hdbscan Documentation

Release 0.8

Leland McInnes, John Healy, Steve Astels

September 03, 2016

Contents

1 User Guide / Tutorial 3

1.1 Basic Usage of HDBSCAN* for Clustering 3

1.2 Getting More Information About a Clustering 5

1.3 Parameter Selection for HDBSCAN* 11

1.4 Outlier Detection 18

2 Background on Clustering with HDBSCAN 23

2.1 How HDBSCAN Works 23

2.2 Comparing Python Clustering Algorithms 30

2.3 Benchmarking Performance and Scaling of Python Clustering Algorithms 43

3 API Reference 55

3.1 API Reference 55

4 Indices and tables 57

The hdbscan library is a suite of tools to use unsupervised learning to find clusters, or dense regions, of a dataset. The primary algorithm is HDBSCAN* as proposed by Campello, Moulavi, and Sander. The library provides a high performance implementation of this algorithm, along with tools for analysing the resulting clustering.

CHAPTER 1

User Guide / Tutorial

1.1 Basic Usage of HDBSCAN* for Clustering

We have some data, and we want to cluster it. How exactly do we do that, and what do the results look like? If you are very familiar with sklearn and it's API, particularly for clustering, then you can probably skip this tutorial – `hdbscan` implements exactly this API, so you can use it just as you would any other sklearn clustering algorithm. If, on the other hand, you aren't that familiar with sklearn, fear not, and read on. Let's start with the simplest case first – we have data in a nice tidy dataframe format.

1.1.1 The Simple Case

Let's generate some data with, say 2000 samples, and 10 features. We can put it in a dataframe for a nice clean table view of it.

```
blobs, labels = make_blobs(n_samples=2000, n_features=10)
```

```
pd.DataFrame(blobs).head()
```

So now we need to import the hdbscan library.

```
import hdbscan
```

Now, to cluster we need to generate a clustering object.

```
clusterer = hdbscan.HDBSCAN()
```

We can then use this clustering object and fit it to the data we have. This will return the clusterer object back to you – just in case you want do some method chaining.

```
clusterer.fit(blobs)
```

```
HDBSCAN(algorithm='best', alpha=1.0, approx_min_span_tree=True,
    gen_min_span_tree=False, leaf_size=40, memory=Memory(cachedir=None),
    metric='euclidean', min_cluster_size=5, min_samples=None, p=None)
```

At this point we are actually done! We've done the clustering! But where are the results? How do I get the clusters? The clusterer object knows, and stores the result in an attribute `labels_`.

```
clusterer.labels_
```

```
array([2, 2, 2, ..., 2, 2, 0])
```

So it is an array of integers. What are we to make of that? It is an array with an integer for each data sample. Samples that are in the same cluster get assigned the same number. The cluster labels are 0 up numbers. We can thus determine the number of clusters found by checking what the largest cluster label is.

```
clusterer.labels_.max()
```

```
2
```

So we have a total of three clusters, with labels 0, 1, and 2. Importantly HDBSCAN is noise aware – it has a notion of data samples that are not assigned to any cluster. This is handled by assigning these samples the label -1. But wait, there's more. The `hdbscan` library implements soft clustering, where wach data point is assigned a cluster membership score ranging from 0.0 to 1.0. A score of 0.0 represents a sample that is not in the cluster at all (all noise points will get this score) while a score of 1.0 represents a sample that is at the heart of the cluster (note that this is not the spatial centroid notion of core). You can access these scores via the `probabilities_` attribute.

```
clusterer.probabilities_
```

```
array([ 0.83890858,  1.        ,  0.72629904, ...,  0.79456452,
        0.65311137,  0.76382928])
```

1.1.2 What about different metrics?

That is all well and good, but even data that is embedded in a vector space may not want to consider distances between data points to be pure Euclidean distance. What can we do in that case? We are still in good shape, since `hdbscan` supports a wide variety of metrics, which you can set when creating the clusterer object. For example we can do the following:

```
clusterer = hdbscan.HDBSCAN(metric='manhattan')
clusterer.fit(blobs)
clusterer.labels_
```

```
array([1, 1, 1, ..., 1, 1, 0])
```

What metrics are supported? Because we simply steal metric computations from sklearn we get a large number of metrics readily available.

```
hdbscan.dist_metrics.METRIC_MAPPING
```

```
{'braycurtis': hdbscan.dist_metrics.BrayCurtisDistance,
 'canberra': hdbscan.dist_metrics.CanberraDistance,
 'chebyshev': hdbscan.dist_metrics.ChebyshevDistance,
 'cityblock': hdbscan.dist_metrics.ManhattanDistance,
 'dice': hdbscan.dist_metrics.DiceDistance,
 'euclidean': hdbscan.dist_metrics.EuclideanDistance,
 'hamming': hdbscan.dist_metrics.HammingDistance,
 'haversine': hdbscan.dist_metrics.HaversineDistance,
 'infinity': hdbscan.dist_metrics.ChebyshevDistance,
 'jaccard': hdbscan.dist_metrics.JaccardDistance,
 'kulsinski': hdbscan.dist_metrics.KulsinskiDistance,
 'l1': hdbscan.dist_metrics.ManhattanDistance,
 'l2': hdbscan.dist_metrics.EuclideanDistance,
 'mahalanobis': hdbscan.dist_metrics.MahalanobisDistance,
 'manhattan': hdbscan.dist_metrics.ManhattanDistance,
 'matching': hdbscan.dist_metrics.MatchingDistance,
 'minkowski': hdbscan.dist_metrics.MinkowskiDistance,
 'p': hdbscan.dist_metrics.MinkowskiDistance,
 'pyfunc': hdbscan.dist_metrics.PyFuncDistance,
 'rogerstanimoto': hdbscan.dist_metrics.RogersTanimotoDistance,
 'russellrao': hdbscan.dist_metrics.RussellRaoDistance,
 'seuclidean': hdbscan.dist_metrics.SEuclideanDistance,
 'sokalmichener': hdbscan.dist_metrics.SokalMichenerDistance,
 'sokalsneath': hdbscan.dist_metrics.SokalSneathDistance,
 'wminkowski': hdbscan.dist_metrics.WMinkowskiDistance}
```

1.1.3 Distance matrices

What if you don’t have a nice set of points in a vector space, but only have a pairwise distance matrix providing the distance between each pair of points? This is a common situation. Perhaps you have a complex custom distance measure; perhaps you have strings and are using Levenstein distance, etc. Again, this is all fine as `hdbscan` supports a special metric called `precomputed`. If you create the clusterer with the metric set to `precomputed` then the clusterer will assume that, rather than being handed a vector of points in a vector space, it is recieving an all pairs distance matrix.

```
distance_matrix = pairwise_distances(blobs)
clusterer = hdbscan.HDBSCAN(metric='precomputed')
clusterer.fit(distance_matrix)
clusterer.labels_
```

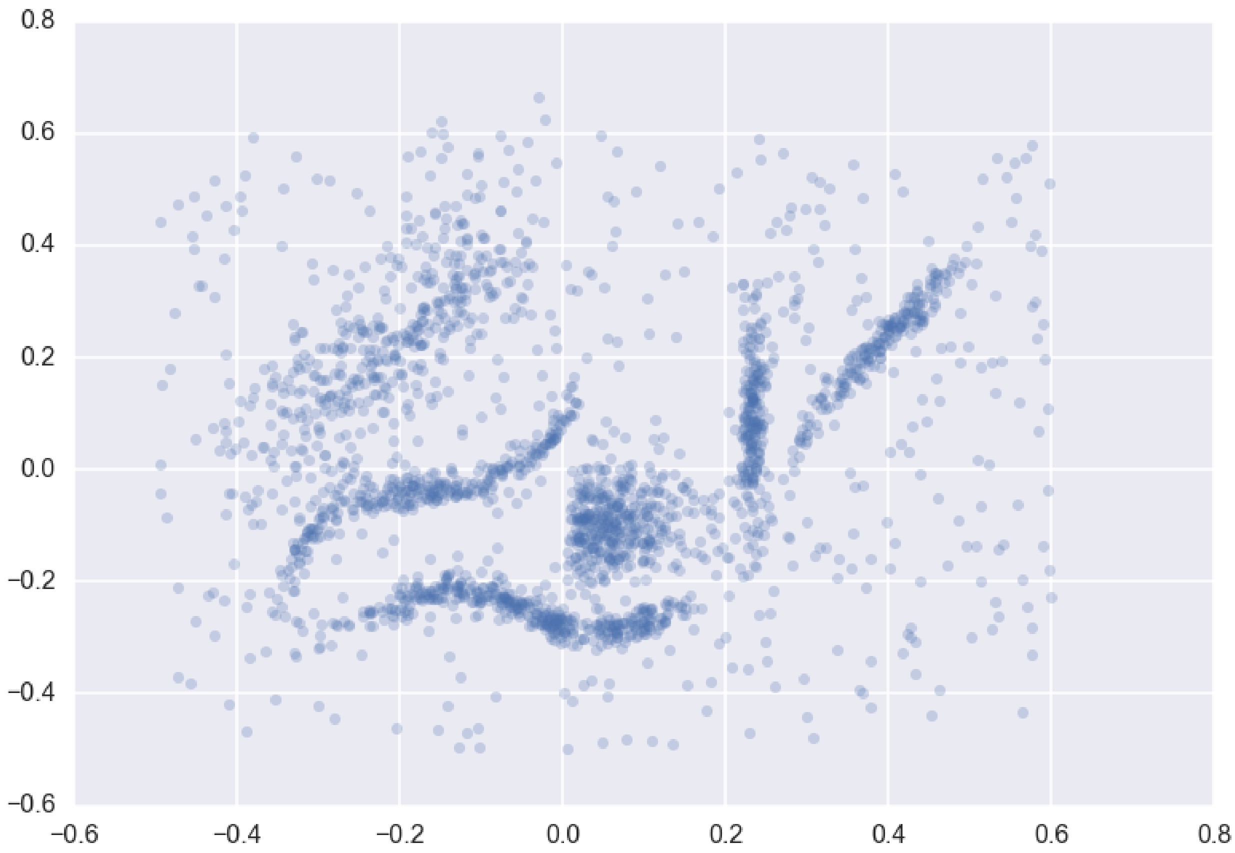
```
array([1, 1, 1, ..., 1, 1, 2])
```

Note that this result only appears different due to a different labelling order for the clusters.

1.2 Getting More Information About a Clustering

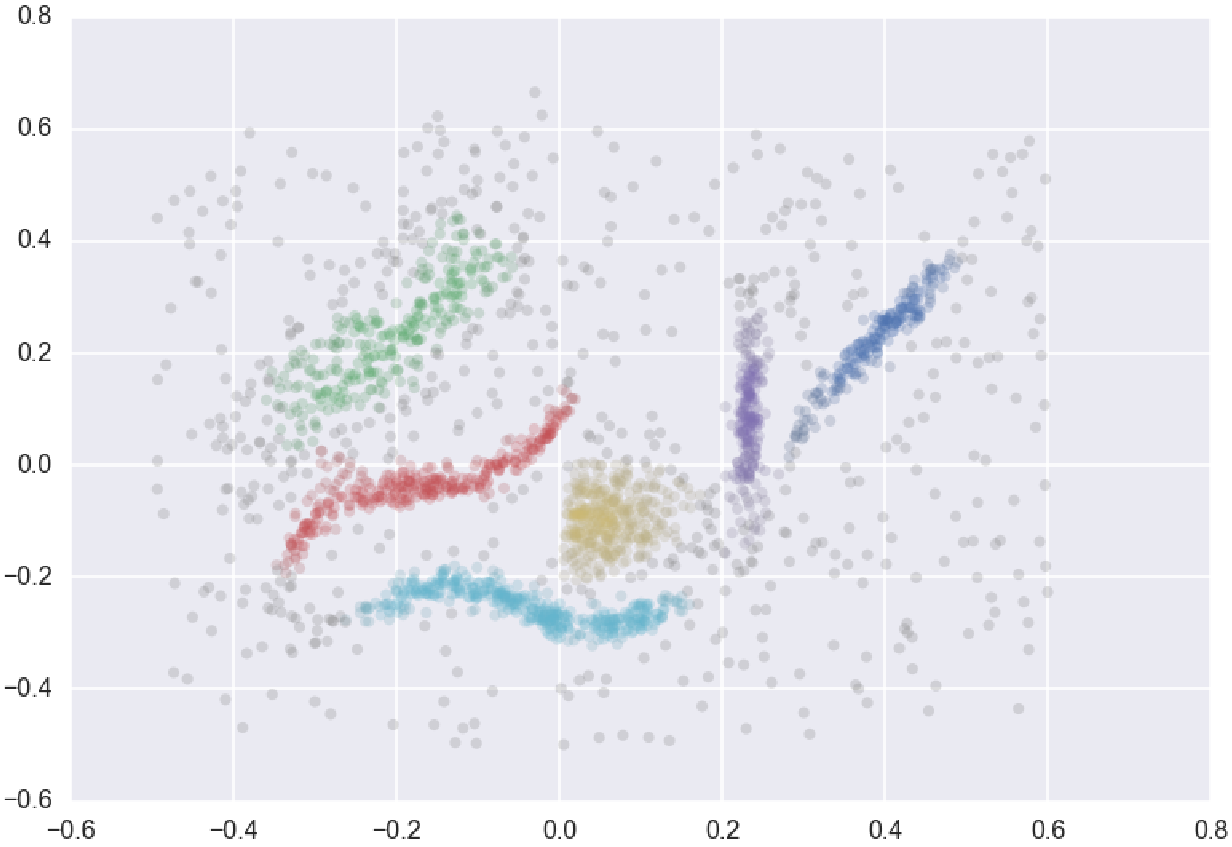
Once you have the basics of clustering sorted you may want to dig a little deeper than just the cluster labels returned to you. Fortunately the hdbscan library provides you with the facilities to do this. During processing HDBSCAN* builds a hierarchy of potential clusters, from which is extracts the flat clustering returned. It can be informative to look at that hierarchy, and potentially make use of the extra information contained therein.

Suppose we have a dataset for clustering



We can cluster the data as normal, and visualize the labels with different colors (and even the cluster membership strengths as levels of saturation)

```
clusterer = hdbscan.HDBSCAN(min_cluster_size=15).fit(data)
color_palette = sns.color_palette('deep', 8)
cluster_colors = [color_palette[x] if x >= 0
                  else (0.5, 0.5, 0.5)
                  for x in clusterer.labels_]
cluster_member_colors = [sns.desaturate(x, p) for x, p in
                         zip(cluster_colors, clusterer.probabilities_)]
plt.scatter(*data.T, s=50, linewidth=0, c=cluster_member_colors, alpha=0.25)
```



1.2.1 Condensed Trees

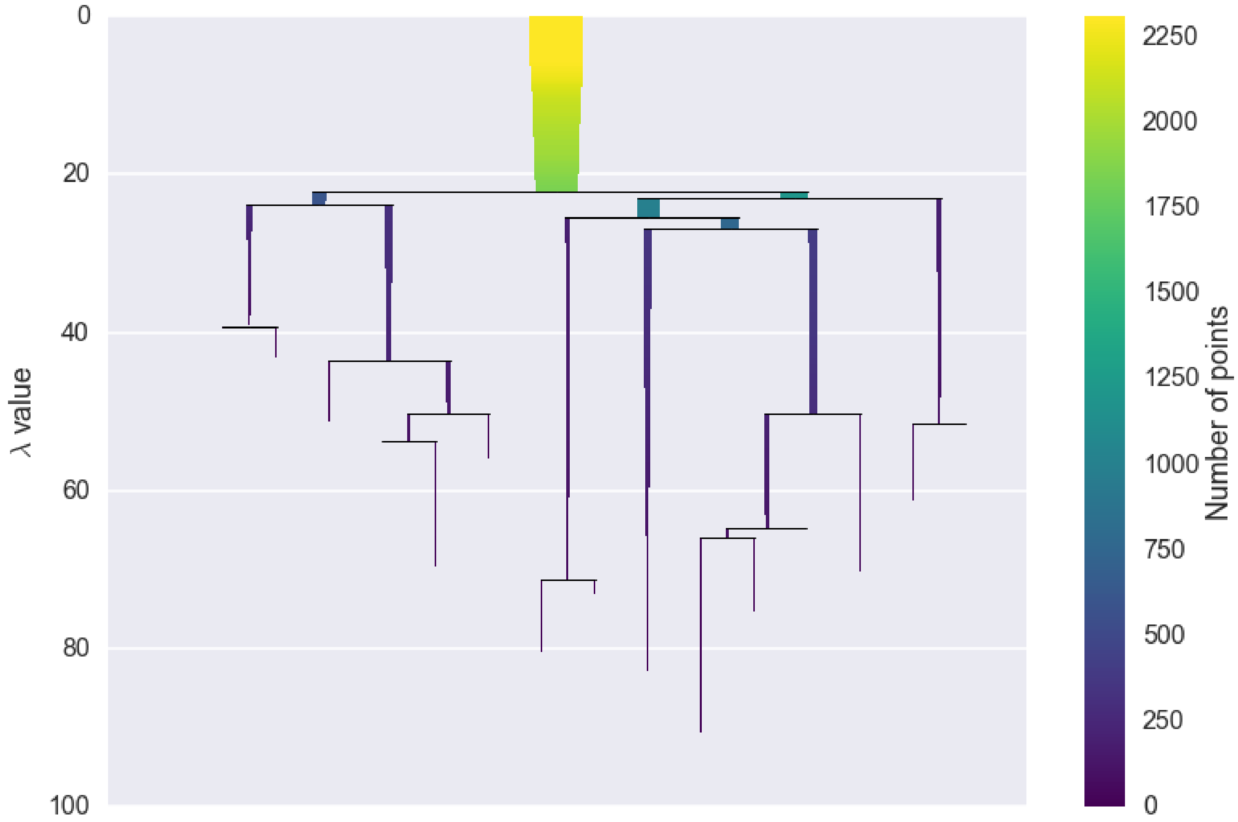
The question now is what does the cluster hierarchy look like – which clusters are near each other, or could perhaps be merged, and which are far apart. We can access the basic hierarchy via the `condensed_tree_` attribute of the clusterer object.

```
clusterer.condensed_tree_
```

```
<hdbscan.plots.CondensedTree at 0x10ea23a20>
```

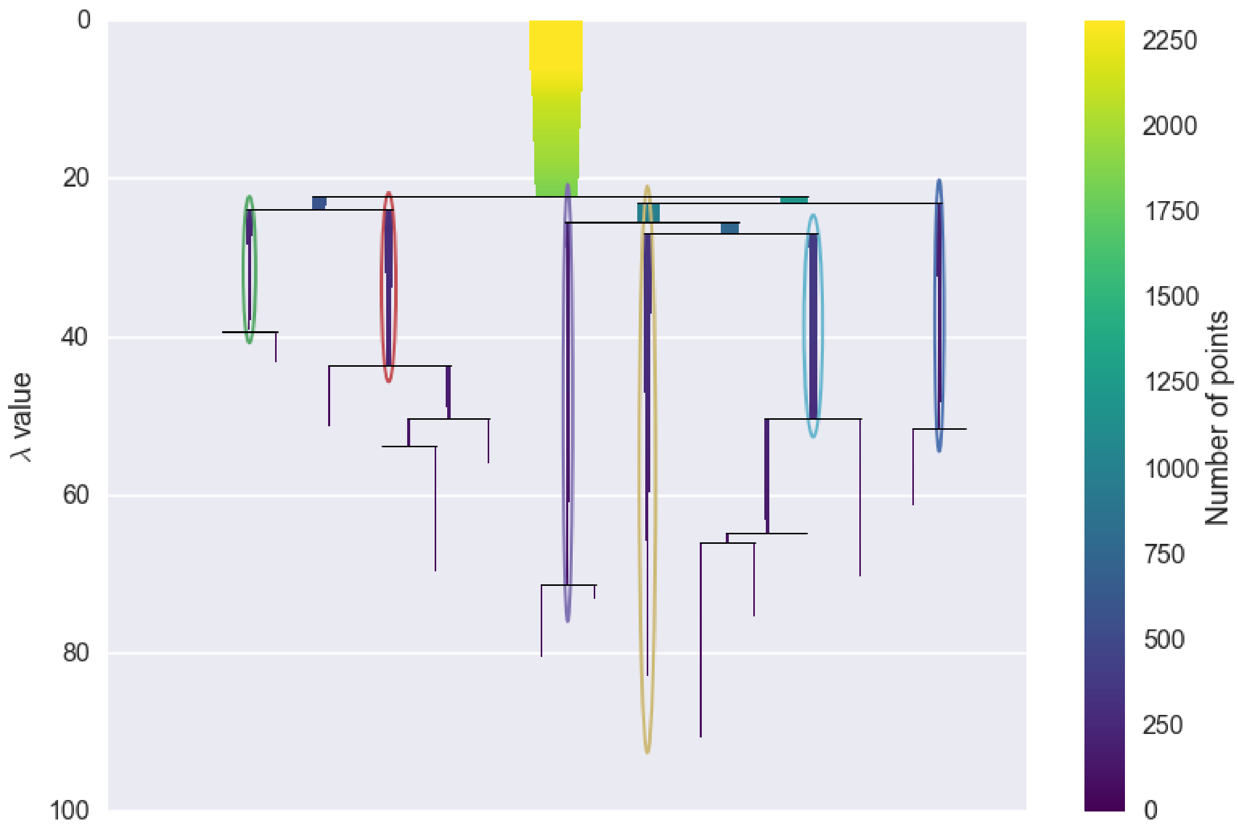
This merely gives us a CondensedTree object. If we want to visualize the hierarchy we can call the `plot` method:

```
clusterer.condensed_tree_.plot()
```



We can now see the hierarchy as a dendrogram, the width (and color) or each branch representing the number of points in the cluster at that level. If we wish to know which branches were selected by the HDBSCAN* algorithm we can pass `select_clusters=True`. You can even pass a selection palette to color the selections according to the cluster labelling.

```
clusterer.condensed_tree_.plot(select_clusters=True,
                               selection_palette=sns.color_palette('deep', 8))
```



From this we can see, for example, that the yellow cluster, at the center of the plot, forms early (breaking off from the pale blue and purple clusters) and persists for a long time. By comparison the green cluster, which also forming early, quickly breaks apart and then vanishes altogether (shattering into clusters all smaller than the `min_cluster_size` of 15).

You can also see that the pale blue cluster breaks apart into several subclusters that in turn persist for quite some time – so there is some interesting substructure to the pale blue cluster that is not present, for example, in the dark blue cluster.

In this was a simple visual analysis of the condensed tree can tell you a lot more about the structure of your data. This is not all we can do with condensed trees however. For larger and more complex datasets the tree itself may be very complex, and it may be desireable to run more interesting analytics over the tree itself. This can be achieved via several converter methods: `to_networkx`, `to_pandas`, and `to_numpy`.

First we'll consider `to_networkx`

```
clusterer.condensed_tree_.to_networkx()
```

```
<networkx.classes.digraph.DiGraph at 0x11d8023c8>
```

As you can see we get a networkx directed graph, which we can then use all the regular networkx tools and analytics on. The graph is richer than the visual plot above may lead you to believe however:

```
g = clusterer.condensed_tree_.to_networkx()
g.number_of_nodes()
```

```
2338
```

The graph actually contains nodes for all the points falling out of clusters as well as the clusters themselves. Each node has an associated `size` attribute, and each edge has a `weight` of the lambda value at which that edge forms.

This allows for much more interesting analyses.

Next we have the `to_pandas` method, which returns a panda dataframe where each row corresponds to an edge of the networkx graph:

```
clusterer.condensed_tree_.to_pandas().head()
```

Here the `parent` denotes the id of the parent cluster, the `child` the id of the child cluster (or, if the child is a single data point rather than a cluster, the index in the dataset of that point), the `lambda_val` provides the lambda value at which the edge forms, and the `child_size` provides the number of points in the child cluster. As you can see the start of the dataframe has singleton points falling out of the root cluster, with each `child_size` equal to 1.

