Master of Science in Computer Engineering
Data Science

Clustering algorithms based on segmentation techniques and artificial neural networks

Supervisor: prof. Paolo GARZA
Candidate: Giuseppe CAMPAGNOLO

Year 2019

Politecnico di Torino, Turin, Italy

Contents

Thanks

Working on this thesis consisted mainly of designing new algorithms by dealing at lower level with the code, and this filled me with a lot of satisfaction since I have a soft spot for programming. The first person I would like to thank is the professor who supported me during these last months as a supervisor. I had moments of discouragement due to the fact that some tests led to nothing, however he was always able to transmit me confidence (especially at the beginning) by helping me follow interesting research directions. His collaboration was fundamental for me. Thanks also to my family who saw me proceeding along my graduation course up to its crowning. I admit that it is complicated to explain the subject of my thesis to non-IT people, many examples must be given. But, most of all, I appreciate the efforts of my father who finally (after five years) has learned what my university orientation is. Better late than never, right? Moreover, I would like to mention my university friends with whom I had the opportunity to compare myself, year after year. It is always nice to cultivate beneficial friendships with which to grow together (and even exchange engineer jokes). Of course, without them everything would have been much more arid and heavy. Finally, I would like to thank my girl for having supported me during the hardest times, when I was sitting at that table and I was programming all day almost without any pause. Definitely, engineering tried to spoil my life, fortunately I resisted until the end. The truth is that studying knows no holidays, every moment is always good to practice. My wish is that at least from now on in the job's world Sunday is really Sunday.

Chapter 1

Introduction

The computer science of the last twenty years has seen the gold rush take place for data. Every avant-garde company has moved towards the extraction of added value from the enormous amount of data already present in its databases, as if they were gold mines. But what kind of added value can data offer? The answer is very broad: a good analysis of data can guide important commercial choices for the sale of a product, it can be the basis of a network monitoring system to identify possible cyber attacks, it can provide relevant information on natural disasters for to be able to foresee them, and a lot of other possibilities. The set of all these things is indicated in literature with the name of *data mining*. Although the idea of analyzing data was born long time ago, the related processing techniques have been designed and implemented only recently: this is the reason why data mining is a hot topic of the moment.

Based on the problem you want to solve, there are several data mining techniques available: classification, association rules, regression, clustering. This thesis faces *clustering*, first referring to some well-known clustering algorithms and then designing new others based on different approaches, hopefully better in some respects. So, what is the clustering problem?

Suppose to have a 2D data set (i.e. each instance is a data-point with two attributes) which can therefore be represented on the Cartesian plane: each data-point corresponds to a point in the plan. Clustering is *the process by which the data-points are organized into groups - called clusters - so as to minimize the intra-cluster distances and maximize the inter-cluster ones.* Even if this

definition could appear cumbersome, however the concept is basically simple: for each data-point you must assign a label indicating which cluster the data-point itself belongs to, in order to put together "similar" (i.e. near) data-points and put separated "dissimilar" (i.e. far) data-points; this subject will be dealt extensively later. Furthermore consider that, in order to define clustering, the concept of *distance* between data-points is needed: the simplest choice is to make the distance between two data-points coincide with their Euclidean distance. In the general case, instances could have an arbitrary number of attributes, not necessarily two.

Clustering is an *unsupervised* problem. This means that new unlabeled data sets are labeled without any training or any prior information help (unlike what happens in supervised problems, e.g. classification). This task is really simple for humans to be performed when 2D images are considered, because the human eye is able to group objects naturally according to some interesting laws (e.g. proximity, similarity, continuity in the direction, etc). But how to get a machine to do it? This question represents the great clustering challenge.

Clustering is so important because in the real world there are a lot of *applications* which require it (i.e. data segmentation): network traffic, text documents, marketing and sales, biology research, medical exams, financial world, image processing, web analytics, robotics, etc. Such fact is more than sufficient to justify the efforts invested in drafting this thesis.

Chapter 2

State of the art clustering algorithms

The introduction chapter has made you get in touch with the clustering problem; now an overview of the known solutions will follow. Unfortunately, no single solution - which always goes well - has yet been found for the clustering problem, rather there are different clustering algorithms based on different approaches. This is due to the fact that the nature of clusters is very varied. So it is useful to make a brief digression on *cluster categorization* before analyzing any clustering algorithm. After such cluster categorization section, there will be three further sections describing the most renowned clustering algorithms respectively: the *k-means algorithm*, the *DBSCAN algorithm* and the *hierarchical clustering algorithm*. Finally a summary section will follow.

2.1 Clusters categorization

Typically clusters are categorized according to two grouping criteria:

- distinction based on cluster overlap:
  - *partitional clustering*: each data-point can belong only to one and only one cluster;
  - *hierarchical clustering*: each data-point can belong to more than one cluster, since clusters are organized in a hierarchy.

- distinction based on cluster shape:
  - *well separated* clusters: intra-cluster distances are much lower than inter-cluster distances (see Figure 2.1);

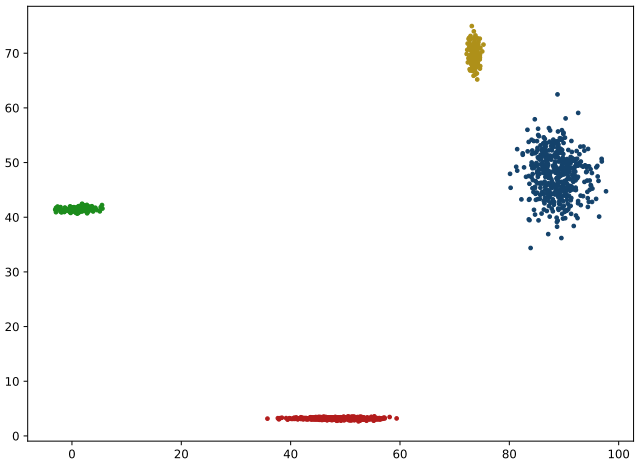

Figure 2.1: Well separated clusters

  - *center-based* clusters: each cluster has a circular shape of arbitrary radius with a center data-point (see Figure 2.2);

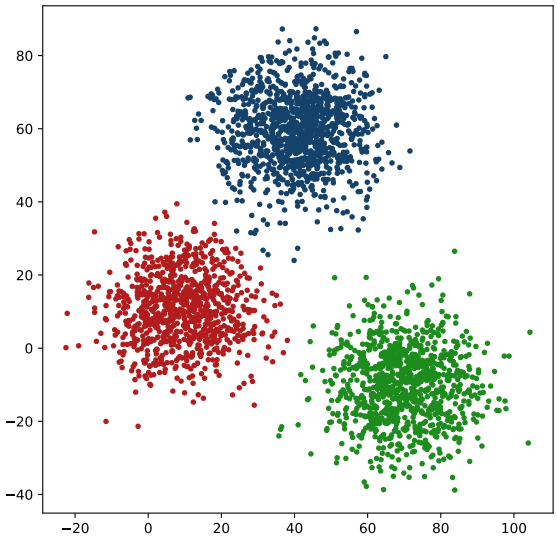

Figure 2.2: Center-based clusters

– *density-based* clusters: each cluster is characterized by a density different from that of the other clusters (see Figure 2.3);

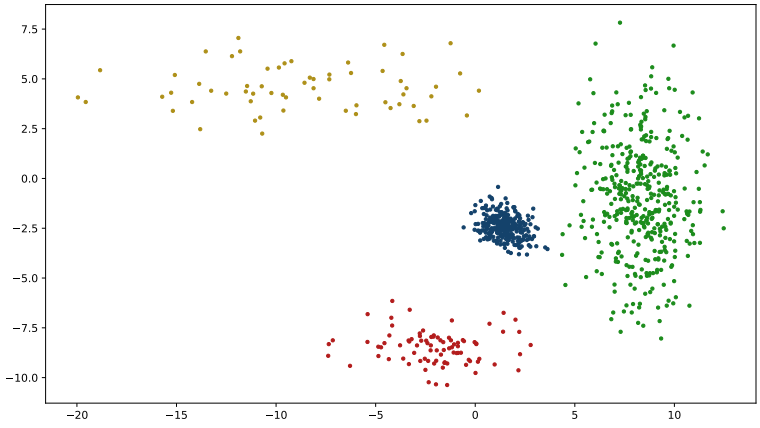

Figure 2.3: Density-based clusters

– *contiguity based* clusters: each cluster is characterized by a continuous arbitrary shape (see Figure 2.4).

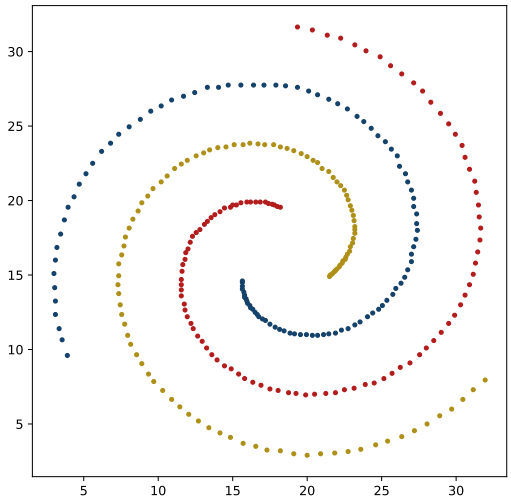

Figure 2.4: Contiguity based clusters

This second grouping criterion is not always strict, in the sense that there are more articulate data sets whose clusters can fall into more than one type, as depicted in Figure 2.5 and in Figure 2.6.

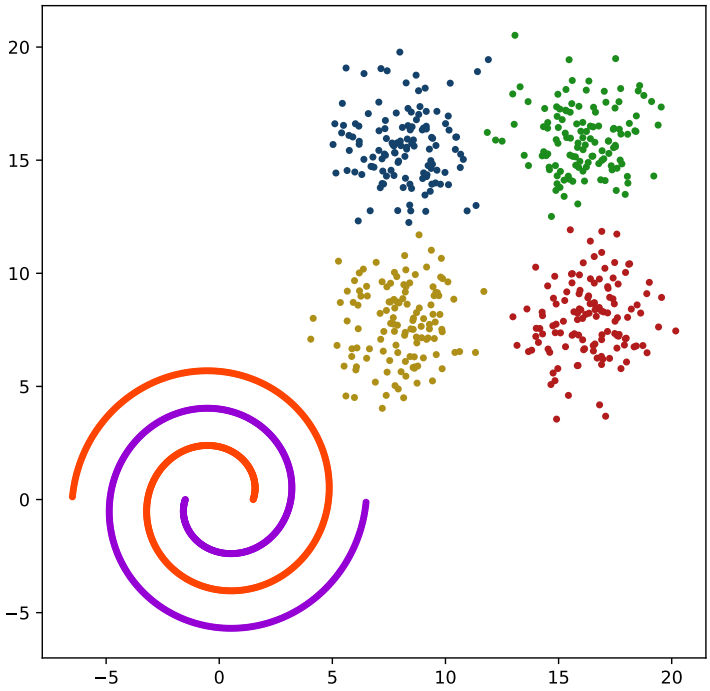

Figure 2.5: Center-based and contiguity-based clusters

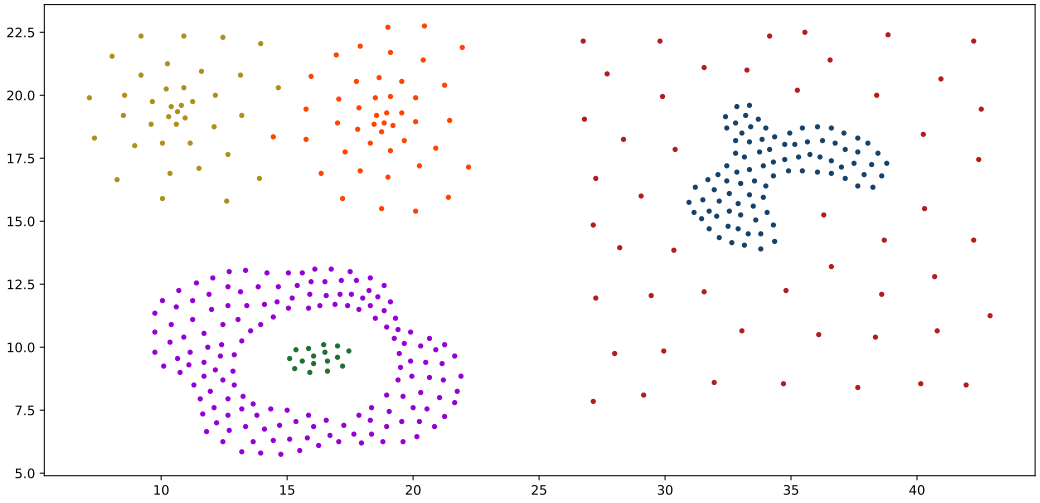

Figure 2.6: Density-based and contiguity-based clusters

Moreover, there is also another aspect that complicates things: the eventual presence of noise (see Figure 2.7).

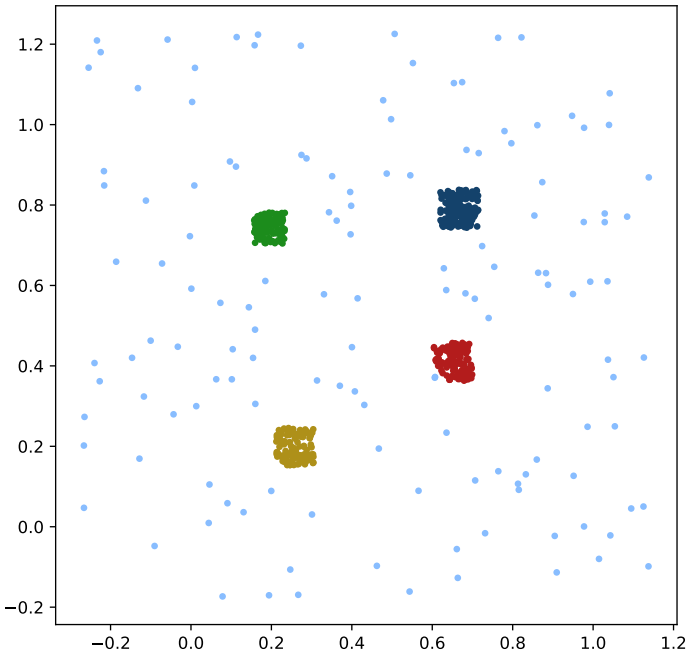

Figure 2.7: Well-separated clusters with noise (in light-blue)

As the name suggests, noise represents useless data that does not add value to the analysis, on the contrary its effect dirties useful data. Typically it has a low density and, since detecting it is not so simple, not all clustering algorithms are able to manage noise.

