persistable

Luis Scoccola and Alexander Rolle

Jan 30, 2024

CONTENTS

Persistent and stable clustering (Persistable) is a density-based clustering algorithm intended for exploratory data analysis. What distinguishes Persistable from other clustering algorithms is its visualization capabilities. Persistable's interactive mode lets you visualize multi-scale and multi-density cluster structure present in the data. This is used to guide the choice of parameters that lead to the final clustering.

Make sure you are using Python 3. Persistable depends on the following python packages, which will be installed automatically when you install with *pip*: *numpy*, *scipy*, *scikit-learn*, *cython*, *plotly*, *dash*, *diskcache*, *multiprocess*, *psutil*. To install from pypi, simply run the following:

```
pip install persistable-clustering
```

CHAPTER

ONE

CONTENTS

1.1 Quick start

This is a quick guide to using Persistable's interactive mode. For a more thorough introduction, see *Introduction to Persistable*.

As a running example, we'll use a synthetic dataset from the hdbscan GitHub repository. This is a great test dataset, since it is quite challenging for most clustering algorithms, but easy to visualize.

```
import numpy as np
import matplotlib.pyplot as plt
from urllib.request import urlopen
from io import BytesIO

# fetch the data from the hdbscan repo
url = "https://github.com/scikit-learn-contrib/hdbscan/blob/
↪4052692af994610adc9f72486a47c905dd527c94/notebooks/clusterable_data.npy?raw=true"
f = urlopen(url)
rf = f.read()
data = np.load(BytesIO(rf))

# plot the data
plt.figure(figsize=(10,10))
plt.scatter(data[:,0], data[:,1], alpha=0.5)
plt.show()
```

To get started, create a Persistable object:

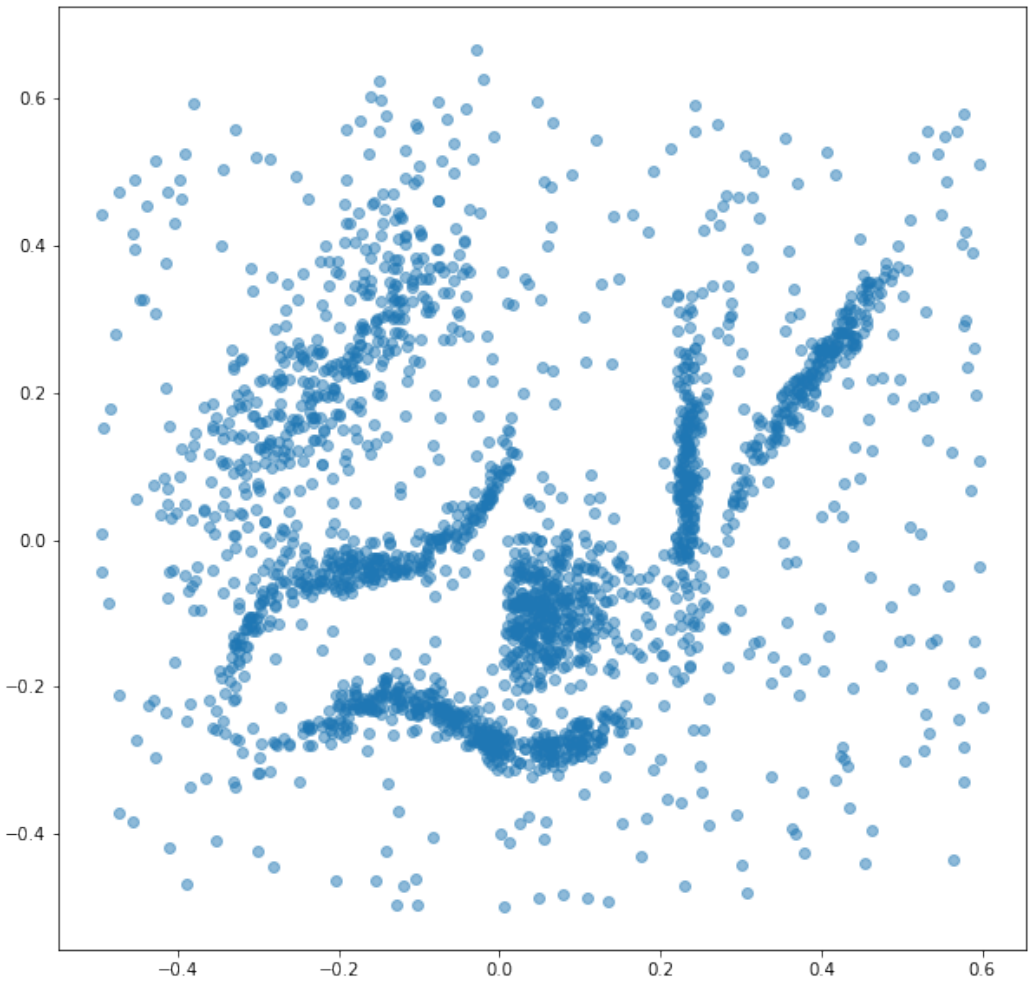
```
p = persistable.Persistable(data, n_neighbors="all")
```

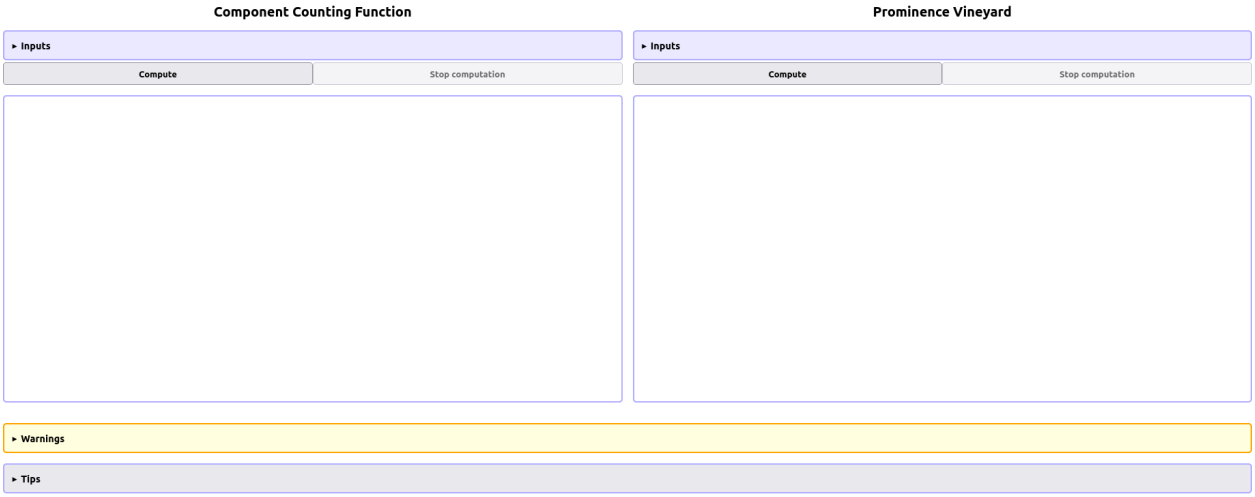
The `n_neighbors` parameter gives an upper bound on how many neighbors of a data point we consider in all future computations. Persistable primarily looks at the nearest neighbors of a data point to estimate density, so it suffices to look at a local neighborhood of the point. Thus `n_neighbors` can be a small fraction of the total size of the dataset. If the dataset is small and you aren't worried about computation time, you can set `n_neighbors` to `"all"`.

Next we launch Persistable's interactive mode:

```
pi = persistable.PersistableInteractive(p)
port = pi.start_ui()
```

The variable `port` contains the port in `localhost` serving the GUI, which `8050` by default, but may be different if that port is in use. Accessing `localhost:[port]` opens a window that is initially pretty empty:


0.6
0.4
0.2
0.0
−0.2
−0.4
−0.4
−0.2
0.0
0.2
0.4
0.6



To get started, press the "Compute" button under "Component Counting Function". After the computation is finished, you should see a plot of the component counting function:

To zoom in or out, adjust the parameters in the "Inputs" box.

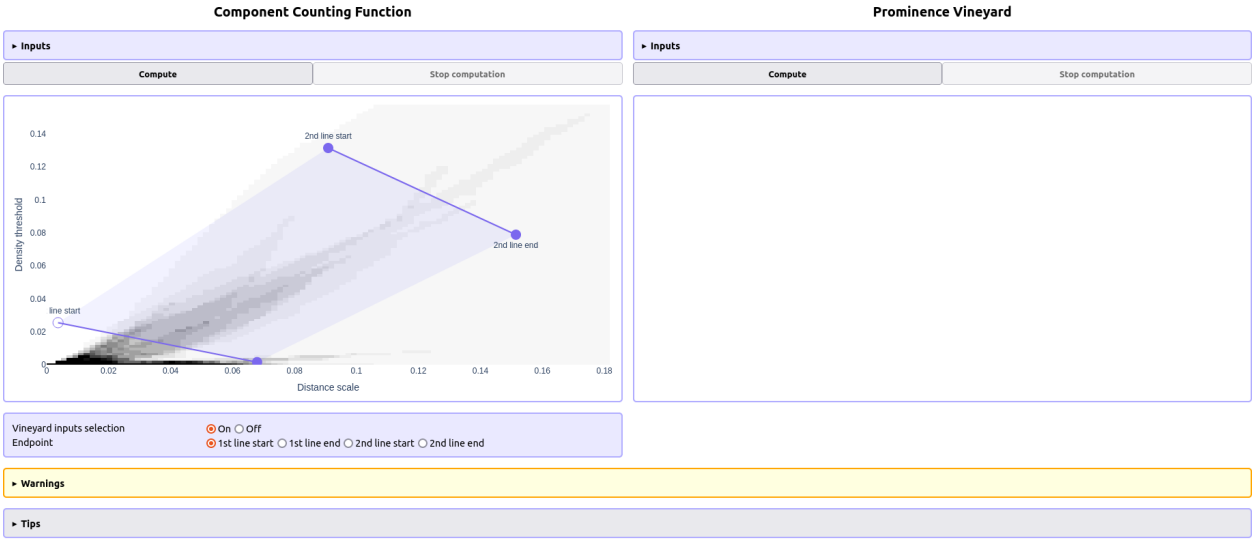
The component counting function shows how many clusters the data has, with respect to a distance scale parameter (horizontal axis) and a density threshold (vertical axis). These are closely related to the parameters of the DBSCAN clustering algorithm. See *Introduction to Persistable* for more details.

Unlike DBSCAN, Persistable will not fix either of these parameters. Instead, the output of Persistable will depend on a "slice" of this parameter space: a line running down and to the right. Any slice determines a hierarchical clustering of the data.

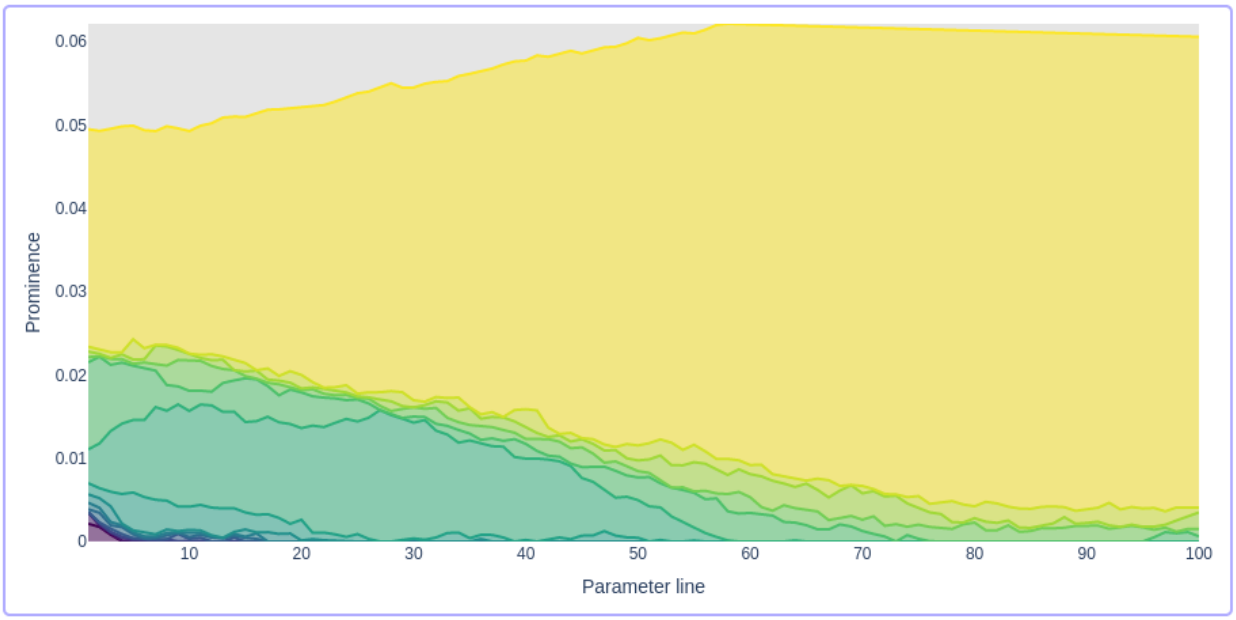
The next step in Persistable's interactive mode is the **Prominence Vineyard**. To use this tool, choose *two* slices:

To do this, turn the "Vineyard inputs selection" on, and choose the slices by choosing their endpoints. You can choose the endpoints by clicking on the component counting function plot, or by entering the coordinates of the endpoints in the "Inputs" box under "Prominence Vineyard".

