Practical 4: Clustering

1 Overview

The aim of these lab exercises is to give you some hands-on experience with **clustering**, with a focus on numeric data. For these exercises, you will work with a synthetic data set.

2 Getting Started

Start by generating a *synthetic* data set for the clustering exercises.

1. Randomly generate a synthetic dataset of 1000 data points (x, y) with 2 features for clustering. This will make it easy to plot the data, though of course, you could have many more than 2 features. Use the *scikit-learn* function datasets.make_blobs(), as shown below:

```python
#--
# generate some random data for clustering
#--
import sklearn.datasets as data

X, clusters = data.samples_generator.make_blobs( n_samples=1000, n_features
    =2, cluster_std=1.0 )
```

   The function parameters are:

   - n_features = defines how many attributes to generate for each instance;
   - n_samples = the number of instances to generate; and
   - cluster_std = the standard deviation for the clusters.

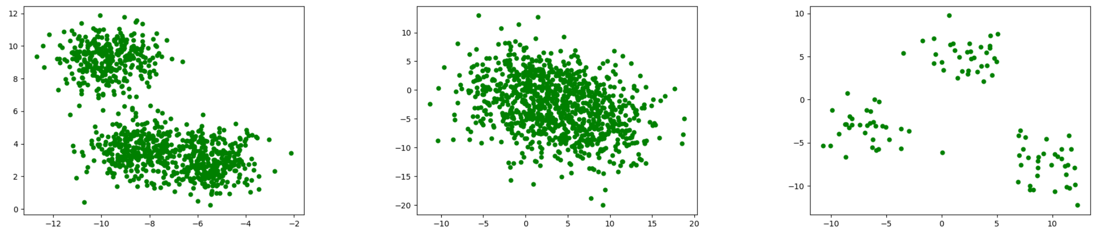
   The function returns:

   - X, the generated 2D array of features, of size *n_samples* × *n_features*; and
   - clusters, an array of target cluster numbers, also of size *n_samples*, which can be used for checking the validity of your clustering results.

2. Note the cluster_std argument to the function make_blobs(). This defines the standard deviation for the clusters. By default, this is set to 1.0. Try increasing it (say to 2.0 and then 5.0) and plotting the results. You will see more disorder in the synthetic data set as the standard deviation increases. My result for cluster_std =4.0 is shown in Figure 1b.

3. There is another optional argument to the function make_blobs(), which is called random_state. This argument can be used to specify a seed for the random number generator, so that you could replicate the results returned from this function if you want to run the script multiple times on the same data set.

4. It is always good to plot the data before trying to analyse it. Your plot should look something like my examples shown in Figure 1a (produced with different argument values, as indicated).



(a) cluster_std =1.0, n_samples=1000,

(b) cluster_std =4.0 n_samples=1000,

(c) cluster_std =2.0, n_samples=100,

Figure 1: Example synthetic data set for clustering

3 Write K-Means Yourself

The **K-Means** method is the basis of many popular clustering algorithms. So let's start by writing the simple steps "by hand" without using a scikit-learn function (you'll do that later, in part 4).

Pseudo-code is shown below:

```
1 initialise K clusters by randomly selecting K centres
2 loop until convergence:
3    compute distance from each instance in the data set to each of the K centres
4    for each instance, assign it to the cluster with the closest centre
5    re-compute the cluster centres, given the new cluster membership
```

In order to instantiate this code, you need to make some decisions:

1. *What value should you choose for* K *?*
   We discussed this question in lecture. Sometimes there are features of the clustering task that help you choose a value for K, such as having 7 chefs and needing to group a set of food delivery orders into 7 groups, one per chef. But that is not always the case, so sometimes you have to guess. But you can compute clusterings for multiple values of K and select the one which gives you the best result, using the metrics we discussed in class (more below).

2. *How do you know when the solution has* **converged***?*
   When you assign instances to clusters (line 4 in the code above), check if the assignment has changed from the previous iteration. The solution has converged when none of the assignments (for any instance) has changed. If the cluster assignments (membership) does not change, then re-computing the cluster centres (line 5) will not produce changes in cluster centre values. So you know that you can stop iterating.