Summing up, the above illustrated categorization shows that clusters can be very varied, so the idea behind any clustering algorithm must take it into account. Please pay attention to the fact that, however, such overview is not exhaustive. In fact there are other more refined grouping criteria (e.g. exclusive versus non-exclusive, fuzzy versus non-fuzzy) which were not presented because they are not relevant to this thesis.

2.2 K-means algorithm

2.2.1 Description

The k-means (proposed in 1957) is a partitional clustering algorithm that fits well all those data sets whose clusters are center-based. It takes in input - in addition to the data set to be clustered - the k parameter, saying how many clusters are in the data set. Unfortunately, there is no golden rule of thumb for determining the right value of k, as will be discussed in the limitations subsection. That said, the algorithm makes use of the concept of *centroid*: the centroid of a cluster is the algebraic mean position of all the data-points in the cluster itself (hence the name k-means). Note that a centroid may not necessarily be an existing data-point in the data set.

Basically, the idea behind the k-means algorithm is very simple. Listing 2.1 is a high level pseudo-code showing the main algorithm steps:

Listing 2.1: Alg.1 - k-means

```
1 input: data_points, k
2 select random k data_points as cluster centroids
3 do {
4     assign each data_point to the nearest centroid
5     recalculate the new centroids
6 } while (centroids are changing);
```

In line 2 of Listing 2.1 the k centroids are initialized randomly: each centroid is representative of the cluster which includes it, so there are k centroids for k clusters. The step in line 4 of Listing 2.1 implicitly performs a for loop: each data-point in the data set is assigned to the nearest centroid, in detail the cluster Id of such data-point is set equal to that of the centroid in question. Please consider that to calculate distances between data-points you need to define a metric. The most commonly chosen metric is the Euclidean distance. In short, at the end of this step the whole data set is labeled. In line 5 of Listing 2.1 there is another implicit for loop: for each cluster - basing on the labels assigned in line 4 of Listing 2.1 - its new centroid is recalculated by computing the algebraic

mean position among all the data-points of the cluster in question. Note that the centroids are supposed to change at least at the first iteration, since their initial choice is random (hence, at the start, most likely the situation is not yet in a state of convergence). Finally, as the while condition in line 6 of Listing 2.1 indicates, lines 4-5 of Listing 2.1 are repeated as long as the centroids change (or until it has been reached a maximum number of iterations).

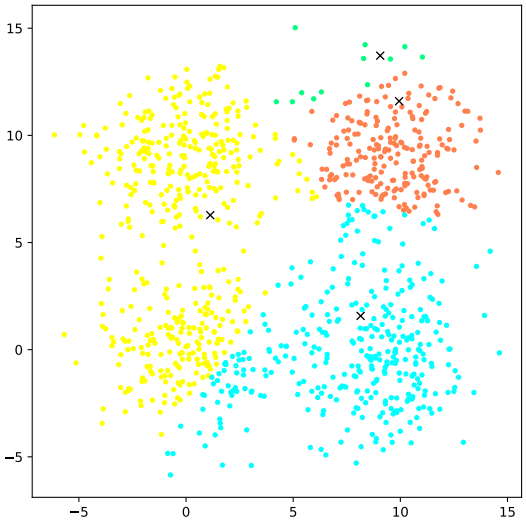

Figure 2.8: Iteration 1

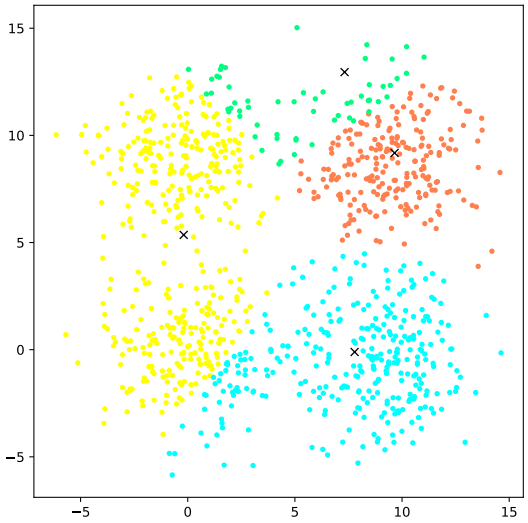

Figure 2.9: Iteration 2

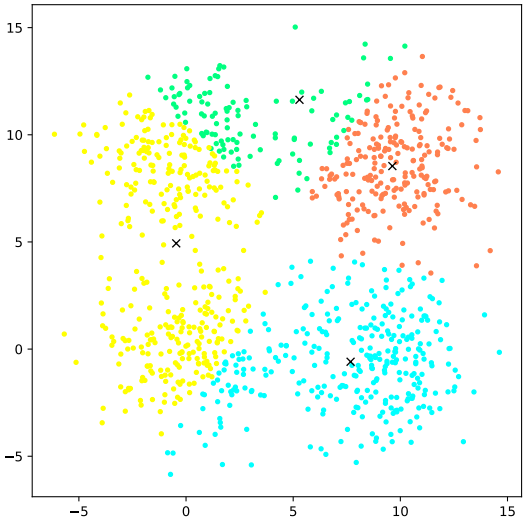

Figure 2.10: Iteration 3

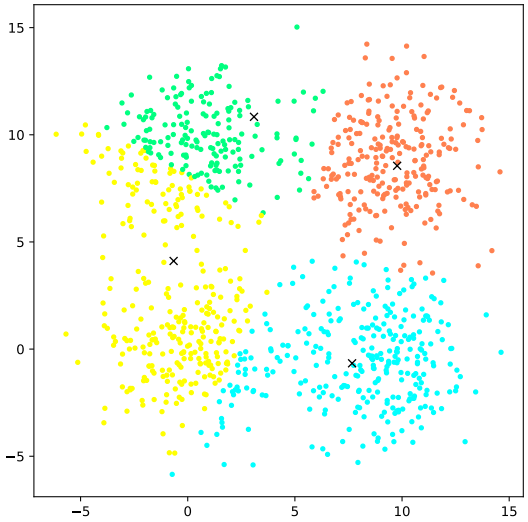

Figure 2.11: Iteration 4

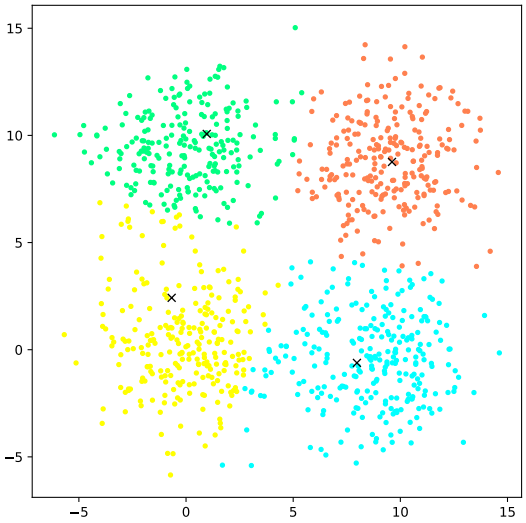

Figure 2.12: Iteration 5

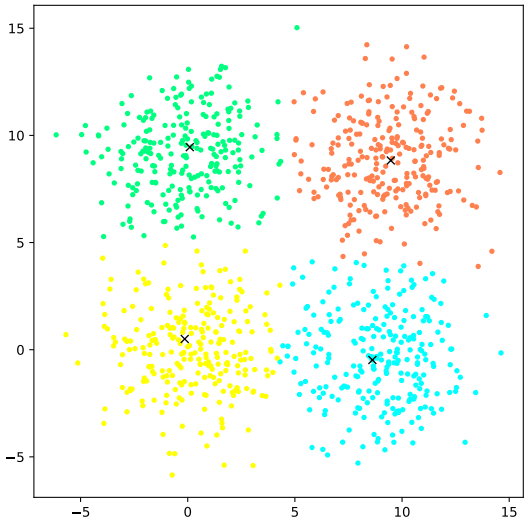

Figure 2.13: Iteration 6

Figures 2.8-2.13 are a graphic example (where $k = 4$) showing how the k-means algorithm works by highlighting the change of the centroids (marked with X) in the various iterations. As you can see, the stop condition is reached after 6 iterations. In fact from interaction 7 onwards the centroids no longer change. In this case things are going very well because clusters are center-based. Definitely, the k-means algorithm tends to find clusters of globular shape, each of them centered at a point called centroid.

2.2.2 Limitations

The simplicity of the k-means algorithm is counterbalanced by a whole series of limitations:

- Knowing *the right value of the k parameter* a priori is not a trivial task. In order to face such inconvenience a parameters' search is performed, which consists in running the algorithm with different values of k and then finding the most appropriate value of it through graphic methods, such as the elbow method.

- The k-means algorithm encounters difficulties when clusters have *different sizes/densities* (see Figure 2.14) or *non-globular shapes* (see Figure 2.15). These are the greatest limitations.

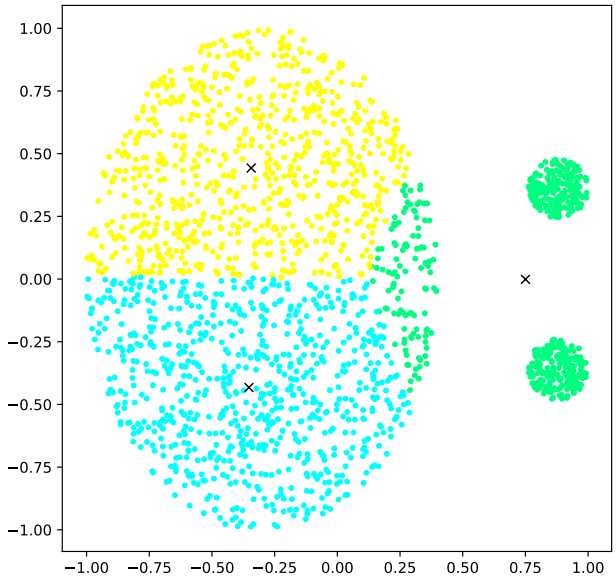

Figure 2.14: Clusters with different sizes and density

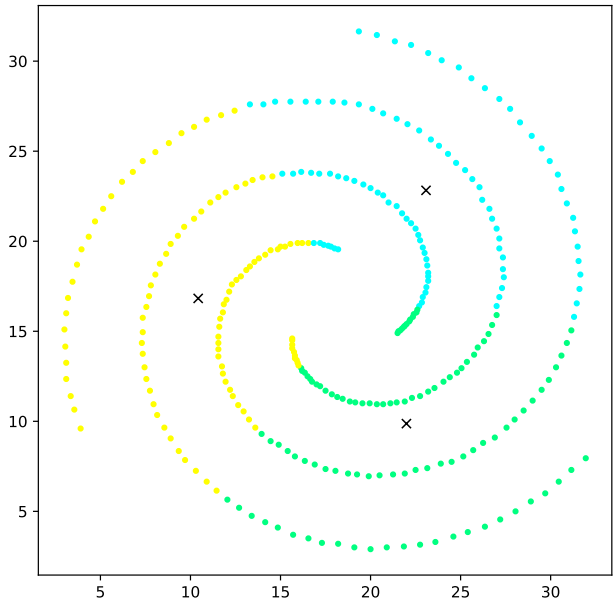

Figure 2.15: Clusters with non-globular shape

The most adopted method to address this problem consists in finding more clusters than there are and then merging them opportunely through post-processing operations (see Figure 2.16).

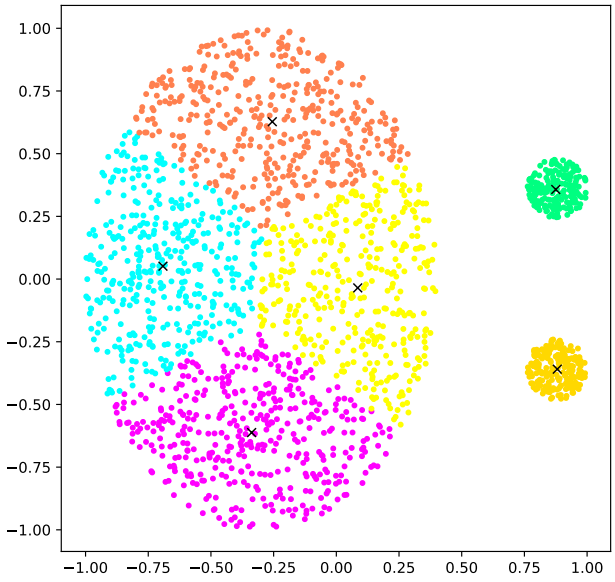

Figure 2.16: K-means with $k = 6$

Unfortunately this approach does not always work, especially for those strongly contiguity-based clusters (see Figure 2.17).