It's often a good strategy to choose the first slice in a region with many clusters, and the second slice in a region with few clusters, as above.



Now press “Compute” under “Prominence Vineyard”. After the computation is complete, the vineyard is displayed:

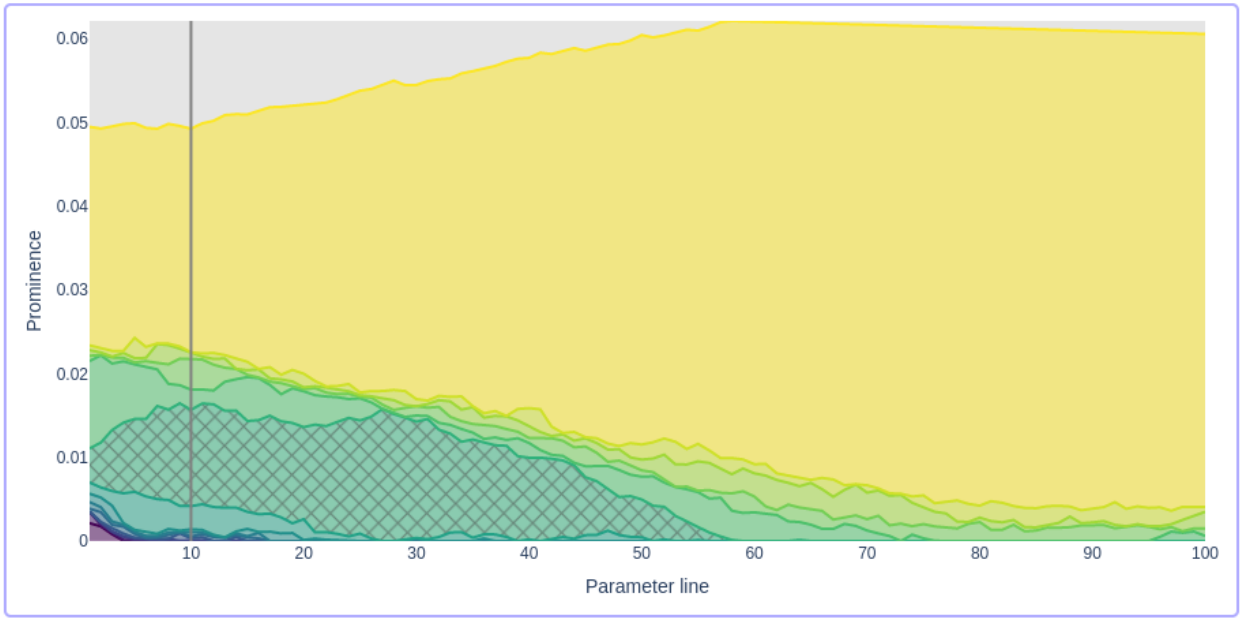


The Prominence Vineyard is constructed by taking a family of slices that interpolate between the two user-selected slices. The curves (“vines”) in the Prominence Vineyard represent clusters in the data that evolve with the choice of slice. The larger their prominence value (plotted on the vertical axis), the more likely they are to represent real structure in the data.

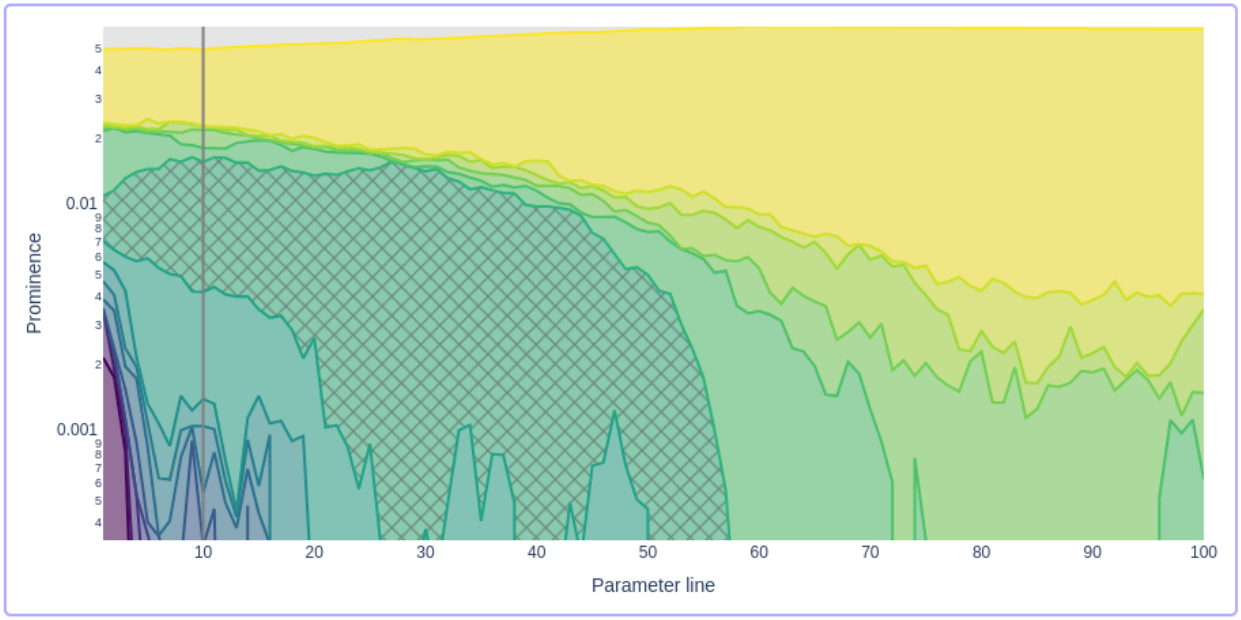
To get a clustering, turn on the “Parameter selection” option under “Prominence Vineyard”. The “Gap number” determines the number of clusters in the output. As you change this number, the corresponding “gap” in the Prominence Vineyard is marked. Here we have gap number 6:

The larger the gap you choose, the more stable the resulting clustering will be; thus, it’s usually a good idea to choose a large gap: a gap that extends over many slices (horizontally), and that covers a large range of prominence values (vertically).

When looking for a clustering with more than 2 or 3 clusters, it’s often helpful to display the prominences in log scale,



using the option in the “Inputs” box under “Prominence Vineyard”:



After choosing a gap number, choose a slice where this gap is large, and you’ve made all the choices necessary to get a clustering. So, press the “Choose parameter” button.

Now where’s the clustering? You can get cluster labels for the data points from the PersistableInteractive object:

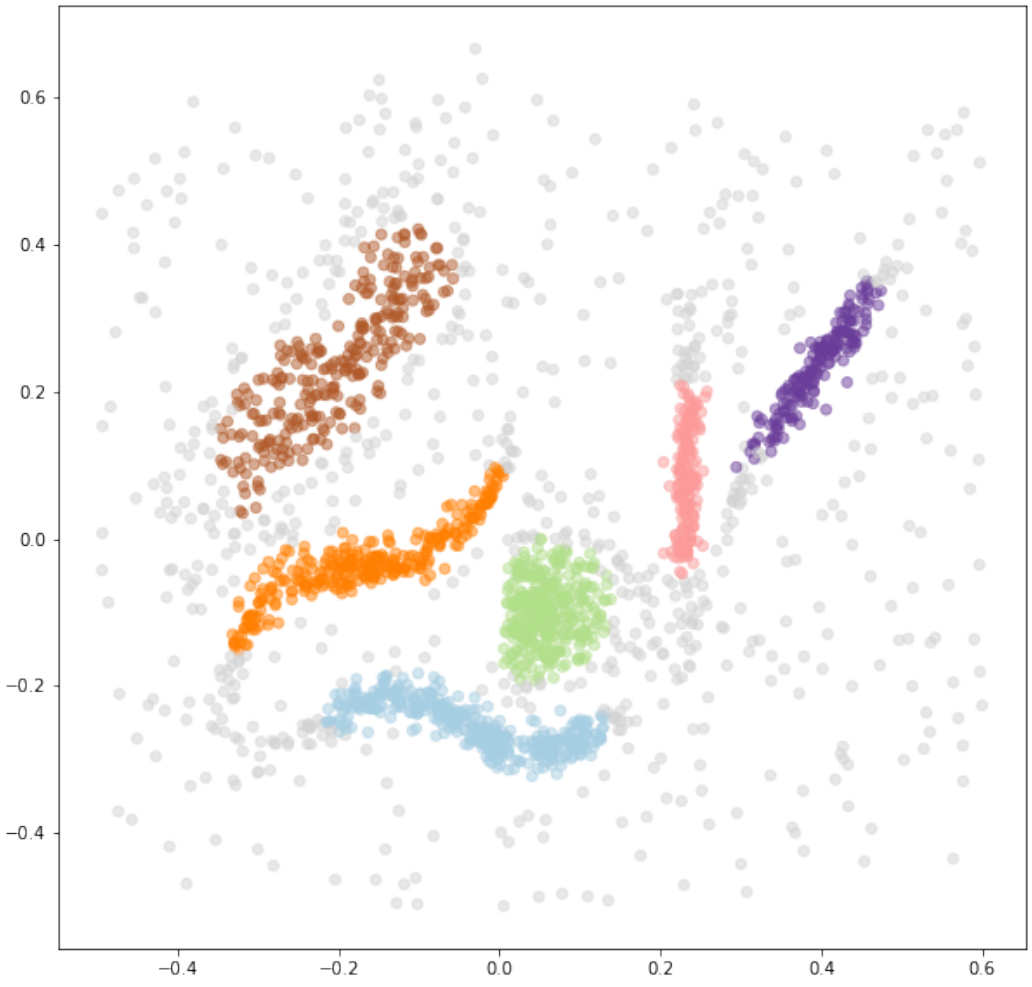
```
labels = pi.cluster()
```

Points labelled -1 are unclustered (noise), and the clusters are labelled starting from 0. On our running example, we can visualize the result by plotting the points with colors corresponding to the labels:

```
# choose color scheme
import matplotlib
cNorm  = matplotlib.colors.Normalize(vmin=0, vmax=np.amax(labels))
scalarMap = matplotlib.cm.ScalarMappable(norm=cNorm, cmap='Paired')
noise_color = (211/255, 211/255, 211/255, 1) # light grey
colors = [noise_color if x==-1 else scalarMap.to_rgba(x) for x in labels]

# plot data with clusters indicated by colors
plt.figure(figsize=(10,10))
plt.scatter(data[:,0], data[:,1], c=colors, alpha=0.5)

plt.show()
```



1.2 Introduction to Persistable

This tutorial explains how Persistable works. We'll focus especially on Persistable's interactive parameter selection features. See *Quick start* for a fast introduction to using the software.

As a running example, we'll use a synthetic dataset from the hdbscan GitHub repository. This is a great test dataset, since it is quite challenging for most clustering algorithms, but easy to visualize.

```
import numpy as np
import matplotlib.pyplot as plt
from urllib.request import urlopen
from io import BytesIO

# fetch the data from the hdbscan repo
url = "https://github.com/scikit-learn-contrib/hdbscan/blob/
↪4052692af994610adc9f72486a47c905dd527c94/notebooks/clusterable_data.npy?raw=true"
f = urlopen(url)
rf = f.read()
data = np.load(BytesIO(rf))

# plot the data
plt.figure(figsize=(10,10))
plt.scatter(data[:,0], data[:,1], alpha=0.5)
plt.show()
```

1.2.1 Background: the DBSCAN algorithm

To understand how Persistable works, it's helpful to first understand the DBSCAN algorithm, perhaps the best-known approach to density-based clustering. (In fact, we will discuss a minor modification of the original algorithm, sometimes called DBSCAN*. This handles the so-called "border" points in a way that is more consistent with a statistical intepretation of clustering in terms of level sets of a density.)