Finally we have the `to_numpy` function, which returns a numpy record array:

```
clusterer.condensed_tree_.to_numpy()
```

```
array([(2309, 2048, 5.016525967983049, 1),
       (2309, 2006, 5.076503128308643, 1),
       (2309, 2024, 5.279133057912248, 1), ...,
       (2318, 1105, 86.5507370650292, 1), (2318, 965, 86.5507370650292, 1),
       (2318, 954, 86.5507370650292, 1)],
      dtype=[('parent', '<i8'), ('child', '<i8'), ('lambda_val', '<f8'), ('child_size', '<i8')])
```

This is equivalent to the pandas dataframe, but is in pure numpy and hence has no pandas dependencies if you do not wish to use pandas.

1.2.2 Single Linkage Trees

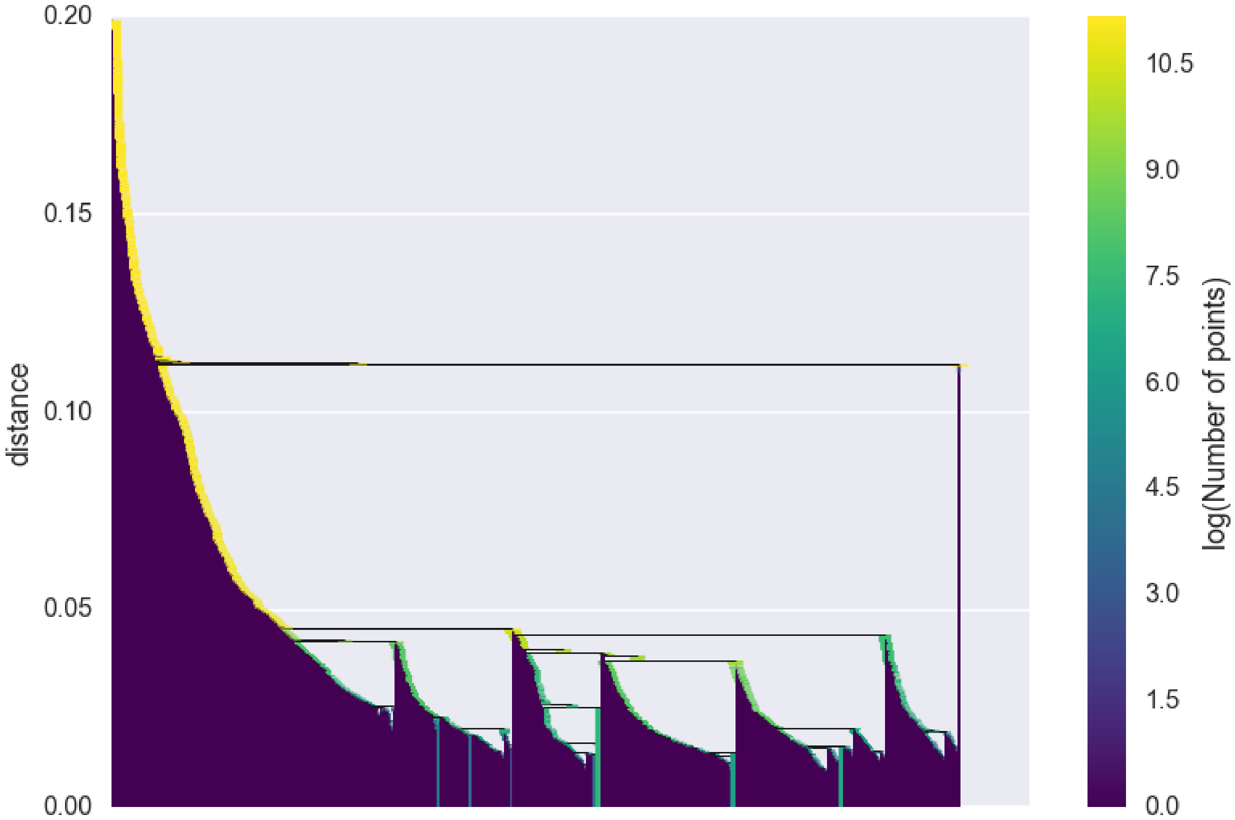
We have still more data at our disposal however. As noted in the How HDBSCAN Works section, prior to providing a condensed tree the algorithm builds a complete dendrogram. We have access to this too via the `single_linkage_tree` attribute of the clusterer.

```
clusterer.single_linkage_tree_
```

```
<hdbscan.plots.SingleLinkageTree at 0x121d4b128>
```

Again we have an object which we can then query for relevant information. The most basic approach is the `plot` method, just like the condensed tree.

```
clusterer.single_linkage_tree_.plot()
```



As you can see we gain a lot from condensing the tree in terms of better presenting and summarising the data. There is a lot less to be gained from visual inspection of a plot like this (and it only gets worse for larger datasets). The plot function support most of the same fucntionality as the dendrogram plotting from `scipy.cluster.hierarchy`, so you can view various truncations of th tree if necessary. In practice, however, you are more likely to be interested in access the raw data for further analysis. Again we have `to_networkx`, `to_pandas` and `to_numpy`. This time the `to_networkx` provides a direct networkx version of what you see above. The numpy and pandas results conform to the single linkage hierarchy format of `scipy.cluster.hierarchy`, and can be passed to routines there if necessary.

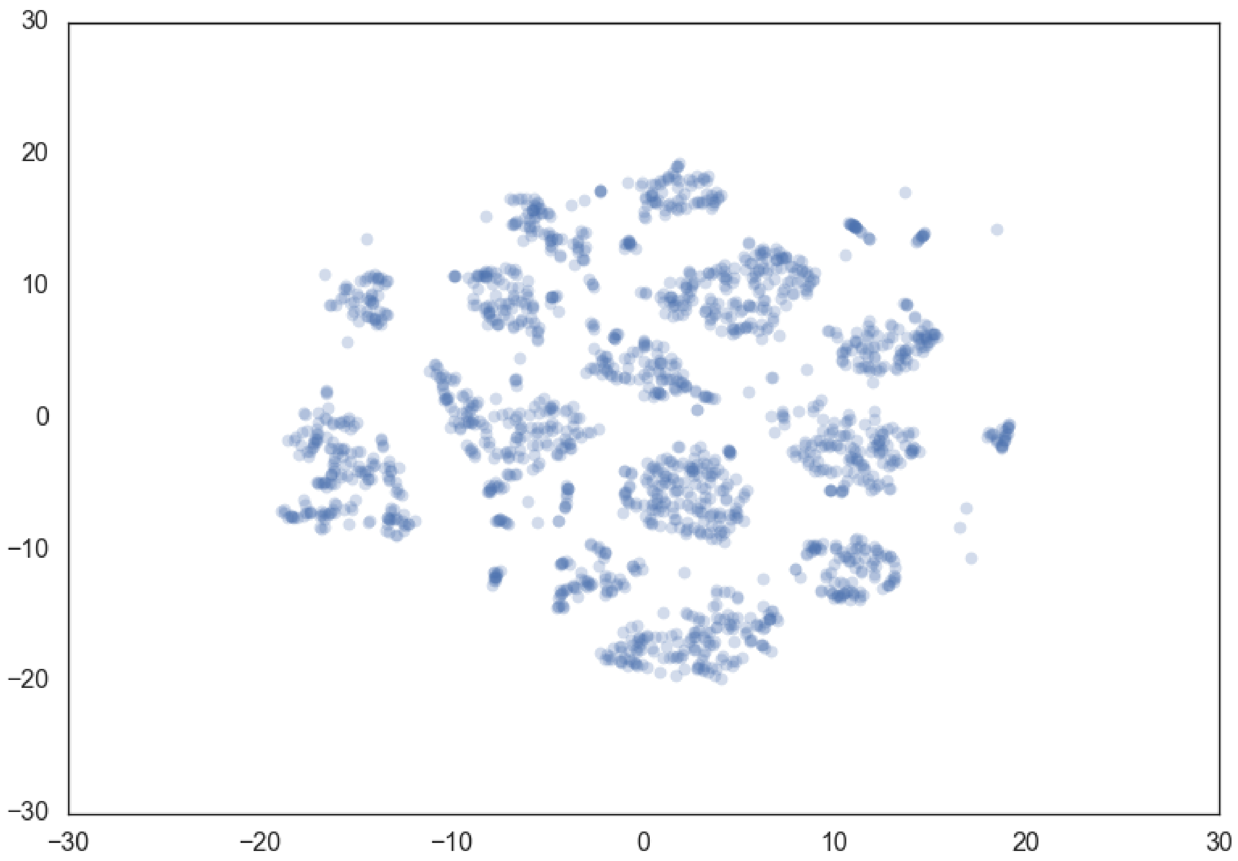
1.3 Parameter Selection for HDBSCAN*

While the HDBSCAN class has a large number of parameters that can be set on initialization, in practice there are a very small number of parameters that have significant practical effect on clustering. We will first consider those major parameters, and consider how one may go about choosing them effectively. With that out of the way we'll look at the remaining parameters and see what their effects are – many just effect performance for various different use cases.

1.3.1 Selecting `min_cluster_size`

The primary parameter to effect the resulting clustering is `min_cluster_size`. Ideally this is a relatively intuitive parameter to select – set it to the smallest size grouping that you sih to consider a cluster. It can have slightly non-obvious effects however. Let's consider the digits dataset from sklearn. We can project the data into two dimensions to visualize it via t-SNE.

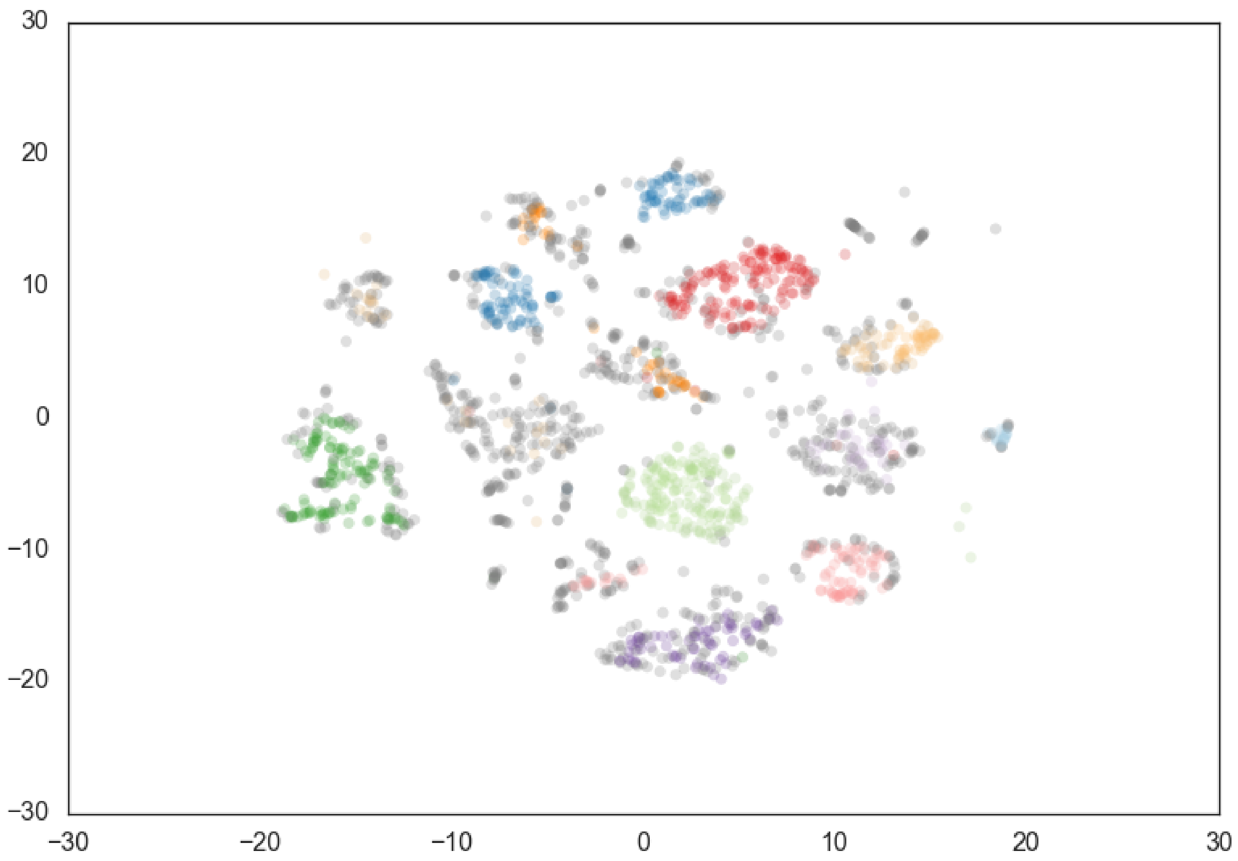
> digits = datasets.load_digits() data = digits.data projection = TSNE().fit_transform(data) plt.scatter(*projection.T, **plot_kwds)



If we cluster this data in the full 64 dimensional space with hdbscan we can see some effects from varying the `min_cluster_size`.

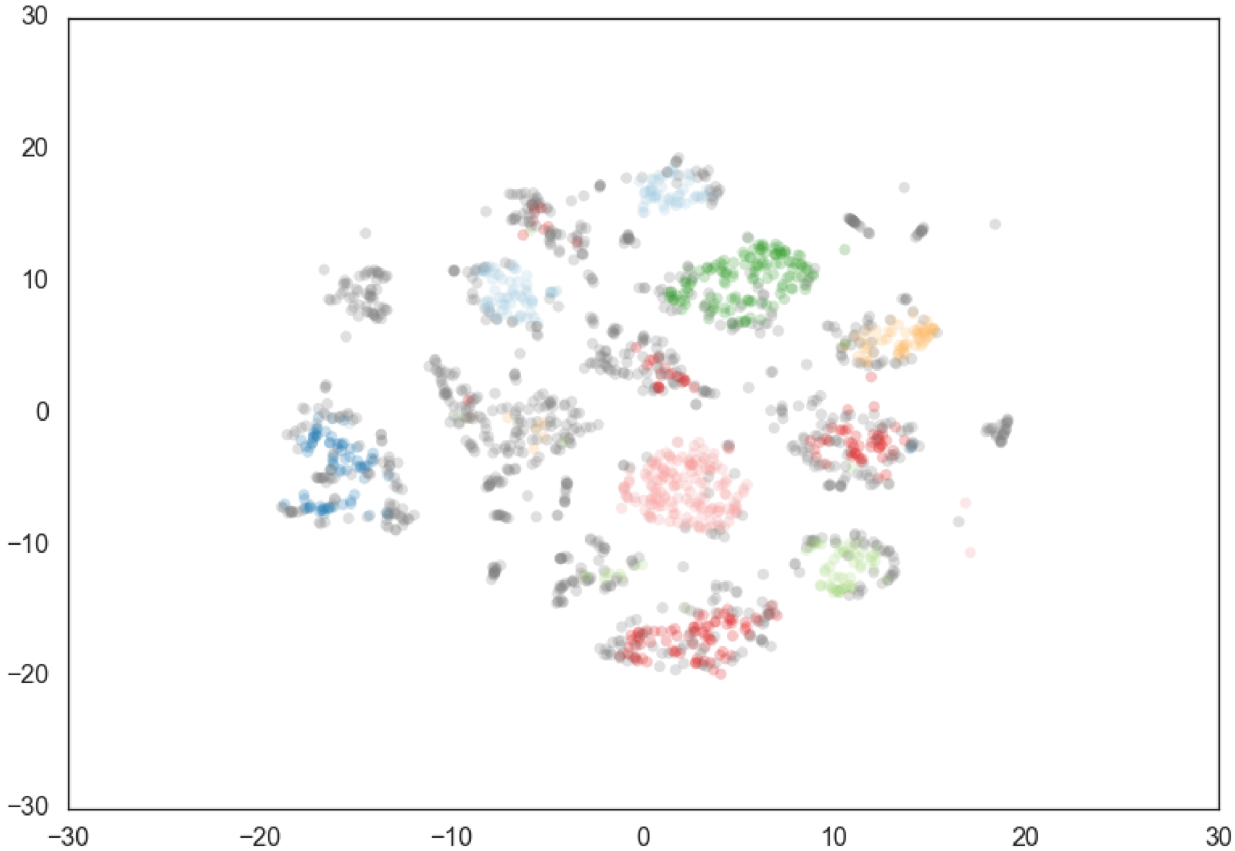
We start with a `min_cluster_size` of 15.

```
clusterer = hdbscan.HDBSCAN(min_cluster_size=15).fit(data)
color_palette = sns.color_palette('Paired', 12)
cluster_colors = [color_palette[x] if x >= 0
                  else (0.5, 0.5, 0.5)
                  for x in clusterer.labels_]
cluster_member_colors = [sns.desaturate(x, p) for x, p in
                         zip(cluster_colors, clusterer.probabilities_)]
plt.scatter(*projection.T, s=50, linewidth=0, c=cluster_member_colors, alpha=0.25)
```



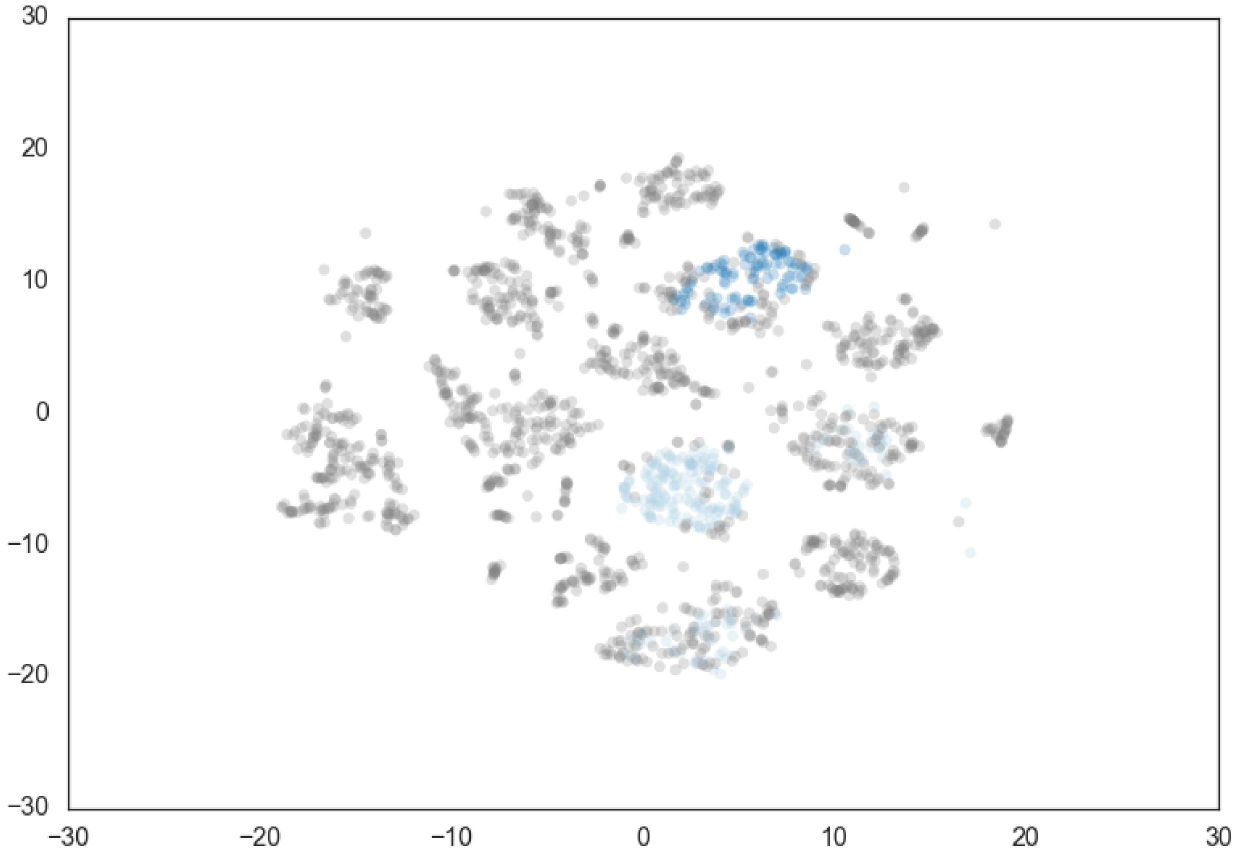
Increasing the `min_cluster_size` to 30 reduces the number of clusters, merging some together. This is a result of HDBSCAN* reoptimizing which flat clustering provides greater stability under a slightly different notion of what constitutes cluster.

```
clusterer = hdbscan.HDBSCAN(min_cluster_size=30).fit(data)
color_palette = sns.color_palette('Paired', 12)
cluster_colors = [color_palette[x] if x >= 0
                  else (0.5, 0.5, 0.5)
                  for x in clusterer.labels_]
cluster_member_colors = [sns.desaturate(x, p) for x, p in
                         zip(cluster_colors, clusterer.probabilities_)]
plt.scatter(*projection.T, s=50, linewidth=0, c=cluster_member_colors, alpha=0.25)
```

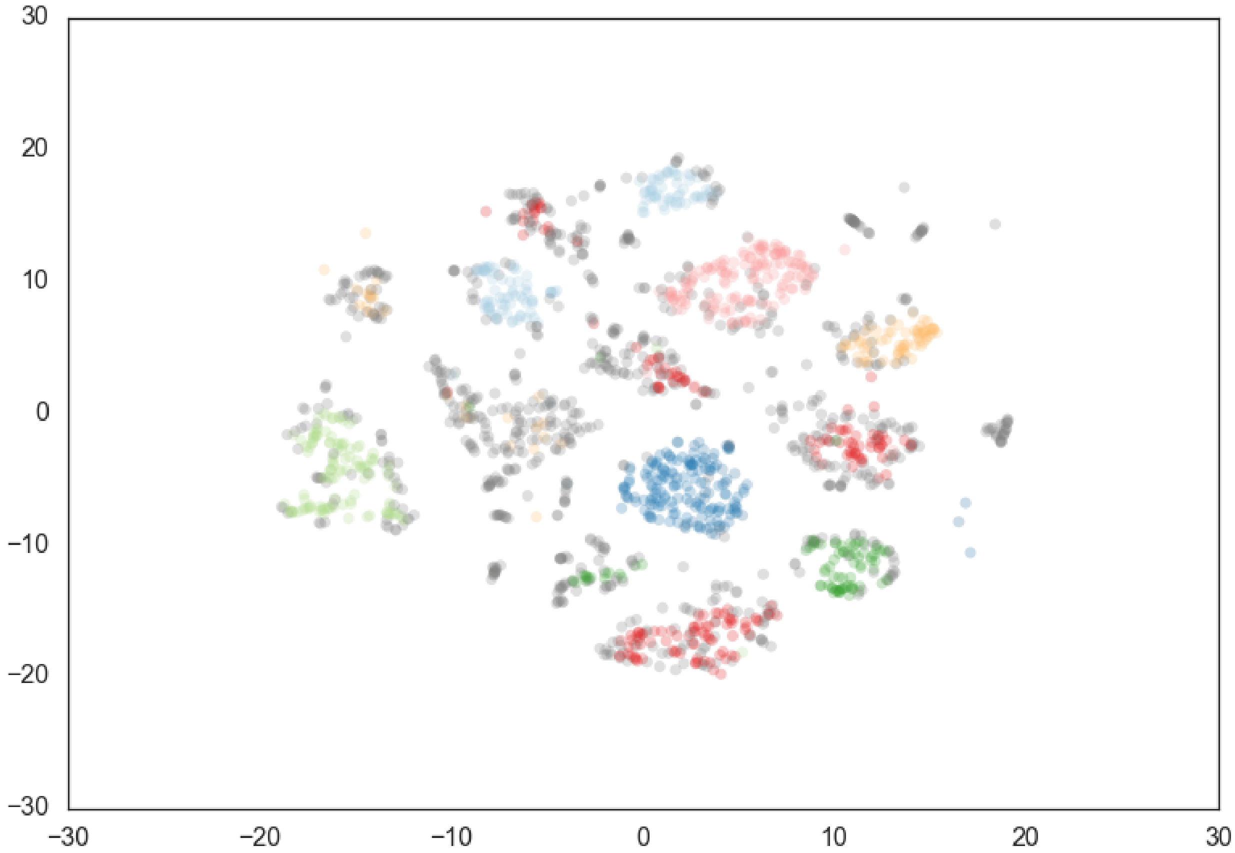


Doubling the min_cluster_size again to 60 gives us just two clusters – the really core clusters. This is somewhat as expected, but surely some of the other clusters that we had previously had more than 60 members? Why are they being considered noise? The answer is that HDBSCAN* has a second parameter min_samples. The implementation defaults this value (if it is unspecified) to whatever min_cluster_size is set to. We can recover some of our original clusters by explicitly providing min_samples at the original value of 15.

```
clusterer = hdbscan.HDBSCAN(min_cluster_size=60).fit(data)
color_palette = sns.color_palette('Paired', 12)
cluster_colors = [color_palette[x] if x >= 0
                  else (0.5, 0.5, 0.5)
                  for x in clusterer.labels_]
cluster_member_colors = [sns.desaturate(x, p) for x, p in
                         zip(cluster_colors, clusterer.probabilities_)]
plt.scatter(*projection.T, s=50, linewidth=0, c=cluster_member_colors, alpha=0.25)
```



```
clusterer = hdbscan.HDBSCAN(min_cluster_size=60, min_samples=15).fit(data)
color_palette = sns.color_palette('Paired', 12)
cluster_colors = [color_palette[x] if x >= 0
                  else (0.5, 0.5, 0.5)
                  for x in clusterer.labels_]
cluster_member_colors = [sns.desaturate(x, p) for x, p in
                         zip(cluster_colors, clusterer.probabilities_)]
plt.scatter(*projection.T, s=50, linewidth=0, c=cluster_member_colors, alpha=0.25)
```