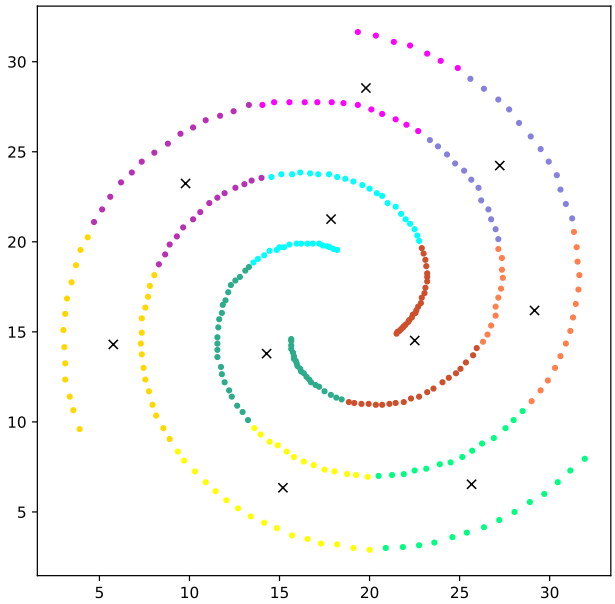

Figure 2.17: K-means with $k = 10$

- The *initial random choice of the centroids* is not always a good idea as there are situations in which the algorithm does not converge in the desired clustering. Most of the time it takes a pre-processing heuristic to select initial the centroids in a clever way.

- The k-means algorithm is not able to *detect noise*.

2.3 DBSCAN algorithm

2.3.1 Description

The DBSCAN is a partitional clustering algorithm proposed for the first time in 1996, after about 30 years from the publication of the k-means algorithm: this is a symptom of a more complicated and sophisticated approach which required more research time. DBSCAN stands for Density-Based Spatial Clustering of Application with Noise: as the name suggests, this algorithm can detect noise by construction. If the data set to be clustered has clusters all *at the same density*, then clustering will most likely succeed, regardless of whether clusters are well separated, center-based or contiguity-based. In detail, the DBSCAN algorithm is divided in two phases (*classification phase* and *labeling phase*) and it requires two parameters (*eps* and *MinPts*).

The classification phase consists in classifying each data-point of the unlabeled data set as *core-point*, *border-point* or *noise*. Suppose, for the moment, to have already set the *eps* and *MinPts* parameters, whose meaning is explained in the following list. A data-point is:

- core-point if it has at least *MinPts* data-points within *eps*;

- border-point if it has less than *MinPts* data-points within *eps*, but at least one of these is a core-point;

- noise (or outlier) if it is neither core-point nor border-point.

At the end of this phase, all the data-points are classified as core, border or noise.

The labeling phase consists in assigning to each data-point its final label (obviously data-points with the same label are in the same cluster). It can be described by the high level pseudo-code in Listing 2.2.

Listing 2.2: Alg.2 - DBSCAN (part 1)

```
1  def labeling_phase():
2    labels = array of -1 (length = total number of data-points)
3    current_label = 0
4    for core_point in core_points:
5      assign_labels(core_point, current_label)
6      current_label++
```

The above pseudo-code is similar to that used for the visit in amplitude of graphs. In line 2 of Listing 2.2 the cells of the `labels` array are initialized with -1, which means noise; at the end of the labeling phase, those cells that remain at -1 (i.e. those cells that have not been updated) will correspond to noise data-points. In line 3 of Listing 2.2 the `current_label` variable is defined and initialized to 0, in fact final labels are integers greater than or equal to 0. The increase of `current_label` in line 6 of Listing 2.2 corresponds to a cluster change. As you can see, all the intelligence of this phase lies in the `assign_labels` recursive function, invoked for each core-point in line 5 of Listing 2.2. The high level implementation of such function is in Listing 2.3.

Listing 2.3: Alg.2 - DBSCAN (part 2)

```
7  # Recursive function
8  def assign_labels(data_point, current_label):
9    if (data_point.get_label != -1)
10     return
11   data_point.set_label(current_label)
12   if (is_border(data_point)):
13     return
14   for neighbor in eps_neighbors_of(data_point):
15     assign_labels(neighbor, current_label)
```

At each level of recursion, the `assign_labels` function takes as parameters the current data-point to process and the current label to assign. In line 9 of Listing 2.3 there is the first stop condition of the recursion: if the current data-point has already been labeled (its label is no longer equal to -1) then the function must return. Otherwise, its label is set with the value of `current_label` (see line 11 of Listing 2.3). Then, in line 12 of Listing 2.3 there is the second stop condition of the recursion: if the current data-point is a border-point (this is known thanks to the previous classification phase) then the function must return because in the next for loop in line 14 of Listing 2.3 only core-points must be iterated. In particular, the `esp_neighbors_of` function returns the neighbors with distance at most `eps` from the data-point passed as parameter. Note that these neighbors cannot be noise but only core-points or border-points, since - as just said - the data-point passed as parameter to the `esp_neighbors_of` function is for sure a core-point. So, for each of these neighbors the `assign_labels` function is invoked recursively in line 15 of Listing 2.3.

As anticipated, the workhorse of the DBSCAN algorithm is its ability to detect noise (see Figure 2.18) and to properly deal with clusters of arbitrary shapes, as long as they are at constant density (see Figure 2.19); in the following figures black data-points are noise.

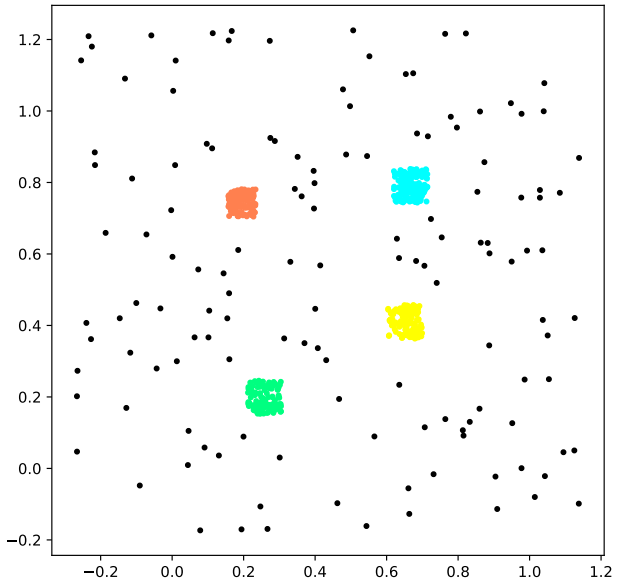

Figure 2.18: DBSCAN with $eps = 0.05$, $MinPts = 20$

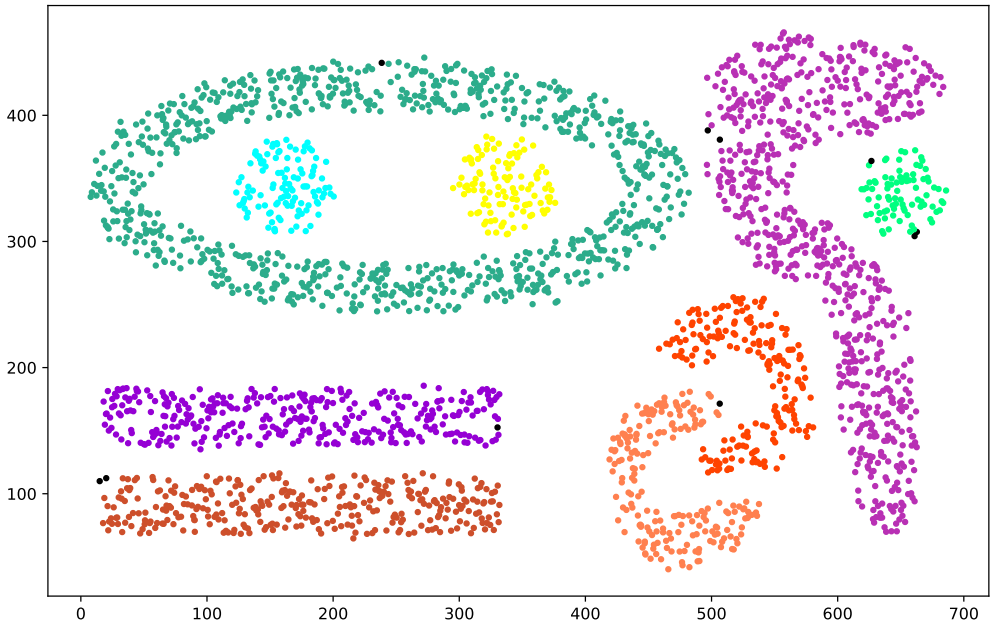

Figure 2.19: DBSCAN with $eps = 15$, $MinPts = 10$

Obtaining these results was almost revolutionary when DBSCAN was proposed because, until then, it was not yet possible to treat such complex data sets. Nevertheless, the vast majority of data sets looks just like those shown in these figures, so this algorithm has taken on some relevance.

2.3.2 Limitations

The DBSCAN algorithm is a good alternative to the k-means algorithm in all those situations where the latter fails. However, there are at least two problems associated to the use of a density-based approach:

- Which are appropriate values of *eps* and *MinPts*? This is a big obstacle because - as depicted in the previous two figures - in order to obtain the right clustering, these parameters can be very variable and their meaning is more complex than the k parameter of the k-means algorithm. In this regard, a quantitative palliative was defined in order to automatically determine the values of such parameters: Without going into details, it consists in analyzing an increasing function on the 2D plane and finding

its "knee" point to derive appropriate values of *eps* and *MinPts*. Unfortunately it does not always work, since sometimes there are more than one knee thus the parameters' search is not enough to address the problem. Most of the time the human intervention is often required, definitely.

- As already mentioned above, the DBSCAN algorithm fails when cluster has different densities (see Figure 2.20).

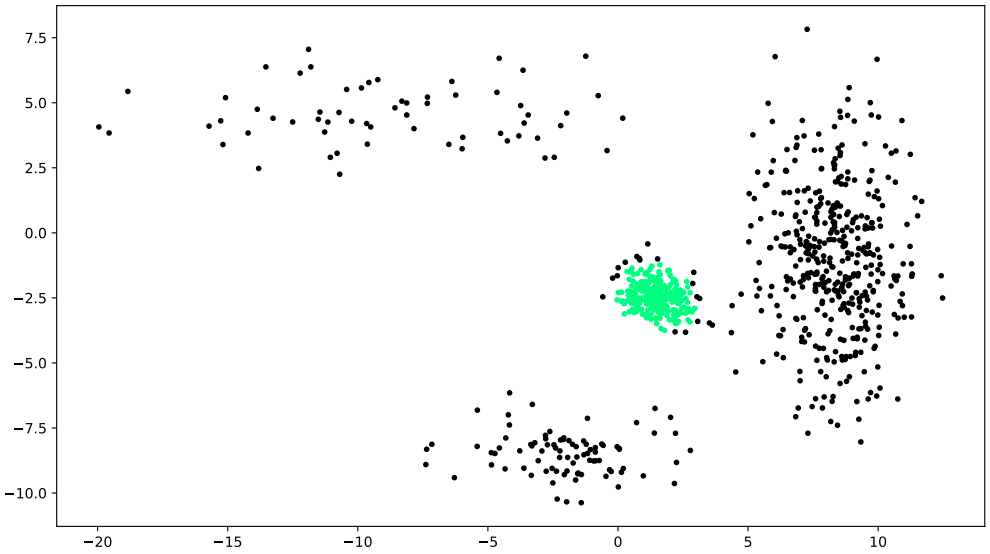

Figure 2.20: DBSCAN with $eps = 0.5$, $MinPts = 25$

As you can see in Figure 2.20, almost all data-points have been classified as noise because - setting $eps = 0.5$ and $MinPts = 25$ - these are at a lower density than that of the green cluster. So, in this case, the DBSCAN algorithm is able to find only the green cluster. On the other hand, trying to set "less restrictive" parameters, the opposite effect is achieved, as depicted in Figure 2.21.

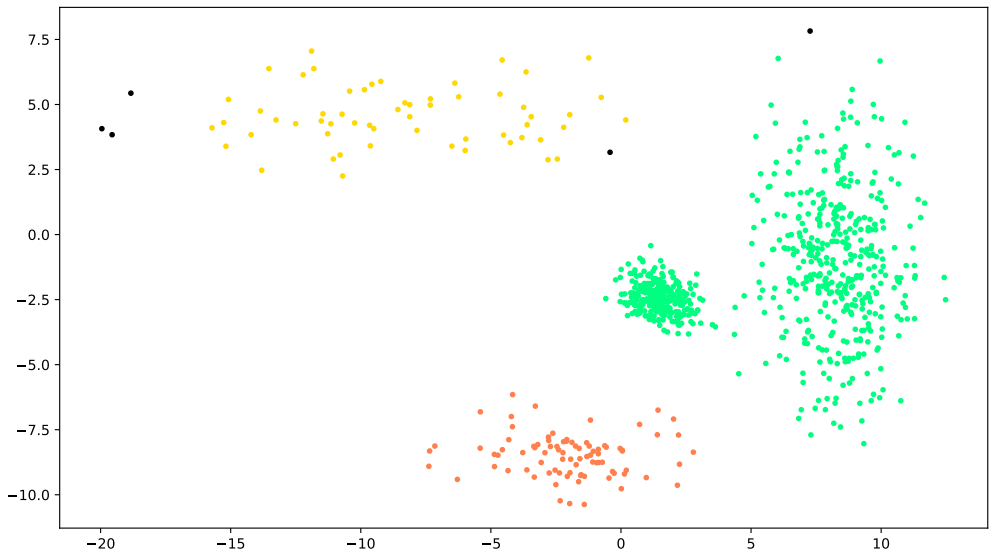

Figure 2.21: DBSCAN with $eps = 2$, $MinPts = 4$

This time, the yellow and orange clusters are found correctly, but the other two on the right were mistakenly joined. This inconvenience is due to the fact that - for data sets like this - there is no pair of parameters (*eps*, *MinPts*) which is fine for all the different cluster densities. However, such problem is easier to address than the previous one. Most commonly the DBSCAN algorithm is iterated several times - with different and appropriate parameters - so as to find first the highest density clusters and then those with lower density, discarding from time to time those already found.

2.4 Hierarchical clustering algorithm

2.4.1 Description

As the name suggests, the hierarchical clustering algorithm (proposed for the first time in 1950 by Polish researchers) falls into the category of algorithms that consider data-points organized according to a hierarchy. As a consequence of this, each data-point may belong to more than one cluster at different layers. It requires the setting of a parameter k indicating the number of clusters to be found. As we will see later, this is a disadvantage because the value of k is not

always known. That said, this algorithm is available in two variants:

- *agglomerative hierarchical clustering* (bottom-up approach);
- *divisive hierarchical clustering* (top-down approach);

In order to understand, look at the following high level implementation of the agglomerative variant (see Listing 2.4).