Say our data comes in the form of a finite set of points X with a metric (like the points plotted in 2D above). The parameters of DBSCAN are a distance scale s (sometimes called epsilon) and a density threshold k (sometimes called min_samples). Now, the "core points" *C(X)* are those points of X with at least k neighbors in their s-neighborhood:

$$C(X)_{s,k} = \{x \in X : |B(x,s)| \geq k\}.$$

Here, *B(x,s)* is the set of points whose distance from x is no more than s.

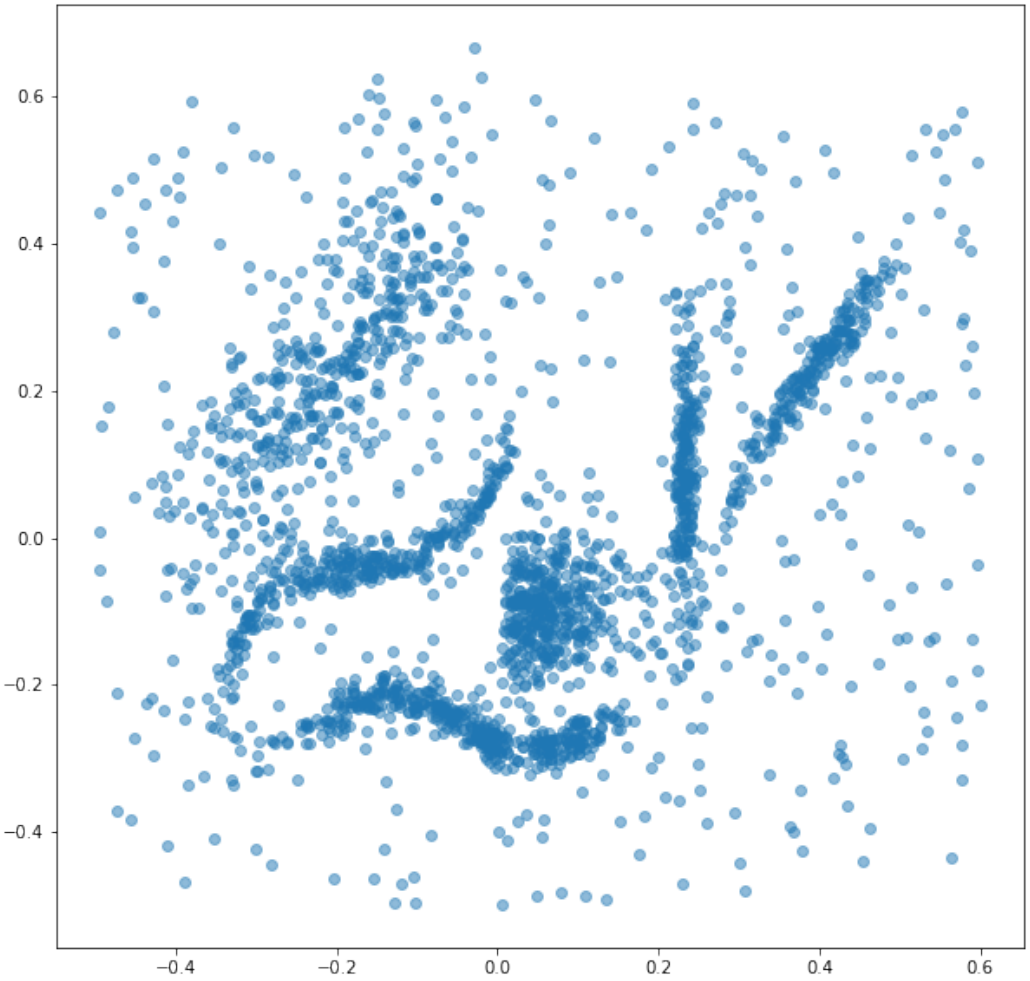
The DBSCAN graph takes these core points as vertices, and puts an edge between two core points if the distance between them is no more than s:

$$\begin{aligned} \text{vertices } G(X)_{s,k} &= C(X)_{s,k}. \\ \text{edges } G(X)_{s,k} &= \{\{x,y\} \in C(X)_{s,k} : d(x,y) \leq s\}. \end{aligned}$$

Now, the DBSCAN clustering of X (with respect to s and k) is just the set of connected components of the DBSCAN graph .

As the values of the parameters s and k vary, we get a whole hierarchy of graphs that encode cluster structure in the data. These are hierarchical in the sense that if two data points are clustered together in the graph , then they remain together for any larger s or smaller k.

We can get a feel for this 2-parameter hierarchy of graphs by plotting the **component counting function**: for each *(s,k)*, we simply count the number of components in the graph .


0.6
0.4
0.2
0.0
−0.2
−0.4
−0.4
−0.2
0.0
0.2
0.4
0.6

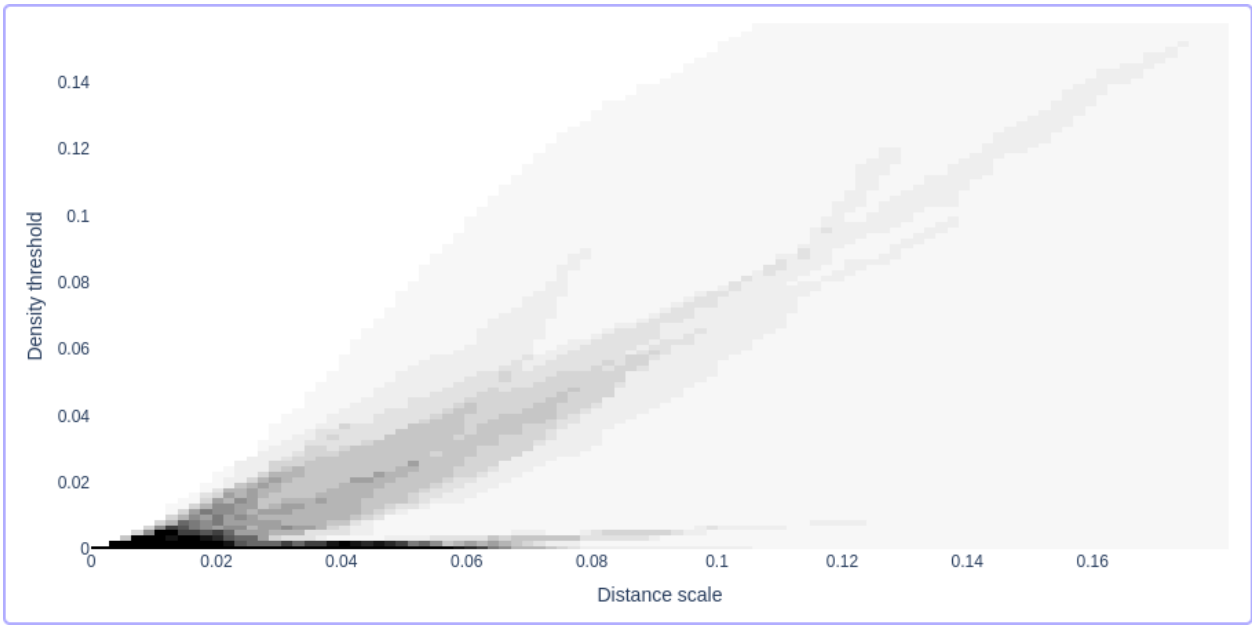


Fig. 1: The component counting function of our running example. The darker the shade of grey, the more components.

In this plot, we see the typical behavior of the DBSCAN graphs. When *s* is small and *k* is large, no points qualify as core points, and we see no clusters. When *s* is large and *k* is small, we have many core points and many edges, and the graph is connected.

In between these two regimes, there is a band in which interesting things are happening.

As a descriptor of our data, the component counting function is usually too coarse to really see what's going on. But at least it gives us an idea of the range of values of *s* and *k* that lead to interesting cluster structure.

Rather than fix a single choice of the parameters *s* and *k* like DBSCAN, Persistable takes a multi-scale approach, which we will describe next.

1.2.2 Clustering with Persistable

Conceptually, Persistable constructs a single clustering of data from the DBSCAN graphs in two steps.

Step 1: Reduce from the 2-parameter hierarchy of graphs *G(X)* to a 1-parameter hierarchy by taking a *slice*.

A slice is defined by a line in the *(s,k)*-space:

$$k = ms + b \quad \text{with} \quad m < 0 \quad \text{and} \quad b > 0$$

which gives us a 1-parameter hierarchy of graphs:

$$G(X)_r = G(X)_{r,mr+b}\,.$$

We always take $m < 0$, so that *k* decreases as *s* increases. This has the effect that, as *r* increases, it gets easier for points of to be in the same component. We get a 1-parameter hierarchical clustering of the data by taking the components of .

Since *k* and *s* both change with *r*, this hierarchical clustering reflects cluster structure at a range of density thresholds and distance scales.

An important feature of Persistable is an interactive tool for choosing slices. We'll discuss this tool, and how to choose a slice in practice, after we discuss the second step of Persistable.

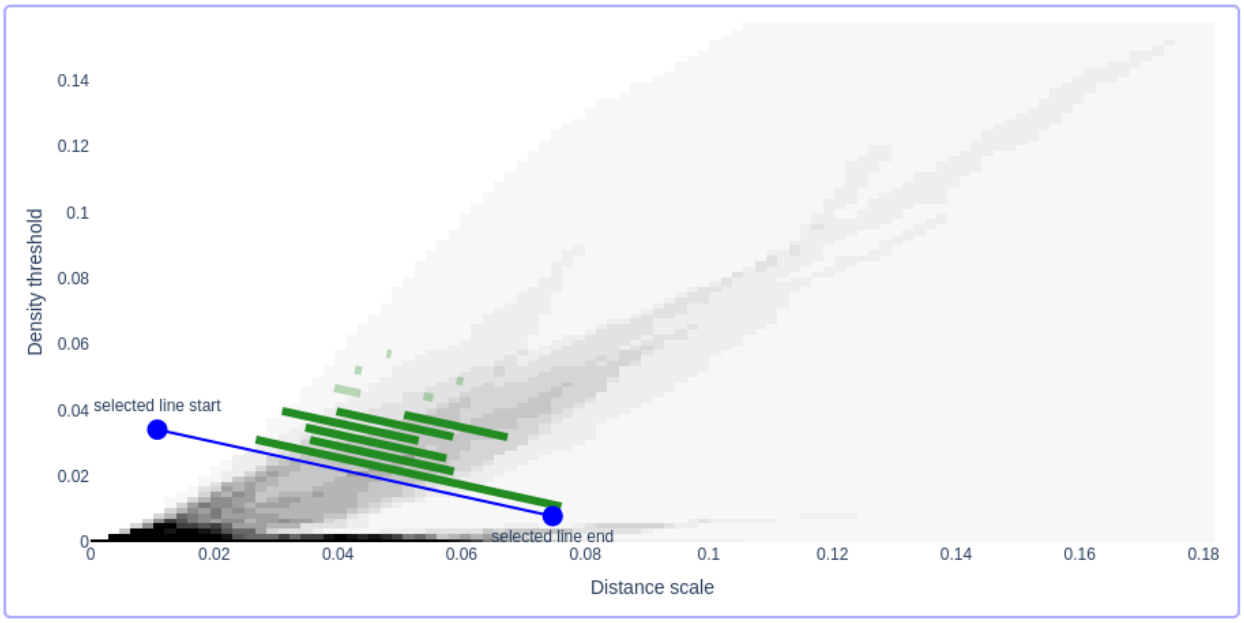


Fig. 2: A slice of *G(x)* is plotted in blue, along with its barcode, in green.

Step 2: Extract a single clustering of the data from the 1-parameter slice using the notion of *persistence* from topological data analysis.

In the picture above, a slice of *G(X)* is plotted in blue, and its *barcode* is plotted in green. The barcode of a 1-parameter hierarchical clustering is a visualizable summary of the hierarchy. A bar is born when a cluster is born in the hierarchy, and when two clusters merge, one of the corresponding bars ends, according to the Elder rule. For more on this construction, see *The barcode of a hierarchical clustering*.

