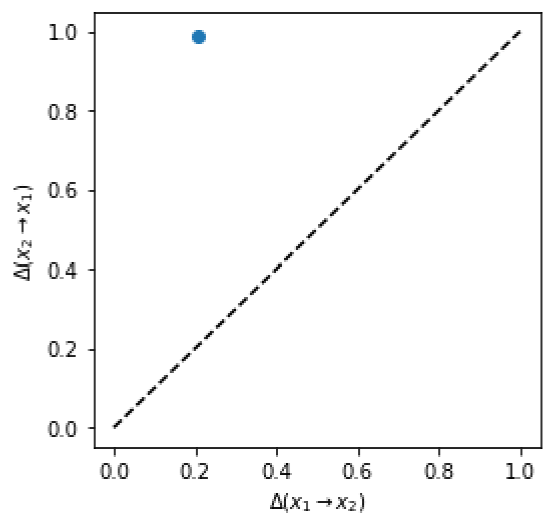
[21]:
```
imb01, imb10 = d.return_inf_imb_two_selected_coords([0], [1])
```

[22]:
```
print(imb01, imb10)
```

```
0.20694200000000001 0.99035000000000001
```

[23]:
```
# plot information imbalance plane
plt.figure(figsize=(4, 4))

#plot_inf_imb_plane(imbalances, coord_list, labels)
plt.scatter(imb01, imb10)

plt.plot([0, 1], [0, 1], 'k--')
plt.xlabel(r'$\Delta(x_1 \rightarrow x_2) $')
plt.ylabel(r'$\Delta(x_2 \rightarrow x_1) $')
```

[23]:
```
Text(0, 0.5, '$\\Delta(x_2 \\rightarrow x_1) $')
```



The information imbalance from x_1 to x_2 is much lower than that from x_2 to x_1. The x_1 space is detected as more informative, in spite of the much larger variance of the data along x_2.

CHAPTER

SEVEN

INDICES AND TABLES

- genindex
- modindex
- search

PYTHON MODULE INDEX

INDEX