Listing 2.4: Alg.3 - Agglomerative hierarchical clustering

```
1  input: data_points, k
2  C = empty array of sets
3  for (data_point in data_points) {
4    C.append({data_point})
5  }
6  while (C.size > k) {
7    find (c1, c2) such that
8            dist(c1, c2) == min(C[i], C[j]) for all (i,j)
9    C.remove(c1)
10   C.remove(c2)
11   C.append({c1, c2})
12 }
```

The basic idea is to start by considering each data-point as a cluster in its own right and then gradually merging these clusters together. So, the `C` array (initialized in line 2 of Listing 2.4) is an array of clusters `c[i]`: in particular each cell of the `C` array is a set representing a cluster and containing all the data-points which belong to the latter. As you can see, at the end of the for loop in line 3 of Listing 2.4 the size of `C` is equal to the number of data-points. Finally, the algorithm stops when such size is less or equal to `k`, as indicated by the while condition in line 6 of Listing 2.4. At each iteration inside this while loop, the 2 nearest clusters are found inside the `C` array (see lines 7-8 of Listing 2.4) and then they are joined. The merge action between themselves clearly results in lines 9-11 of Listing 2.4.

Going into the particular, there is a low level detail not focused in the above pseudo-code: in some iterations there could be more than two clusters with a

distance equal to the minimum, therefore more than two clusters to be merged. This means that in line 12 of Listing 2.4 the size of the `C` array could be decreased by a value greater than 1. But this implies that, at the end of the while loop in line 6 of Listing 2.4, the size of the `C` array could be less than `k`. Please consider that this is a low level implementation detail; in general only one merge at each iteration is allowed. Moreover, pay attention to the `dist` function used in line 8 of Listing 2.4: it computes inter-cluster distances (i.e. distance between groups of data-points), but how is this type of distances defined? Typically it coincides with the maximum, minimum or average among all the combinations of distances between data-points of different clusters. However, there are also other more sophisticated techniques such as distance between centroids or Ward's Method. Besides, even if not reported, consider that the divisive variant is diametrically opposed to this described above.

In Figure 2.22 a toy data set consisting of five data-points is shown. This sample is used to explain how the agglomerative hierarchical clustering performs.

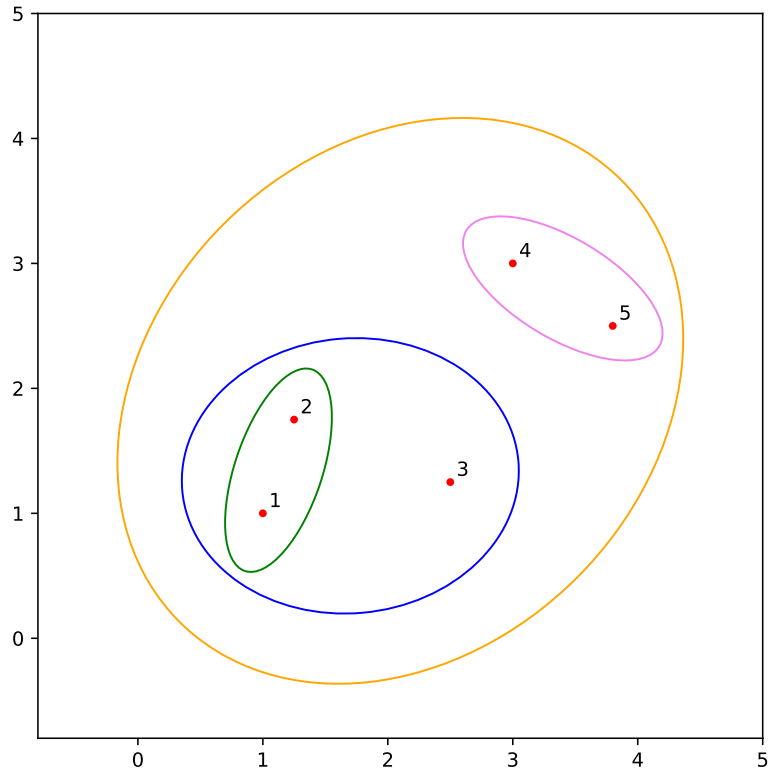

Figure 2.22: Hierarchical clusters

It is clearly depicted that the nearest data-points between them are 1-2 and 4-5, so they form clusters (in green and in violet) at the bottom of the hierarchy. Then the cluster including data-point 3 (in blue) follows and finally - at the top of the hierarchy - there is the cluster including all the data-points (in orange).

In order to capture better the concept of hierarchy, the dendrogram was introduced (see Figure 2.23). It is a tree like diagram where the x-axis indicates data-points and the y-axis indicates inter-cluster distances (in this example the minimum measure is adopted, as mentioned above). The dendrogram is an useful tool to display in a simple way in which levels and between which groups of data-points merges occur.

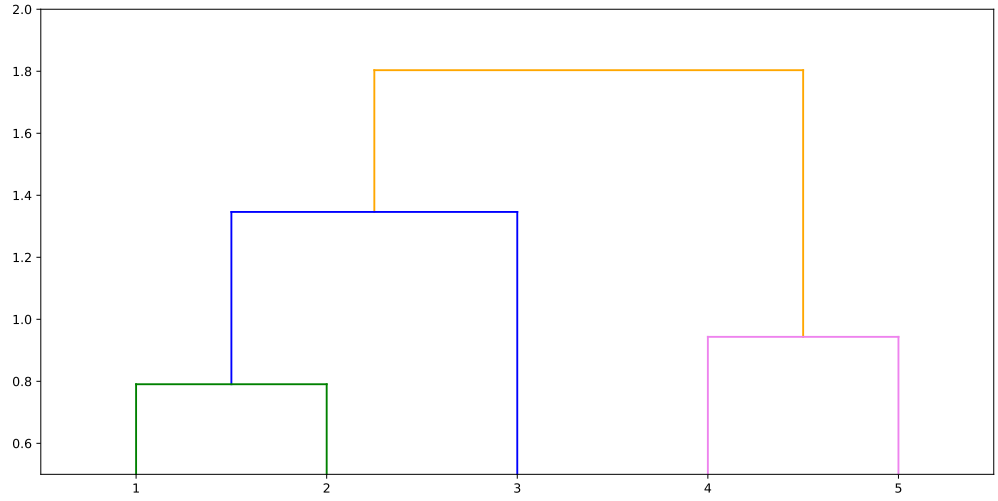

Figure 2.23: Dendrogram

Got to this point, the interpretation of the k parameter is given by a horizontal line that cuts the dendrogram to such a height that at the end k clusters are found. About this, check out the following Figures 2.24-2.25: for example, if $k = 2$ then the previous external orange cluster is discarded.

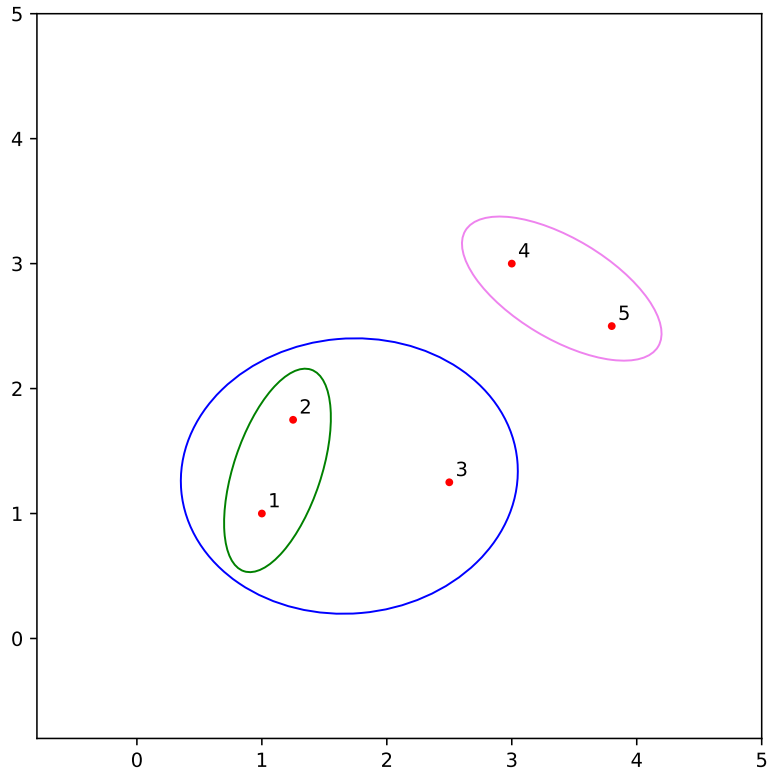

Figure 2.24: Hierarchical clusters with $k = 2$

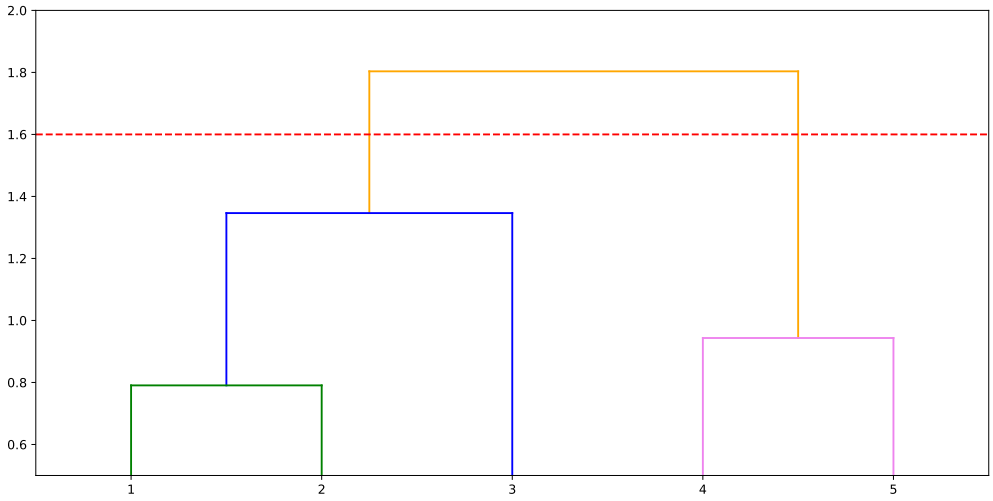

Figure 2.25: Dendrogram with $k = 2$

Similarly to k-means, the hierarchical clustering algorithm performs well when underlying data are organized in well-separated (see Figure 2.26) or center-based clusters (see Figure 2.27).

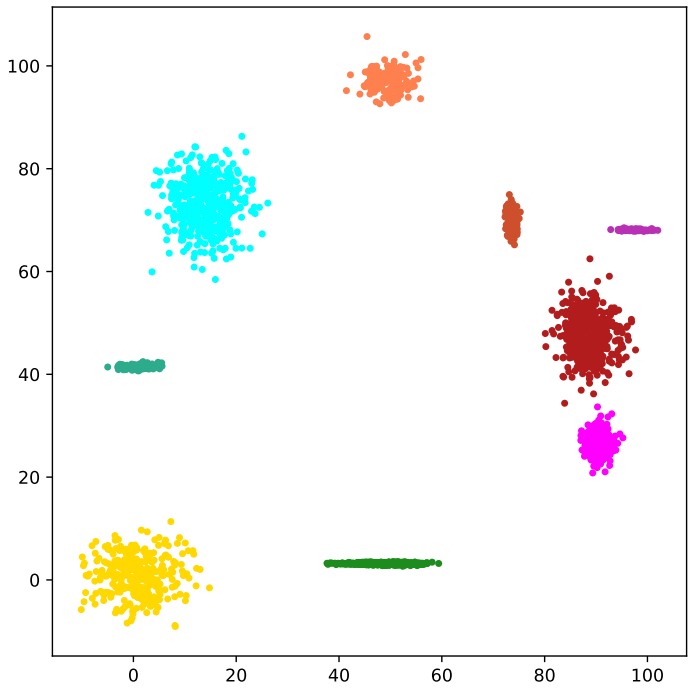

Figure 2.26: Well-separated clusters ($k = 9$)

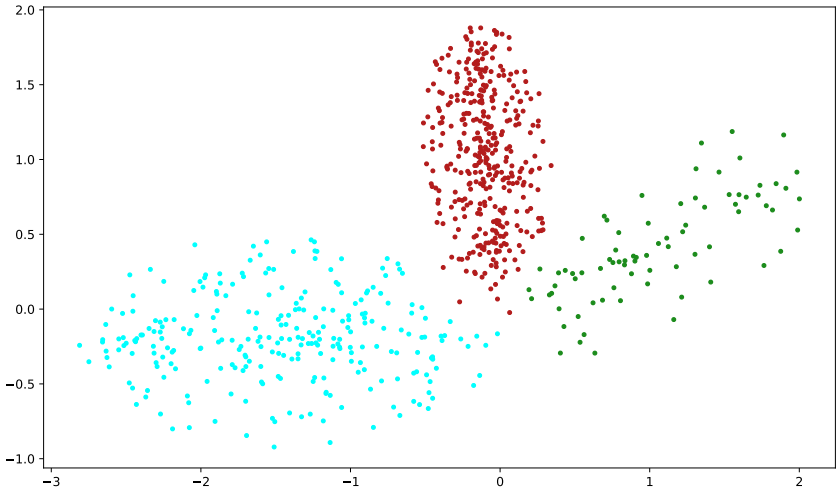

Figure 2.27: Center-based cluster ($k = 3$)

It can be said that the hierarchical clustering algorithm adopts a *greedy approach* because at each iteration (with a local vision) the nearest clusters are merged. Unfortunately it rarely provides the best solution. On the other hand, it is easy to understand and to implement (e.g. respect to more complex algorithms like DBSCAN).