In this example, there are six fairly long bars (in dark green), as well as some shorter bars (in light green). Longer bars represent clusters that survive across a bigger range of scales, and are thus more likely to represent real structure in the data, rather than an artifact of the algorithm.

To get a clustering of the data from the slice, we only have to choose how many bars we want to keep. If we choose *n* bars, the output of Persistable will consist of *n* clusters. These clusters correspond to the *n* longest bars in the barcode, so the strategy is to look at the barcode and try to draw a line between long bars and short bars.

In this example, 6 bars is a reasonable choice. Now, the *persistence-based flattening algorithm* extracts a clustering with 6 clusters. See *The persistence-based flattening algorithm* for a description of this algorithm.

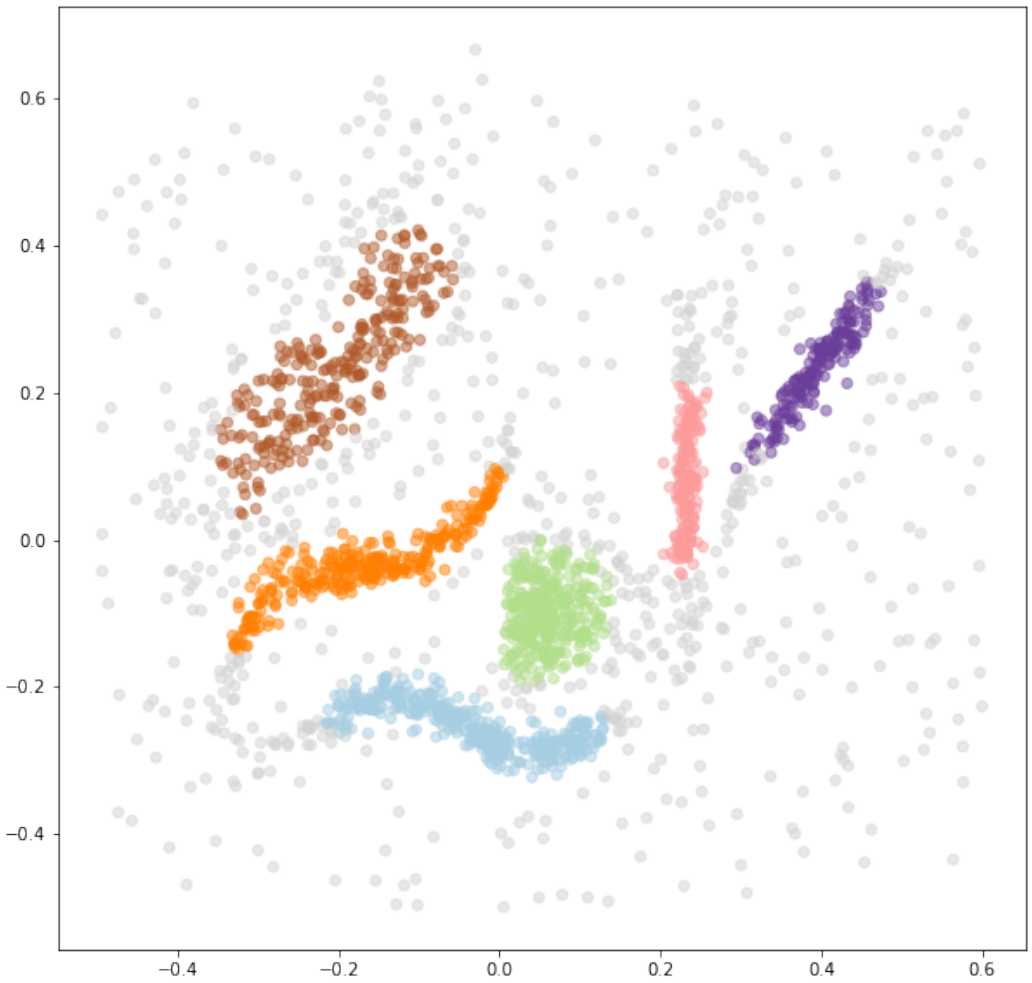
Going back to our example dataset, we get the following result (gray points do not belong to any cluster):

1.2.3 Choosing a slice

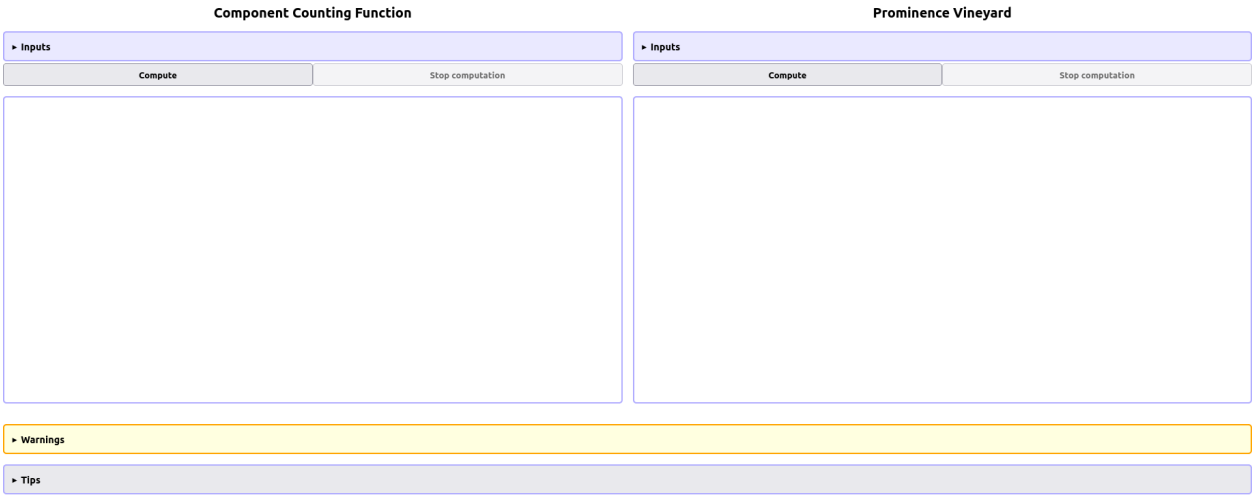
Persistable's interactive mode provides several visualization tools that help the user choose a slice. The rest of this tutorial is an introduction to these tools. For a quick guide, see *Quick start*.

To plot the component counting function using Persistable, we use Persistable's interactive mode:

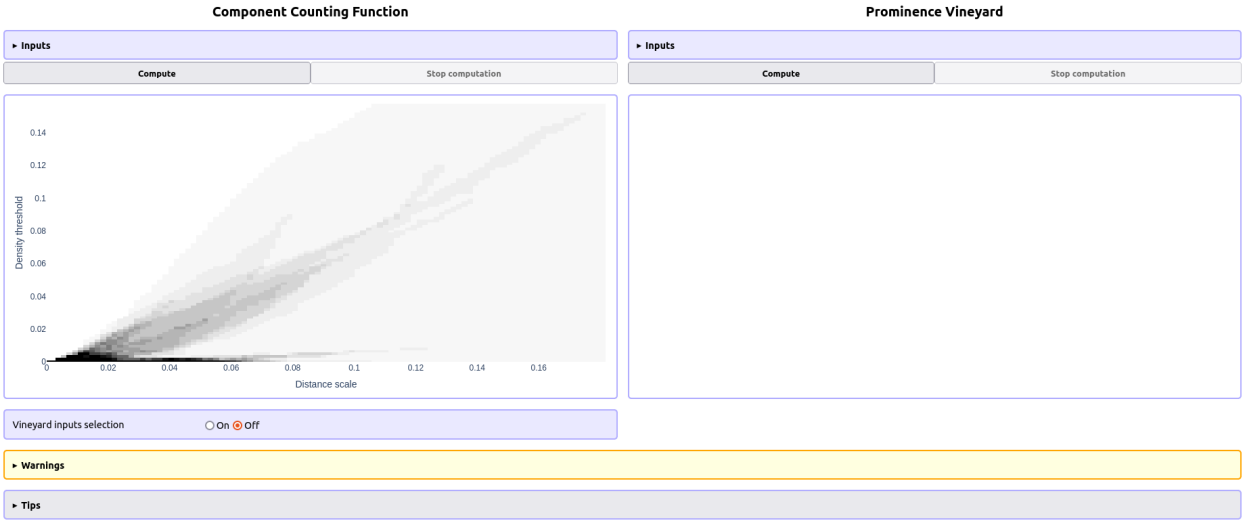
```
p = persistable.Persistable(data, n_neighbors="all")
pi = persistable.PersistableInteractive(p)
port = pi.start_ui()
```


0.6
0.4
0.2
0.0
−0.2
−0.4
−0.4
−0.2
0.0
0.2
0.4
0.6

The variable `port` contains the port in `localhost` serving the GUI, which `8050` by default, but may be different if that port is in use. Accessing `localhost:[port]` opens a window that is initially pretty empty:



To get started, press the "Compute" button under "Component Counting Function". After the computation is finished, you should see a plot of the component counting function:



To zoom in or out, adjust the parameters in the "Inputs" box.

The component counting function can give us an idea of where the interesting cluster structure is, but just from looking at this plot, it's not usually clear which slice we want to choose.

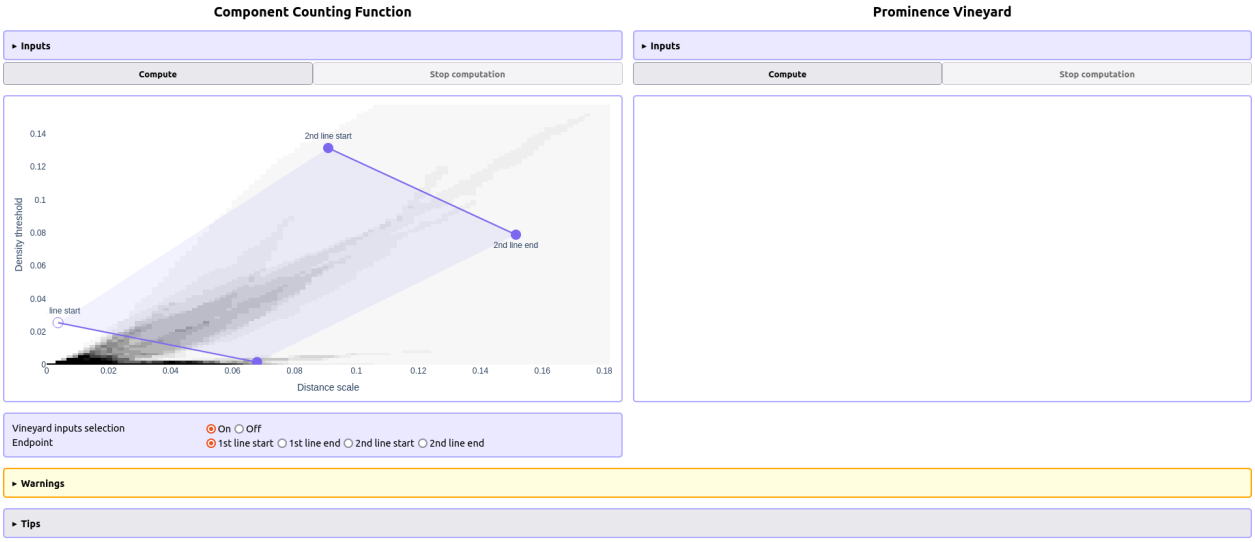
The next step in Persistable's interactive mode is the **Prominence Vineyard**. To use this tool, choose *two* slices:

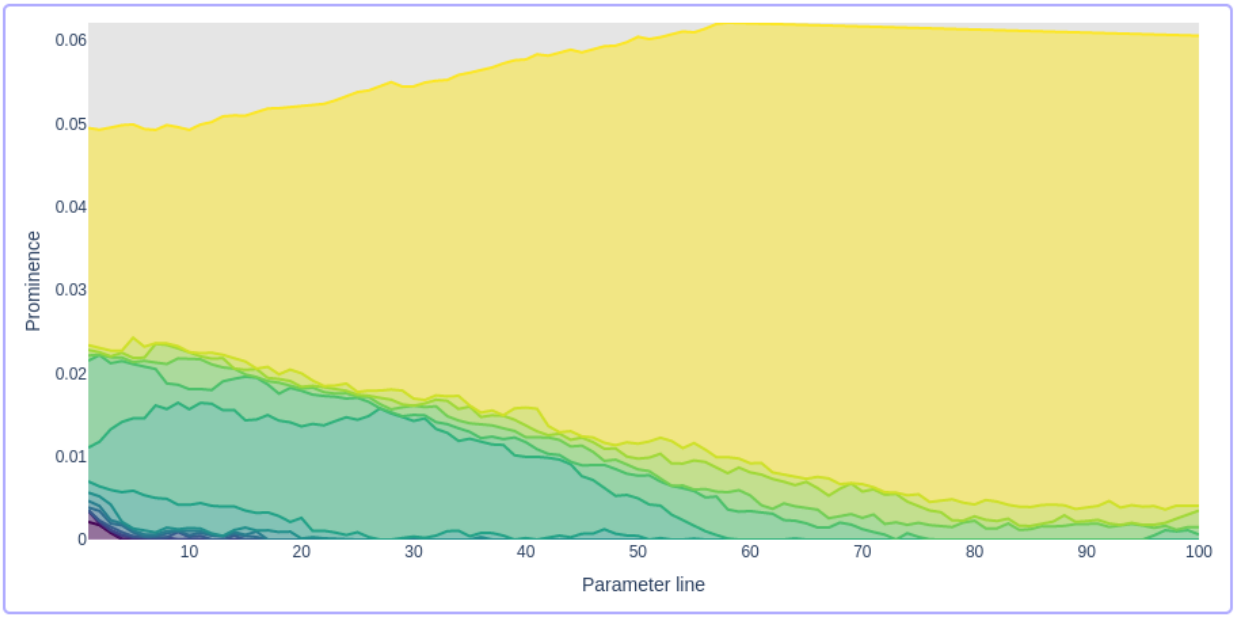
To do this, turn the "Vineyard inputs selection" on, and choose the slices by choosing their endpoints. You can choose the endpoints by clicking on the component counting function plot, or by entering the coordinates of the endpoints in the "Inputs" box under "Prominence Vineyard".

It's often a good strategy to choose the first slice in a region with many clusters, and the second slice in a region with few clusters, as above.

Now press "Compute" under "Prominence Vineyard". After the computation is complete, the vineyard is displayed:

What is the meaning of this plot?


Component Counting Function
Prominence Vineyard
▸ Inputs
Compute
Stop computation
2nd line start
2nd line end
line start
Density threshold
Distance scale
Vineyard inputs selection
On
Off
Endpoint
1st line start
1st line end
2nd line start
2nd line end
▸ Warnings
▸ Tips


Prominence
Parameter line

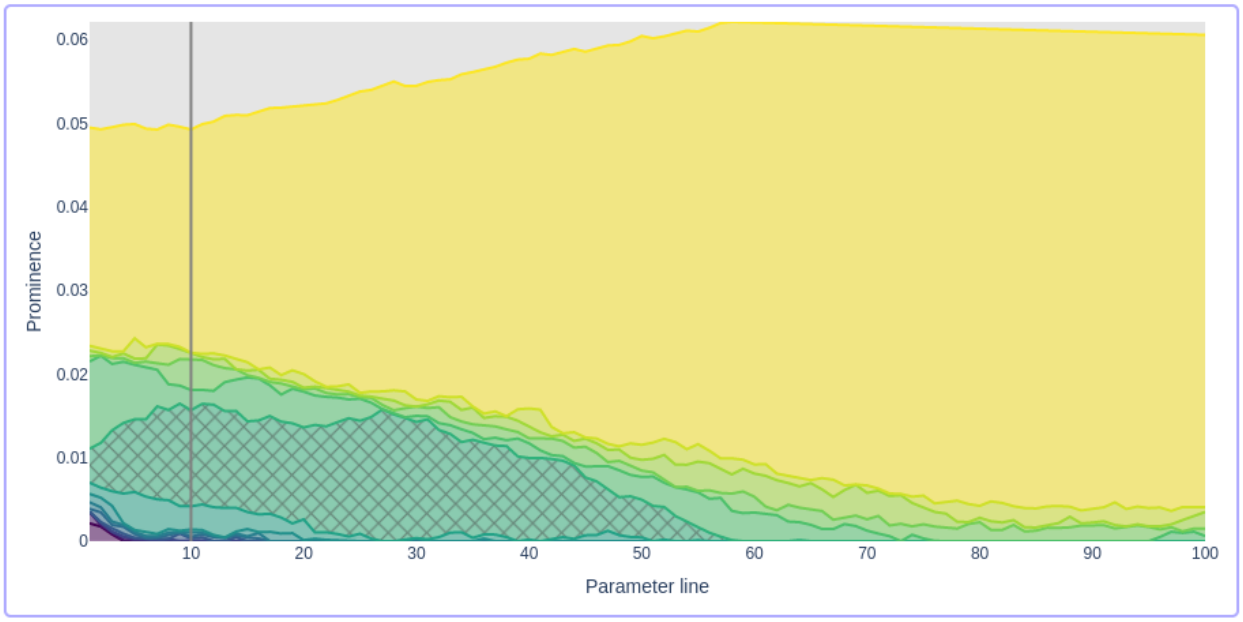
First, recall that the barcode of a hierarchical clustering gives us a coarse summary of its structure. We can get an even coarser summary by just remembering the length of each bar, which is also called the *prominence* of the bar. This list of prominences tells us how many clusters are in the hierarchy, and how long each one persists in the hierarchy.

Now, the Prominence Vineyard is constructed by taking a family of slices that interpolate between the two user-selected slices, and for each interpolating slice, plotting the prominences of the bars in its barcode. These prominences trace out curves that we call *vines*. The result is a visual summary of how the hierarchical clustering we get changes as we change the slice.

To get a clustering of the data, all we have to do now is choose a slice and choose how many bars (i.e., clusters) to keep. Recall that we are looking to draw a line between long bars and short bars. In the Prominence Vineyard plot, this corresponds to a gap between large and small prominences.

To choose a slice and a gap, turn on the "Parameter selection" option under "Prominence Vineyard".

In this example, we see a huge gap between the largest and second-largest vines. This is typical, since the most prominent bar in a hierarchical clustering is typically much longer than all the others. After this, the gap between the 6th and 7th vines stands out:



This is particularly noticeable if we display prominences on log scale, using the option in the "Inputs" box under "Prominence Vineyard":

Setting the "Gap number" to 6 and choosing a slice (i.e., a vertical line in the Prominence Vineyard plot) where this gap is large, we've made all the choices we need to get a clustering. So, press the "Choose parameter" button.

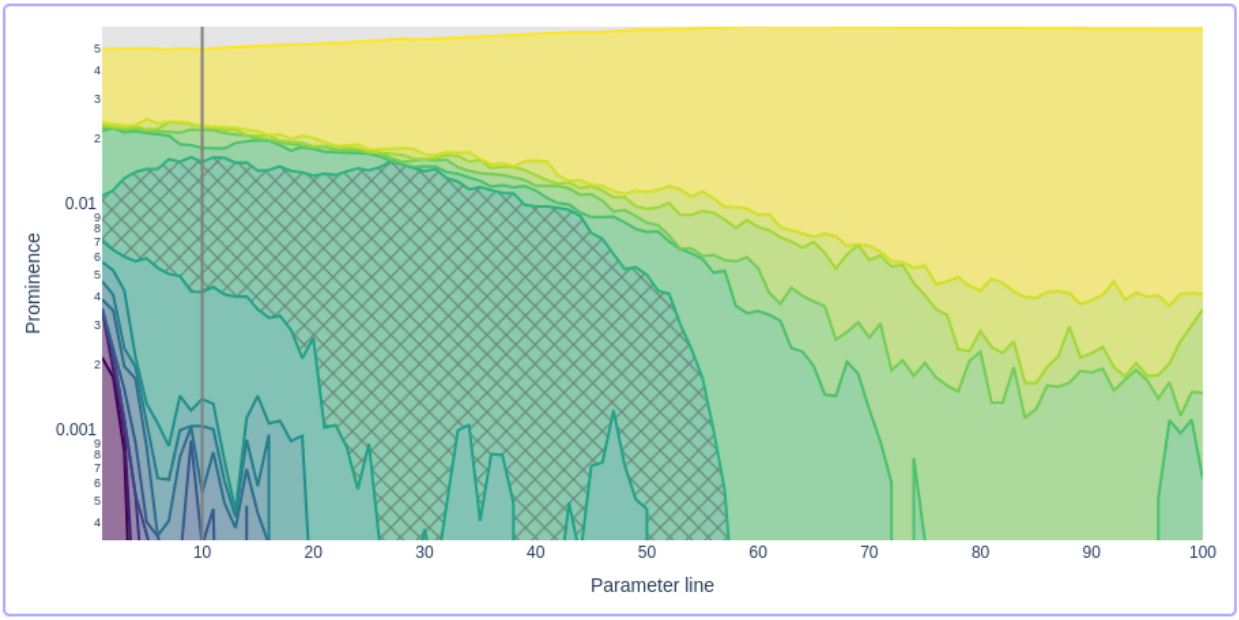
Now where's the clustering? We can get cluster labels for the data points from the PersistableInteractive object:

```
cluster_labels = pi.cluster()
```

Points labelled -1 are noise, and the clusters are labelled starting from 0. We can visualize the result by plotting the points with colors corresponding to the labels:

```
# choose color map
import matplotlib
cNorm  = matplotlib.colors.Normalize(vmin=0, vmax=np.amax(cluster_labels))
scalarMap = matplotlib.cm.ScalarMappable(norm=cNorm, cmap='Paired')
```

(continues on next page)



(continued from previous page)

```
noise_color = (211/255, 211/255, 211/255, 1) # light grey

# plot data with clusters indicated by colors
plt.figure(figsize=(10,10))
plt.scatter(data[:,0], data[:,1], c=[noise_color if x==-1 else scalarMap.to_rgba(x) for␣
↪x in cluster_labels], alpha=0.5)

plt.show()
```

1.3 A bit of theory

1.3.1 The barcode of a hierarchical clustering

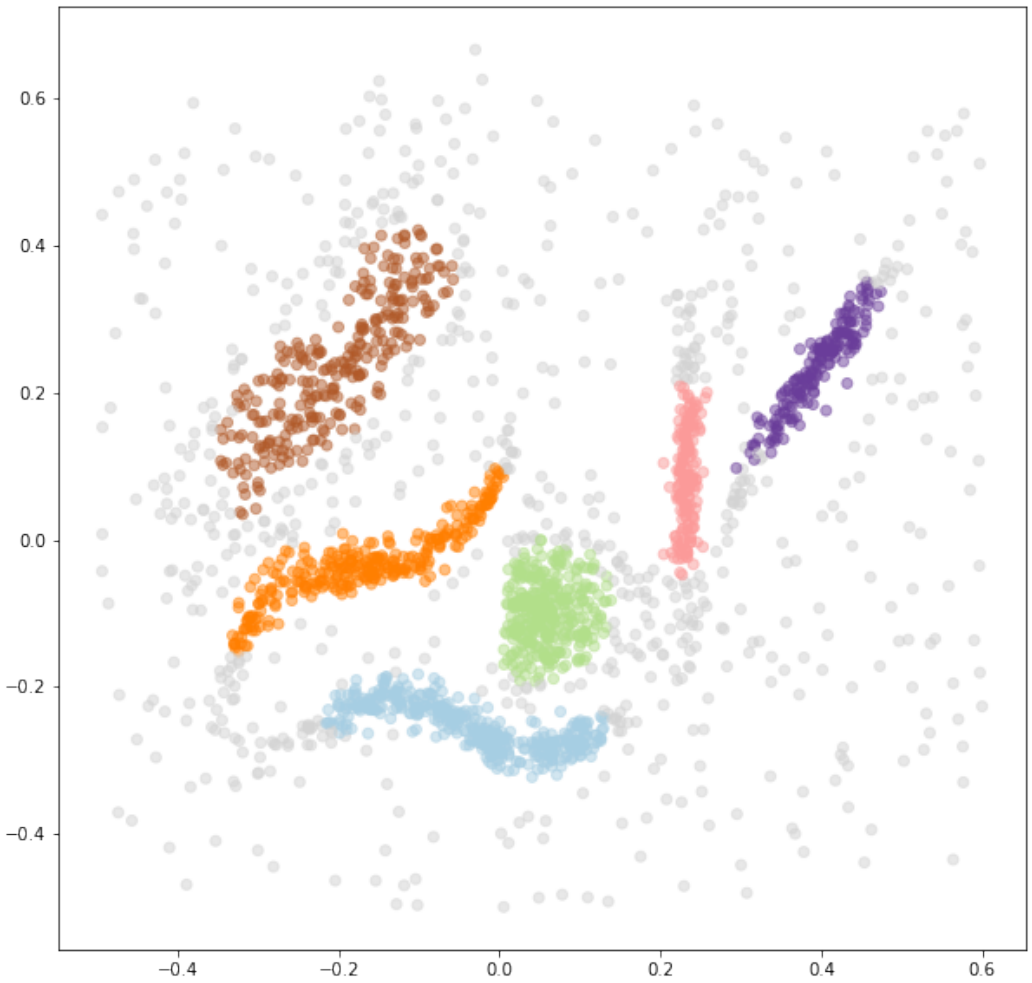
Barcodes are a visualization tool for data analysis that play an important role in persistent homology. This tutorial explains how barcodes provide a simple visualization of a hierarchical clustering.