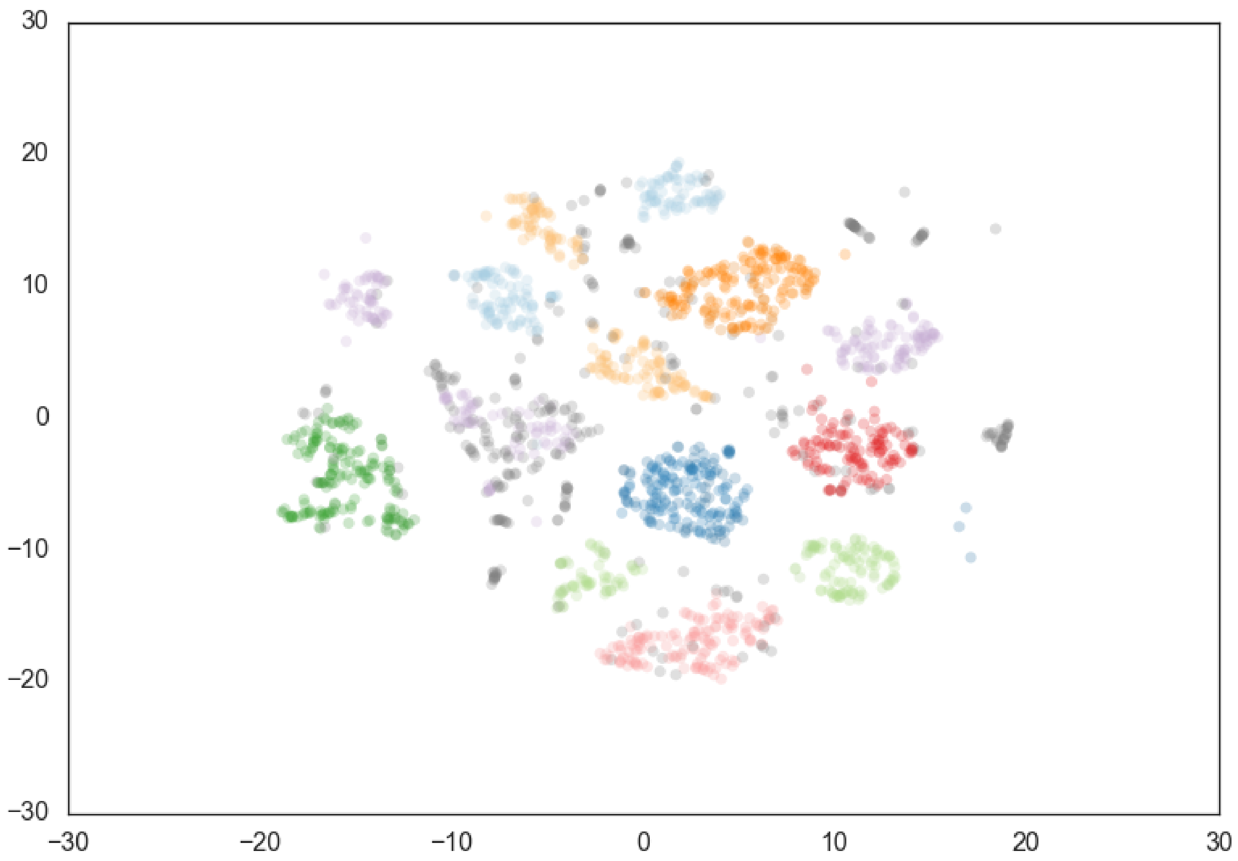
As you can see this results in us recovering something much closer to our original clustering, only now with some of the smaller clusters pruned out. Thus `min_cluster_size` does behave more closely to our intuitions, but only if we fix `min_samples`. If you wish to explore different `min_cluster_size` settings with a fixed `min_samples` value, especially for larger dataset sizes, you can cache the hard computation, and recompute onlythe relatively cheap flat cluster extraction using the `memory` parameter, which makes use of `joblib` [link].

1.3.2 Selecting `min_samples`

Since we have seen that `min_samples` clearly has a dramatic effect on clustering, the question becomes: how do we select this parameter? The simplest intuition for what `min_samples` does is provide a measure of how conservative you want you clustering to be. The larger the value of `min_samples` you provide, the more conservative the clustering – more points will be declared as noise, and clusters will be restricted to progressively more dense areas. We can see this in practice by leaving the `min_cluster_size` at 60, but reducing `min_samples` to 1.

```
clusterer = hdbscan.HDBSCAN(min_cluster_size=60, min_samples=1).fit(data)
color_palette = sns.color_palette('Paired', 12)
cluster_colors = [color_palette[x] if x >= 0
                  else (0.5, 0.5, 0.5)
                  for x in clusterer.labels_]
cluster_member_colors = [sns.desaturate(x, p) for x, p in
                         zip(cluster_colors, clusterer.probabilities_)]
plt.scatter(*projection.T, s=50, linewidth=0, c=cluster_member_colors, alpha=0.25)
```

```
<matplotlib.collections.PathCollection at 0x120978438>
```

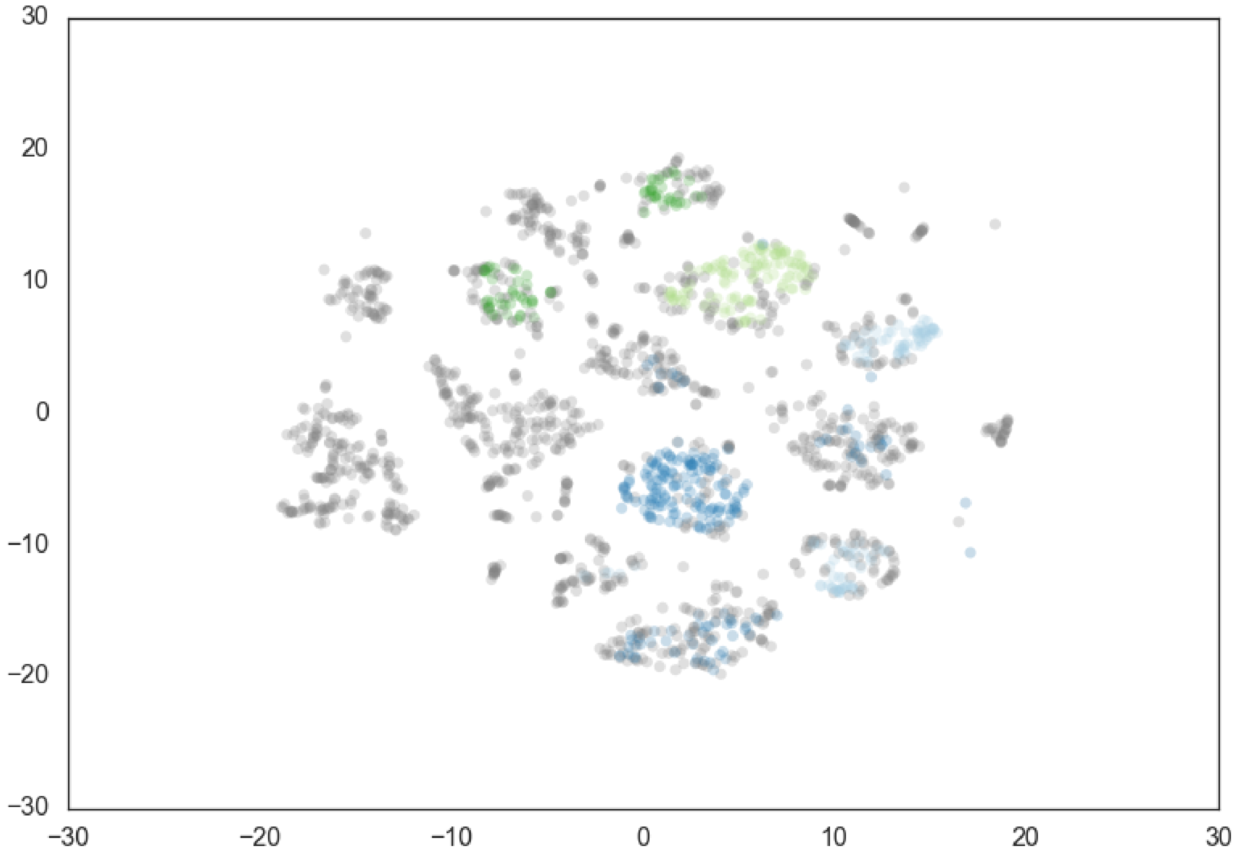


Now most points are clustered, and there are much fewer noise points. Steadily increasing min_samples will, as we saw in the examples above, make the clustering progressivly more conservative, culiminating in the example above where min_samples was set to 60 and we had only two clusters with most points declared as noise.

1.3.3 Selecting alpha

A further parameter that effects the resulting clustering is alpha. In practice it is best not to mess with this paramter – ultimately it is part of the RobustSingleLinkage code, but flows naturally into HDBSCAN*. If, for some reason, min_samples is not providing you what you need, stop, rethink things, and try again with min_samples. If you still need to play with another parameter (and you shouldn't), then you can try setting alpha. The alpha parameter provides a slightly different approach to determining how conservative the clustering is. By default alpha is set to 1.0. Increasing alpha will make the clustering more conservative, but on a much tighter scale, as we can see by setting alpha to 1.3.

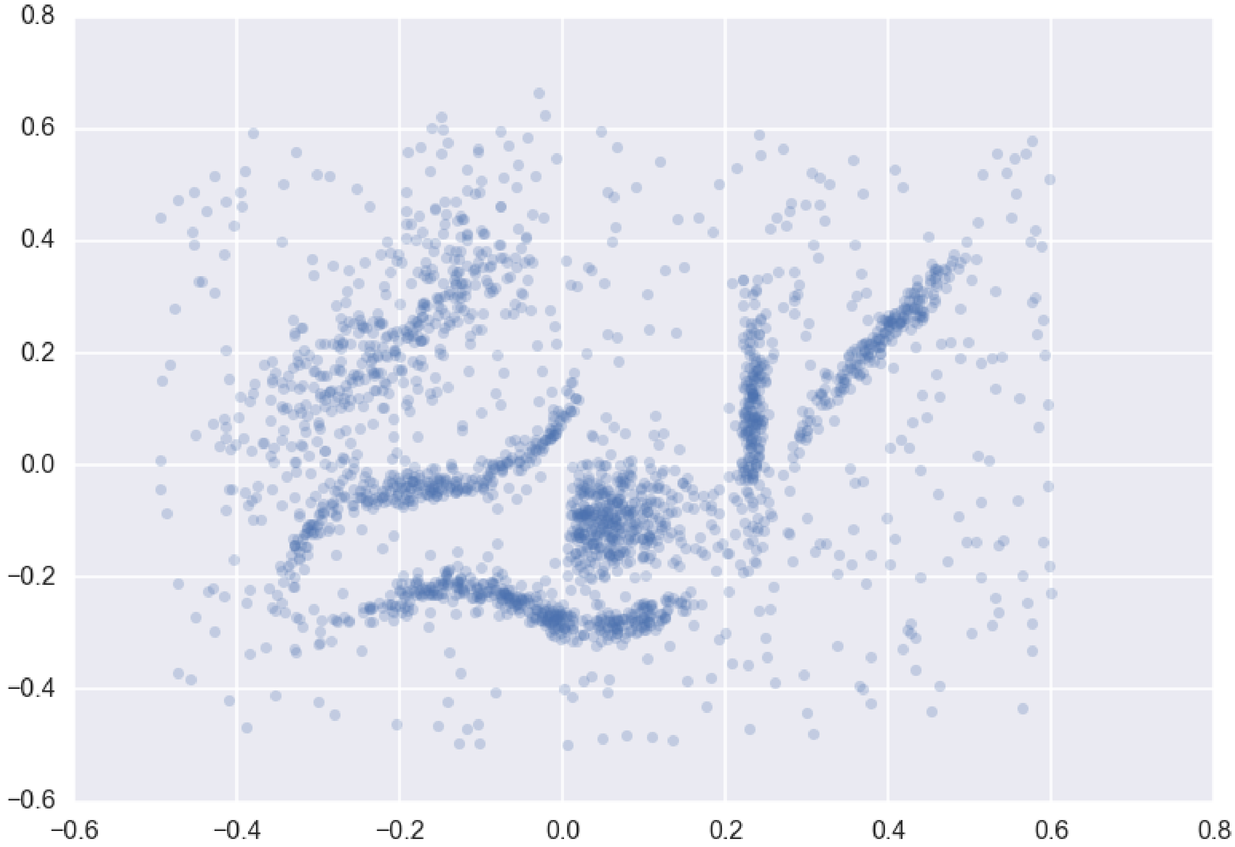
```
clusterer = hdbscan.HDBSCAN(min_cluster_size=60, min_samples=15, alpha=1.3).fit(data)
color_palette = sns.color_palette('Paired', 12)
cluster_colors = [color_palette[x] if x >= 0
                  else (0.5, 0.5, 0.5)
                  for x in clusterer.labels_]
cluster_member_colors = [sns.desaturate(x, p) for x, p in
                         zip(cluster_colors, clusterer.probabilities_)]
plt.scatter(*projection.T, s=50, linewidth=0, c=cluster_member_colors, alpha=0.25)
```



1.4 Outlier Detection

The hdbscan library supports the GLOSH outlier detection algorithm, and does so within the HDBSCAN clustering class. The GLOSH outlier detection algorithm is related to older outlier detection methods such as LOF and LOCI. It is a fast and flexible outlier detection system, and supports a notion of local outliers. This means that it can detect outliers that may be noticeably different from points in its local region (for example points not on a local submanifold) but that are not necessarily outliers globally. So how do we find outliers? We proceed identically to the basic use of HDBSCAN*. We start with some data, and fit it with an HDBSCAN object.

```
plt.scatter(*data.T, s=50, linewidth=0, c='b', alpha=0.25)
```



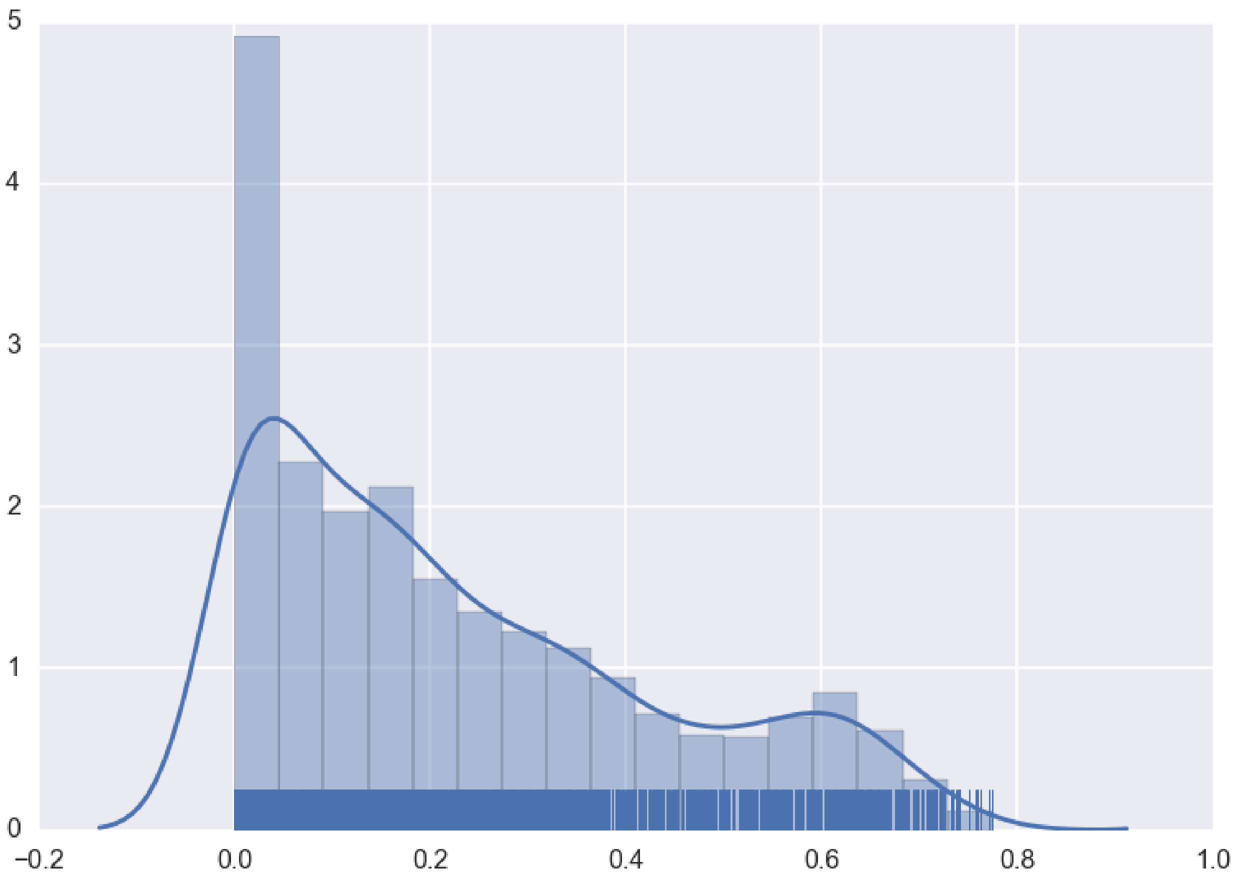
```
clusterer = hdbscan.HDBSCAN(min_cluster_size=15).fit(data)
```

The clusterer object now has an attribute (computed when first accessed) called `outlier_scores_`. This provides a numpy array with a value for each sample in the original dataset that was fit with the clusterer. The higher the score, the more likely the point is to be an outlier. In practice it is often best to look at the distributions of outlier scores.

```
clusterer.outlier_scores_
```

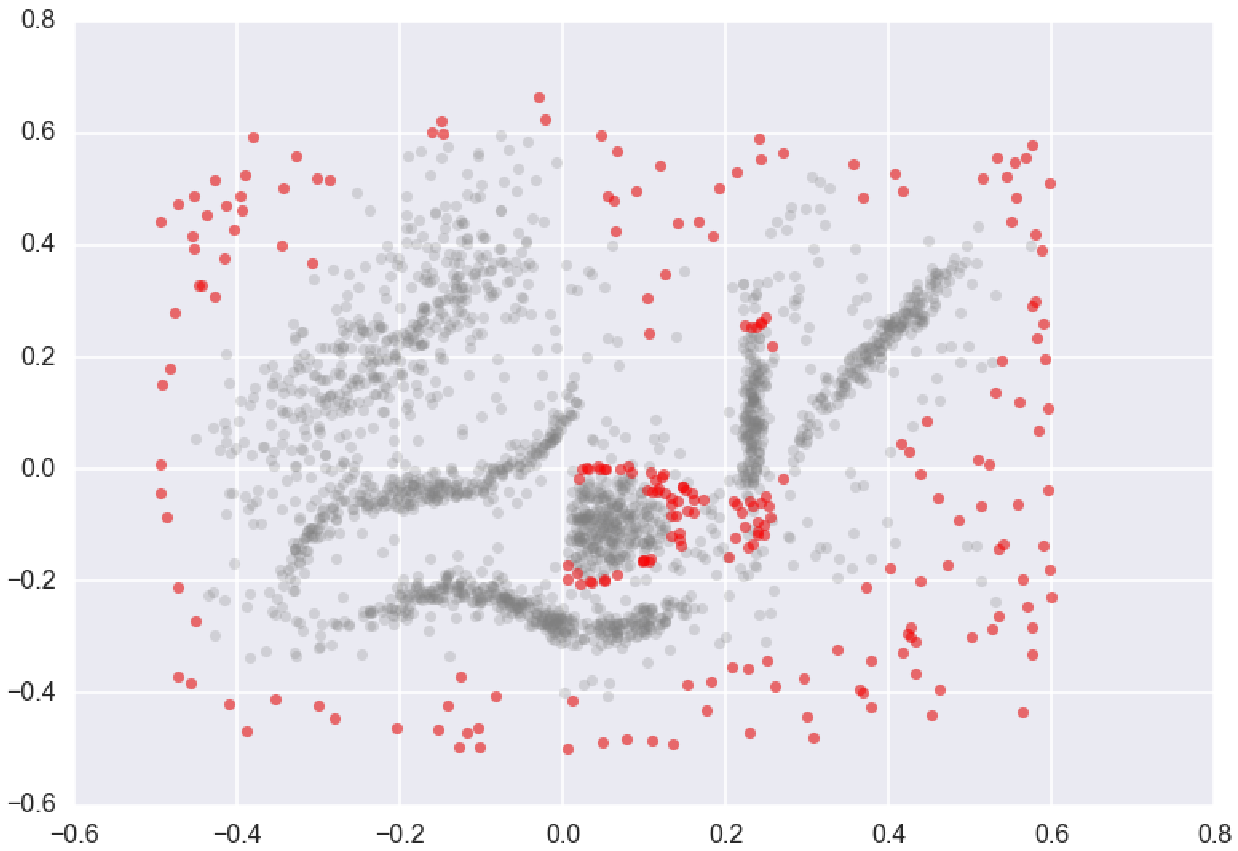
```
array([ 0.14791852,  0.14116731,  0.09171929, ...,  0.62050534,
        0.56749298,  0.20681685])
```

```
sns.distplot(clusterer.outlier_scores_[np.isfinite(clusterer.outlier_scores_)], rug=True)
```



We can pull off upper quantiles to detect outliers, which we can then plot.

```
threshold = pd.Series(clusterer.outlier_scores_).quantile(0.9)
outliers = np.where(clusterer.outlier_scores_ > threshold)[0]
plt.scatter(*data.T, s=50, linewidth=0, c='gray', alpha=0.25)
plt.scatter(*data[outliers].T, s=50, linewidth=0, c='red', alpha=0.5)
```



Note that not only are the outlying border points highlighted as outliers, but points at the edge of the central ball like cluster, and just below the vertical band cluster, are also designated as outliers. This is because those two clusters are extremely dense, and the points at the edge of this cluster are close enough to the cluster that they should be part of it, but far enough from the being core parts of the cluster that they are extremely unlikely and hence anomalous.