2.4.2 Limitations

The limitations of the hierarchical clustering algorithm are very similar to those of the k-means algorithm:

- it is often not known a priori *what is the right value of* k;
- there are drawbacks when clusters have *different sizes/densities* (see Figure 2.29) or *non-globular shapes* (see Figure 2.28);

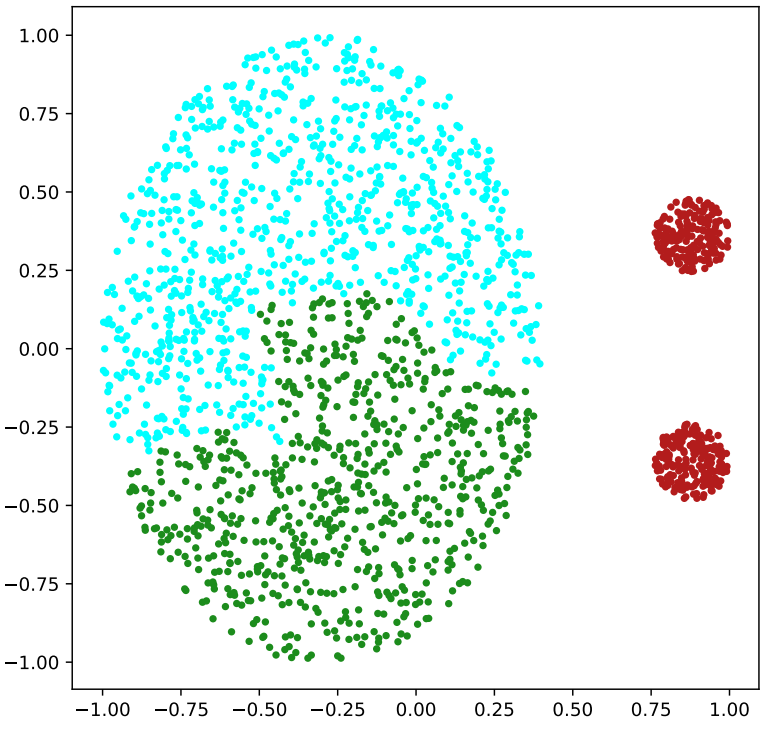

Figure 2.28: Different sizes/densities cluster ($k = 4$)

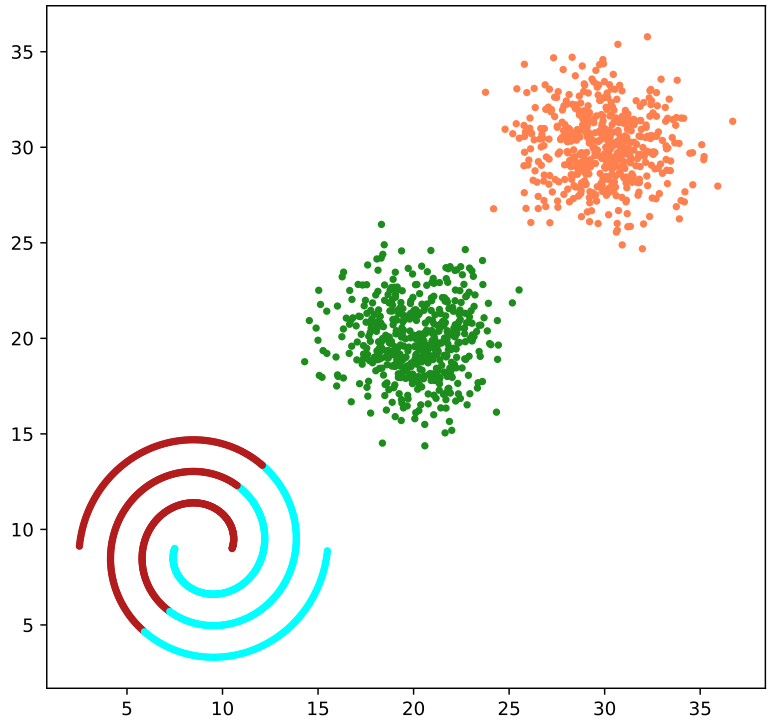

Figure 2.29: Non-globular clusters ($k = 3$)

- the hierarchical clustering algorithm is not able to *detect noise*.

2.5 Summary considerations

The overview described in the previous sections implicitly says that the "perfect" clustering algorithm does not yet exist. In other words, there is no algorithm suitable for every situation since the nature of the clustering problem intrinsically presents three big obstacles:

- there are different types of clusters (e.g. density-based, contiguity-based, well separated, etc);

- there are some features related to the specific data set, even if the type of clusters is the same (e.g. the number of clusters, the minimum number of data-points in a cluster, etc);

- the notion of a cluster can be ambiguous. For example, how many clusters are in the data set in Figure 2.30? Two (see Figure 2.31) or four (see Figure 2.32)?

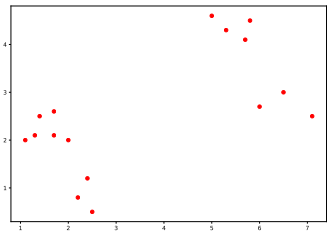

Figure 2.30: Data

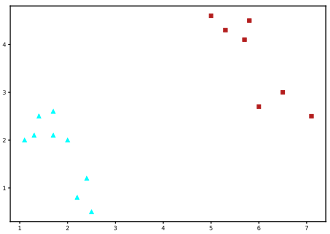

Figure 2.31: 2 Clusters

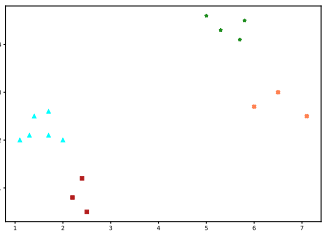

Figure 2.32: 4 Clusters

This is the reason why the *human intervention* is still required to choose - based on the specific data set - the right algorithm with the right parameters. For example, if the data set to be clustered has center-based clusters then the k-means algorithm is surely a good choice, instead if it has contiguity-based clusters maybe it is better to use the DBSCAN algorithm. Please note, however, that the difference between center-based and contiguity-based is something that humans perceive easily, instead machines are not so able. Moreover, also the choice of the parameters is often left to humans, maybe with the help of some diagrams showing a good range in which look for. It is also true that there are variants of the k-means algorithm (bisecting k-means, k-medoids, etc) and the DBSCAN algorithm (OPTICS, etc) which implement more sophisticated solutions to the problem, but it is not enough.

Having acknowledged such limits, the aim of this thesis is to find a clustering algorithm that fits most of (hopefully all) cluster types, requiring few (hopefully zero) parameters to set. Addressing this topic was motivated by an initial idea that has been refined and improved from time to time. As a consequence of this, several algorithms have been designed and implemented, gradually becoming more and more sophisticated. The main ones are ExpansionClustering (version I, version II, version III) and BridgeClustering.

Chapter 3

Proposed clustering algorithms: ExpansionClustering

3.1 ExpansionClustering - version I

This section - like the following ones inside this chapter - is dedicated to the description of an algorithm by organizing the contents into three subsections: the first one is a *graphic illustration* of the algorithm through the use of images, the second one is an *analysis of the weaknesses* looking at possible improvements and finally the third one contains the *pseudo-code* for a more detailed understanding of the algorithm itself.

3.1.1 Graphic illustration

Suppose to have an unlabeled data set like the one in Figure 3.1. This specific data set is useful as sample to explain how the algorithm works. However at the same time some considerations will be gradually done to deduce general ideas. In the following analysis, for graphic simplicity, only 2D data sets are treated so that they can be shown in the Cartesian plane.

The purpose of the ExpansionClustering algorithm - as a clustering algorithm - is to assign a cluster label to each data-point in order to split the whole data set in clusters: the data set depicted in Figure 3.1 has 9 well separated clusters, easily identifiable by the human eye (basically without ambiguity).

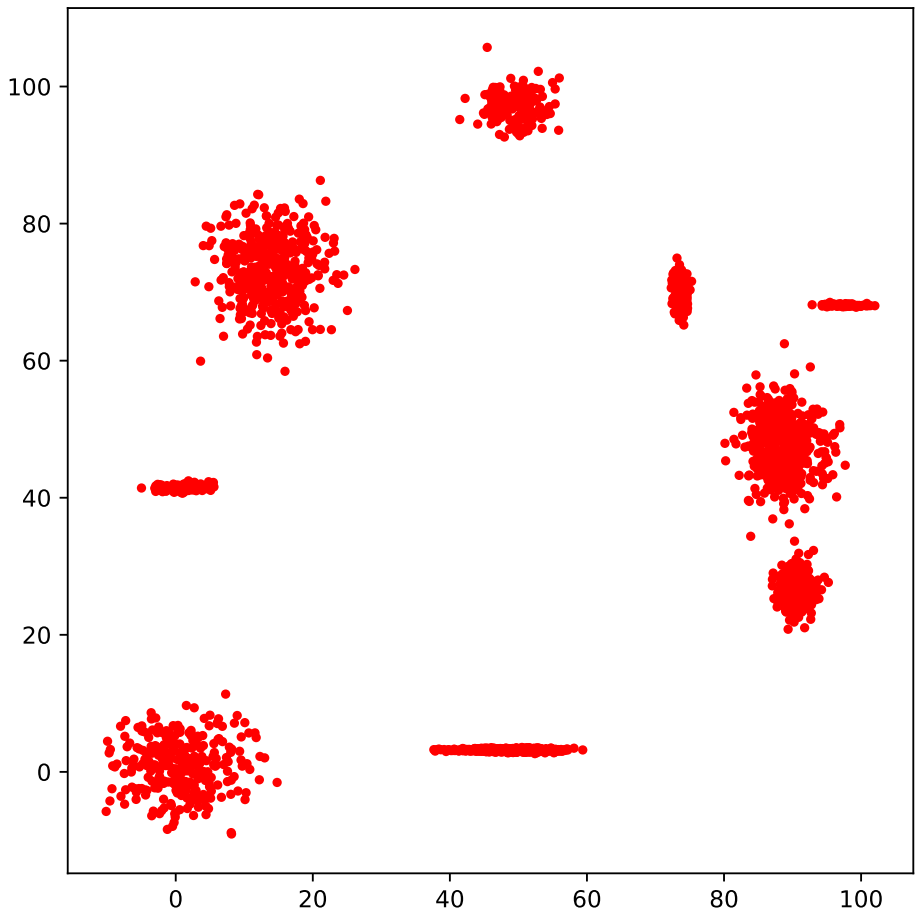

Figure 3.1: Unlabeled data

The starting idea conceived to face the clustering problem is based on the following two phases:

1. the *expansion phase* that consists in creating an image of data-points properly expanded according to some shape (circular or polygonal). This image represents the input for the next phase;

2. the *segmentation phase* that consists in segmenting the above image. The goal is to ensure that data expansions better delineate the shapes of the various clusters, thus favoring the segmentation operation. Then, the obtained segments will correspond to the desired clusters (the details about this are below).

3.1.1.1 The expansion phase

The expansion phase, as just mentioned, consists in expanding each data-point "like an oil stain" according to a circular (see Figure 3.2) or polygonal (see Figure 3.3) shape. In the case of the circular shape, each data-point has:

- an *expansion radius* whose length is half the distance from its nearest neighbor (i.e. two circles are tangent if the corresponding data-points are the nearest to each other reciprocally);

- an *expansion color* in the white-black percentage scale:

  - black if the expansion radius is equals to the minimum value (i.e. the smallest circles);

  - white (i.e. not plotted, invisible) if the expansion radius is equals to the maximum value (i.e. the biggest circles) in order to manage noise, as we will see later;

  - gray shadows for the cases in the middle.

Intuitively, the larger the expansion radius, the more tending to white will be the color. So, the small circles are almost black and the big circles are almost white.

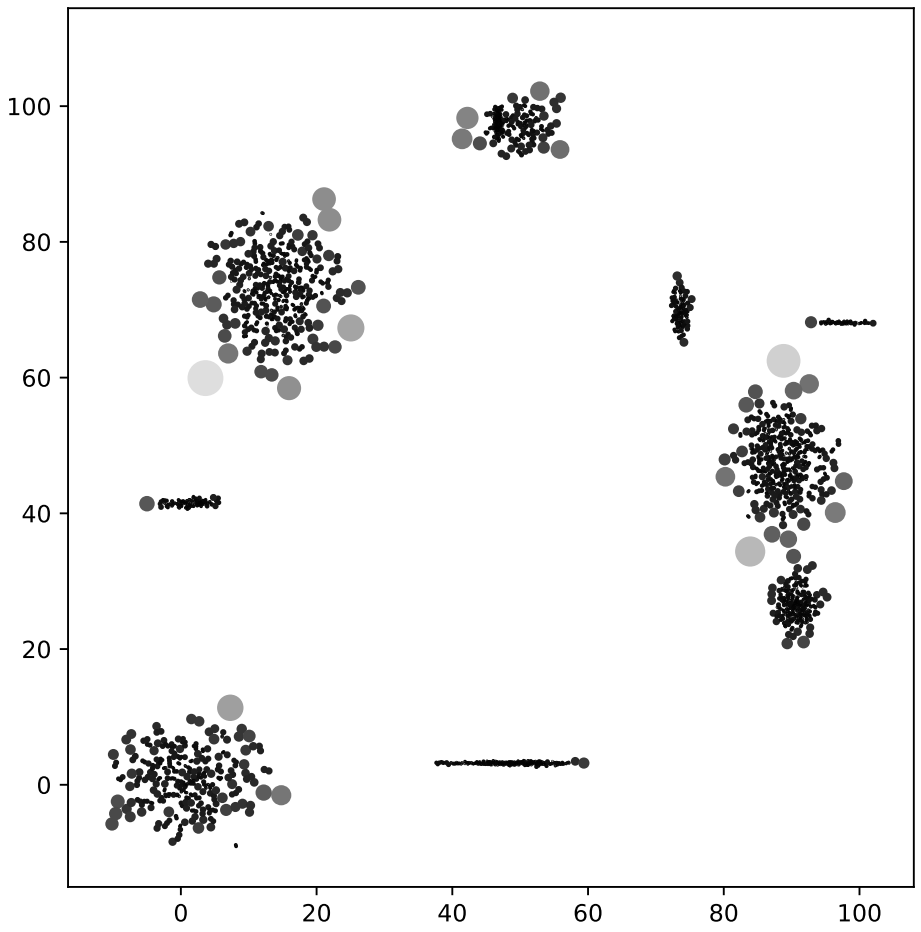

Figure 3.2: Expanded data in circular shape

The above convention about the expansion radius and the expansion color is applied also to the case of the polygonal shape (see Figure 3.3), with the only difference that here - in the expansion radius computing - the apothem is considered instead of the radius (i.e. side-to-side tangent hexagons).