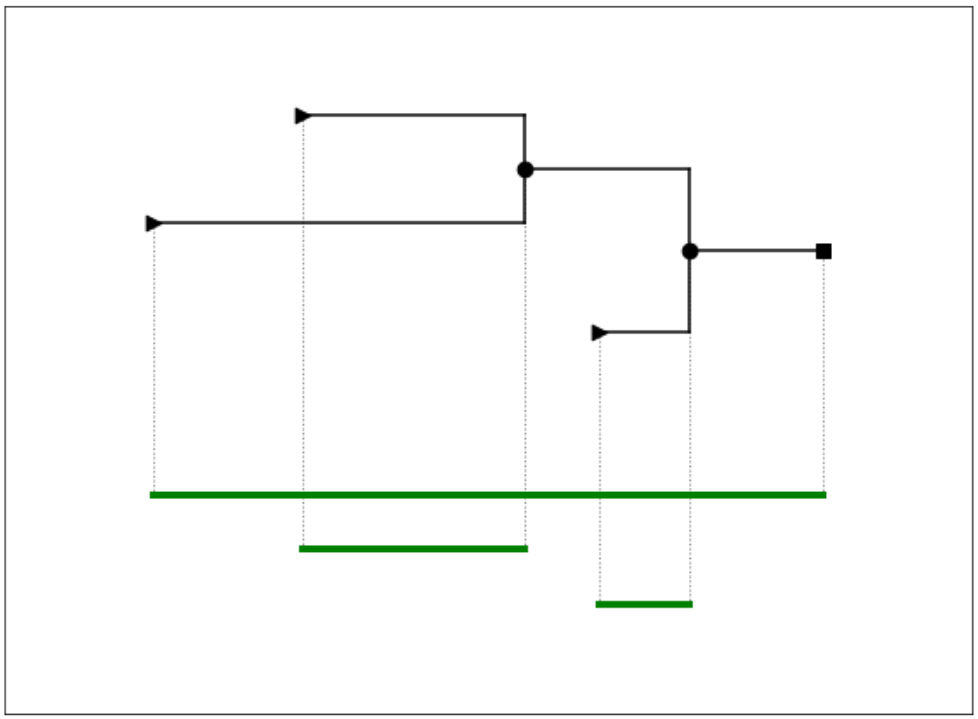
A (1-parameter) *hierarchical clustering* of a dataset X is a parameterized family of clusterings $H(X)$ of X. Above (in black) is a schematic picture of a hierarchical clustering.

For real values r in some interval of the real line, we have a clustering of the data, and as the parameter r grows, the clustering can change in the following ways:

1. A new cluster can be added to the hierarchy (the triangles in the diagram).
2. Two clusters can merge (the dots).
3. The hierarchical clustering can end (the square).
4. New data points can be added to a cluster (not shown in the diagram).

An example of such a hierarchical clustering is the output of the single-linkage algorithm. Persistable constructs hierarchical clusterings using graphs that encode both spatial relations among data points and density (see *Introduction to*


0.6
0.4
0.2
0.0
−0.2
−0.4
−0.4
−0.2
0.0
0.2
0.4
0.6



Persistable).

For a rigorous definition of hierarchical clustering, see Section 2 of Stable and consistent density-based clustering.

The barcode of a hierarchical clustering *H(X)* consists of *bars*, which are simply intervals of the real line. In the picture above, the barcode is displayed in green. It can be constructed as follows:

1. When a new cluster enters *H(X)* at parameter *r*, start a new bar with left endpoint *r*.
2. If two clusters merge at parameter *r*, take the cluster that entered the hierarchy at the larger parameter value, and end its bar, by setting the right endpoint to be *r*.
3. If the hierarchical clustering ends at parameter *r*, set the right endpoint of all bars that do not yet have a right endpoint to *r*.

Rule 2 is called the *Elder rule*, since the elder bar (that entered the hierarchy first), survives, while the younger bar dies.

For a rigorous description of this algorithm, see Section 5 of Stable and consistent density-based clustering (in particular, Algorithm 1).

The barcode of a hierarchical clustering forgets some information. In particular, when two clusters merge, the barcode does not record which two clusters merged. This forgetfulness makes the barcode quite simple, which makes it easy to read even for fairly complicated hierarchical clusterings. See *Clustering with Persistable* to see how we use the barcode of a hierarchical clustering to choose parameters for Persistable.

1.3.2 The persistence-based flattening algorithm

Given a hierarchical clustering of data, the persistence-based flattening algorithm extracts a single clustering of the data. We call this a "flattening" algorithm because it reduces a 1-parameter hierarchical clustering to a flat (0-parameter) clustering.

This tutorial provides a short description of the persistence-based flattening algorithm. For details, see Section 6 of Stable and consistent density-based clustering. This method is inspired by the ToMATo algorithm; see *Related tools for density-based clustering*.

The term "persistence" comes from the theory of persistent homology, and in particular, we will use the *barcode* of a hierarchical clustering to describe this algorithm:

See *The barcode of a hierarchical clustering* for an introduction to this notion.

A simple flattening algorithm is to just take the *leaves* of the hierarchical clustering (the blue dashed edges):

Recall that bars in the barcode are created whenever a new cluster enters the hierarchy, so the bars are in one-to-one correspondence with the leaves of the hierarchical clustering.

To be more explicit, let's add some data points to this picture:

The picture represents a hierarchical clustering of the set *{a,b,c,d,e}*. Point *a* enters the hierarchy at the parameter $r = 0$, and point *b* enters at $r=1$. At $r=1.5$, point *c* enters the hierarchy, and is immediately clustered with *a*. The clusters *{a,c}* and *{b}* merge at $r=2.5$, and so on.

The clustering of the data we get by taking the leaves has three clusters: *{a,c}*, *{b}*, and *{d}*. The point *e* is not assigned to any cluster, since it never lived inside a leaf.

A nice property of the leaf clustering is that it can identify cluster structure happening at different scales. In this example, once the point *d* has entered the hierarchy, points *a*, *b*, and *c* are all clustered together. So, we can't get the leaf clustering just by fixing a value of *r* and taking the hierarchical clustering at that value.

However, the leaf clustering can give poor results if the hierarchy has many spurious leaves, as in this example:

The persistence-based flattening attempts to avoid this problem by first *pruning* the hierarchy, and then taking the leaf clustering. For this we use the barcode of the hierarchical clustering as a visual guide.

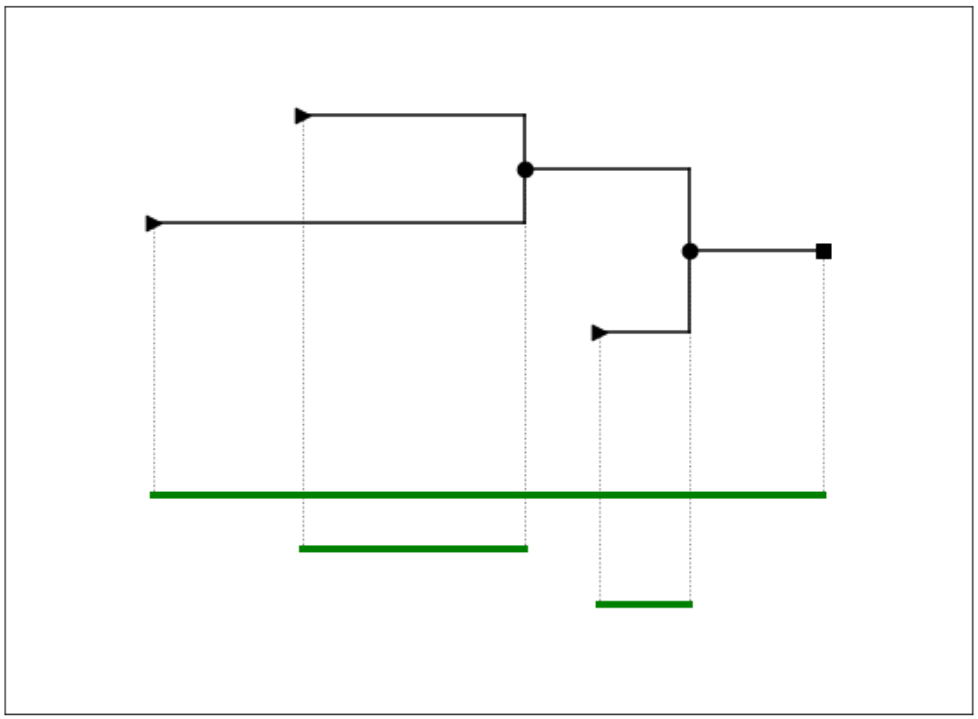
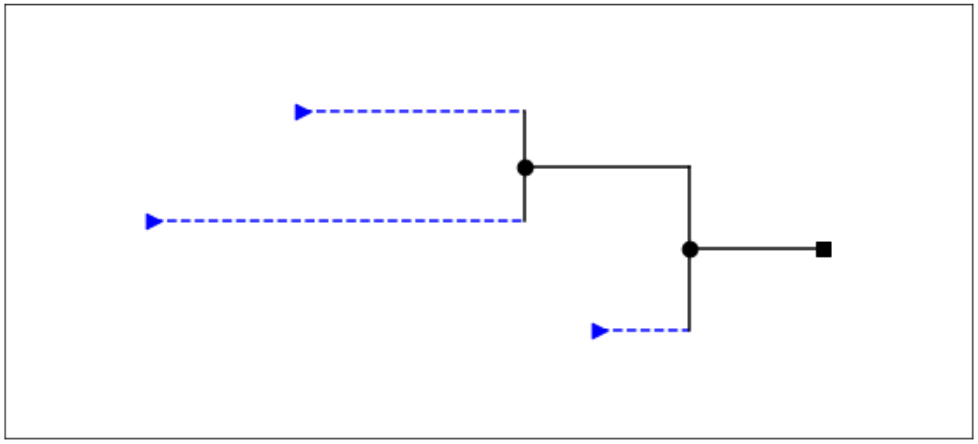
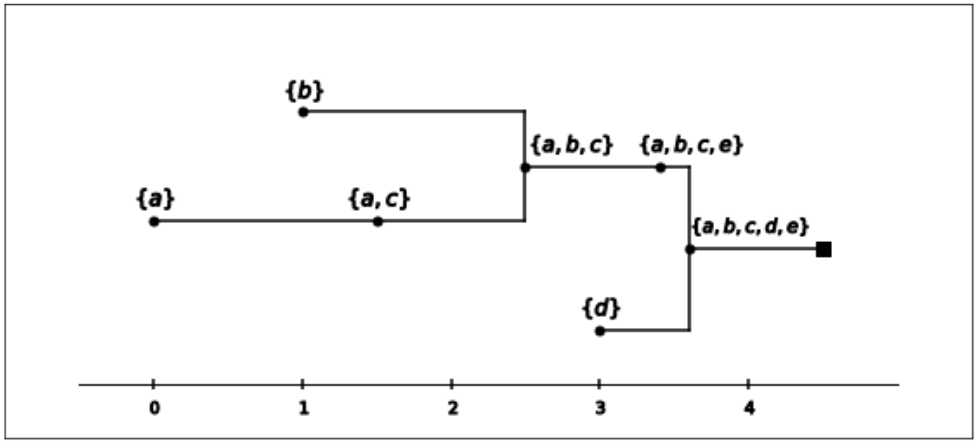
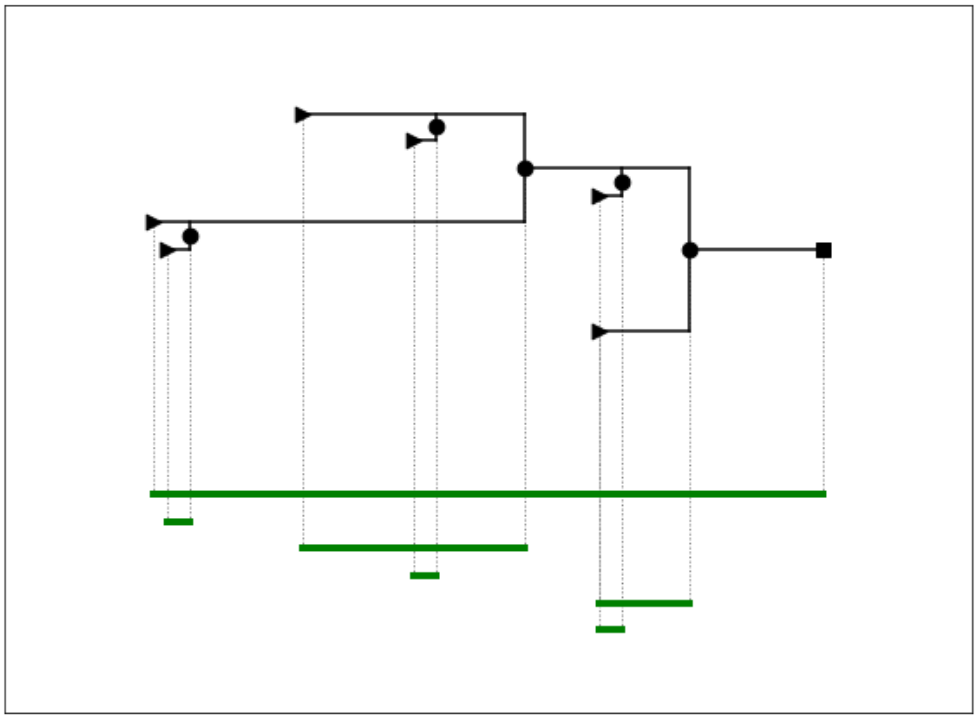