CHAPTER 2

Background on Clustering with HDBSCAN

2.1 How HDBSCAN Works

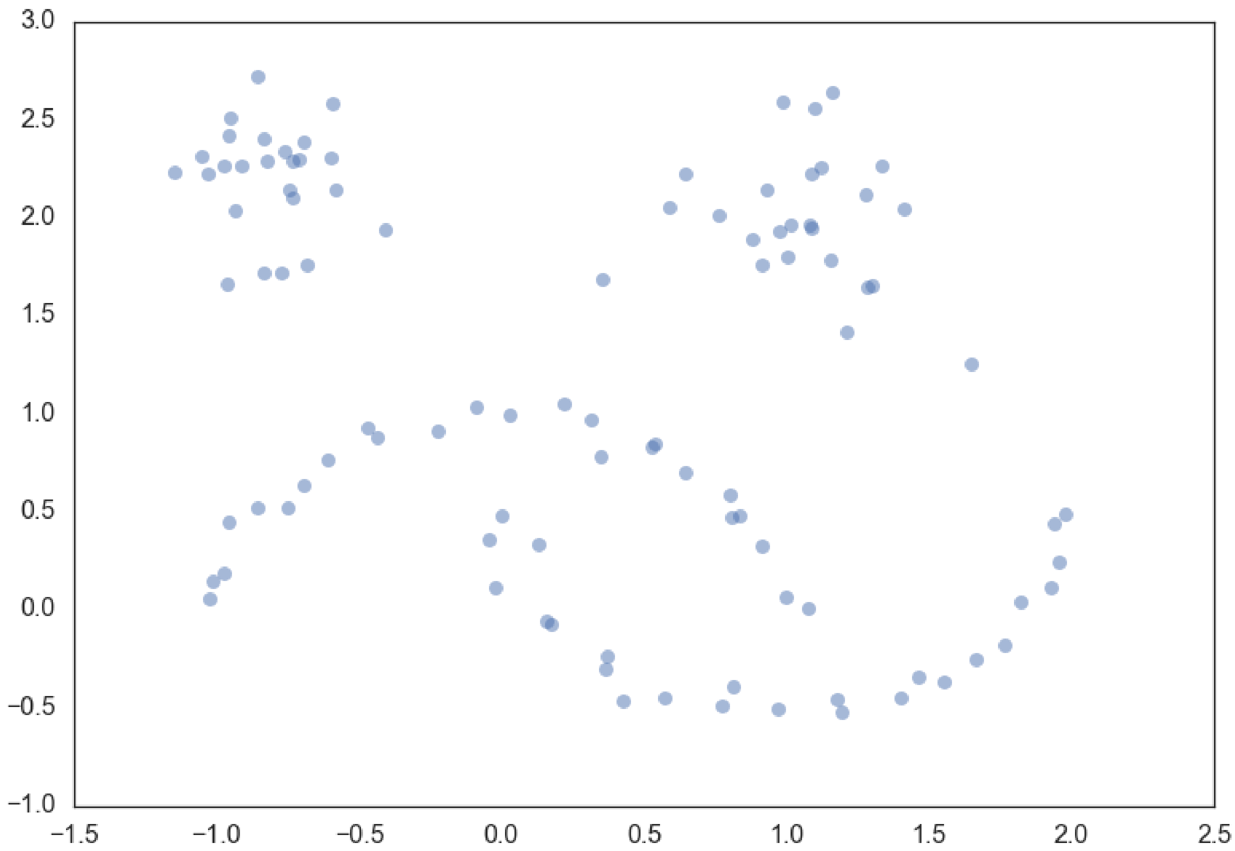
HDBSCAN is a clustering algorithm developed by Campello, Moulavi, and Sander. It extends DBSCAN by converting it into a hierarchical clustering algorithm, and then using a technique to extract a flat clustering based in the stability of clusters. The goal of this notebook is to give you an overview of how the algorithm works and the motivations behind it. In contrast to the HDBSCAN paper I'm going to describe it without reference to DBSCAN. Instead I'm going to explain how I like to think about the algorithm, which aligns more closely with Robust Single Linkage with flat cluster extraction on top of it.

Before we get started we'll load up most of the libraries we'll need in the background, and set up our plotting (because I believe the best way to understand what is going on is to actually see it working in pictures).

```
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns
import sklearn.datasets as data
%matplotlib inline
sns.set_context('poster')
sns.set_style('white')
sns.set_color_codes()
plot_kwds = {'alpha' : 0.5, 's' : 80, 'linewidths':0}
```

The next thing we'll need is some data. To make for an illustrative example we'll need the data size to be fairly small so we can see what is going on. It will also be useful to have several clusters, preferably of different kinds. Fortunately sklearn has facilities for generating sample clustering data so I'll make use of that and make a dataset of one hundred data points.

```
moons, _ = data.make_moons(n_samples=50, noise=0.05)
blobs, _ = data.make_blobs(n_samples=50, centers=[(-0.75,2.25), (1.0, 2.0)], cluster_std=0.25)
test_data = np.vstack([moons, blobs])
plt.scatter(test_data.T[0], test_data.T[1], color='b', **plot_kwds)
```



Now, the best way to explain HDBSCAN is actually just use it and then go through the steps that occurred along the way teasing out what is happening at each step. So let's load up the hdbscan library and get to work.

```
import hdbscan
```

```
clusterer = hdbscan.HDBSCAN(min_cluster_size=5, gen_min_span_tree=True)
clusterer.fit(test_data)
```

```
HDBSCAN(algorithm='best', alpha=1.0, approx_min_span_tree=True,
    gen_min_span_tree=True, leaf_size=40, memory=Memory(cachedir=None),
    metric='euclidean', min_cluster_size=5, min_samples=None, p=None)
```

So now that we have clustered the data – what actually happened? We can break it out into a series of steps

1. Transform the space according to the density/sparsity.
2. Build the minimum spanning tree of the distance weighted graph.
3. Construct a cluster hierarchy of connected components.
4. Condense the cluster hierarchy based on minimum cluster size.
5. Extract the stable clusters from the condensed tree.

2.1.1 Transform the space

To find clusters we want to find the islands of higher density amid a sea of sparser noise – and the assumption of noise is important: real data is messy and has outliers, corrupt data, and noise. The core of the clustering algorithm is single linkage clustering, and it can be quite sensitive to noise: a single noise data point in the wrong place can act as a bridge between islands, gluing them together. Obviously we want our algorithm to be robust against noise so we need to find a way to help 'lower the sea level' before running a single linkage algorithm.

How can we characterize 'sea' and 'land' without doing a clustering? As long as we can get an estimate of density we can consider lower density points as the 'sea'. The goal here is not to perfectly distinguish 'sea' from 'land' – this is an initial step in clustering, not the ouput – just to make our clustering core a little more robust to noise. So given an identification of 'sea' we want to lower the sea level. For practical purposes that means making 'sea' points more distant from each other and from the 'land'.

That's just the intuition however. How does it work in practice? We need a very inexpensive estimate of density, and the simplest is the distance to the *k*th nearest neighbor. If we have the distance matrix for our data (which we will need imminently anyway) we can simply read that off; alternatively if our metric is supported (and dimension is low) this is the sort of query that kd-trees are good for. Let's formalise this and (following the DBSCAN, LOF, and HDBSCAN literature) call it the **core distance** defined for parameter *k* for a point *x* and denote as $\mathrm{core}_k(x)$. Now we need a way to spread apart points with low density (correspondingly high core distance). The simple way to do this is to define a new distance metric between points which we will call (again following the literature) the **mutual reachability distance**. We define mutual reachability distance as follows:

$$d_{\mathrm{mreach-}k}(a,b) = \max\{\mathrm{core}_k(a), \mathrm{core}_k(b), d(a,b)\}$$

where $d(a,b)$ is the original metric distance between *a* and *b*. Under this metric dense points (with low core distance) remain the same distance from each other but sparser points are pushed away to be at least their core distance away from any other point. This effectively 'lowers the sea level' spreading sparse 'sea' points out, while leaving 'land' untouched. The caveat here is that obviously this is dependent upon the choice of *k*; larger *k* values interpret more points as being in the 'sea'. All of this is a little easier to understand with a picture, so let's use a *k* value of five. Then for a given point we can draw a circle for the core distance as the circle that touches the fifth nearest neighbor, like so:

Pick another point and we can do the same thing, this time with a different set of neighbors (one of them even being the first point we picked out).

And we can do that a third time for good measure, with another set of five nearest neighbors and another circle with slightly different radius again.

Now if we want to know the mutual reachability distance between the blue and green points we can start by drawing in and arrow giving the distance between green and blue:

This passes through the blue circle, but not the green circle – the core distance for green is larger than the distance between blue and green. Thus we need to mark the mutual reachability distance between blue and green as larger – equal to the radius of the green circle (easiest to picture if we base one end at the green point).

On the other hand the mutual reachablity distance from red to green is simply distance from red to green since that distance is greater than either core distance (i.e. the distance arrow passes through both circles).

In general there is underlying theory to demonstrate that mutual reachability distance as a transform works well in allowing single linkage clustering to more closely approximate the hierarchy of level sets of whatever true density distribution our points were sampled from.

2.1.2 Build the minimum spanning tree

Now that we have a new mutual reachability metric on the data we want start finding the islands on dense data. Of course dense areas are relative, and different islands may have different densities. Conceptually what we will do is the following: consider the data as a weighted graph with the data points as vertices and an edge between any two points with weight equal to the mutual reachability distance of those points.

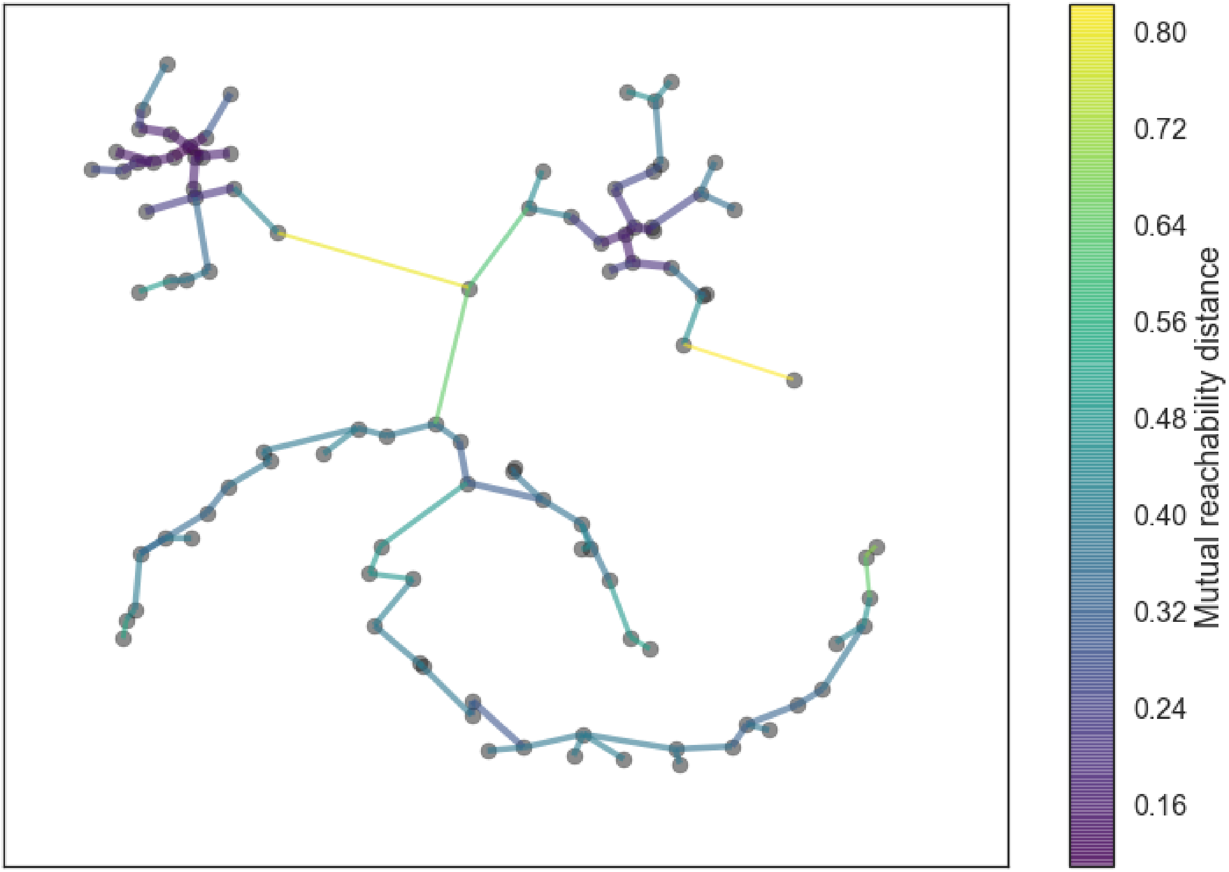
Now consider a threshold value, starting high, and steadily being lowered. Drop any edges with weight above that threshold. As we drop edges we will start to disconnect the graph into connected components. Eventually we will have a hierarchy of connected components (from completely connected to completely disconnected) at varying threshold levels.

In practice this is very expensive: there are n^2 edges and we don't want to have to run a connected components algorithm that many times. The right thing to do is to find a minimal set of edges such that dropping any edge from the set causes a disconnection of components. But we need more, we need this set to be such that there is no lower weight edge that could connect the components. Fortunately graph theory furnishes us with just such a thing: the minimum spanning tree of the graph.

We can build the minimum spanning tree very efficiently via Prim's algorithm – we build the tree one edge at a time, always adding the lowest weight edge that connects the current tree to a vertex not yet in the tree. You can see the tree HDBSCAN constructed below; note that this is the minimum spanning tree for *mutual reachability distance* which is different from the pure distance in the graph. In this case we had a *k* value of 5.

In the case that the data lives in a metric space we can use even faster methods, such as Dual Tree Boruvka to build the minimal spanning tree.

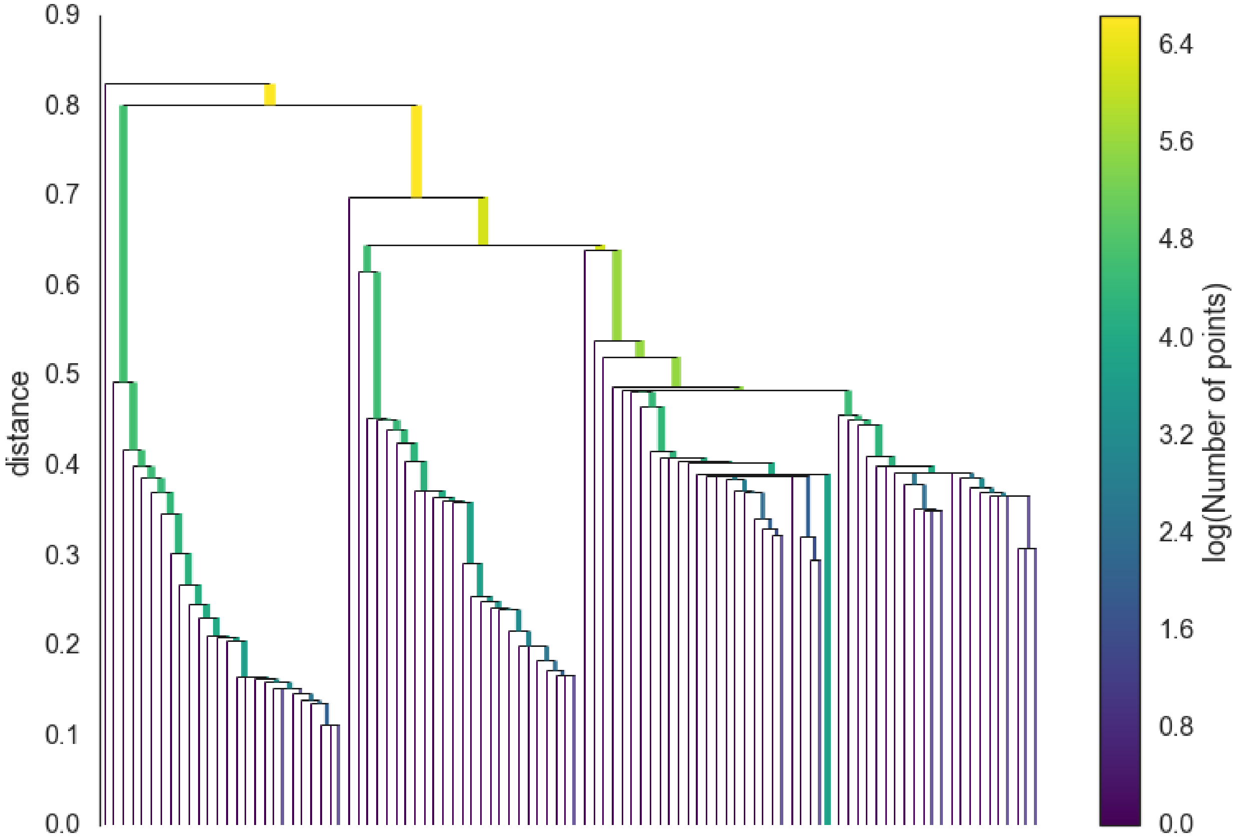
```
clusterer.minimum_spanning_tree_.plot(edge_cmap='viridis',
                                      edge_alpha=0.6,
                                      node_size=80,
                                      edge_linewidth=2)
```



2.1.3 Build the cluster hierarchy

Given the minimal spanning tree, the next step is to convert that into the hierarchy of connected components. This is most easily done in the reverse order: sort the edges of the tree by distance (in increasing order) and then iterate through, creating a new merged cluster for each edge. The only difficult part here is to identify the two clusters each edge will join together, but this is easy enough via a union-find data structure. We can view the result as a dendrogram as we see below:

```
clusterer.single_linkage_tree_.plot(cmap='viridis', colorbar=True)
```

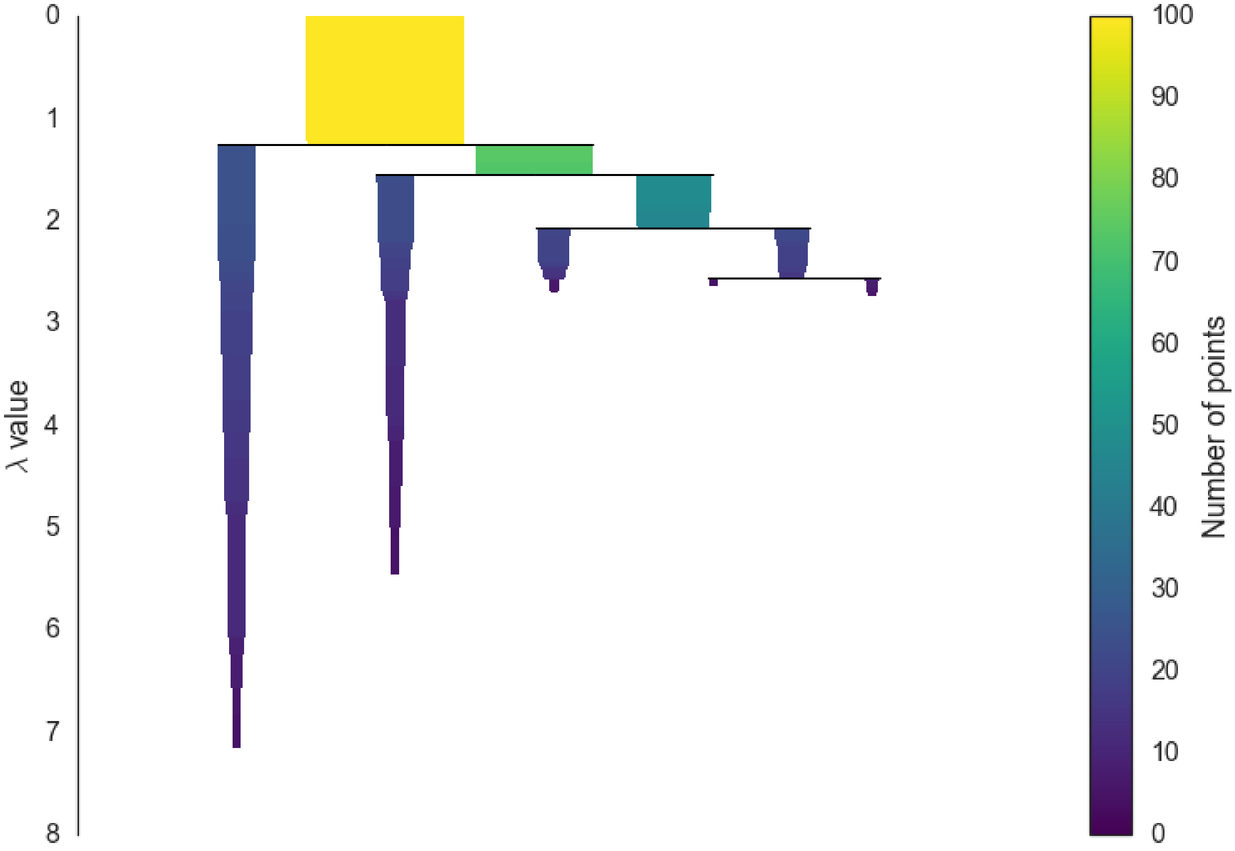


This brings us to the point where robust single linkage stops. We want more though; a cluster hierarchy is good, but we really want a set of flat clusters. We could do that by drawing a a horizontal line through the above diagram and selecting the clusters that it cuts through. This is in practice what DBSCAN effectively does (declaring any singleton clusters at the cut level as noise). The question is, how do we know where to draw that line? DBSCAN simply leaves that as a (very unintuitive) parameter. Worse, we really want to deal with variable density clusters and any choice of cut line is a choice of mutual reachability distance to cut at, and hence a single fixed density level. Ideally we want to be able to cut the tree at different places to select our clusters. This is where the next steps of HDBSCAN begin and create the difference from robust single linkage.

2.1.4 Condense the cluster tree

The first step in cluster extraction is condensing down the large and complicated cluster hierarchy into a smaller tree with a little more data attached to each node. As you can see in the hierarchy above it is often the case that a cluster split is one or two points splitting off from a cluster; and that is the key point – rather than seeing it as a cluster splitting into two new clusters we want to view it as a single persistent cluster that is 'losing points'. To make this concrete we need a notion of **minimum cluster size** which we take as a parameter to HDBSCAN. Once we have a value for minimum cluster size we can now walk through the hierarchy and at each split ask if one of the new clusters created by the split has fewer points than the minimum cluster size. If it is the case that we have fewer points than the minimum cluster size we declare it to be 'points falling out of a cluster' and have the larger cluster retain the cluster identity of the parent, marking down which points 'fell out of the cluster' and at what distance value that happened. If on the other hand the split is into two clusters each at least as large as the minimum cluster size then we consider that a true cluster split and let that split persist in the tree. After walking through the whole hierarchy and doing this we end up with a much smaller tree with a small number of nodes, each of which has data about how the size of the cluster at that node descreases over varying distance. We can visualize this as a dendrogram similar to the one above – again we can have the width of the line represent the number of points in the cluster. This time, however, that width varies over the length of the line as points fall our of the cluster. For our data using a minimum cluster size of 5 the result looks like this:

```
clusterer.condensed_tree_.plot()
```



This is much easier to look at and deal with, particularly in as simple a clustering problem as our current test dataset. However we still need to pick out clusters to use as a flat clustering. Looking at the plot above should give you some ideas about how one might go about doing this.

2.1.5 Extract the clusters

Intuitively we want the choose clusters that persist and have a longer lifetime; short lived clusters are ultimately probably merely artifcacts of the single linkage approach. Looking at the previous plot we could say that we want to choose those clusters that have the greatest area of ink in the plot. To make a flat clustering we will need to add a further requirement that, if you select a cluster, then you cannot select any cluster that is a descendant of it. And in fact that intuitive notion of what should be done is exactly what HDBSCAN does. Of course we need to formalise things to make it a concrete algorithm.