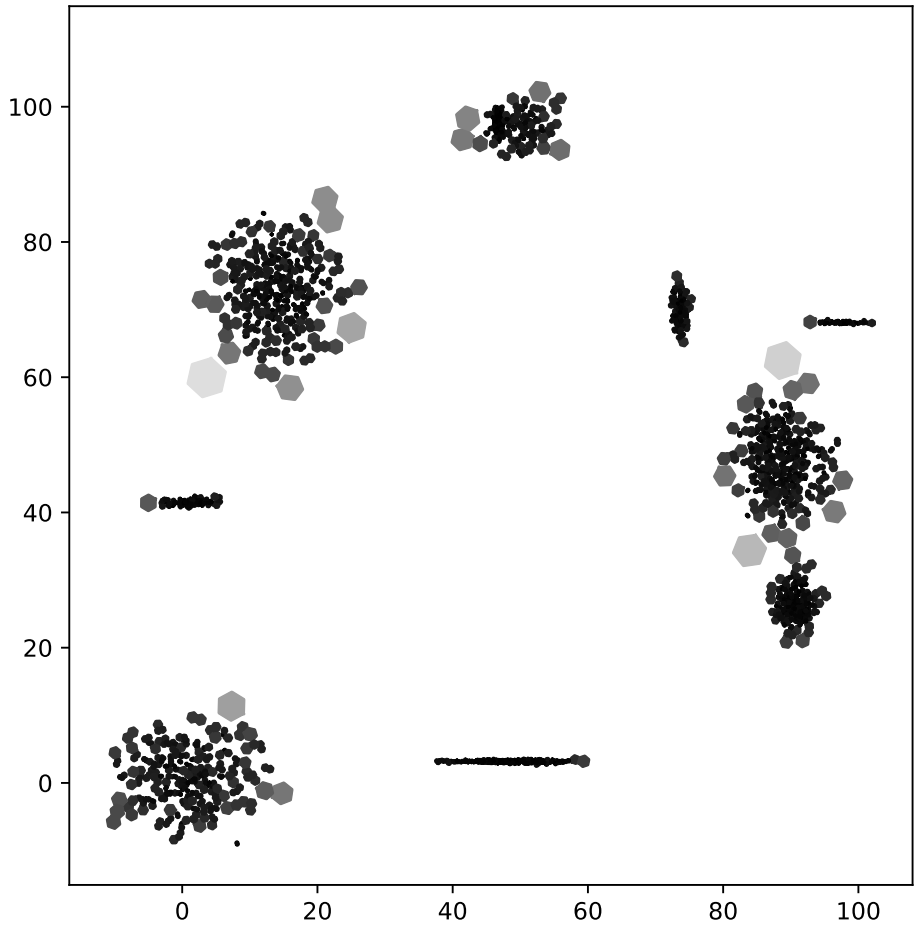

Figure 3.3: Expanded data in hexagonal shape

3.1.1.2 The segmentation phase

Considering, for the moment, the case of the circular shape, the output of the expansion phase is the Figure 3.2. The segmentation phase consists in taking such image as input, segmenting it and returning another image showing the segments found. Graphically these latter segments should coincide with the clusters you are looking for. The segmentation task is performed by using the *watershed algorithm* with default parameter settings (see Figure 3.4).

At the beginning the intuition suggested that the polygonal shape had favored the segmentation, but in reality - after a sufficient number of tests - this was proved to be false, so from now on only circular expansions will be considered since they are less complex (both for programming and for execution).

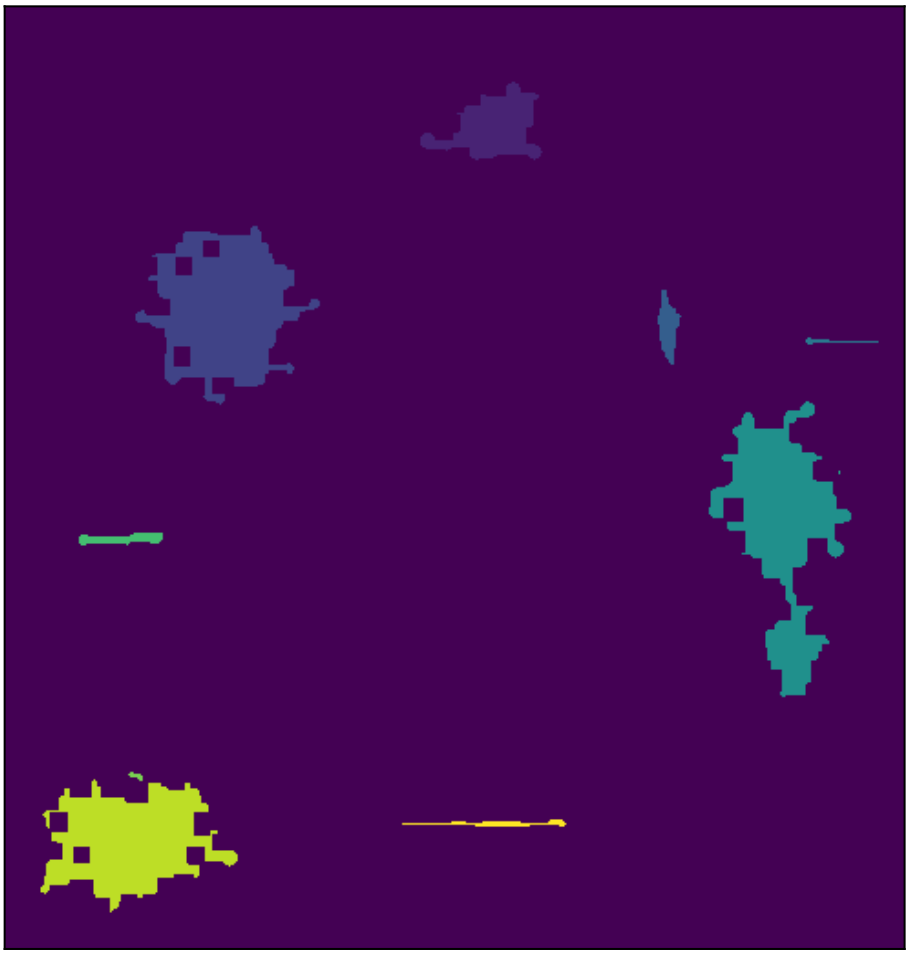

Figure 3.4: Segments obtained from expanded data

The segments shown in the Figure 3.4 are colored areas including most data-points; however there are also data-points that do not fall in any segment, but in the background. That said, the goal is to get the cluster labels from these segments. How to do this? The idea is trivial:

- the data-points that fall in a segment will take the cluster label associated with the segment itself;

- the data-points that fall in the background will take the cluster label associated with the nearest segment, for example by adopting the knn classification technique.

At first sight, the result achieved by segmentation (see Figure 3.4) seems quite good graphically: the number of found segments is 8 rather than 9. But there are situations in which this approach fails, like for the data set in Figure 3.5 with its segmentation in Figure 3.6

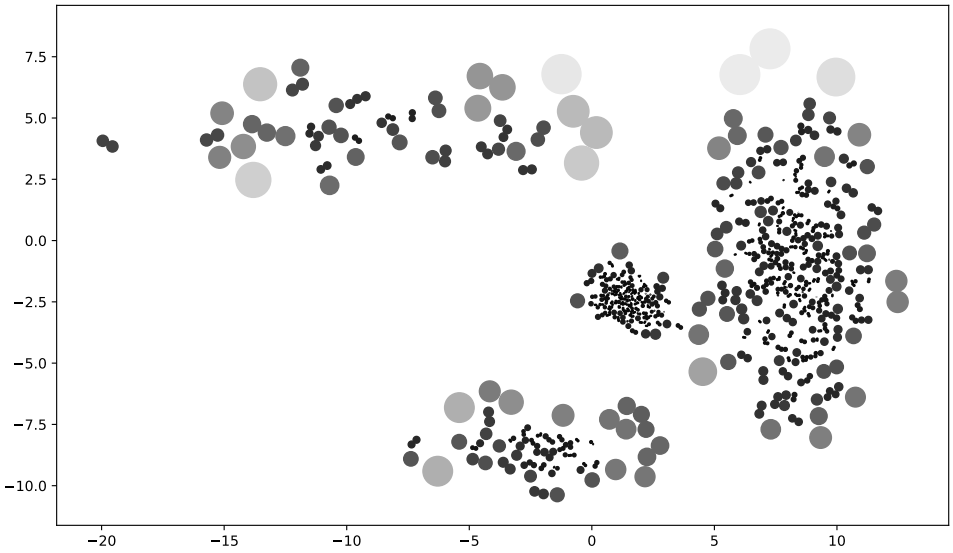

Figure 3.5: Expanded data in circular shape

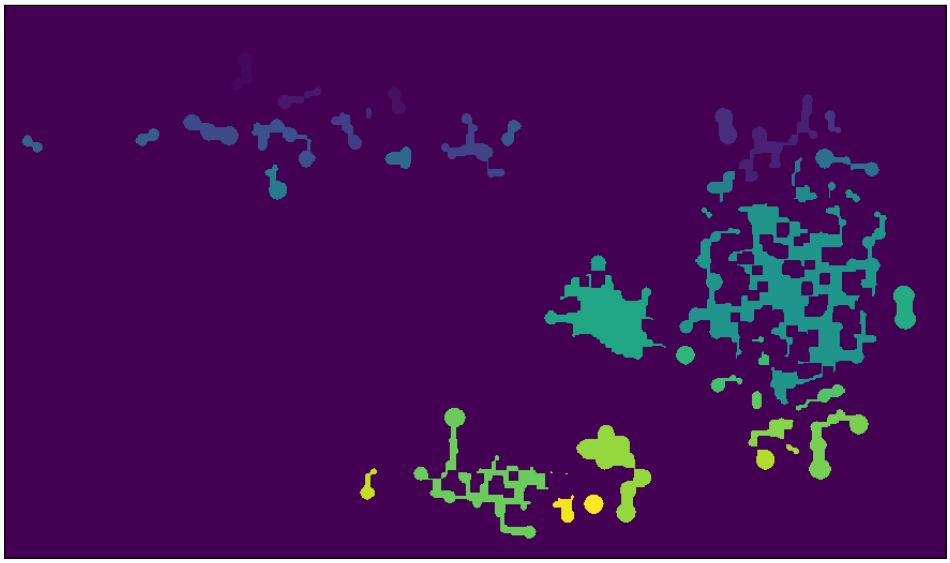

Figure 3.6: Segments obtained from expanded data

In this case, unfortunately, the watershed algorithm returns too many segments (around 40) while the real number of clusters is only 4. Furthermore, there are many situations like this in which the above procedure is not enough.

3.1.2 Analysis of weaknesses

Trying to generalize, the problem discussed at the end of the segmentation phase subsection is that each data-point is seen in correlation only with the first nearest neighbor, and this causes the presence of a lot of small circles which typically generates too many segments. Understood this, a better approach envisages that for each data-point also other k nearest neighbors are considered in the expansion radius computing, and not only the first (the details of such computing are in the version II section). In this way, if the first k nearest neighbors of a data-point are "sparse" then its expansion radius will be bigger, otherwise it remains small as in the previous images. Doing so, the circles should flow into the desired shapes for clusters.

But what if also too big circles are allowed? Then the opposite effect would happen, that is, there will be some circles big enough to merge different clusters (unwanted behavior). So, another parameter is needed, acting as threshold to filter only the expansion rays under itself: this is called the *ert* parameter (expansion-radius-threshold).

All these drafts of improvements will be taken up and deepened in the next section, representing the starting point for the ExpansionClustering version II algorithm.

3.1.3 Pseudo-code

The following listings are the pseudo-code which helps to understand the steps of the EC version I at a lower level.

Listing 3.1: EC version I (part 1)

```
1 data = read_data() # 2D array of (x,y) data-points
2 n = len(data) # number of data-points
```

This is the initial step in which the unlabeled data set is read somehow (e.g. from a file or from the network). The `data` in line 1 of Listing 3.1 is a (n x 2) array containing a 2D data-point `data[i]` for each row i.

3.1.3.1 The expansion phase

Listing 3.2: EC version I (part 2)

```
3 # EXPANSION PHASE ----------------------------------------
4 nn_distances = nearest_neighbors(data, k=1)
```

The `nearest_neighbors` in line 4 of Listing 3.2 is a function that computes, for each data-point in the `data` 2D array (see line line 1 of Listing 3.1), the distances from its first `k` nearest neighbors (ordered from the nearest one), where `k` is specified as a parameter. Since in this case `k=1`, the `nearest_neighbors` function returns a (n x 1) array `nn_distances` that for each row i contains the distance from the data-point `data[i]` to its first nearest neighbor.

Listing 3.3: EC version I (part 3)

```
5 exp_rays = nn_distances / 2
```

The `exp_rays` in line line 5 of Listing 3.3 is a (n x 1) array that contains, for each row i, the expansion radius of the data-point `data[i]`. Such array is equal to half of the `nn_distances` array (ref. expansion phase → expansion radius). The "`/ 2`" operation is to be intended cell by cell.