$\rightarrow$ **Implement this pseudo-code,** using the 2-feature synthetic data set that you generated earlier in part 2. I suggest that you use use 100 samples (n_samples=100), otherwise it will take a while for your code to run (since your step-by-step manual effort will probably be less efficient than the scikit-learn version).

Test your code. Display some intermediate clusterings to visualise whether it's working. Some of my examples are shown in Figure 2 for $K = 4$.

Once you have gotten your K-Means code to work, then you should evaluate your solution. We talked about two main metrics: **within clusters** and **between clusters**. Let's compute each of these by hand.

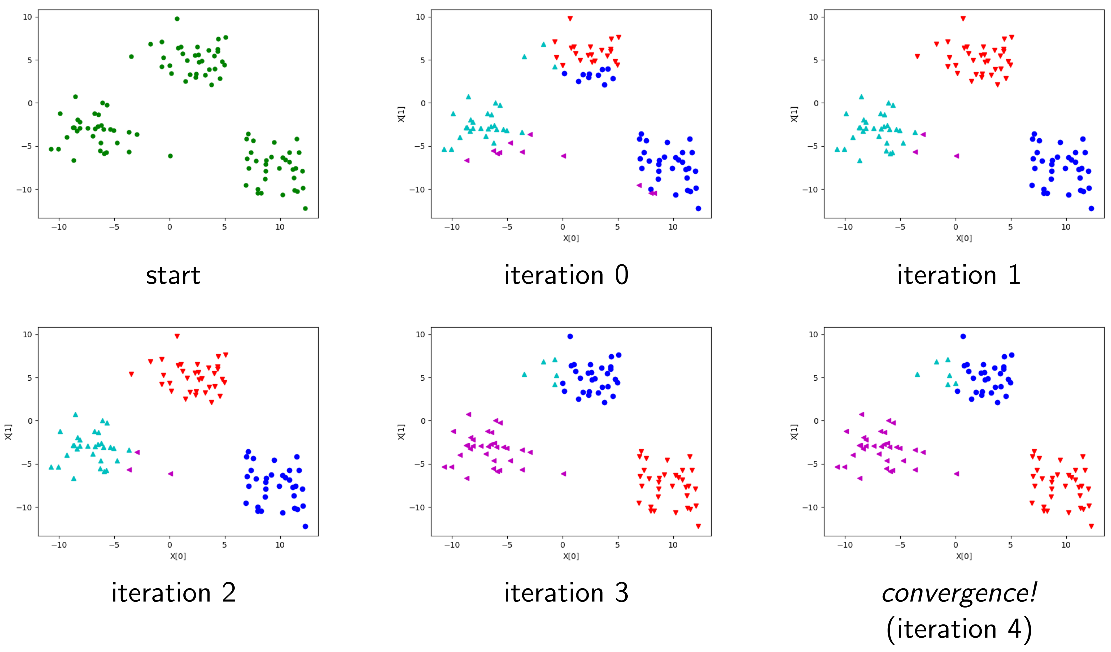


Figure 2: Example results for my version of K-Means, with $K = 4$

Note that, for simplicity here, we will use **cluster centres** for our computations. But you can refer back to the lecture notes to consider other ways of computing these metrics.

$\rightarrow$ Compute the Within Cluster score

The equation for computing the overall Within Cluster score, WC, is:

$$WC = \sum_{i=1}^{K} \langle \sum_{x_j \in C_i} d(x_j, c_i)^2 \rangle$$

where:

- K = number of clusters
- C_i = the i-th cluster
- x_j = the j-th instance in the data set (data point)
- c_i = centre of cluster i
- d = distance function, where $d(x_j, c_i)^2$ is the *within-cluster sum-of-squares distance*, measured from each point (x_j) in the cluster to the centre of its cluster (c_i)

Remember that we are trying to **minimise** the Within Clusters score (smaller values are better).