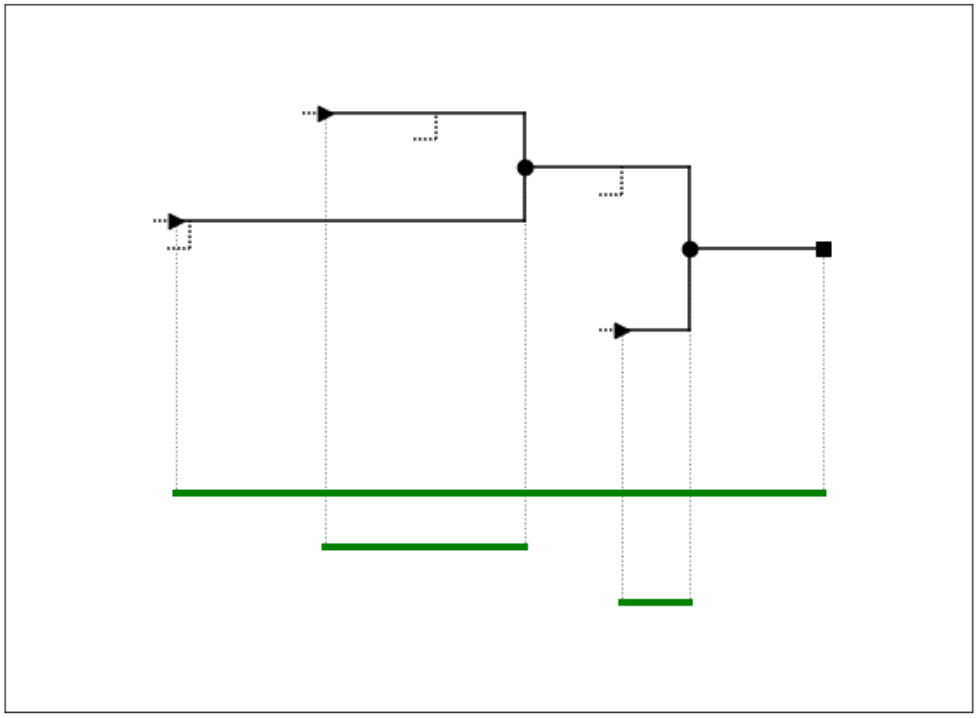
Fig. 3: A schematic picture of a hierarchical clustering (in black), and its barcode (in green).




{b}
{a, b, c}
{a, b, c, e}
{a}
{a, c}
{a, b, c, d, e}
{d}
0
1
2
3
4



In the example above, we might guess that the three very short bars correspond to spurious leaves. So, we can choose a value t that is greater than the length of the three short bars, and prune the hierarchy by shortening each leaf by t:



To see what happens to the data points inside the leaves, consider data points a and b living inside different leaves in the original hierarchical clustering:

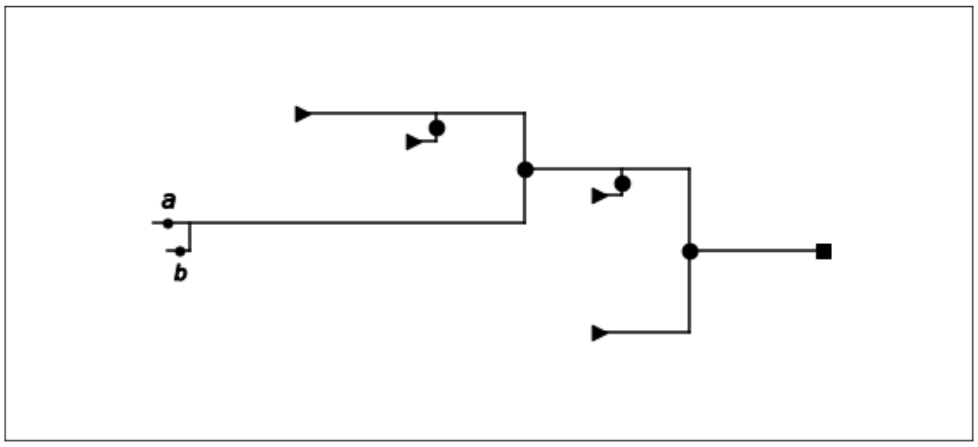
After pruning, these points are pushed rightwards along the hierarchy, and live inside the same leaf:

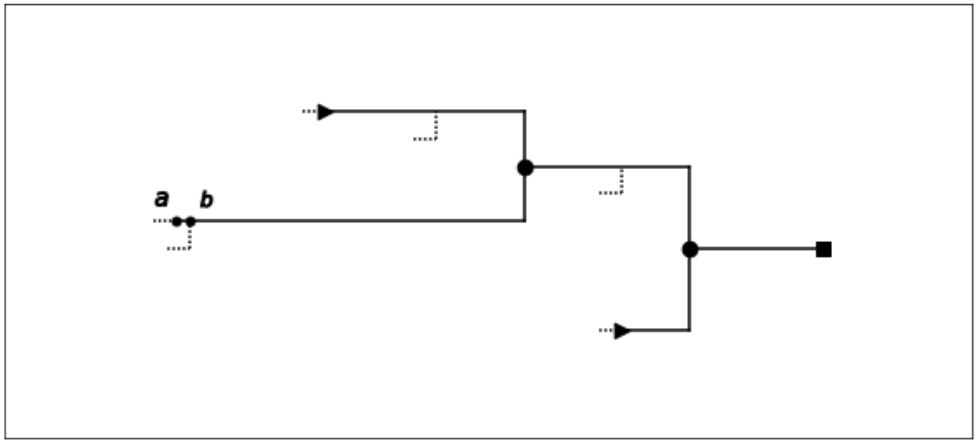
Notice that, when we prune before taking the leaf clustering, we're not just removing low-persistence clusters from the end result. Since these low-persistence leaves can "interrupt" high-persistence leaves, pruning allows larger clusters to form.

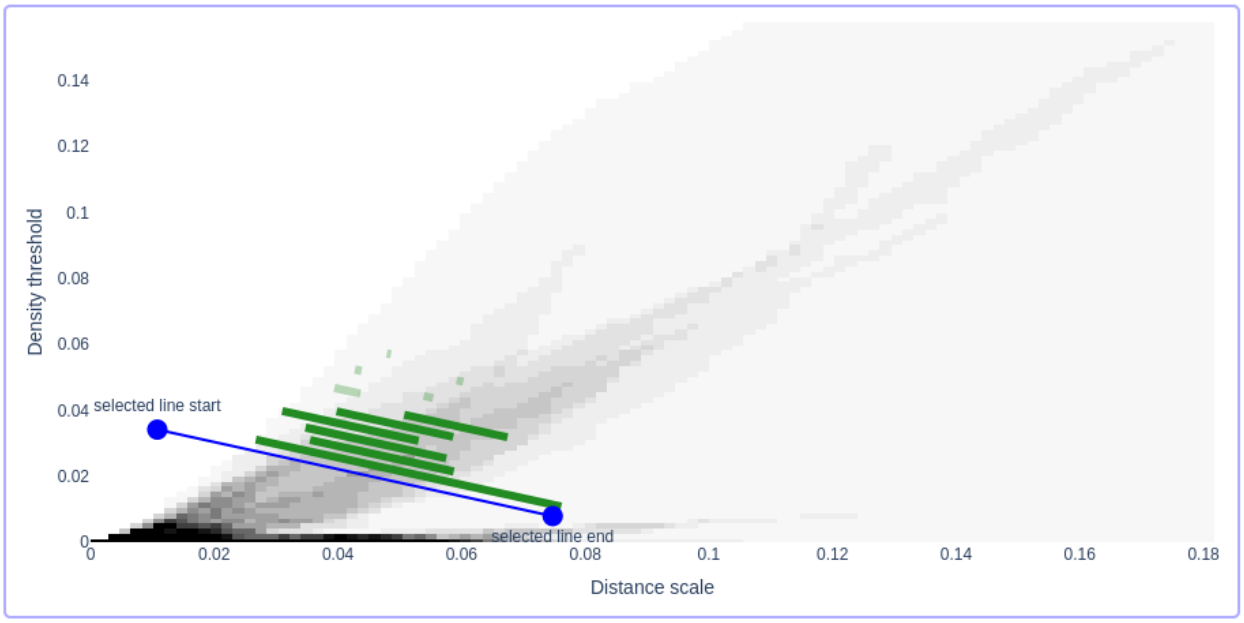
As an example, consider the dataset from *Introduction to Persistable*. There, we considered a hierarchical clustering obtained by taking a slice of the DBSCAN graphs:

If we choose to keep 7 bars, the persistence-based flattening produces the following result:

If we choose to keep only 6 bars, the two clusters at the bottom are able to merge:


a
b


a
b



1.4 Related tools for density-based clustering

The theory behind Persistable was developed in the paper Stable and consistent density-based clustering. The introduction to the paper describes the previous work by many researchers that we are building on.

This page provides links to some clustering tools that users of Persistable may also be interested in.