Listing 3.4: EC version I (part 4)

```
6 exp_min = min(exp_rays)
7 exp_max = max(exp_rays)
8 exp_colors = (exp_rays - exp_min) / (exp_max - exp_min)
```

The `exp_colors` in line line 8 of Listing 3.4 is a (n x 1) array that contains, for each row i, a floating number between [0,1] saying what is the expansion color of the data-point `data[i]`; such value is in the black-white percentage scale (ref. to expansion phase → expansion color). In fact, as you can see, the values of the `exp_colors` array are the values of the `exp_rays` array normalized in the [0,1] range. Doing so (see Listing 3.5):

Listing 3.5: EC version I (part 5)

```
9  if (exp_colors[i] == 0) -> black expansion for data[i]
10 if (exp_colors[i] == 1) -> white expansion for data[i]
```

which means that the smallest circles will be black and the largest ones white.

Listing 3.6: EC version I (part 6)

```
11 exp_image = plot_data_expansions(exp_rays, exp_colors)
```

This is the last step of the expansion phase. The `exp_image` in line line 11 of Listing 3.6 is a data structure representing the image of the expanded data set. To do this task, the `plot_data_expansions` function needs the `exp_rays` array to set the radius and `exp_colors` array to set the color of each data-point expansion.

3.1.3.2 The segmentation phase

Listing 3.7: EC version I (part 7)

```
12 # SEGMENTATION PHASE ---------------------------------------
13 seg_image = watershed(exp_image)
14 assigned_labels = assign_labels(data, seg_image)
```

The segmentation phase is implemented by using the watershed algorithm (see line 13 of Listing 3.7) which segments the previous `exp_image` and returns a new image (`seg_image`) representing the segments found (like those in Figure 3.4). According to the `seg_image`, the cluster labels are assigned to the unlabeled data set by using the `assign_labels` function (see line 14 of Listing 3.7).

Listing 3.8: EC version I (part 8)

```
15 real_labels # (n x 1) array known a priori
16 error = error_measurement(real_labels, assigned_labels)
```

Finally, if the real labels are known a priori, then it is possible to perform an error measurement by counting all the data-points for which a mismatch between the real label and the assigned label has been verified (see Listing 3.8).

3.2 ExpansionClustering - version II

3.2.1 Graphic illustration

The ExpansionClustering version I lays the foundations for the general method, that is the *expansion* of a data set followed by the *segmentation* of the resulting image. In the version II there are still these two phases but, as mentioned in the analysis of weaknesses of version I, two parameters are introduced:

- the k parameter saying how many nearest neighbors must be considered around each data-point;

- the *ert* parameter saying what is the threshold for filtering the expansion rays.

In light of these changes, the concept of the expansion radius of a data-point need to be reviewed.

3.2.1.1 The expansion phase

In the previous section the expansion radius of a data-point was defined as half the distance between the data-point itself and its nearest neighbor, but here there are k nearest neighbors to take into account. So, a new definition is needed: *the expansion radius of a data-point is the average of the distances from its first k nearest neighbors.* The expansion color, instead, is the same as before.

Intuitively this approach makes sense because each data-point is influenced not only by the first nearest neighbor but also by some k others around. As a consequence of this, some expansion rays now are expected to be larger than before since in general the average is greater than the distance from the first neighbor individually. This means that some circles become larger basically (see Figure 3.7).

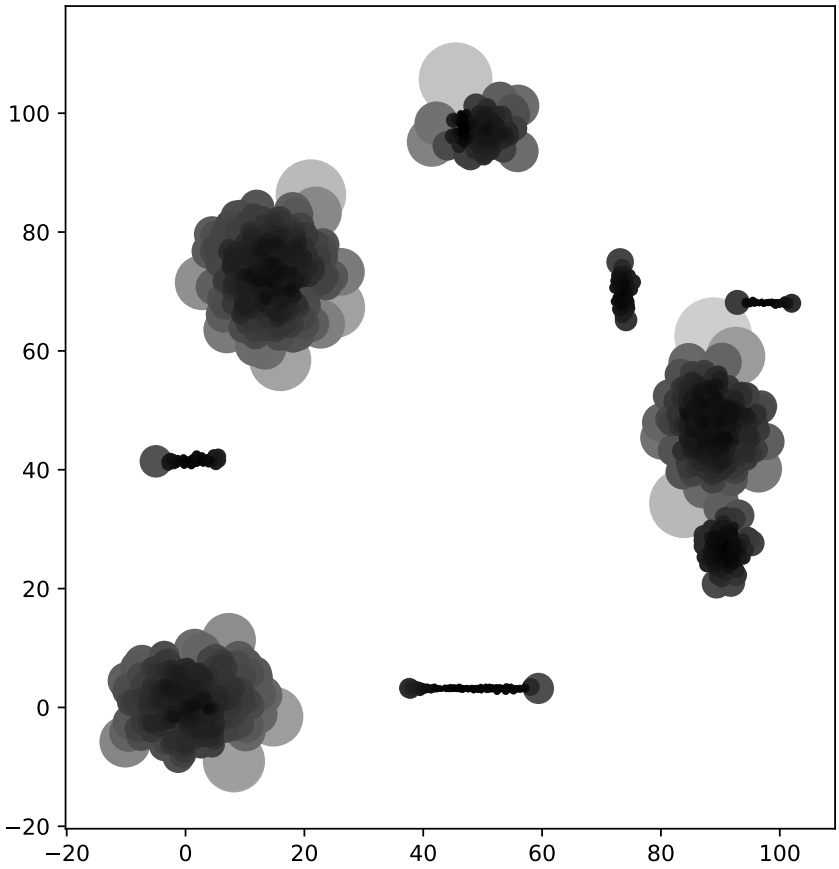

Figure 3.7: Expanded data with $k = 10$

As expected, some cluster shapes become to appear - and this is a good sign - but this is not yet the right solution since the number of segments found will be still wrong. This inconvenience is due to the fact that some circular expansions (at the border) should not appear in the image since they "dirty" the subsequent segmentation.

In order to address this problem, the *ert* parameter is used. So, after having computed the expansion rays, they are normalized in the [0,1] range; then only the expansion rays under the *ert* threshold (that is a floating number between 0 and 1 too) will be filtered and plotted with its expansion color. Essentially, based on the value of *ert*, some circles - larger than the others - will not be plotted to not interfere at the borders of the clusters, so there will be no very light shades of gray (see Figure 3.8).

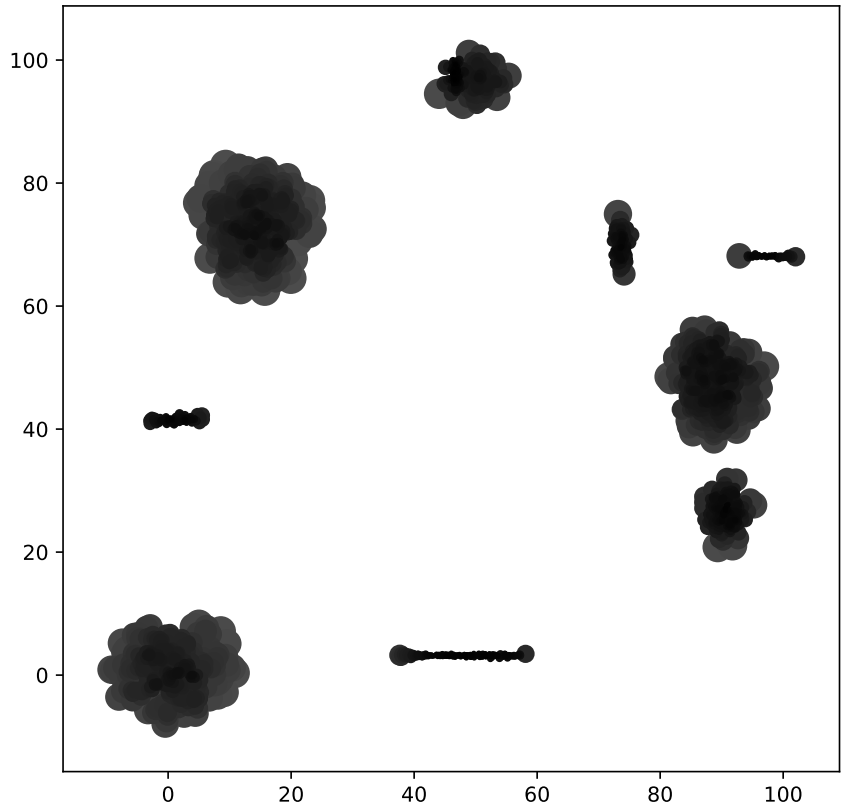

Figure 3.8: Expanded data with $k = 10$, $ert = 0.3$

3.2.1.2 The segmentation phase

The segmentation phase is entirely equal to that one of the ExpansionClustering algorithm version I. Even if the segmentation image is not reported, as you can see in Figure 3.8 the result is optimal now, in fact the watershed algorithm finds exactly the 9 wanted clusters. But a cumbersome (and previously absent) disadvantage has not yet been focused: two parameters have been introduced, k and ert.

3.2.2 Analysis of weaknesses

What is the right value for k and ert? For the previous data set a good result is reached by choosing $k = 10$ and $ert = 0.3$ but, as already seen, in general clusters are really different from each other, so it is very difficult to find a fixed value for k and ert that works in all circumstances.

At this point, a parameters' search on a 2D space (k x ert) was performed to understand what are appropriate parameters for 15 very different data sets. In detail, the ExpansionClustering version II was executed 9 x 13 = 117 times for each data set, setting the parameters obtained by the Cartesian product between the following two sets:

- *k-set* = { 0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7, 0.8, 0.9 }

- *ert-set* = { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 20, 50, 100 }

that is ($k = 0.1$, $ert = 1$), ($k = 0.1$, $ert = 2$), ..., ($k = 0.2$, $ert = 1$), ...

Some successful outcomes are reported in Figures 3.10-3.12 (unlabeled data to the left, expanded data to the right). The segmentation images are not reported to avoid overloading the representation and, anyway, the expansion images shown to the right are already explanatory about the segments which will be found.

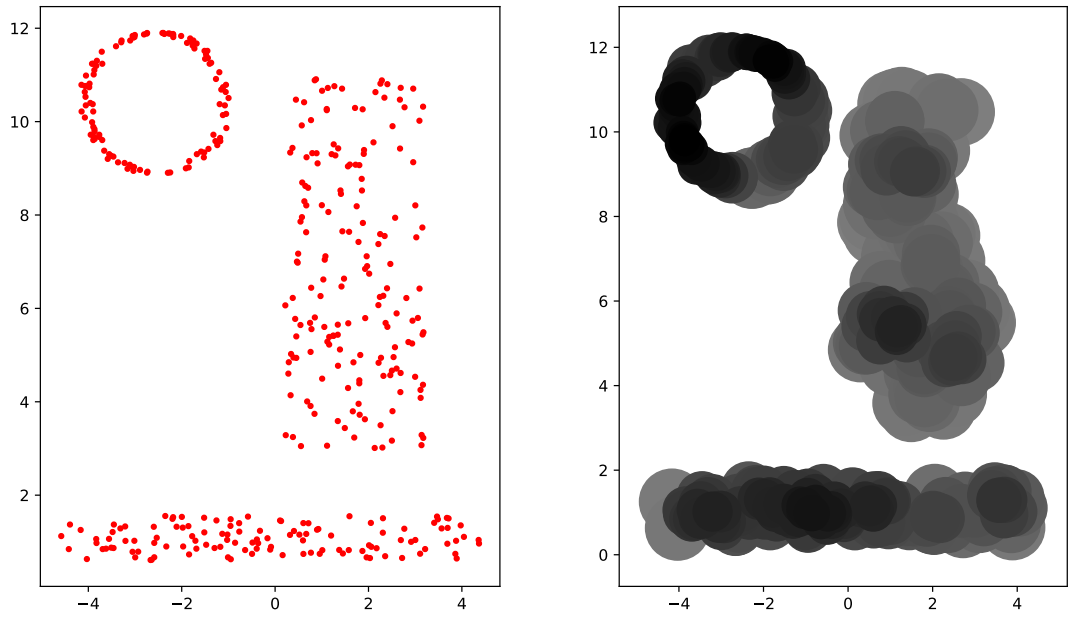

Figure 3.9: Density-based and contiguity-based clusters ($k = 20$, $ert = 0.5$)

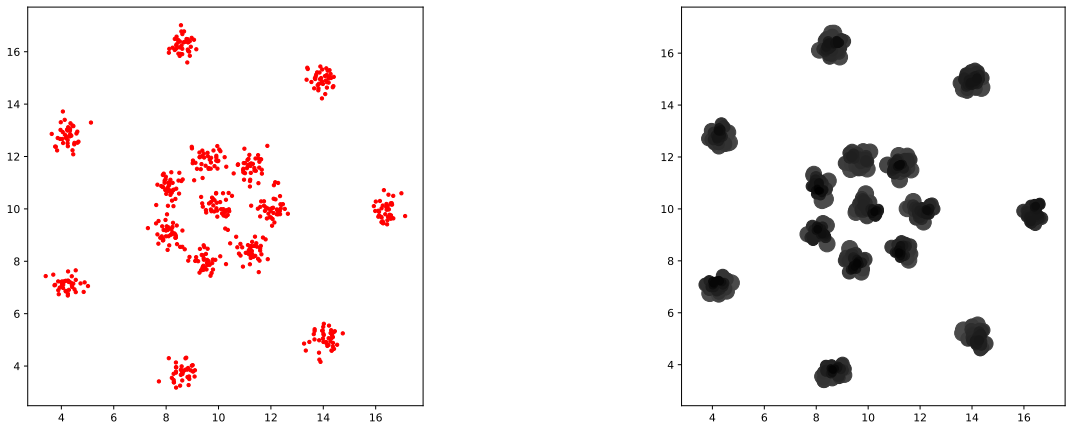

Figure 3.10: Center-based clusters ($k = 10$, $ert = 0.3$)