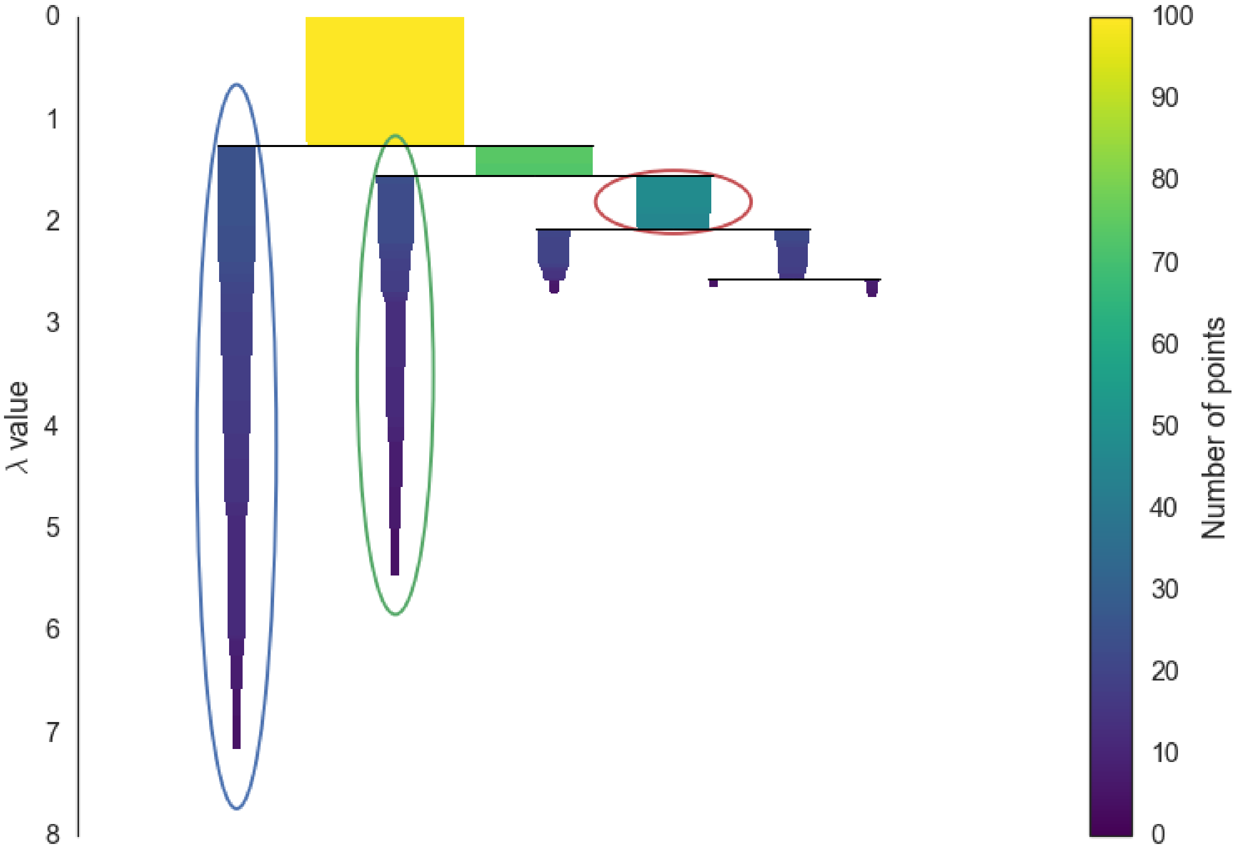
First we need a different measure than distance to consider the persistence of clusters; instead we will use $\lambda = \frac{1}{\mathrm{distance}}$. For a given cluster we can then define values λ_{birth} and λ_{death} to be the lambda value when the cluster split off and became it's own cluster, and the lambda value (if any) when the cluster split into smaller clusters respectively. In turn, for a given cluster, for each point p in that cluster we can define the value λ_p as the lambda value at which that point 'fell out of the cluster' which is a value somewhere between λ_{birth} and λ_{death} since the point either falls out of the cluster at some point in the cluster's lifetime, or leaves the cluster when the cluster splits into two smaller clusters. Now, for each cluster compute the **stability** to as

$\sum_{p\in\text{cluster}}(\lambda_p - \lambda_{\text{birth}})$.

Declare all leaf nodes to be selected clusters. Now work up through the tree (the reverse topological sort order). If the sum of the stabilities of the child clusters is greater than the stability of the cluster then we set the cluster stability to be the sum of the child stabilities. If, on the other hand, the cluster's stability is greater than the su of it's children then we declare the cluster to be a selected cluster, and unselect all its descendants. Once we reach the root node we call the current set of selected clusters our flat clsutering and return that.

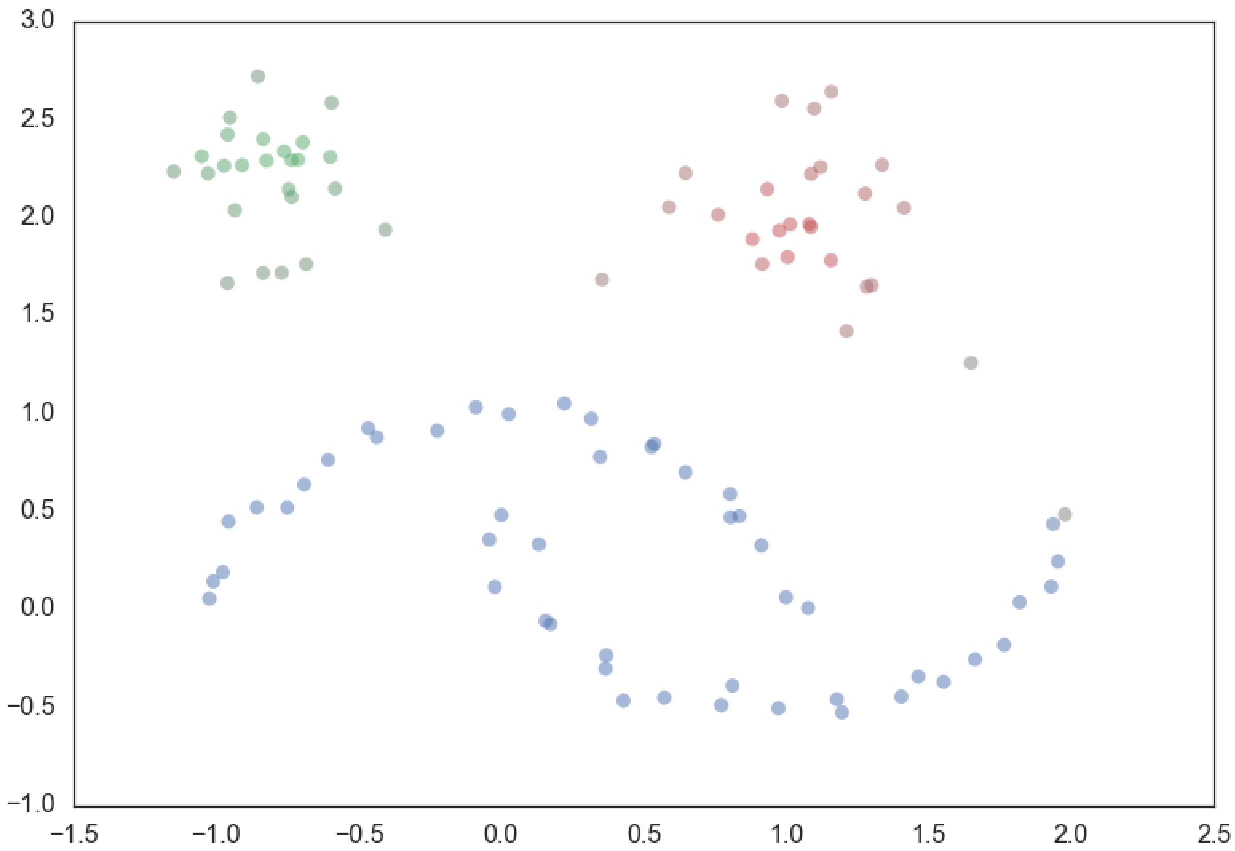
Okay, that was wordy and complicated, but it really is simply performing our 'select the clusters in the plot with the largest total ink area' subject to descendant constraints that we explained earlier. We can select the clusters in the condensed tree dendrogram via this algorithm, and you get what you expect:

```
clusterer.condensed_tree_.plot(select_clusters=True, selection_palette=sns.color_palette())
```



Now that we have the clusters it is a simple enough matter to turn that into cluster labelling as per the sklearn API. Any point not in a selected cluster is simply a noise point (and assigned the label -1). We can do a little more though: for each cluster we have the λ_p for each point p in that cluster; If we simply normalize those values (so they range from zero to one) then we have a measure of the strength of cluster membership for each point in the cluster. The hdbscan library returns this as a `probabilities_` attribute of the clusterer object. Thus, with labels and membership strengths in hand we can make the standard plot, choosing a color for points based on cluster label, and desaturating that color according the strength of membership (and make unclustered points pure gray).

```
palette = sns.color_palette()
cluster_colors = [sns.desaturate(palette[col], sat)
                  if col >= 0 else (0.5, 0.5, 0.5) for col, sat in
                  zip(clusterer.labels_, clusterer.probabilities_)]
plt.scatter(test_data.T[0], test_data.T[1], c=cluster_colors, **plot_kwds)
```



And that is how HDBSCAN works. It may seem somewhat complicated – there are a fair number of moving parts to the algorithm – but ultimately each part is actually very straightforward and can be optimized well. Hopefully with a better understanding both of the intuitions and some of the implementation details of HDBSCAN you will feel motivated to try it out. The library continues to develop, and will provide a base for new ideas including a near parameterless Persistent Density Clustering algorithm, and a new semi-supervised clustering algorithm.

2.2 Comparing Python Clustering Algorithms

There are a lot of clustering algorithms to choose from. The standard `sklearn` clustering suite has thirteen different clustering classes alone. So what clustering algorithms should you be using? As with every question in data science and machine learning it depends on your data. A number of those thirteen classes in `sklearn` are specialised for certain tasks (such as co-clustering and bi-clustering, or clustering features instead data points). Obviously an algorithm specializing in text clustering is going to be the right choice for clustering text data, and other algorithms specialize in other specific kinds of data. Thus, if you know enough about your data, you can narrow down on the clustering algorithm that best suits that kind of data, or the sorts of important properties your data has, or the sorts of clustering you need done. All well and good, but what if you don't know much about your data? If, for example, you are 'just looking' and doing some exploratory data analysis (EDA) it is not so easy to choose a specialized algorithm. So, what algorithm is good for exploratory data analysis?

2.2.1 Some rules for EDA clustering

To start, lets' lay down some ground rules of what we need a good EDA clustering algorithm to do, then we can set about seeing how the algorithms available stack up.

- **Don't be wrong!**: If you are doing EDA you are trying to learn and gain intuitions about your data. In that case it is far better to get no result at all than a result that is wrong. Bad results lead to false intuitions which in turn send you down completely the wrong path. Not only do you not understand your data, you *misunderstand* your data. This means a good EDA clustering algorithm needs to conservative in int's clustering; it should be willing to not assign points to clusters; it should not group points together unless they really are in a cluster; this is true of far fewer algorithms than you might think.
- **Intuitive Parameters**: All clustering algorithms have parameters; you need some knobs to turn to adjust things. The question is: how do you pick settings for those parameters? If you know little about your data it can be hard to determine what value or setting a parameter should have. This means parameters need to be intuitive enough that you can hopefully set them without having to know a lot about your data.
- **Stable Clusters**: If you run the algorithm twice with a different random initialization, you should expect to get roughly the same clusters back. If you are sampling your data, taking a different random sample shouldn't radically change the resulting cluster structure (unless your sampling has problems). If you vary the clustering algorithm parameters you want the clustering to change in a somewhat stable predictable fashion.
- **Performance**: Data sets are only getting bigger. You can sub-sample (but see *stability*), but ultimately you need a clustering algorithm that can scale to large data sizes. A clustering algorithm isn't much use if you can only use it if you take such a small sub-sample that it is no longer representative of the data at large!

There are other nice to have features like soft clusters, or overlapping clusters, but the above desiderata is enough to get started with because, oddly enough, very few clustering algorithms can satisfy them all!

2.2.2 Getting set up

If we are going to compare clustering algorithms we'll need a few things; first some libraries to load and cluster the data, and second some visualisation tools so we can look at the results of clustering.

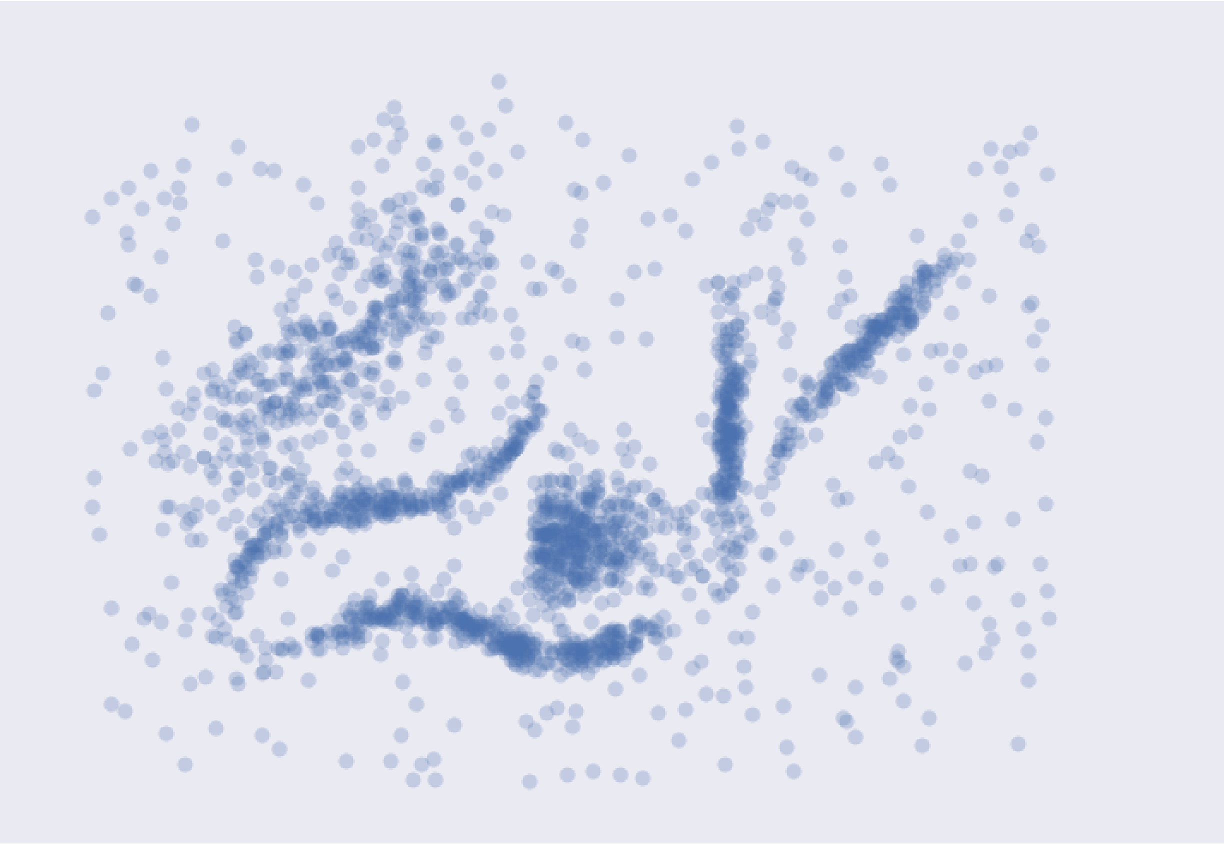
```
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns
import sklearn.cluster as cluster
import time
%matplotlib inline
sns.set_context('poster')
sns.set_color_codes()
plot_kwds = {'alpha' : 0.25, 's' : 80, 'linewidths':0}
```

Next we need some data. In order to make this more interesting I've constructed an artificial dataset that will give clustering algorithms a challenge – some non-globular clusters, some noise etc.; the sorts of things we expect to crop up in messy real-world data. So that we can actually visualize clusterings the dataset is two dimensional; this is not something we expect from real-world data where you generally can't just visualize and see what is going on.

```
data = np.load('clusterable_data.npy')
```

So let's have a look at the data and see what we have.

```
plt.scatter(data.T[0], data.T[1], c='b', **plot_kwds)
frame = plt.gca()
frame.axes.get_xaxis().set_visible(False)
frame.axes.get_yaxis().set_visible(False)
```



It's messy, but there are certainly some clusters that you can pick out by eye; determining the exact boundaries of those clusters is harder of course, but we can hope that our clustering algorithms will find at least some of those clusters. So, on to testing ...

2.2.3 Testing Clustering Algorithms

To start let's set up a little utility function to do the clustering and plot the results for us. We can time the clustering algorithm while we're at it and add that to the plot since we do care about performance.

```python
def plot_clusters(data, algorithm, args, kwds):
    start_time = time.time()
    labels = algorithm(*args, **kwds).fit_predict(data)
    end_time = time.time()
    palette = sns.color_palette('deep', np.unique(labels).max() + 1)
    colors = [palette[x] if x >= 0 else (0.0, 0.0, 0.0) for x in labels]
    plt.scatter(data.T[0], data.T[1], c=colors, **plot_kwds)
    frame = plt.gca()
    frame.axes.get_xaxis().set_visible(False)
    frame.axes.get_yaxis().set_visible(False)
    plt.title('Clusters found by {}'.format(str(algorithm.__name__)), fontsize=24)
    plt.text(-0.5, 0.7, 'Clustering took {:.2f} s'.format(end_time - start_time), fontsize=14)
```

Before we try doing the clustering, there are some things to keep in mind as we look at the results.

- In real use cases we *can't* look at the data and realise points are not really in a cluster; we have to take the clustering algorithm at its word.
- This is a small dataset, so poor performance here bodes very badly.

On to the clustering algorithms.

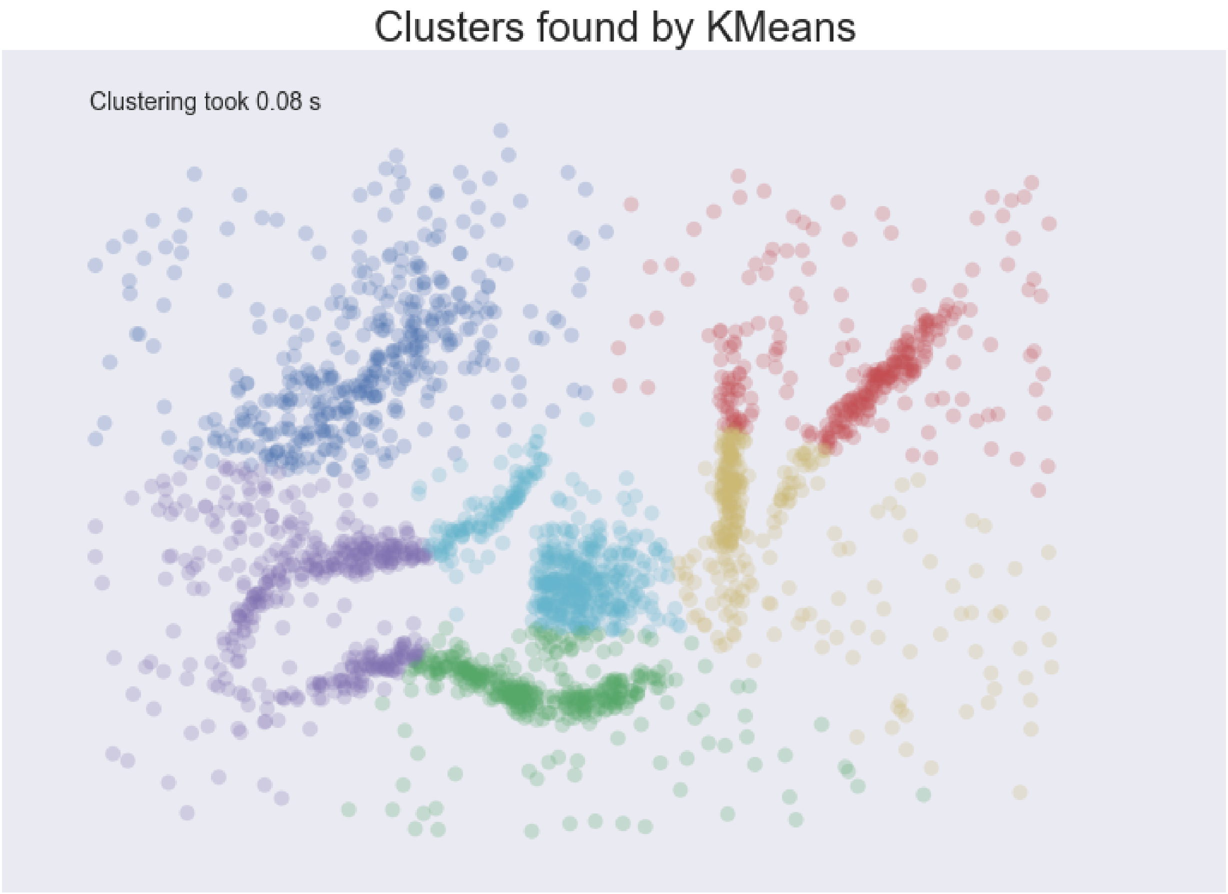
2.2.4 K-Means

K-Means is the 'go-to' clustering algorithm for many simply because it is fast, easy to understand, and available everywhere (there's an implementation in almost any statistical or machine learning tool you care to use). K-Means has a few problems however. The first is that it isn't a clustering algorithm, it is a partitioning algorithm. That is to say K-means doesn't 'find clusters' it partitions your dataset into as many (assumed to be globular) chunks as you ask for by attempting to minimize intra-partition distances. That leads to the second problem: you need to specify exactly how many clusters you expect. If you know a lot about your data then that is something you might expect to know. If, on the other hand, you are simply exploring a new dataset then 'number of clusters' is a hard parameter to have any good intuition for. The usually proposed solution is to run K-Means for many different 'number of clusters' values and score each clustering with some 'cluster goodness' measure (usually a variation on intra-cluster vs inter-cluster distances) and attempt to find an 'elbow'. If you've ever done this in practice you know that finding said elbow is usually not so easy, nor does it necessarily correlate as well with the actual 'natural' number of clusters as you might like. Finally K-Means is also dependent upon initialization; give it multiple different random starts and you can get multiple different clusterings. This does not engender much confidence in any individual clustering that may result.

So, in summary, here's how K-Means seems to stack up against out desiderata: * **Don't be wrong!**: K-means is going to throw points into clusters whether they belong or not; it also assumes you clusters are globular. K-Means scores very poorly on this point. * **Intuitive parameters**: If you have a good intuition for how many clusters the dataset your exploring has then great, otherwise you might have a problem. * **Stability**: Hopefully the clustering is stable for your data. Best to have many runs and check though. * **Performance**: This is K-Means big win. It's a simple algorithm and with the right tricks and optimizations can be made exceptionally efficient. There are few algorithms that can compete with K-Means for performance. If you have truly huge data then K-Means might be your only option.

But enough opinion, how does K-Means perform on our test dataset? Let's have look. We'll be generous and use our knowledge that there are six natural clusters and give that to K-Means.

```
plot_clusters(data, cluster.KMeans, (), {'n_clusters':6})
```



We see some interesting results. First, the assumption of perfectly globular clusters means that the natural clusters have been spliced and clumped into various more globular shapes. Worse, the noise points get lumped into clusters as well: in some cases, due to where relative cluster centers ended up, points very distant from a cluster get lumped in. Having noise pollute your clusters like this is particularly bad in an EDA world since they can easily mislead your intuition and understanding of the data. On a more positive note we completed clustering very quickly indeed, so at least we can be wrong quickly.

2.2.5 Affinity Propagation

Affinity Propagation is a newer clustering algorithm that uses a graph based approach to let points 'vote' on their preferred 'exemplar'. The end result is a set of cluster 'exemplars' from which we derive clusters by essentially doing what K-Means does and assigning each point to the cluster of it's nearest exemplar. Affinity Propagation has some advantages over K-Means. First of all the graph based exemplar voting means that the user doesn't need to specify the number of clusters. Second, due to how the algorithm works under the hood with the graph representation it allows for non-metric dissimilarities (i.e. we can have dissimilarities that don't obey the triangle inequality, or aren't symmetric). This second point is important if you are ever working with data isn't naturally embedded in a metric space of some kind; few clustering algorithms support, for example, non-symmetric dissimilarities. Finally Affinity Propagation does, at least, have better stability over runs (but not over parameter ranges!).