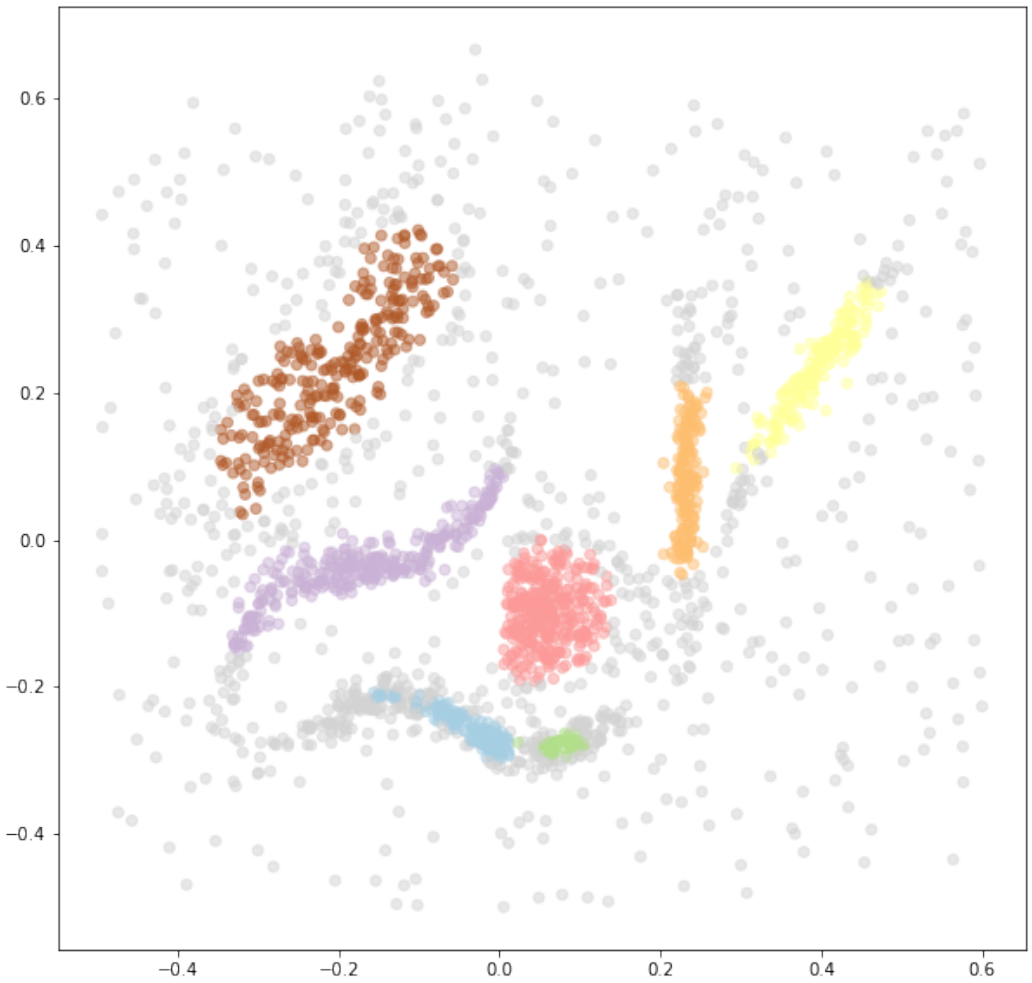
- hdbscan. A high performance implementation of HDBSCAN clustering. The hierarchical clustering method used by HDBSCAN is closely related to the hierarchies used by Persistable, and this implementation of Persistable uses the high-performance algorithms for hierarchical clustering implemented here.
- RIVET. A tool for 2-parameter persistent homology. In particular, this software implements tools for visualizing the *degree-Rips* persistent homology of a finite metric space. This is closely related to the visualization of the component counting function provided by Persistable, though we use different computational methods. The visualization tools provided by RIVET provide an even more fine-grained view of the DBSCAN graphs.
- ToMATo, implemented in the GUDHI library. ToMATo is another approach to density-based clustering. This was the first algorithm to use persistence to choose the number of clusters.

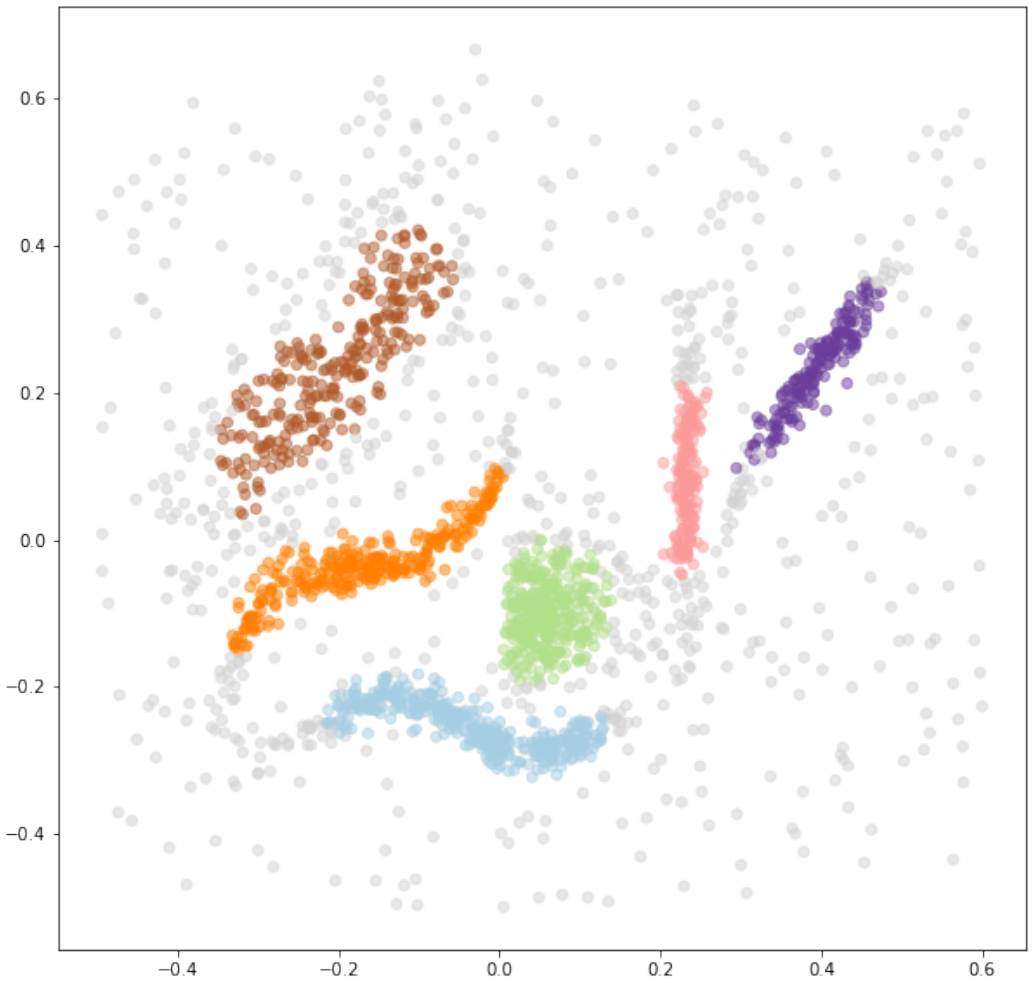
1.5 API Reference

1.5.1 Index

Classes

persistable.Persistable	Density-based clustering on finite metric spaces.
persistable.PersistableInteractive	Graphical user interface for doing parameter selection for `Persistable`.


0.6
0.4
0.2
0.0
−0.2
−0.4
−0.4
−0.2
0.0
0.2
0.4
0.6


0.6
0.4
0.2
0.0
−0.2
−0.4
−0.4
−0.2
0.0
0.2
0.4
0.6

persistable.Persistable methods

persistable.Persistable.cluster	Cluster dataset passed at initialization.

persistable.PersistableInteractive methods

persistable.PersistableInteractive.start_ui	Serves the GUI with a given persistable instance.
persistable.PersistableInteractive.cluster	Clusters the dataset with which the Persistable instance that was initialized.
persistable.PersistableInteractive.save_ui_state	Save state of input fields in the UI as a Python object.

1.5.2 Details

class persistable.**Persistable**(*X*, *metric='minkowski'*, *measure=None*, *subsample=None*, *n_neighbors='auto'*, *debug=False*, *threading=False*, *n_jobs=4*, ***kwargs*)

Density-based clustering on finite metric spaces.

X: ndarray (n_samples, n_features)
A numpy vector of shape (samples, features) or a distance matrix.

metric: string, optional, default is "minkowski"
A string determining which metric is used to compute distances between the points in X. It can be a metric in `KDTree.valid_metrics` or `BallTree.valid_metrics` (which can be found by `from sklearn.neighbors import KDTree, BallTree`) or `"precomputed"` if X is a distance matrix.

measure: None or ndarray(n_samples), default is None
A numpy vector of length (samples) of non-negative numbers, which is intepreted as a measure on the data points. If None, the uniform measure where each point has weight 1/samples is used. If the measure does not sum to 1, it is normalized.

subsample: None or int, optional, default is None
Number of datapoints to subsample. The subsample is taken to have a measure that approximates the original measure on the full dataset as best as possible, in the Prokhorov sense. If metric is `minkowski` and the dimensionality is not too big, computing the sample takes time O(log(size_subsample) * size_data), otherwise it takes time O(size_subsample * size_data).

n_neighbors: int or string, optional, default is "auto"
Number of neighbors for each point in X used to initialize datastructures used for clustering. If set to `"all"` it will use the number of points in the dataset, if set to `"auto"` it will find a reasonable default.

debug: bool, optional, default is False
Whether to print debug messages.

threading: bool, optional, default is False
Whether to use python threads for parallel computation with `joblib`. If false, the backend `loky` is used. In this case, using threads is significantly slower because of the GIL, but the backend `loky` does not work well in some systems.

n_jobs: int, default is 1
Number of processes or threads to use to fit the data structures, for exaple to compute the nearest neighbors of all points in the dataset.

****kwargs:**
Passed to `KDTree` or `BallTree`.

`cluster`(*n_clusters*, *start*, *end*, *flattening_mode='conservative'*, *keep_low_persistence_clusters=False*)

Cluster dataset passed at initialization.

n_clusters: int
Integer determining how many clusters the final clustering must have. Note that the final clustering can have fewer clusters if the selected parameters do not allow for so many clusters.

start: (float, float)
Two-element list, tuple, or numpy array representing a point on the positive plane determining the start of the segment in the two-parameter hierarchical clustering used to do persistence-based clustering.

end: (float, float)
Two-element list, tuple, or numpy array representing a point on the positive plane determining the end of the segment in the two-parameter hierarchical clustering used to do persistence-based clustering.

flattening_mode: string, optional, default is "conservative"
If "exhaustive", flatten the hierarchical clustering using the approach of 'Persistence-Based Clustering in Riemannian Manifolds' Chazal, Guibas, Oudot, Skraba. If "conservative", use the more stable approach of 'Stable and consistent density-based clustering' Rolle, Scoccola. The conservative approach usually results in more unclustered points.

keep_low_persistence_clusters: bool, optional, default is False
Only has effect if `flattening_mode` is set to "exhaustive". Whether to keep clusters that are born below the persistence threshold associated to the selected n_clusters. If set to True, the number of clusters can be larger than the selected one.

returns:
A numpy array of length the number of points in the dataset containing integers from -1 to the number of clusters minus 1, representing the labels of the final clustering. The label -1 represents noise points, i.e., points deemed not to belong to any cluster by the algorithm.

`class` persistable.`PersistableInteractive`(*persistable*)

Graphical user interface for doing parameter selection for `Persistable`.

persistable: Persistable
Persistable instance with which to interact with the user interface.

`cluster`(*flattening_mode='conservative'*, *keep_low_persistence_clusters=False*)

Clusters the dataset with which the Persistable instance that was initialized.

flattening_mode: string, optional, default is "conservative"
If "exhaustive", flatten the hierarchical clustering using the approach of 'Persistence-Based Clustering in Riemannian Manifolds' Chazal, Guibas, Oudot, Skraba. If "conservative", use the more stable approach of 'Stable and consistent density-based clustering' Rolle, Scoccola. The conservative approach usually results in more unclustered points.

keep_low_persistence_clusters: bool, optional, default is False
Only has effect if `flattening_mode` is set to "exhaustive". Whether to keep clusters that are born below the persistence threshold associated to the selected n_clusters. If set to True, the number of clusters can be larger than the selected one.

returns:
A numpy array of length the number of points in the dataset containing integers from -1 to the number of clusters minus 1, representing the labels of the final clustering. The label -1 represents noise points, i.e., points deemed not to belong to any cluster by the algorithm.

`save_ui_state()`

Save state of input fields in the UI as a Python object. The output can then be used as the optional input of the `start_ui()` method.

returns: dictionary

`start_ui(`*ui_state=None*, *port=8050*, *debug=False*, *jupyter_mode='external'*`)`

Serves the GUI with a given persistable instance.

ui_state: dictionary, optional
The state of a previous UI session, as a Python object, obtained by calling the method `save_ui_state()`.

port: int, optional, default is 8050
Integer representing which port of localhost to try use to run the GUI. If port is not available, we look for one that is available, starting from the given one.

debug: bool, optional, default is False
Whether to run Dash in debug mode.

jupyter_mode: string, optional, default is "external"
How to display the application when running inside a jupyter notebook. Options are "external" to serve the app in a port returned by this function, "inline" to open the app inline in the jupyter notebook. "jupyterlab" to open the app in a separate tab in JupyterLab.

return: int
Returns the port of localhost used to serve the UI.

INDEX