$\rightarrow$ **Compute the Between Clusters score**

The equation for computing the overall Between Cluster score, BC, is:

$$BC = \sum_{1 \leq i < l \leq K} d(c_i, c_l)^2$$

where:
- $d(c_i, c_l)^2$ is the *between-cluster sum-of-squares distance*, measured from the centre of each cluster i to the centre of every other cluster l

Remember that we are trying to **maximise** the Between Clusters score (larger values are better).

$\rightarrow$ **Compute the Overall Clustering score**

The equation for computing the overall clustering score ($score$) is:

$$score = \frac{BC}{WC}$$

where:
- BC is the overall between cluster score; and
- WC is the overall within cluster score.

Because we are trying to maximise the Between Clusters score (the numerator, or top of the fraction) and minimise the Within Clusters score (the denominator, or bottom of the fraction), we are looking for larger values of the overall clustering score.

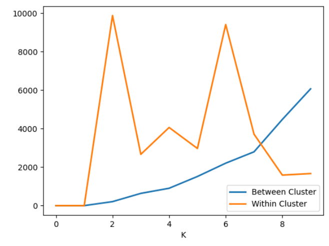
Remember from lecture, however, that this is a pretty crude measure of clustering goodness. In the next sections, we will explore some other metrics.

$\rightarrow$ **Try different values of** K and compare the overall clustering scores to see which K is best for your data set. Figure 3 shows my results for different values of K. I've also plotted the values of BC and WC for the different K values (Figure 3a).

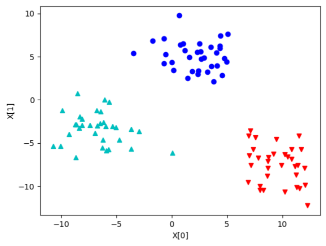
$\star$ Some Notes on K-Means

One thing to note about K-Means is this: because you initialise with random cluster centres, you will get different results each time you run your code. (This is even true if you use the random_state argument to the make_blobs() function so that you are always clustering the same set of instances.) So you may want to run the code multiple times in order to try and determine empirically which value of K is best for this data set. For example, you could run 30 times, for statistical significance, and compare the average scores for each K over the 30 trials. You can use statistical significance testing (e.g., **t-test**) to determine whether the results for any particular K are reliably better than others.

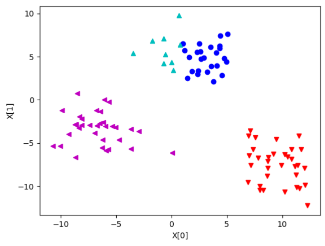
Again, though, remember that BC/WC is a crude measure of clustering "goodness". So if you choose to do these experiments, then I suggest that you employ one of the more sophisticated scoring methods (shown in part 4) for your evaluation.



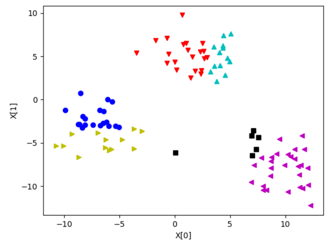
(a) BC and WC



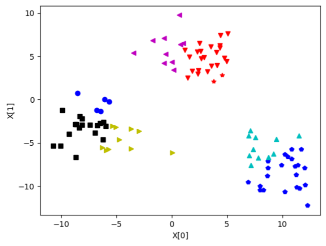
$K = 3$



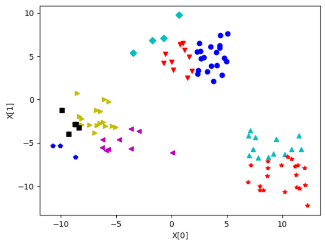
$K = 4$



$K = 6$



$K = 8$



$K = 9$

Figure 3: Example results for my version of K-Means, varying K. All plots show results after convergence and overall clustering score.

4 Use K-Means from scikit-learn

Now that you have written **K-Means** by hand, you hopefully have a strong understanding of how it works. So it is safe to start using the one provided by **scikit-learn**, which should run quite a bit faster than your code. (Unless you want to spend time optimising your code...)