The weak points of Affinity Propagation are similar to K-Means. Since it partitions the data just like K-Means we expect to see the same sorts of problems, particularly with noisy data. While Affinity Propagation eliminates the need to specify the number of clusters, it has 'preference' and 'damping' parameters. Picking these parameters well can be difficult. The implementation in `sklearn` default preference to the median dissimilarity. This tends to result in a very large number of clusters. A better value is something smaller (or negative) but data dependent. Finally Affinity

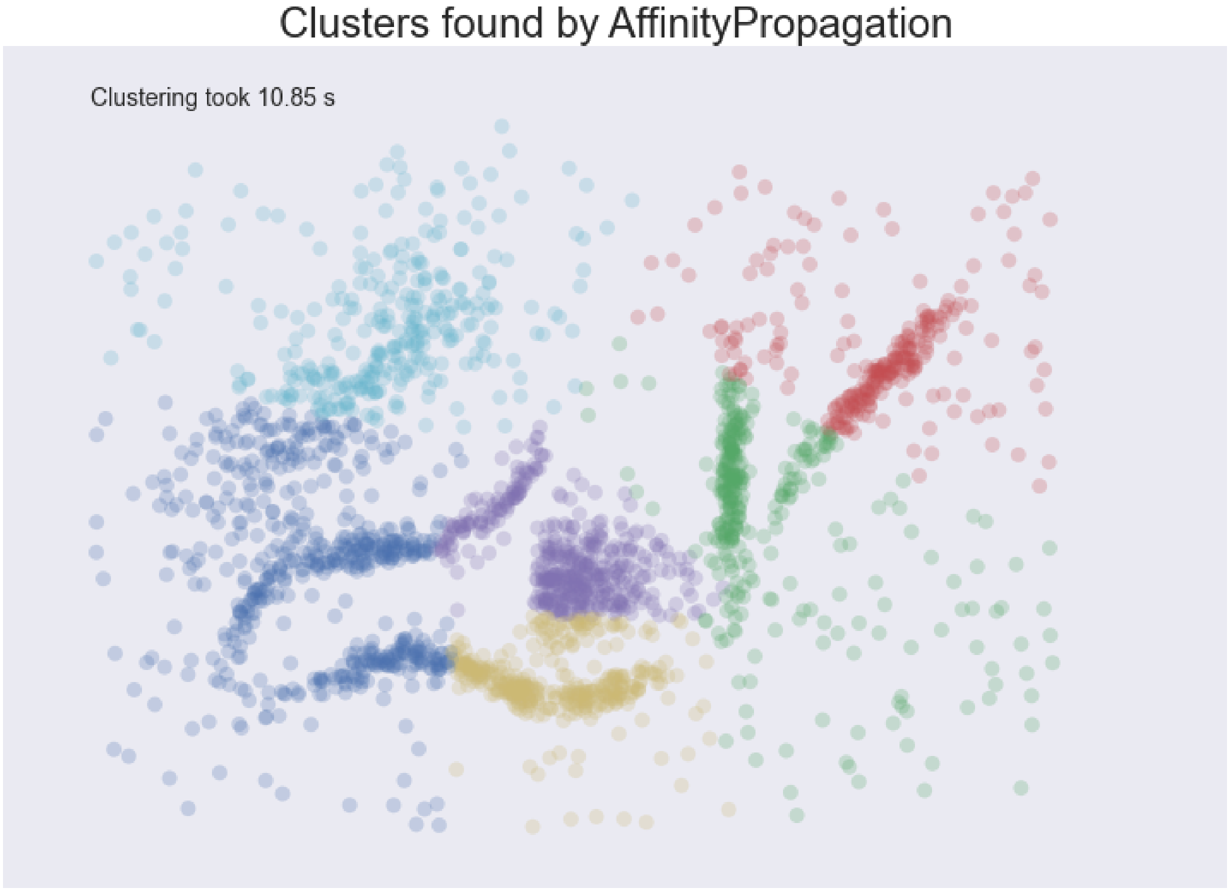
Propagation is *slow*; since it supports non-metric dissimilarities it can't take any of the shortcuts available to other algorithms, and the basic operations are expensive as data size grows.

So, in summary, over our desiderata we have:

- **Don't be wrong**: The same issues as K-Means; Affinity Propagation is going to throw points into clusters whether they belong or not; it also assumes you clusters are globular.
- **Intuitive Parameters**: It can be easier to guess at preference and damping than number of clusters, but since Affinity Propagation is quite sensitive to preference values it can be fiddly to get "right". This isn't really that much of an improvement over K-Means.
- **Stability**: Affinity Propagation is deterministic over runs.
- **Performance**: Affinity Propagation tends to be very slow. In practice running it on large datasets is essentially impossible without a carefully crafted and optimized implementation (i.e. not the default one available in `sklearn`).

And how does it look in practice on our chosen dataset? I've tried to select a preference and damping value that gives a reasonable number of clusters (in this case six) but feel free to play with the parameters yourself and see if you can come up with a better clustering.

```
plot_clusters(data, cluster.AffinityPropagation, (), {'preference':-5.0, 'damping':0.95})
```



The result is eerily similar to K-Means and has all the same problems. The globular clusters have lumped together splied parts of various 'natural' clusters. The noise points have been assigned to clusters regardless of being significant outliers. In other words, we'll have a very poor intuitive understanding of our data based on these 'clusters'. Worse still it took us several seconds to arrive at this unenlightening conclusion.

2.2.6 Mean Shift

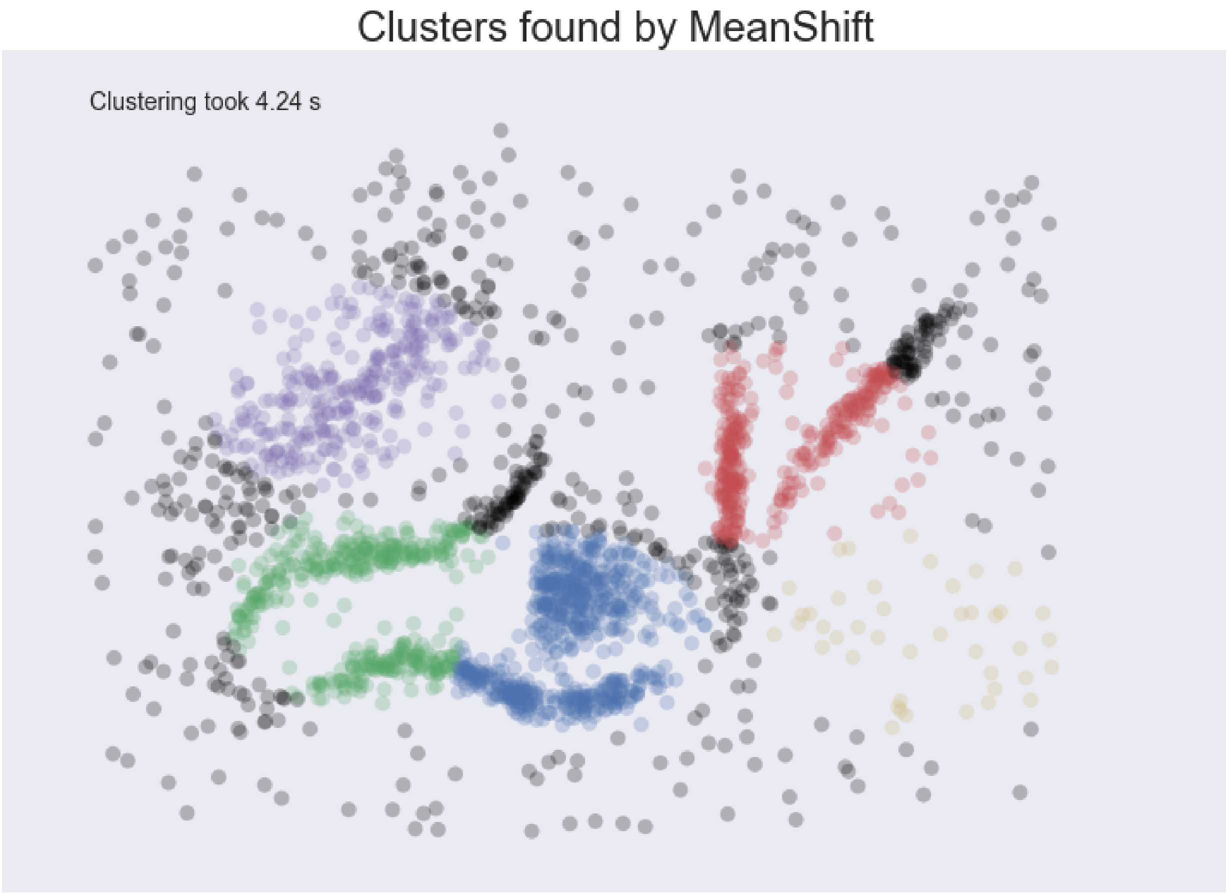
Mean shift is another option if you don't want to have to specify the number of clusters. It is centroid based, like K-Means and affinity propagation, but can return clusters instead of a partition. The underlying idea of the Mean Shift algorithm is that there exists some probability density function from which the data is drawn, and tries to place centroids of clusters at the maxima of that density function. It approximates this via kernel density estimation techniques, and the key parameter is then the bandwidth of the kernel used. This is easier to guess than the number of clusters, but may require some staring at, say, the distributions of pairwise distances between data points to choose successfully. The other issue (at least with the sklearn implementation) is that it is fairly slow depsite potentially having good scaling!

How does Mean Shift fare against out criteria? In principle proming, but in practice ...

- **Don't be wrong!**: Mean Shift doesn't cluster every point, but it still aims for globular clusters, and in practice it can return less than ideal results (see below for example). Without visual validation it can be hard to know how wrong it may be.
- **Intuitive parameters**: Mean Shift has more intuitive and meaningful parameters; this is certainly a strength.
- **Stability**: Mean Shift results can vary a lot as you vary the bandwidth parameter (which can make selection more difficult than it first appears. It also has a random initialisation, which means stability under runs can vary (if you reseed the random start).
- **Performance**: While Mean Shift has good scalability in principle (using ball trees) in practice the sklearn implementation is slow; this is a serious weak point for Mean Shift.

Let's see how it works on some actual data. I spent a while trying to find a good bandwidth value that resulted in a reasonable clustering. The choice below is about the best I found.

```
plot_clusters(data, cluster.MeanShift, (0.175,), {'cluster_all':False})
```



We at least aren't polluting our clusters with as much noise, but we certainly have dense regions left as noise and clusters that run across and split what seem like natural clusters. There is also the outlying yellow cluster group that doesn't make a lot of sense. Thus while Mean Shift had good promise, and is certainly better than K-Means, it's still short of our desiderata. Worse still it took over 4 seconds to cluster this small dataset!

2.2.7 Spectral Clustering

Spectral clustering can best be thought of as a graph clustering. For spatial data one can think of inducing a graph based on the distances between points (potentially a k-NN graph, or even a dense graph). From there spectral clustering will look at the eigenvectors of the Laplacian of the graph to attempt to find a good (low dimensional) embedding of the graph into Euclidean space. This is essentially a kind of manifold learning, finding a transformation of our original space so as to better represent manifold distances for some manifold that the data is assumed to lie on. Once we have the transformed space a standard clustering algorithm is run; with `sklearn` the default is K-Means. That means that the key for spectral clustering is the transformation of the space. Presuming we can better respect the manifold we'll get a better clustering – we need worry less about K-Means globular clusters as they are merely globular on the transformed space and not the original space. We unfortunately retain some of K-Means weaknesses: we still partition the data instead of clustering it; we have the hard to guess 'number of clusters' parameter; we have stability issues inherited from K-Means. Worse, if we operate on the dense graph of the distance matrix we have a very expensive initial step and sacrifice performance.

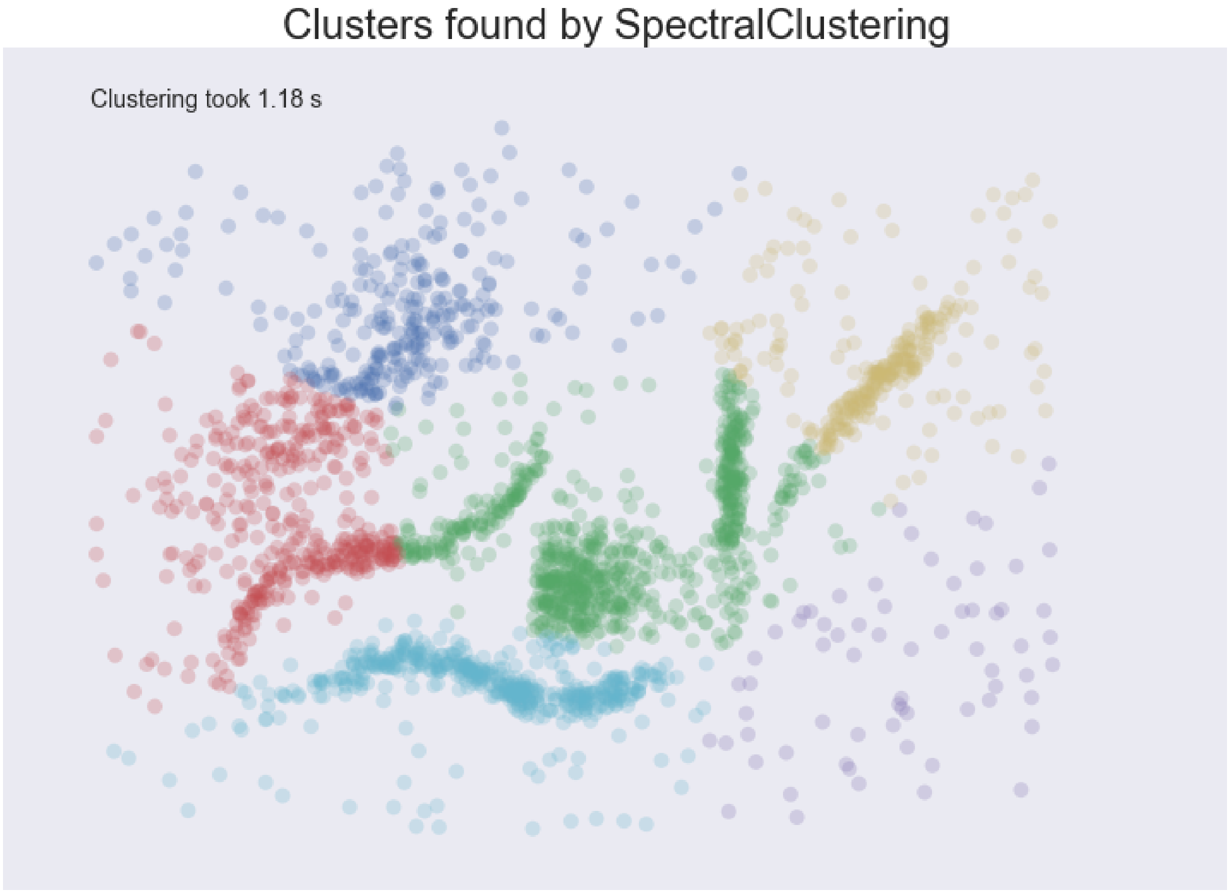
So, in summary:

- **Don't be wrong!**: We are less wrong, in that we don't have a purely globular cluster assumption; we do still have partitioning and hence are polluting clusters with noise, messing with our understanding of the clusters and hence the data.

- **Intuitive parameters**: We are no better than K-Means here; we have to know the correct number of clusters, or hope to guess by clustering over a range of parameter values and finding some way to pick the 'right one'.
- **Stability**: Slightly more stable than K-Means due to the transformation, but we still suffer from those issues.
- **Performance**: For spatial data we don't have a sparse graph (unless we prep one ourselves) so the result is a somewhat slower algorithm.

Let's have a look at how it operates on our test dataset. Again, we'll be generous and give it the six clusters to look for.

```
plot_clusters(data, cluster.SpectralClustering, (), {'n_clusters':6})
```



Spectral clustering performed *better* on the long thin clusters, but still ended up cutting some of them strangely and dumping parts of them in with other clusters. We also still have the issue of noise points polluting our clusters, so again our intuitions are going to be led astray. Performance was a distinct improvement of Affinity Propagation however. Over all we are doing better, but are still a long way from achieving our desiderata.

2.2.8 Agglomerative Clustering

Agglomerative clustering is really a suite of algorithms all based on the same idea. The fundamental idea is that you start with each point in it's own cluster and then, for each cluster, use some criterion to choose another cluster to merge with. Do this repeatedly until you have only one cluster and you get get a hierarchy, or binary tree, of clusters branching down to the last layer which has a leaf for each point in the dataset. The most basic version of this, single linkage, chooses the closest cluster to merge, and hence the tree can be ranked by distance as to when clusters merged/split. More complex variations use things like mean distance between clusters, or distance between cluster centroids etc. to determine which cluster to merge. Once you have a cluster hierarchy you can choose a level or cut (according to some criteria) and take the clusters at that level of the tree. For `sklearn` we usually choose a cut based on a ‘number of clusters’ parameter passed in.

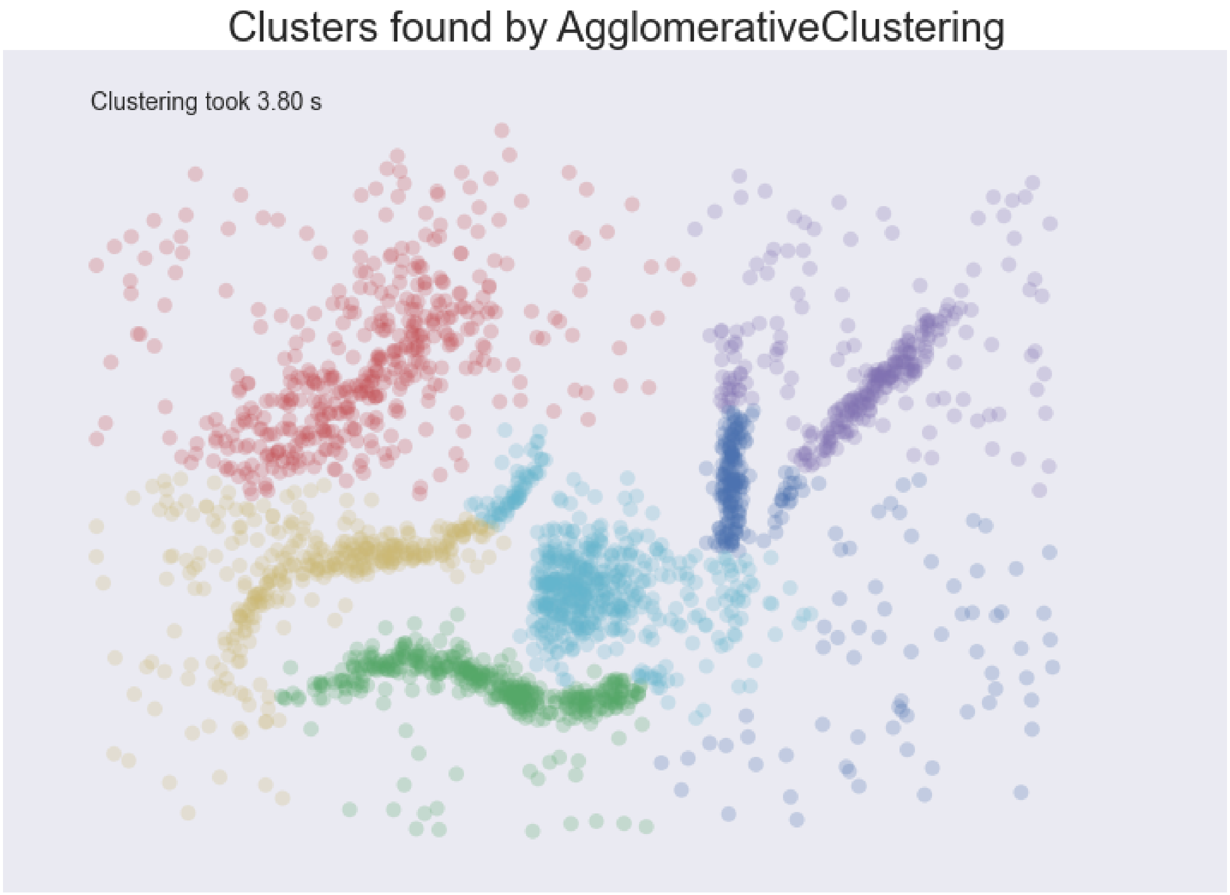
The advantage of this approach is that clusters can grow ‘following the underlying manifold’ rather than being presumed to be globular. You can also inspect the dendrogram of clusters and get more information about how clusters break down. On the other hand, if you want a flat set of clusters you need to choose a cut of the dendrogram, and that can be hard to determine. You can take the `sklearn` approach and specify a number of clusters, but as we’ve already discussed that isn’t a particularly intuitive parameter when you’re doing EDA. You can look at the dendrogram and try to pick a natural cut, but this is similar to finding the ‘elbow’ across varying `k` values for K-Means: in principle it’s fine, and the textbook examples always make it look easy, but in practice on messy real world data the ‘obvious’ choice is often far from obvious. We are also still partitioning rather than clustering the data, so we still have that persistent issue of noise polluting our clusters. Fortunately performance can be pretty good; the `sklearn` implementation is fairly slow, but `‘fastcluster` <https://pypi.python.org/pypi/fastcluster>‘__ provides high performance agglomerative clustering if that’s what you need.

So, in summary:

- **Don’t be wrong!**: We have gotten rid of the globular assumption, but we are still assuming that all the data belongs in clusters with no noise.
- **Intuitive parameters**: Similar to K-Means we are stuck choosing the number of clusters (not easy in EDA), or trying to discern some natural parameter value from a plot that may or may not have any obvious natural choices.
- **Stability**: Agglomerative clustering is stable across runs and the dendrogram shows how it varies over parameter choices (in a reasonably stable way), so stability is a strong point.
- **Performance**: Performance can be good if you get the right implementation.

So, let’s see it clustering data. I chose to provide the correct number of clusters (six) and use Ward as the linkage/merge method. This is a more robust method than say single linkage, but it does tend toward more globular clusters.

```
plot_clusters(data, cluster.AgglomerativeClustering, (), {'n_clusters':6, 'linkage':'ward'})
```



Similar to the spectral clustering we have handled the long thin clusters much better than K-Means or Affinity Propagation. We in fact improved on spectral clustering a bit on that front. We do still have clusters that contain parts of several different natural clusters, but those 'mis-clusterings' are smaller. We also still have all the noise points polluting our clusters. The end result is probably the best clustering we've seen so far, but given the mis-clustering and noise issues we are still not going to get as good an intuition for the data as we might reasonably hope for.