The **sklearn.cluster** package contains the function you want:
`http://scikit-learn.org/stable/modules/generated/sklearn.cluster.KMeans.html`
Sample code is shown in Figure 4

→ **Re-implement your K-Means code** (from part 3) using the **scikit** function.

Once you have fit the model, you will want to look at the outputs:

- km.cluster_centers_ is a $K \times N$-dimensional vector containing the attribute values that define the cluster centres, or *centroids*

```
# load the clustering package definition
import sklearn.cluster as cluster

# initialise the KMeans clustering object
km = cluster.KMeans( n_clusters=K )

# compute the clusters:
km.fit( X )
# (where X is the synthetic data set we generated)
```

Figure 4: Example code for classifying data using K-Means from scikit-learn

- km.labels_ is an M-dimensional vector containing the cluster labels, in the range $[0 \ldots (K-1)]$
- km.inertia_ is the within-cluster score (also called 'inertia'); this should match your hand-crafted *within-cluster sum-of-squares distance* (from part 3)

In addition to the **within-cluster (inertia) score**, you should look at various other metrics, as we discussed in lecture. The **sklearn.metrics** package contains a large number of metrics for various data mining activities (you already used metrics for classification last week). Here, we will concentrate on the metrics defined for **clustering** tasks:

http://scikit-learn.org/stable/modules/classes.html#clustering-metrics

Figure 5 illustrates how to call the functions that compute the scores we have discussed in lecture.

```python
# load the metrics package definition
import sklearn.metrics as metrics

# compute the overall silhouette coefficient (returns a scalar):
SC = metrics.silhouette_score( X, km.labels_, metric='euclidean' )

# compute the calinski-harabasz score (returns a scalar):
CH = metrics.calinski_harabasz_score( X, km.labels_ )
```

Figure 5: Sample code for calling some of the **scikit-learn** clustering metrics

$\rightarrow$ **Add computation of the Silhouette score and the Calinski-Harabasz score to your code.** Compare your hand-computed metrics to those provided by **scikit-learn**.

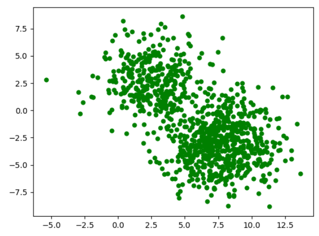
Recall what we look for with these metrics:

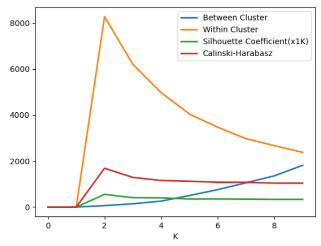
- With the Calinski-Harabasz score, we look for larger values because this metric is higher when clusters are well separated and dense.
- With the Silhouette Coefficient, we look for values closer to 1, which indicates that clusters are well separated. As this value moves toward 0, clusters may be overlapping. If the metric is negative, then the clustering is considered incorrect.

$\rightarrow$ **As previously, try different values of** K and compare the overall clustering scores to see which K is best for your data set. Table 1 shows my results for different values of K. I've also plotted the different score values for the different K values (Figure 6a).

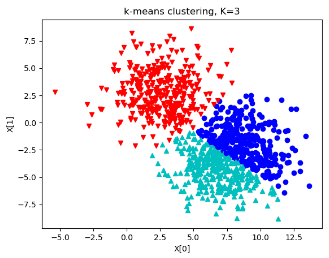
Table 1: Example scores for **scikit** K-Means, varying K

K	WC	BC	*score*	*inertia*	*silhouette*	*calinski-harabasz*
2	8275.00	61.56	0.0074	8275.00	0.5517	1686.69
3	6214.96	143.09	0.0230	6214.96	0.4065	1286.99
4	4972.80	256.36	0.0516	4972.80	0.4000	1154.17
5	4042.99	499.74	0.1236	4042.99	0.3481	1120.84
6	3469.25	756.78	0.2181	3469.25	0.3501	1076.79
7	2971.88	1051.13	0.3537	2971.88	0.3411	1074.15
8	2672.25	1448.75	0.5421	2672.25	0.3441	1038.79
9	2378.47	1803.34	0.7582	2378.47	0.3336	1035.48

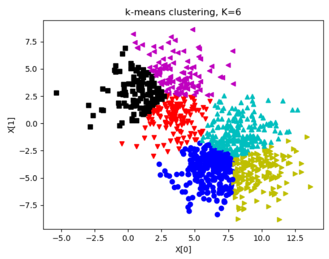

(a) data



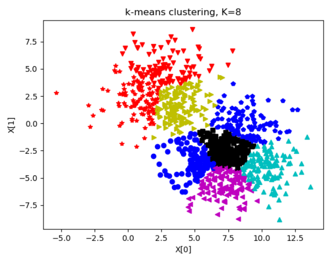
scores



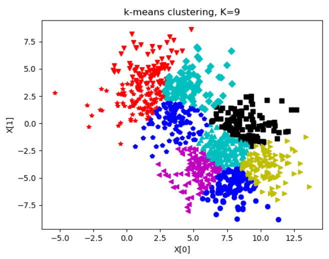
$K = 3$



$K = 6$



$K = 8$



$K = 9$

Figure 6: Example results for **scikit** K-Means, varying K. All plots show results after convergence and overall clustering score. Note that the **Silhouette coefficient** scores are scaled by $x1000$, to show them on the same plot as the other metrics.

5 Write Agglomerative Clustering using **scikit-learn**

Recall from this week's lecture that there are multiple approaches to clustering, and K-Means is only one of them. Another approach is called **Hierarchicial** clustering.

The **Agglomerative Hierarchical** clustering package in scikit-learn is here:

`http://scikit-learn.org/stable/modules/generated/sklearn.cluster.AgglomerativeClustering.html`

Example code for using this package is shown in Figure 7.

Once you have fit the model, you will want to look at the outputs:

```
# load the clustering package definition
import sklearn.cluster as cluster

# initialise the agglomerative clustering object
ac = cluster.AgglomerativeClustering( n_clusters=K, linkage='average', affinity='
    euclidean' )

# compute the clusters:
ac.fit( X )
```

Figure 7: Example code for **scikit** Agglomerative clustering

- ac.labels_ is an M-dimensional vector containing the cluster labels for each instance, in the range $[0 \ldots (K-1)]$
- ac.leaves_ is the number of leaves in the dendrogram which represents the clustering
- ac.children_ is a $(M-1) \times 2$ dimensional vector containing the linkages between samples which indicate how the clusters are formed in the dendrogram

The outputs (aside from the labels) are a bit awkward to understand, but there is a handy function in **scipy** which displays dendrograms:

`https://docs.scipy.org/doc/scipy-0.14.0/reference/generated/scipy.cluster.hierarchy.dendrogram.html`

Note that this function was written to work with the **scipy.cluster.hierarchy.linkage** function, described here:

`https://docs.scipy.org/doc/scipy-0.14.0/reference/generated/scipy.cluster.hierarchy.linkage.html#scipy.cluster.hierarchy.linkage`

However, you will probably find the **sklearn.cluster.AgglomerativeClustering** package more useful generally, including having access to more metrics. Example code showing how to estimate the linkage information and produce a dendrogram from the**scikit** output is contained in Figure 8.