2.2.9 DBSCAN

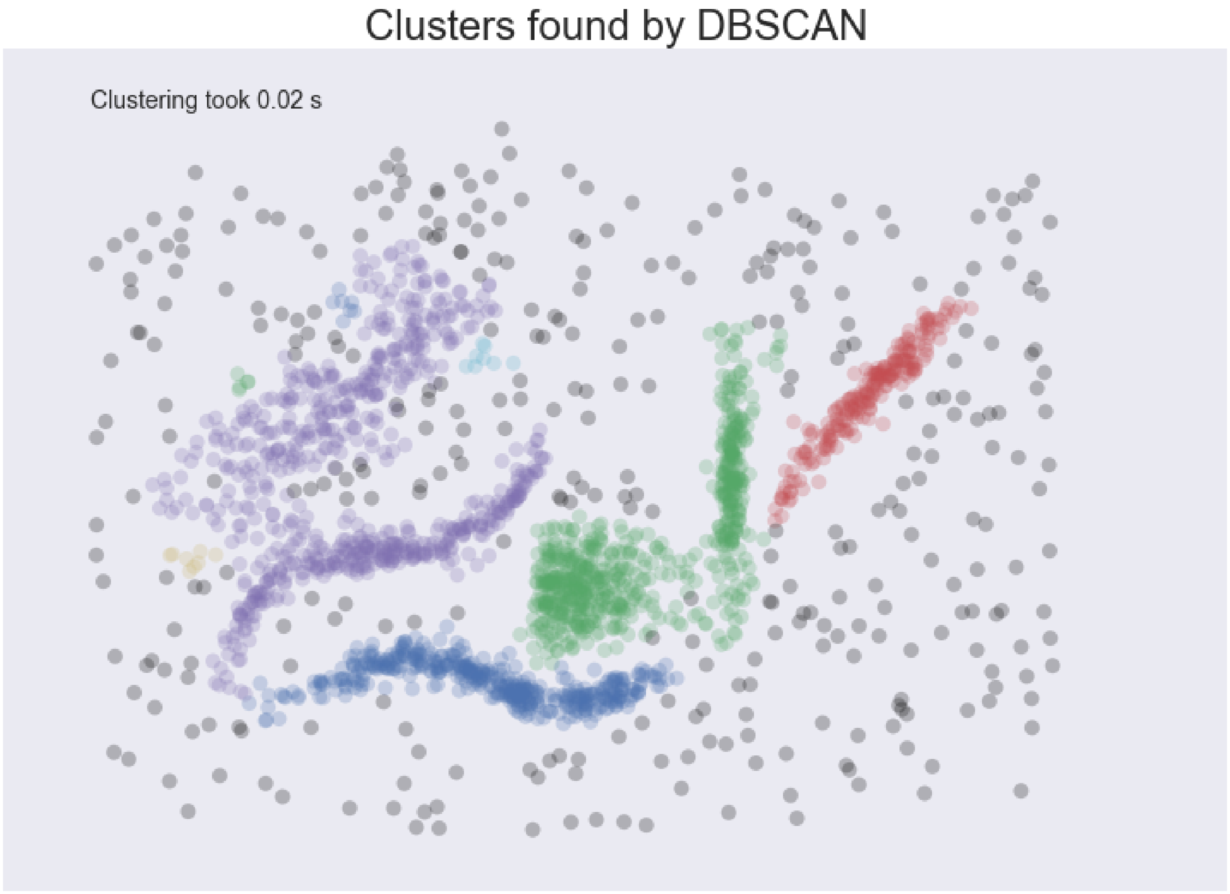
DBSCAN is a density based algorithm – it assumes clusters for dense regions. It is also the first actual clustering algorithm we've looked at: it doesn't require that every point be assigned to a cluster and hence doesn't partition the data, but instead extracts the 'dense' clusters and leaves sparse background classified as 'noise'. In practice DBSCAN is related to agglomerative clustering. As a first step DBSCAN transforms the space according to the density of the data: points in dense regions are left alone, while points in sparse regions are moved further away. Applying single linkage clustering to the transformed space results in a dendrogram, which we cut according to a distance parameter (called epsilon or `eps` in many implementations) to get clusters. Importantly any singleton clusters at that cut level are deemed to be 'noise' and left unclustered. This provides several advantages: we get the manifold following behaviour of agglomerative clustering, and we get actual clustering as opposed to partitioning. Better yet, since we can frame the algorithm in terms of local region queries we can use various tricks such as kdtrees to get exceptionally good performance and scale to dataset sizes that are otherwise unapproachable with algorithms other than K-Means. There are some catches however. Obviously epsilon can be hard to pick; you can do some data analysis and get a good guess, but the algorithm can be quite sensitive to the choice of the parameter. The density based transformation depends on another parameter (`min_samples` in `sklearn`). Finally the combination of `min_samples` and `eps` amounts to a choice of density and the clustering only finds clusters at or above that density; if your data has variable density clusters then DBSCAN is either going to miss them, split them up, or lump some of them together depending on your parameter choices.

So, in summary:

- **Don’t be wrong!**: Clusters don’t need to be globular, and won’t have noise lumped in; varying density clusters may cause problems, but that is more in the form of insufficient detail rather than explicitly wrong. DBSCAN is the first clustering algorithm we’ve looked at that actually meets the ‘Don’t be wrong!’ requirement.
- **Intuitive parameters**: Epsilon is a distance value, so you can survey the distribution of distances in your dataset to attempt to get an idea of where it should lie. In practice, however, this isn’t an especially intuitive parameter, nor is it easy to get right.
- **Stability**: DBSCAN is stable across runs (and to some extent subsampling if you re-parameterize well); stability over varying epsilon and min samples is not so good.
- **Performance**: This is DBSCAN’s other great strength; few clustering algorithms can tackle datasets as large as DBSCAN can.

So how does it cluster our test dataset? I played with a few epsilon values until I got somethign reasonable, but there was little science to this – getting the parameters right can be hard.

```
plot_clusters(data, cluster.DBSCAN, (), {'eps':0.025})
```



This is a pretty decent clustering; we’ve lumped natural clusters together a couple of times, but at least we didn’t carve them up to do so. We also picked up a few tiny clusters in amongst the large sparse cluster. These problems are artifacts of not handling variable density clusters – to get the sparser clusters to cluster we end up lumping some of the denser clusters with them; in the meantime the very sparse cluster is still broken up into several clusters. All in all we’re finally doing a decent job, but there’s still plenty of room for improvement.

2.2.10 HDBSCAN

HDBSCAN is a recent algorithm developed by some of the same people who write the original DBSCAN paper. Their goal was to allow varying density clusters. The algorithm starts off much the same as DBSCAN: we transform the space according to density, exactly as DBSCAN does, and perform single linkage clustering on the transformed space. Instead of taking an epsilon value as a cut level for the dendrogram however, a different approach is taken: the dendrogram is condensed by viewing splits that result in a small number of points splitting off as points 'falling out of a cluster'. This results in a smaller tree with fewer clusters that 'lose points'. That tree can then be used to select the most stable or persistent clusters. This process allows the tree to be cut at varying height, picking our varying density clusters based on cluster stability. The immediate advantage of this is that we can ave varying density clusters; the second benefit is that we have eliminated the epsilon parameter as we no longer need it to choose a cut of the dendrogram. Instead we have a new parameter `min_cluster_size` which is used to determine whether points are 'falling out of a cluster' or splitting to form two new clusters. This trades an unintuitive parameter for one that is not so hard to choose for EDA (what is the minimum size cluster I am willing to care about?).

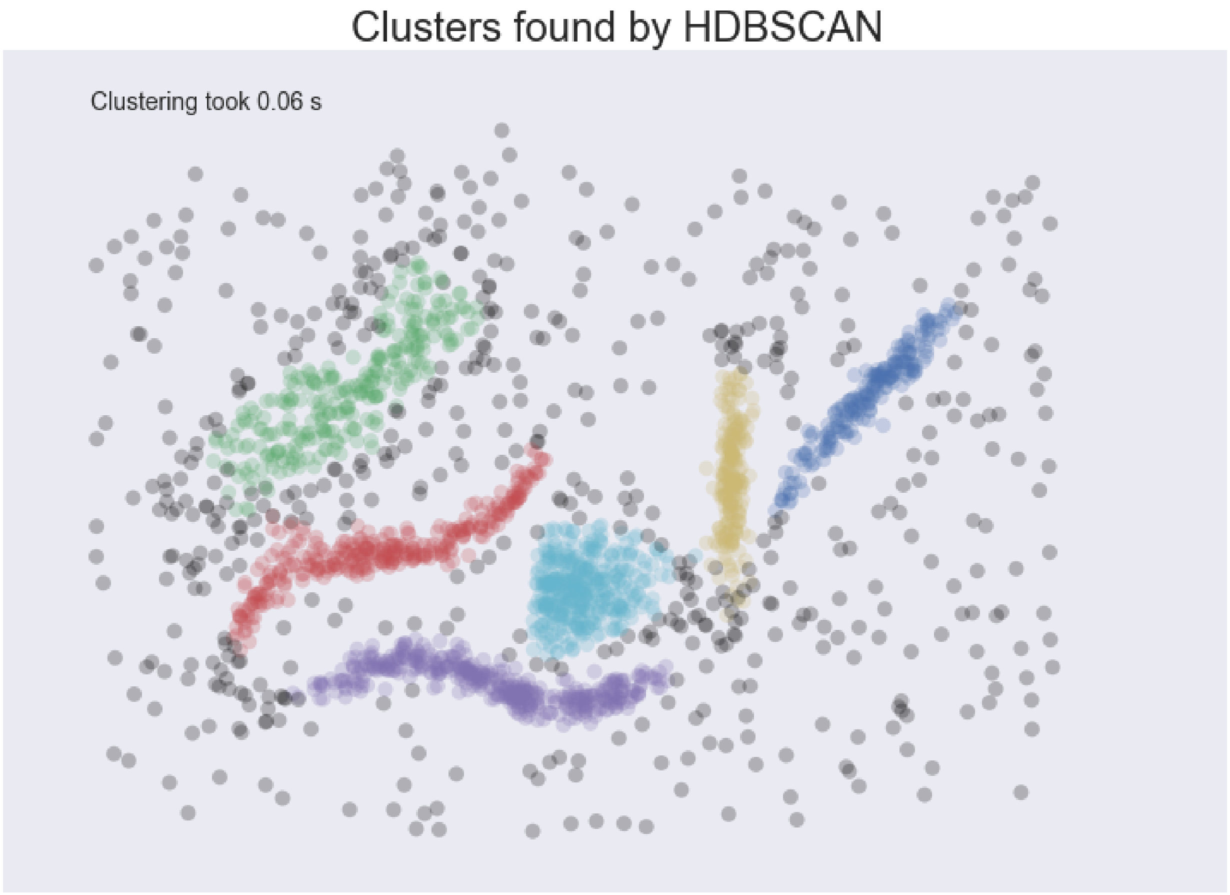
So, in summary:

- **Don't be wrong!**: We inherited all the benefits of DBSCAN and removed the varying density clusters issue. HDBSCAN is easily the strongest option on the 'Don't be wrong!' front.
- **Intuitive parameters**: Choosing a mimnimum cluster size is very reasonable. The only remaining parameter is `min_samples` inherited from DBSCAN for the density based space transformation. Sadly `min_samples` is not that intuitive; HDBSCAN is not that sensitive to it and we can choose some sensible defaults, but this remains the biggest weakness of the algorithm.
- **Stability**: HDBSCAN is stable over runs and subsampling (since the variable density clustering will still cluster sparser subsampled clusters with the same parameter choices), and has good stability over parameter choices.
- **Performance**: When implemented well HDBSCAN can be very efficient. The current implementation has similar performance to `fastcluster`'s agglomerative clustering (and will use `fastcluster` if it is available), but we expect future implementations that take advantage of newer data structure such as cover trees to scale significantly better.

How does HDBSCAN perform on our test dataset? Unfortunately HDBSCAN is not part of `sklearn`. Fortunately we can just import the hdbscan library and use it as if it were part of `sklearn`.

```
import hdbscan
```

```
plot_clusters(data, hdbscan.HDBSCAN, (), {'min_cluster_size':15})
```



I think the picture speaks for itself.

2.3 Benchmarking Performance and Scaling of Python Clustering Algorithms

There are a host of different clustering algorithms and implementations thereof for Python. The performance and scaling can depend as much on the implementation as the underlying algorithm. Obviously a well written implementation in C or C++ will beat a naive implementation on pure Python, but there is more to it than just that. The internals and data structures used can have a large impact on performance, and can even significanty change asymptotic performance. All of this means that, given some amount of data that you want to cluster your options as to algorithm and implementation maybe significantly constrained. I'm both lazy, and prefer empirical results for this sort of thing, so rather than analyzing the implementations and deriving asymptotic performance numbers for various implementations I'm just going to run everything and see what happens.

To begin with we need to get together all the clustering implementations, along with some plotting libraries so we can see what is going on once we've got data. Obviously this is not an exhaustive collection of clustering implementations, so if I've left off your favourite I apologise, but one has to draw a line somewhere.

The implementations being test are:

- Sklearn (which implements several algorithms):
- K-Means clustering
- DBSCAN clustering

- Agglomerative clustering
- Spectral clustering
- Affinity Propagation
- Scipy (which provides basic algorithms):
- K-Means clustering
- Agglomerative clustering
- Fastcluster (which provides very fast agglomerative clustering in C++)
- DeBaCl (Density Based Clustering; similar to a mix of DBSCAN and Agglomerative)
- HDBSCAN (A robust hierarchical version of DBSCAN)

Obviously a major factor in performance will be the algorithm itself. Some algorithms are simply slower – often, but not always, because they are doing more work to provide a better clustering.

```
import hdbscan
import debacl
import fastcluster
import sklearn.cluster
import scipy.cluster
import sklearn.datasets
import numpy as np
import pandas as pd
import time
import matplotlib.pyplot as plt
import seaborn as sns
%matplotlib inline
sns.set_context('poster')
sns.set_palette('Paired', 10)
sns.set_color_codes()
```

```
:0: FutureWarning: IPython widgets are experimental and may change in the future.
```

Now we need some benchmarking code at various dataset sizes. Because some clustering algorithms have performance that can vary quite a lot depending on the exact nature of the dataset we'll also need to run several times on randomly generated datasets of each size so as to get a better idea of the average case performance.

We also need to generalise over algorithms which don't necessarily all have the same API. We can resolve that by taking a clustering function, argument tuple and keywords dictionary to let us do semi-arbitrary calls (fortunately all the algorithms do at least take the dataset to cluster as the first parameter).

Finally some algorithms scale poorly, and I don't want to spend forever doing clustering of random datasets so we'll cap the maximum time an algorithm can use; once it has taken longer than max time we'll just abort there and leave the remaining entries in our datasize by samples matrix unfilled.

In the end this all amounts to a fairly straightforward set of nested loops (over datasizes and number of samples) with calls to sklearn to generate mock data and the clustering function inside a timer. Add in some early abort and we're done.

```
def benchmark_algorithm(dataset_sizes, cluster_function, function_args, function_kwds,
                        dataset_dimension=10, dataset_n_clusters=10, max_time=45, sample_size=2):

    # Initialize the result with NaNs so that any unfilled entries
    # will be considered NULL when we convert to a pandas dataframe at the end
    result = np.nan * np.ones((len(dataset_sizes), sample_size))
    for index, size in enumerate(dataset_sizes):
        for s in range(sample_size):
            # Use sklearns make_blobs to generate a random dataset with specified size
            # dimension and number of clusters
            data, labels = sklearn.datasets.make_blobs(n_samples=size,
                                                       n_features=dataset_dimension,
                                                       centers=dataset_n_clusters)

            # Start the clustering with a timer
            start_time = time.time()
            cluster_function(data, *function_args, **function_kwds)
            time_taken = time.time() - start_time

            # If we are taking more than max_time then abort -- we don't
            # want to spend excessive time on slow algorithms
            if time_taken > max_time:
                result[index, s] = time_taken
                return pd.DataFrame(np.vstack([dataset_sizes.repeat(sample_size),
                                               result.flatten()]).T, columns=['x','y'])
            else:
                result[index, s] = time_taken

    # Return the result as a dataframe for easier handling with seaborn afterwards
    return pd.DataFrame(np.vstack([dataset_sizes.repeat(sample_size),
                                   result.flatten()]).T, columns=['x','y'])
```

2.3.1 Comparison of all ten implementations

Now we need a range of dataset sizes to test out our algorithm. Since the scaling performance is wildly different over the ten implementations we're going to look at it will be beneficial to have a number of very small dataset sizes, and increasing spacing as we get larger, spanning out to 32000 datapoints to cluster (to begin with). Numpy provides convenient ways to get this done via `arange` and vector multiplication. We'll start with step sizes of 500, then shift to steps of 1000 past 3000 datapoints, and finally steps of 2000 past 6000 datapoints.

```
dataset_sizes = np.hstack([np.arange(1, 6) * 500, np.arange(3,7) * 1000, np.arange(4,17) * 2000])
```

Now it is just a matter of running all the clustering algorithms via our benchmark function to collect up all the requsite data. This could be prettier, rolled up into functions appropriately, but sometimes brute force is good enough. More importantly (for me) since this can take a significant amount of compute time, I wanted to be able to comment out algorithms that were slow or I was uninterested in easily. Which brings me to a warning for you the reader and potential user of the notebook: this next step is very expensive. We are running ten different clustering algorithms multiple times each on twenty two different dataset sizes – and some of the clustering algorithms are slow (we are capping out at forty five seconds per run). That means that the next cell can take an hour or more to run. That doesn't mean "Don't try this at home" (I actually encourage you to try this out yourself and play with dataset parameters and clustering parameters) but it does mean you should be patient if you're going to!

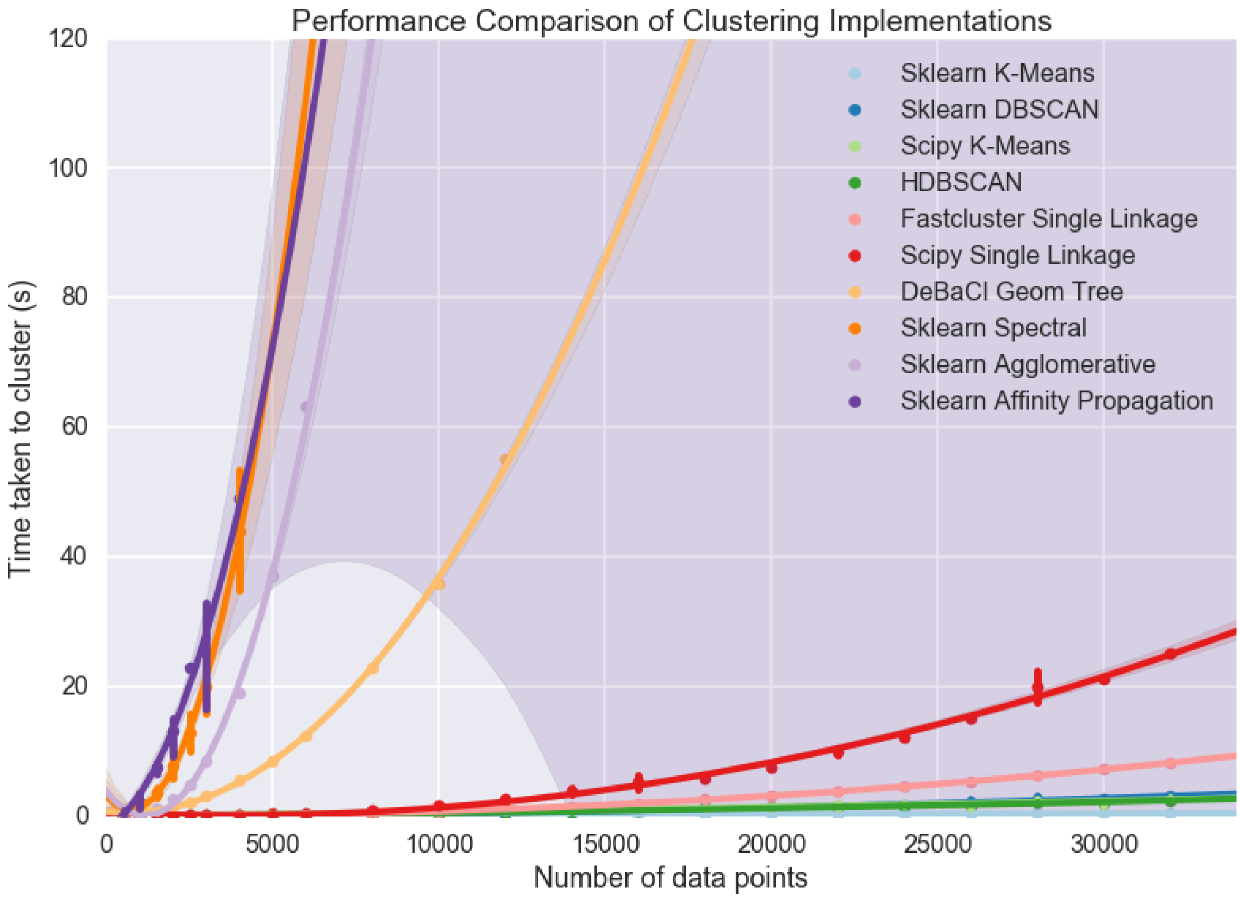
```
k_means = sklearn.cluster.KMeans(10)
k_means_data = benchmark_algorithm(dataset_sizes, k_means.fit, (), {})

dbscan = sklearn.cluster.DBSCAN(eps=1.25)
dbscan_data = benchmark_algorithm(dataset_sizes, dbscan.fit, (), {})

scipy_k_means_data = benchmark_algorithm(dataset_sizes,
                                         scipy.cluster.vq.kmeans, (10,), {})

scipy_single_data = benchmark_algorithm(dataset_sizes,
                                        scipy.cluster.hierarchy.single, (), {})
```

```
fastclust_data = benchmark_algorithm(dataset_sizes,
                                     fastcluster.linkage_vector, (), {})

hdbscan_ = hdbscan.HDBSCAN()
hdbscan_data = benchmark_algorithm(dataset_sizes, hdbscan_.fit, (), {})

debacl_data = benchmark_algorithm(dataset_sizes,
                                  debacl.geom_tree.geomTree, (5, 5), {'verbose':False})

agglomerative = sklearn.cluster.AgglomerativeClustering(10)
agg_data = benchmark_algorithm(dataset_sizes,
                               agglomerative.fit, (), {}, sample_size=4)

spectral = sklearn.cluster.SpectralClustering(10)
spectral_data = benchmark_algorithm(dataset_sizes,
                                    spectral.fit, (), {}, sample_size=6)

affinity_prop = sklearn.cluster.AffinityPropagation()
ap_data = benchmark_algorithm(dataset_sizes,
                              affinity_prop.fit, (), {}, sample_size=3)
```

Now we need to plot the results so we can see what is going on. The catch is that we have several datapoints for each dataset size and ultimately we would like to try and fit a curve through all of it to get the general scaling trend. Fortunately seaborn comes to the rescue here by providing `regplot` which plots a regression through a dataset, supports higher order regression (we should probably use order two as most algorithms are effectively quadratic) and handles multiple datapoints for each x-value cleanly (using the `x_estimator` keyword to put a point at the mean and draw an error bar to cover the range of data).

```
sns.regplot(x='x', y='y', data=k_means_data, order=2,
            label='Sklearn K-Means', x_estimator=np.mean)
sns.regplot(x='x', y='y', data=dbscan_data, order=2,
            label='Sklearn DBSCAN', x_estimator=np.mean)
sns.regplot(x='x', y='y', data=scipy_k_means_data, order=2,
            label='Scipy K-Means', x_estimator=np.mean)
sns.regplot(x='x', y='y', data=hdbscan_data, order=2,
            label='HDBSCAN', x_estimator=np.mean)
sns.regplot(x='x', y='y', data=fastclust_data, order=2,
            label='Fastcluster Single Linkage', x_estimator=np.mean)
sns.regplot(x='x', y='y', data=scipy_single_data, order=2,
            label='Scipy Single Linkage', x_estimator=np.mean)
sns.regplot(x='x', y='y', data=debacl_data, order=2,
            label='DeBaCl Geom Tree', x_estimator=np.mean)
sns.regplot(x='x', y='y', data=spectral_data, order=2,
            label='Sklearn Spectral', x_estimator=np.mean)
sns.regplot(x='x', y='y', data=agg_data, order=2,
            label='Sklearn Agglomerative', x_estimator=np.mean)
sns.regplot(x='x', y='y', data=ap_data, order=2,
            label='Sklearn Affinity Propagation', x_estimator=np.mean)
plt.gca().axis([0, 34000, 0, 120])
plt.gca().set_xlabel('Number of data points')
plt.gca().set_ylabel('Time taken to cluster (s)')
plt.title('Performance Comparison of Clustering Implementations')
plt.legend()
```

```
<matplotlib.legend.Legend at 0x1125dee50>
```



A few features stand out. First of all there appear to be essentially two classes of implementation, with DeBaCl being an odd case that falls in the middle. The fast implementations tend to be implementations of single linkage agglomerative clustering, K-means, and DBSCAN. The slow cases are largely from sklearn and include agglomerative clustering (in this case using Ward instead of single linkage).

For practical purposes this means that if you have much more than 10000 datapoints your clustering options are significantly constrained: sklearn spectral, agglomerative and affinity propagation are going to take far too long. DeBaCl may still be an option, but given that the hdbscan library provides “robust single linkage clustering” equivalent to what DeBaCl is doing (and with effectively the same runtime as hdbscan as it is a subset of that algorithm) it is probably not the best choice for large dataset sizes.

So let’s drop out those slow algorithms so we can scale out a little further and get a closer look at the various algorithms that managed 32000 points in under thirty seconds. There is almost undoubtedly more to learn as we get ever larger dataset sizes.

2.3.2 Comparison of fast implementations

Let’s compare the six fastest implementations now. We can scale out a little further as well; based on the curves above it looks like we should be able to comfortably get to 60000 data points without taking much more than a minute per run. We can also note that most of these implementations weren’t that noisy so we can get away with a single run per dataset size.