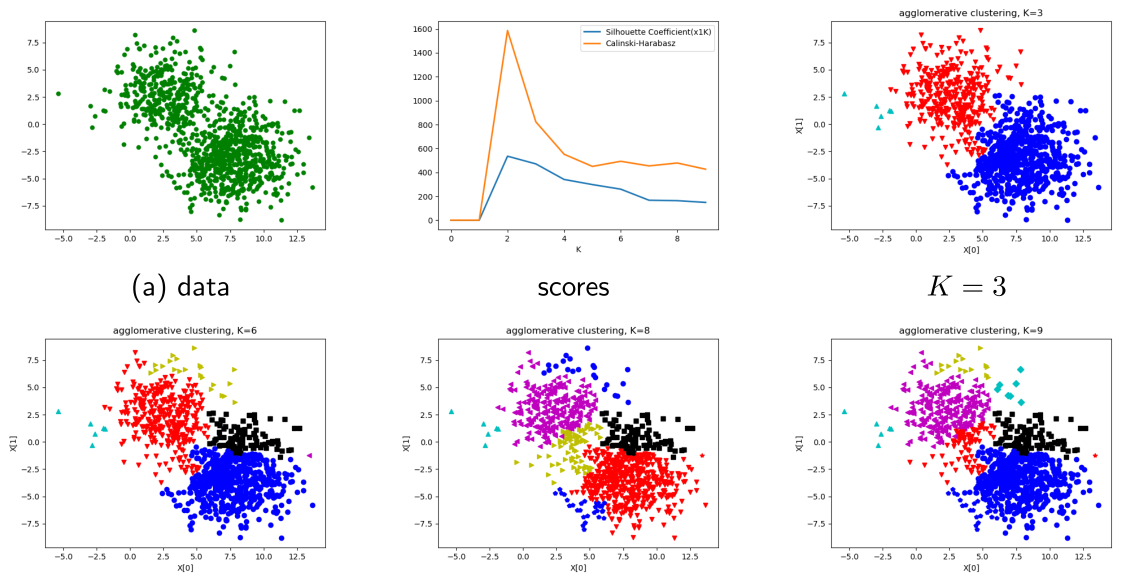
My results for $K = 3$ are shown in Table 2 and plotted in Figure 9.

Table 2: Example scores for **scikit** Agglomerative clustering, varying K

K	*silhouette*	*calinski-harabasz*
2	0.5355	1586.90
3	0.4712	822.91
4	0.3408	551.71
5	0.2981	450.01
6	0.2597	492.92
7	0.1671	454.20
8	0.1635	478.51
9	0.1488	426.48

```python
import numpy as np
import matplotlib.pyplot as plt
import scipy.cluster.hierarchy as hierarchy

plt.figure()
# initialise data structure for scipy dendrogram printing function
Z = np.empty( [ len( hc.children_ ), 4 ], dtype=float )
# steps (A) and (B) below are thanks to:
# https://github.com/scikit-learn/scikit-learn/blob/70cf4a676caa2d2dad2e3f6e4478d64bcb0506f7/examples/cluster/plot_hierarchical_clustering_dendrogram.py
# for hints on how to combine the sklearn agglomerative clustering with the dendrogram plotting function of scipy.
# (A) compute distances between each pair of children: since we don't
# have this information, we can use a uniform one for plotting
cluster_distances = np.arange( hc.children_.shape[0] )
# (B) compute the number of observations contained in each cluster level
cluster_sizes = np.arange( 2, hc.children_.shape[0]+2 )
for i in range( len( hc.children_ )):
    Z[i][0] = hc.children_[i][0]
    Z[i][1] = hc.children_[i][1]
    Z[i][2] = cluster_distances[i]
    Z[i][3] = cluster_sizes[i]
# plot dendrogram
hierarchy.dendrogram( Z )
plt.show()
```

Figure 8: Example code for producing a dendrogram



(a) data — scores — $K = 3$

$K = 6$ — $K = 8$ — $K = 9$

Figure 9: Example results for **scikit** Agglomerative clustering, varying K. All plots show results after convergence and overall clustering score. Note that the **Silhouette coefficient** scores are scaled by $x1000$, to show them on the same plot as the other metrics.

6 Write DBSCAN with **scikit-learn**

Another clustering approach that we discussed in lecture is **Density-based** clustering.

The **DBSCAN** clustering package in scikit-learn is here:

`http://scikit-learn.org/stable/modules/generated/sklearn.cluster.DBSCAN.html`

Example code for using this package is shown in Figure 10.

```
# load the clustering package definition
import sklearn.cluster as cluster

# initialise the DBSCAN clustering object
db = cluster.DBSCAN( eps=0.5, min_samples=1 )

# compute the clusters:
db.fit( X )
```

Figure 10: Example code for **scikit** DBSCAN clustering

My results are shown in Figure 11.