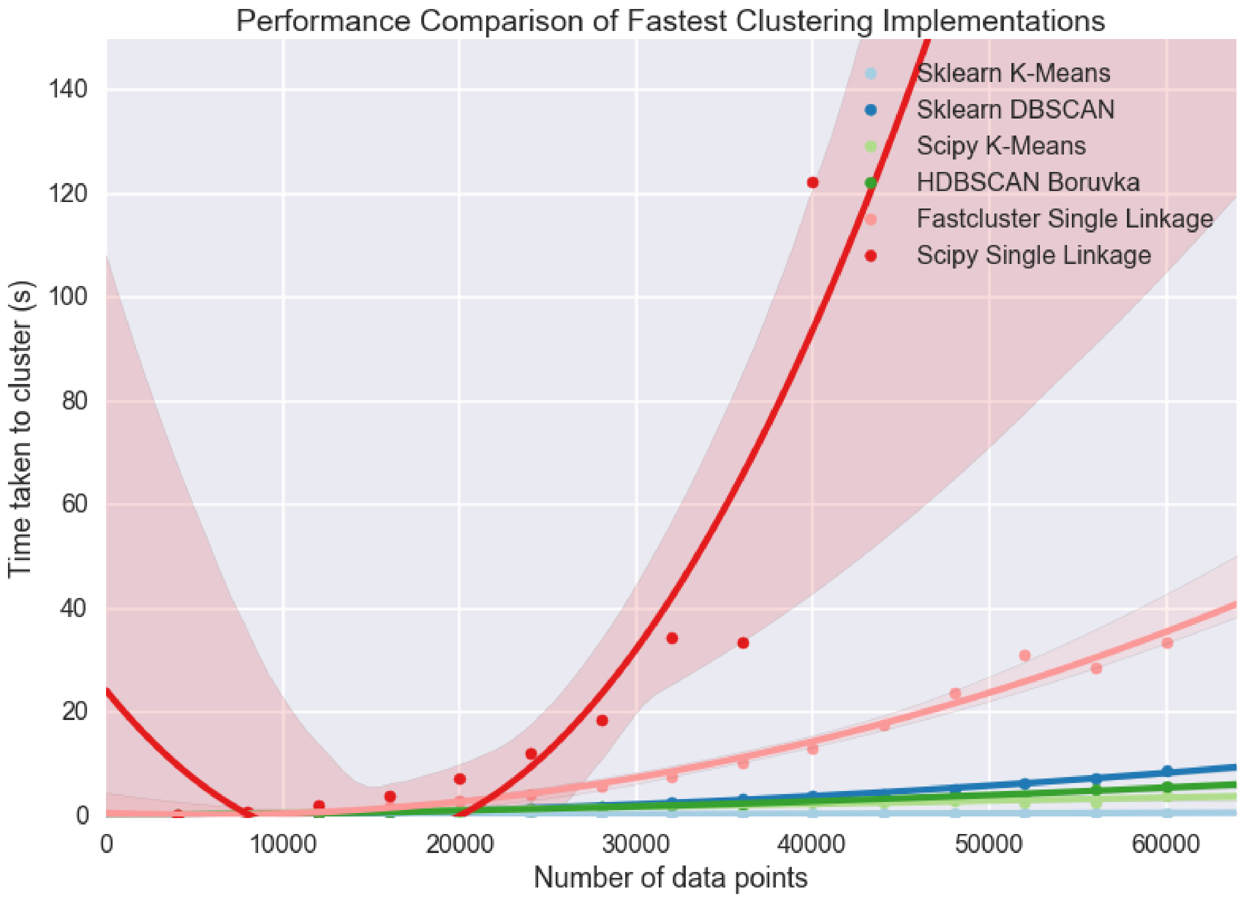
```
large_dataset_sizes = np.arange(1,16) * 4000

hdbscan_boruvka = hdbscan.HDBSCAN(algorithm='boruvka_kdtree')
```

```
large_hdbscan_boruvka_data = benchmark_algorithm(large_dataset_sizes,
                                         hdbscan_boruvka.fit, (), {},
                                                 max_time=90, sample_size=1)

k_means = sklearn.cluster.KMeans(10)
large_k_means_data = benchmark_algorithm(large_dataset_sizes,
                                         k_means.fit, (), {},
                                         max_time=90, sample_size=1)

dbscan = sklearn.cluster.DBSCAN(eps=1.25, min_samples=5)
large_dbscan_data = benchmark_algorithm(large_dataset_sizes,
                                        dbscan.fit, (), {},
                                        max_time=90, sample_size=1)

large_fastclust_data = benchmark_algorithm(large_dataset_sizes,
                                           fastcluster.linkage_vector, (), {},
                                           max_time=90, sample_size=1)

large_scipy_k_means_data = benchmark_algorithm(large_dataset_sizes,
                                               scipy.cluster.vq.kmeans, (10,), {},
                                               max_time=90, sample_size=1)

large_scipy_single_data = benchmark_algorithm(large_dataset_sizes,
                                              scipy.cluster.hierarchy.single, (), {},
                                              max_time=90, sample_size=1)
```

Again we can use seaborn to do curve fitting and plotting, exactly as before.

```
sns.regplot(x='x', y='y', data=large_k_means_data, order=2,
            label='Sklearn K-Means', x_estimator=np.mean)
sns.regplot(x='x', y='y', data=large_dbscan_data, order=2,
            label='Sklearn DBSCAN', x_estimator=np.mean)
sns.regplot(x='x', y='y', data=large_scipy_k_means_data, order=2,
            label='Scipy K-Means', x_estimator=np.mean)
sns.regplot(x='x', y='y', data=large_hdbscan_boruvka_data, order=2,
            label='HDBSCAN Boruvka', x_estimator=np.mean)
sns.regplot(x='x', y='y', data=large_fastclust_data, order=2,
            label='Fastcluster Single Linkage', x_estimator=np.mean)
sns.regplot(x='x', y='y', data=large_scipy_single_data, order=2,
            label='Scipy Single Linkage', x_estimator=np.mean)

plt.gca().axis([0, 64000, 0, 150])
plt.gca().set_xlabel('Number of data points')
plt.gca().set_ylabel('Time taken to cluster (s)')
plt.title('Performance Comparison of Fastest Clustering Implementations')
plt.legend()
```

```
<matplotlib.legend.Legend at 0x116038bd0>
```



Clearly something has gone woefully wrong with the curve fitting for the scipy single linkage implementation, but what exactly? If we look at the raw data we can see.

```
large_scipy_single_data.tail(10)
```

It seems that at around 44000 points we hit a wall and the runtimes spiked. A hint is that I'm running this on a laptop with 8GB of RAM. Both single linkage algorithms use `scipy.spatial.pdist` to compute pairwise distances between points, which returns an array of shape `(n(n-1)/2, 1)` of doubles. A quick computation shows that that array of distances is quite large once we nave 44000 points:

```
size_of_array = 44000 * (44000 - 1) / 2          # from pdist documentation
bytes_in_array = size_of_array * 8               # Since doubles use 8 bytes
gigabytes_used = bytes_in_array / (1024.0 ** 3)  # divide out to get the number of GB
gigabytes_used
```

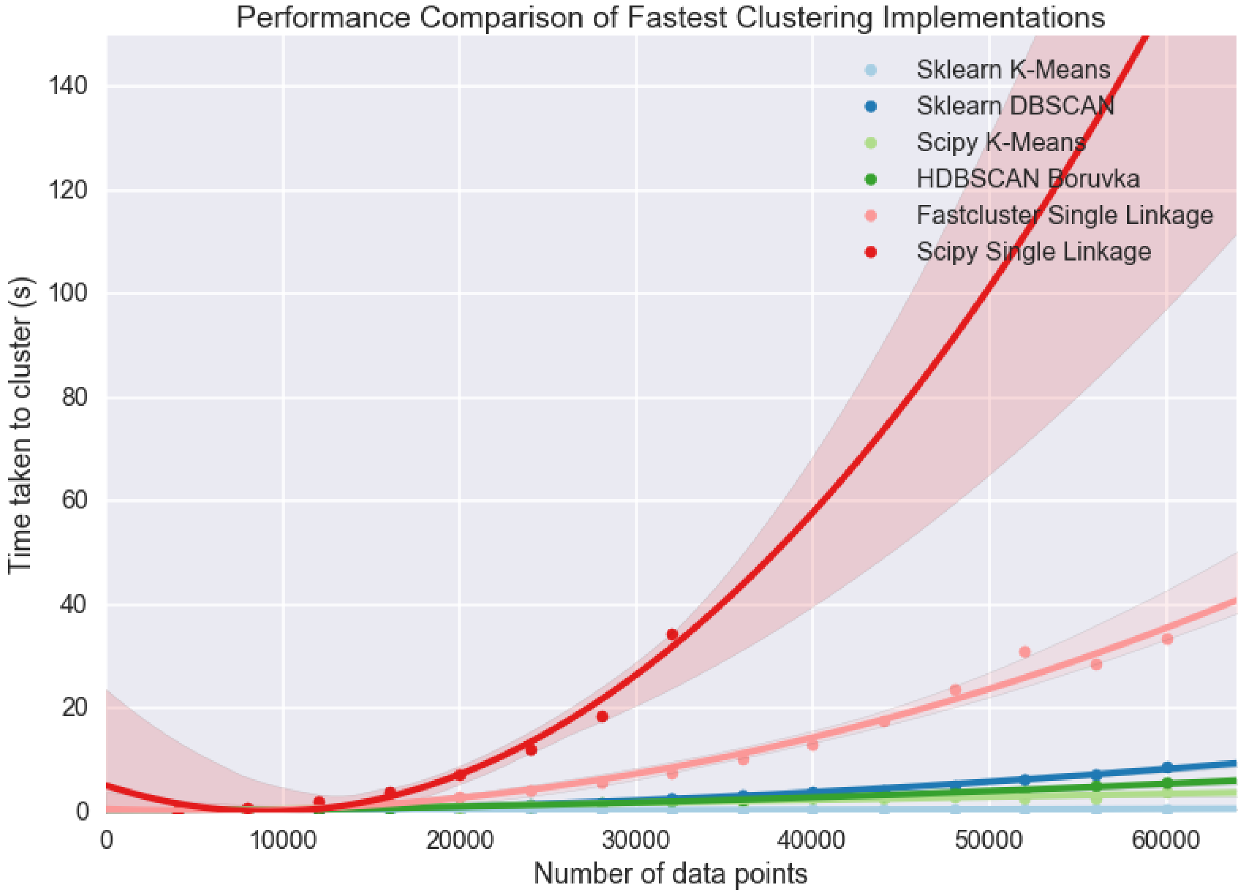
```
7.211998105049133
```

If we assume that my laptop is keeping much other than that distance array in RAM then clearly we are going to spend time paging out the distance array to disk and back and hence we will see the runtimes increase dramatically as we become disk IO bound. If we just leave off the last element we can get a better idea of the curve, but keep in mind that the scipy single linkage implementation does not scale past a limit set by your available RAM.

```
sns.regplot(x='x', y='y', data=large_k_means_data, order=2,
            label='Sklearn K-Means', x_estimator=np.mean)
sns.regplot(x='x', y='y', data=large_dbscan_data, order=2,
            label='Sklearn DBSCAN', x_estimator=np.mean)
sns.regplot(x='x', y='y', data=large_scipy_k_means_data, order=2,
            label='Scipy K-Means', x_estimator=np.mean)
sns.regplot(x='x', y='y', data=large_hdbscan_boruvka_data, order=2,
            label='HDBSCAN Boruvka', x_estimator=np.mean)
sns.regplot(x='x', y='y', data=large_fastclust_data, order=2,
            label='Fastcluster Single Linkage', x_estimator=np.mean)
sns.regplot(x='x', y='y', data=large_scipy_single_data[:8], order=2,
            label='Scipy Single Linkage', x_estimator=np.mean)

plt.gca().axis([0, 64000, 0, 150])
plt.gca().set_xlabel('Number of data points')
plt.gca().set_ylabel('Time taken to cluster (s)')
plt.title('Performance Comparison of Fastest Clustering Implementations')
plt.legend()
```

```
/Users/leland/.conda/envs/hdbscan_dev/lib/python2.7/site-packages/numpy/lib/polynomial.py:595: RankWa
  warnings.warn(msg, RankWarning)
```

```
<matplotlib.legend.Legend at 0x118843210>
```



If we're looking for scaling we can write off the scipy single linkage implementation – if even we didn't hit the RAM limit the $O(n^2)$ scaling is going to quickly catch up with us. Fastcluster has the same asymptotic scaling, but is heavily optimized to being the constant down much lower – at this point it is still keeping close to the faster algorithms. It's asymtotics will still catch up with it eventually however.

In practice this is going to mean that for larger datasets you are going to be very constrained in what algorithms you can apply: if you get enough datapoints only K-Means, DBSCAN, and HDBSCAN will be left. This is somewhat disappointing, paritcularly as K-Means is not a particularly good clustering algorithm, paricularly for exploratory data analysis.

With this in mind it is worth looking at how these last several implementations perform at much larger sizes, to see, for example, when fastscluster starts to have its asymptotic complexity start to pull it away.

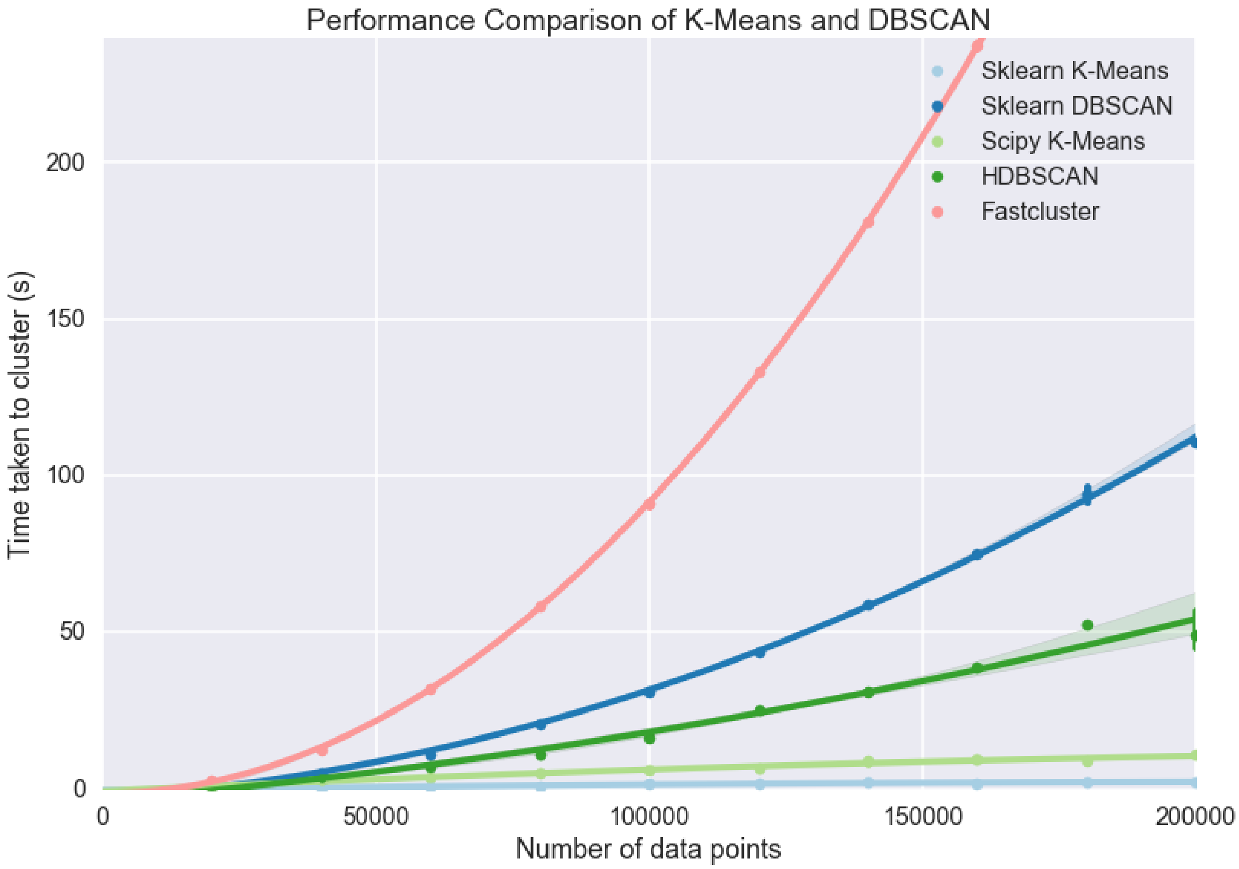
2.3.3 Comparison of high performance implementations

At this point we can scale out to 200000 datapoints easily enough, so let's push things at least that far so we can start to really see scaling effects.

```
huge_dataset_sizes = np.arange(1,11) * 20000

k_means = sklearn.cluster.KMeans(10)
huge_k_means_data = benchmark_algorithm(huge_dataset_sizes,
                                        k_means.fit, (), {},
                                        max_time=120, sample_size=2, dataset_dimension=10)

dbscan = sklearn.cluster.DBSCAN(eps=1.5)
huge_dbscan_data = benchmark_algorithm(huge_dataset_sizes,
                                       dbscan.fit, (), {},
                                       max_time=120, sample_size=2, dataset_dimension=10)

huge_scipy_k_means_data = benchmark_algorithm(huge_dataset_sizes,
                                              scipy.cluster.vq.kmeans, (10,), {},
                                              max_time=120, sample_size=2, dataset_dimension=10)

hdbscan_boruvka = hdbscan.HDBSCAN(algorithm='boruvka_kdtree')
huge_hdbscan_data = benchmark_algorithm(huge_dataset_sizes,
                                       hdbscan_boruvka.fit, (), {},
                                        max_time=240, sample_size=4, dataset_dimension=10)

huge_fastcluster_data = benchmark_algorithm(huge_dataset_sizes,
                                            fastcluster.linkage_vector, (), {},
                                            max_time=240, sample_size=2, dataset_dimension=10)
```

```
sns.regplot(x='x', y='y', data=huge_k_means_data, order=2,
            label='Sklearn K-Means', x_estimator=np.mean)
sns.regplot(x='x', y='y', data=huge_dbscan_data, order=2,
            label='Sklearn DBSCAN', x_estimator=np.mean)
sns.regplot(x='x', y='y', data=huge_scipy_k_means_data, order=2,
            label='Scipy K-Means', x_estimator=np.mean)
sns.regplot(x='x', y='y', data=huge_hdbscan_data, order=2,
            label='HDBSCAN', x_estimator=np.mean)
sns.regplot(x='x', y='y', data=huge_fastcluster_data, order=2,
            label='Fastcluster', x_estimator=np.mean)


plt.gca().axis([0, 200000, 0, 240])
plt.gca().set_xlabel('Number of data points')
plt.gca().set_ylabel('Time taken to cluster (s)')
plt.title('Performance Comparison of K-Means and DBSCAN')
plt.legend()
```

```
<matplotlib.legend.Legend at 0x11d2aff50>
```



Now the some differences become clear. The asymptotic complexity starts to kick in with fastcluster failing to keep up. In turn HDBSCAN and DBSCAN, while having sub-$O(n^2)$ complexity, can't achieve $O(n \log(n))$ at this dataset dimension, and start to curve upward precipitously. Finally it demonstrates again how much of a difference implementation can make: the sklearn implementation of K-Means is far better than the scipy implementation. Since HDBSCAN clustering is a lot better than K-Means (unless you have good reasons to assume that the clusters partition your data and are all drawn from Gaussian distributions) and the scaling is still pretty good I would suggest that unless you have a truly stupendous amount of data you wish to cluster then the HDBSCAN implementation is a good choice.

2.3.4 But should I get a coffee?

So we know which implementations scale and which don't; a more useful thing to know in practice is, given a dataset, what can I run interactively? What can I run while I go and grab some coffee? How about a run over lunch? What if I'm willing to wait until I get in tomorrow morning? Each of these represent significant breaks in productivity – once you aren't working interactively anymore your productivity drops measurably, and so on.

We can build a table for this. To start we'll need to be able to approximate how long a given clustering implementation will take to run. Fortunately we already gathered a lot of that data; if we load up the `statsmodels` package we can fit the data (with a quadratic or $n \log n$ fit depending on the implementation; DBSCAN and HDBSCAN get caught here, since while they are under $O(n^2)$ scaling, they don't have an easily described model, so I'll model them as n^2 for now) and use the resulting model to make our predictions. Obviously this has some caveats: if you fill your RAM with a distance matrix your runtime isn't going to fit the curve.

I've hand built a `time_samples` list to give a reasonable set of potential data sizes that are nice and human readable. After that we just need a function to fit and build the curves.

```
import statsmodels.formula.api as sm

time_samples = [1000, 2000, 5000, 10000, 25000, 50000, 75000, 100000, 250000, 500000, 750000,
               1000000, 2500000, 5000000, 10000000, 50000000, 100000000, 500000000, 1000000000]

def get_timing_series(data, quadratic=True):
    if quadratic:
        data['x_squared'] = data.x**2
        model = sm.ols('y ~ x + x_squared', data=data).fit()
        predictions = [model.params.dot([1.0, i, i**2]) for i in time_samples]
        return pd.Series(predictions, index=pd.Index(time_samples))
    else: # assume n log(n)
        data['xlogx'] = data.x * np.log(data.x)
        model = sm.ols('y ~ x + xlogx', data=data).fit()
        predictions = [model.params.dot([1.0, i, i*np.log(i)]) for i in time_samples]
        return pd.Series(predictions, index=pd.Index(time_samples))
```

Now we run that for each of our pre-existing datasets to extrapolate out predicted performance on the relevant dataset sizes. A little pandas wrangling later and we've produced a table of roughly how large a dataset you can tackle in each time frame with each implementation. I had to leave out the scipy KMeans timings because the noise in timing results caused the model to be unrealistic at larger data sizes. Note how the $O(n \log n)$ algorithms utterly dominate here. In the meantime, for medium sizes data sets you can still get quite a lot done with HDBSCAN.

```
ap_timings = get_timing_series(ap_data)
spectral_timings = get_timing_series(spectral_data)
agg_timings = get_timing_series(agg_data)
debacl_timings = get_timing_series(debacl_data)
fastclust_timings = get_timing_series(large_fastclust_data.ix[:10,:].copy())
scipy_single_timings = get_timing_series(large_scipy_single_data.ix[:10,:].copy())
hdbscan_boruvka = get_timing_series(huge_hdbscan_data, quadratic=True)
#scipy_k_means_timings = get_timing_series(huge_scipy_k_means_data, quadratic=False)
dbscan_timings = get_timing_series(huge_dbscan_data, quadratic=True)
k_means_timings = get_timing_series(huge_k_means_data, quadratic=False)

timing_data = pd.concat([ap_timings, spectral_timings, agg_timings, debacl_timings,
                            scipy_single_timings, fastclust_timings, hdbscan_boruvka,
                            dbscan_timings, k_means_timings
                           ], axis=1)
timing_data.columns=['AffinityPropagation', 'Spectral', 'Agglomerative',
                                        'DeBaCl', 'ScipySingleLinkage', 'Fastcluster',
                                        'HDBSCAN', 'DBSCAN', 'SKLearn KMeans'
                                       ]
def get_size(series, max_time):
    return series.index[series < max_time].max()

datasize_table = pd.concat([
                            timing_data.apply(get_size, max_time=30),
                            timing_data.apply(get_size, max_time=300),
                            timing_data.apply(get_size, max_time=3600),
                            timing_data.apply(get_size, max_time=8*3600)
                            ], axis=1)
datasize_table.columns=('Interactive', 'Get Coffee', 'Over Lunch', 'Overnight')
datasize_table
```

2.3.5 Conclusions

Performance obviously depends on the algorithm chosen, but can also vary significantly upon the specific implementation (HDBSCAN is far better hierarchical density based clustering than DeBaCl, and sklearn has by far the best K-Means implementation). For anything beyond toy datasets, however, your algorithm options are greatly constrained. In my (obviously biased) opinion HDBSCAN is the best algorithm for clustering. If you need to cluster data beyond the scope that HDBSCAN can reasonably handle then the only algorithm options on the table are DBSCAN and K-Means; DBSCAN is the slower of the two, especially for very large data, but K-Means clustering can be remarkably poor – it's a tough choice.

CHAPTER 3

API Reference

3.1 API Reference

Major classes are `HDBSCAN` and `RobustSingleLinkage`.

3.1.1 HDBSCAN

3.1.2 RobustSingleLinkage

CHAPTER 4

Indices and tables

- genindex
- modindex
- search