There are two parameter values that can be used to tune the results of DBSCAN:

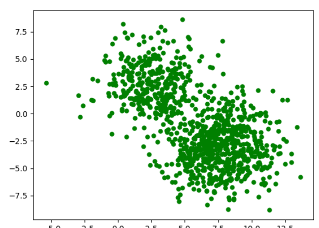
- EPS is the maximum distance between samples allowed for them to be considered in the same cluster; and
- MS (or min_samples) is the minimum number of samples allowed to be contained in a cluster.

In order to determine a good value for EPS, it is helpful to look at the *nearest neighbour* distances between instances in the data set. There is a package in scikit-learn to compute nearest neighbours, as illustrated below:

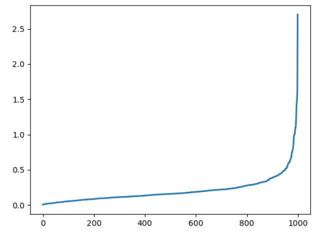
```
# load the clustering package definition
import sklearn.cluster as cluster

# initialise the nearest neighbours object
nn = neighbors.NearestNeighbors( n_neighbors=2, metric='euclidean' )

# fit the data to find the nearest neighbours to every instance
nn.fit( X )

# return the K nearest neighbours (K=n_neighbors argument)
dist, ind = nn.kneighbors( X, n_neighbors=2 )
```

The plot shown in Figure 11b illustrates the *nearest neighbour* distances for each sample, plotted in sorted order. It can be seen that where the distance $= 0.75$ (on the y-axis), we start to see a sharper incline in the distances. So I ran DBSCAN with a number of different values for min_samples using eps=0.75. The results are shown in Table 3 and Figure 11d-f.

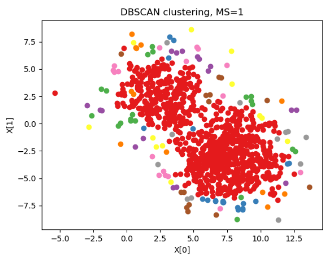
→ **For the last exercise, try implementing DBSCAN on the synthetic data.** Try some different values for the arguments eps and min_samples to see if you can get scores that approach those of the other methods tried in parts 4 and 5.



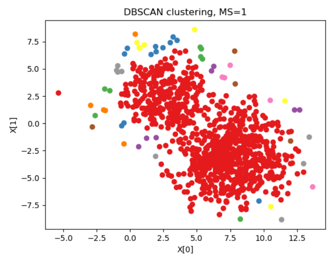
(a) data



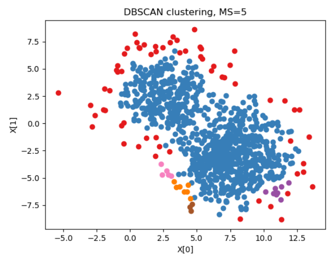
(b) nearest neighbours



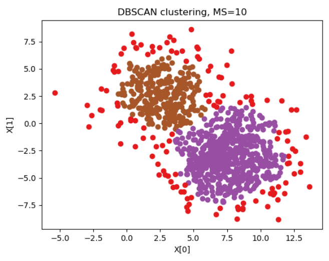
(c) $EPS = 0.5, MS = 1$



(d) $EPS = 0.75, MS = 1$



(e) $EPS = 0.75, MS = 5$



(f) $EPS = 0.75, MS = 10$

Figure 11: Example results for **scikit** DBSCAN clustering, varying EPS (maximum distance between samples) and MS (minimum number of samples).

Table 3: Example scores for **scikit** DBSCAN clustering, varying K

EPS	*min_samples*	K	*silhouette*	*calinski-harabasz*
0.50	1	86	-0.244453	37.56
0.75	1	37	-0.345760	6.67
0.75	5	6	-0.107877	18.69
0.75	10	3	0.432022	474.42