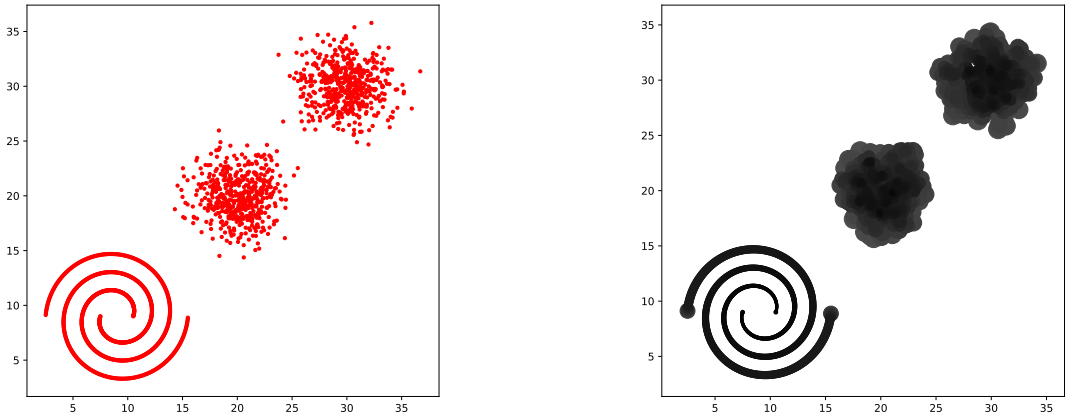

Figure 3.11: Center-based and contiguity-based clusters ($k = 10$, $ert = 0.3$)

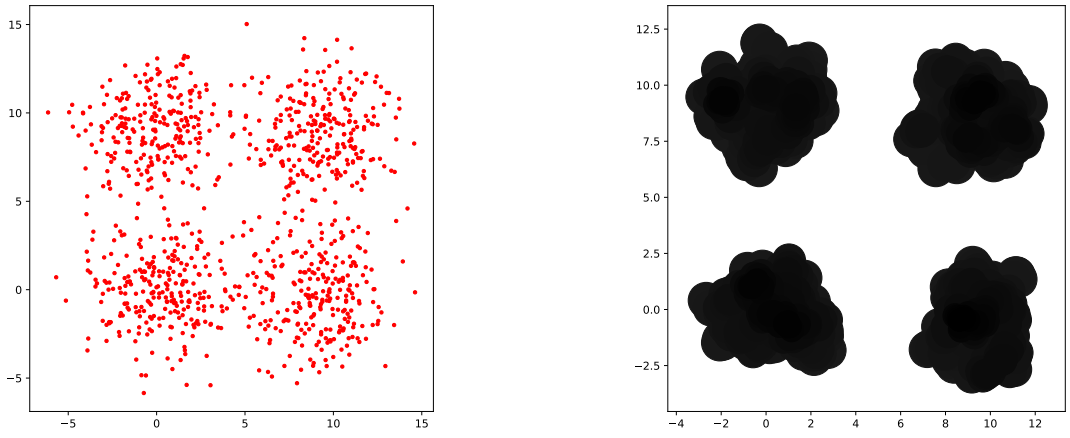

Figure 3.12: Center-based clusters ($k = 20$, $ert = 0.1$)

As you can see, the desired expansions are not obtained always with the same pair of parameters (k, ert). But even worse, there are data sets for which no pair of (k, ert) parameters exists such that the result is acceptable (see Figures 3.13-3.14).

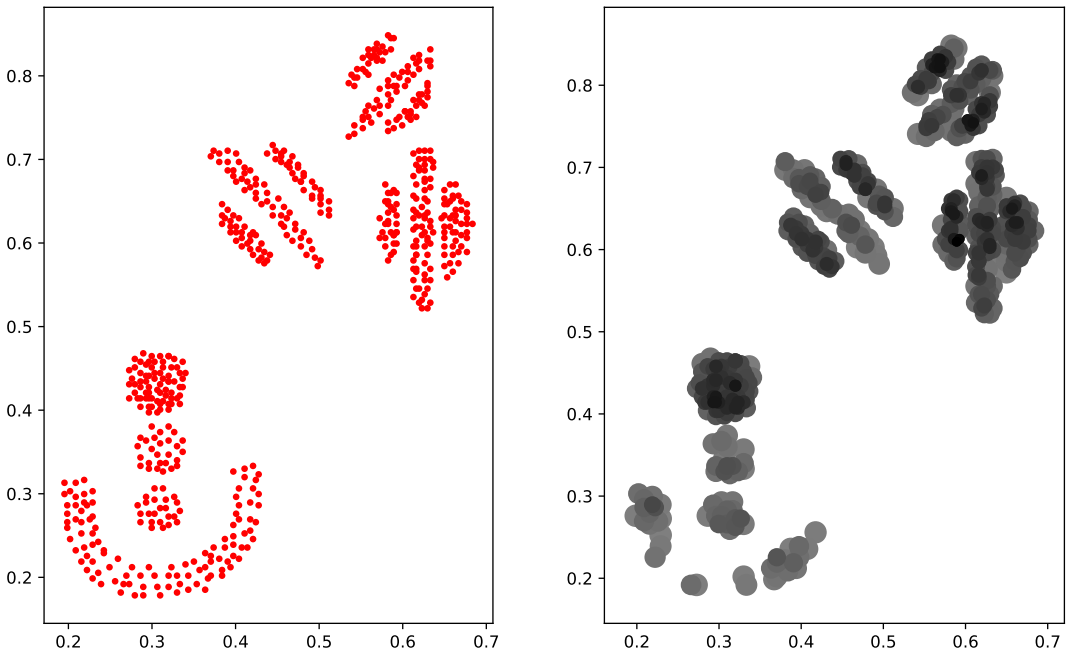

Figure 3.13: Contiguity-based clusters ($k = 5$, $ert = 0.5$)

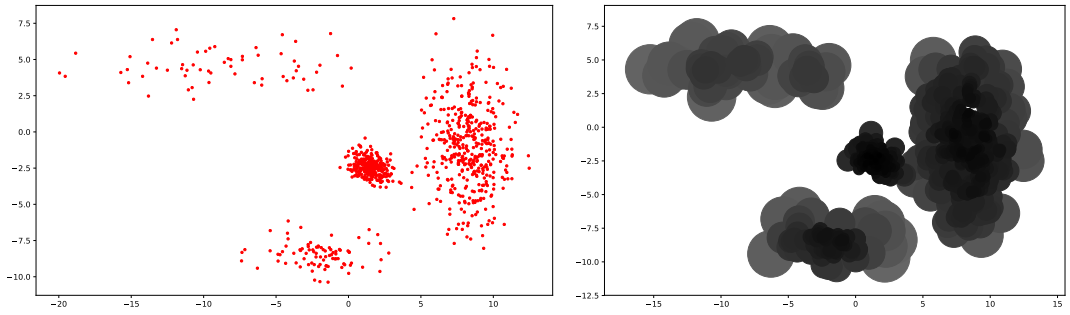

Figure 3.14: Density-based clusters ($k = 10$, $ert = 0.4$)

So, even if the choice of the parameters was somehow automatic, there would still be the unsolved data sets drawback. An useful information to address this problem could be the distinction between *core-points* and *border-points*.

Intuitively, in a data set only core-points should be allowed to make expansions, since border-points often produce big circles which join wrongly different clusters in the same segment. But how to know if a point is core or border?

These basic ideas have led the research towards something more sophisticated which constitutes the heart of the ExpansionClustering version III.

3.2.3 Pseudo-code

The following pseudo-code listings are very similar to those of the version I, hence - for compactness reasons - only the different lines are discussed below. The other lines are the same as in the pseudo-code of the ExpansionClustering algorithm version I.

Listing 3.9: EC version II (part 1)

```
1 data = read_data() # 2D array of (x,y) data-points
2 n = len(data) # number of data-points
3 k = 10
4 ert = 0.3
```

In lines 3-4 of Listing 3.9 the algorithm parameters are set (ref. graphic illustration).

3.2.3.1 The expansion phase

Listing 3.10: EC version II (part 2)

```
5 # EXPANSION PHASE ----------------------------------------
6 nn_distances = nearest_neighbors(data, k)
```

The `nearest_neighbors` in line 6 of Listing 3.10 is a function that computes, for each data-point in the `data` 2D array (see line 1 of Listing 3.9), the distances from its first `k` nearest neighbors (ordered from the nearest one), where `k` is specified as a parameter. Since in this case `k=10`, the `nearest_neighbors` function returns a (n x 10) array `nn_distances` that for each row i contains an array (10 cells long) with the distances from the data-point `data[i]` to its first 10 nearest neighbors.

Listing 3.11: EC version II (part 3)

```
7  exp_rays = [avg(row[:]) for row in nn_distances]
```

The `exp_rays` in line 7 of Listing 3.11 is a (n x 1) array that contains, for each row i, the expansion radius of the data-point `data[i]`, that is the average of the distances from its first 10 nearest neighbors (ref. expansion phase → expansion radius).

Listing 3.12: EC version II (part 4)

```
8  exp_min = min(exp_rays)
9  exp_max = max(exp_rays)
10 exp_colors = (exp_rays - exp_min) / (exp_max - exp_min)
11 exp_colors.filter(x -> x <= ert)
12 exp_image = plot_data_expansions(exp_rays, exp_colors)
```

The filtering action in line 11 of Listing 3.12 implements the behaviour described previously (ref. expansion phase → expansion color).

3.2.3.2 The segmentation phase

Listing 3.13: EC version II (part 5)

```
13 # SEGMENTATION PHASE ---------------------------------------
14 seg_image = watershed(exp_image)
15 assigned_labels = assign_labels(data, seg_image)
16 real_labels # (n x 1) array known a priori
17 error = error_measurement(real_labels, assigned_labels)
```

No change is present for the segmentation phase (see Listing 3.13) compared to the version I.

3.3 ExpansionClustering - version III

3.3.1 Graphic illustration

As discussed in the analysis of weaknesses of version II, the ExpansionClustering version III is based on a method to distinguish core-points from border-points (they are defined in the classification phase subsection). But investigating the problem more deeply, the real limit - from which this need to distinguish data-points arises - is the use of the *ert* parameter. In fact the *ert* filter was applied directly to the expansion radius, but this implies that low-density data-points are not plotted because their expansion rays are over the *ert* threshold, even if the human eye suggests that they should be plotted. The problem here is simply that the expansion radius is an *absolute measure*, so - similarly to what happens to the DBSCAN algorithm - the *ert* filter tends to discard low-density data-points without understanding if they are border-points (it is right to filter) or core-points (it is wrong to filter). In order to improve this filtering action, the *expansion relative* measure will be introduced later.

The ExpansionClustering version III is the final version of the algorithm. It is split in three phases:

- the expansion phase;
- the classification phase;
- the segmentation phase.

3.3.1.1 The expansion phase

The expansion phase is similar to that of the version II. The only difference is that, this time, there is no filtering action with the *ert* threshold, so this parameter is no longer required. Simply, for each data-point, its expansion radius and its expansion color are computed as described for the ExpansionClustering version II (hence the k parameter is still used).

3.3.1.2 The classification phase

The classification phase consists in discerning core-points from border-points, which should be (intuitively) data-points respectively in the center and at the border of a given cluster. This phase makes use of a criterion inferred from an observation on the ExpansionClustering version II execution, in particular by looking at the array of expansion rays. It has been seen that a border-point (differently from a core-point) is characterized by the fact that its expansion radius is often quite different than the expansion rays of its nearest neighbors. Think about one of the previous unsolved data sets (see Figure 3.15) in which, for now, "potential" core-points (in red) and border-points (in yellow/blue) have been labeled by hand.

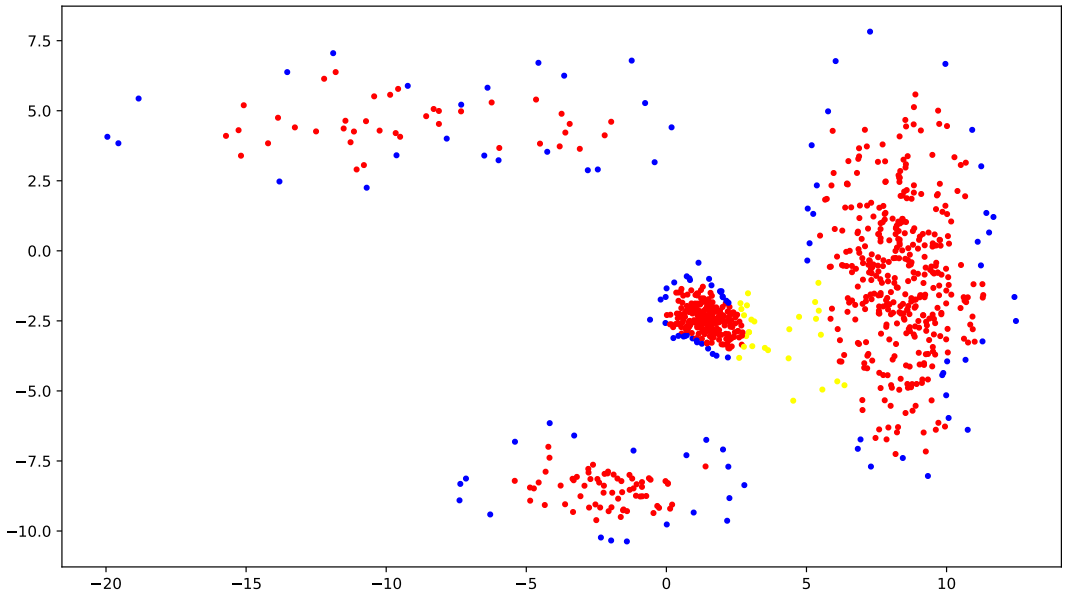

Figure 3.15: Core-points in red, border-points in blue/yellow

For the moment put the k parameter aside (reminding that it says how many nearest neighbors are considered for each data-point) and look at the clusters. Intuitively, a core-point (in red) should have an expansion radius similar to those of its nearest neighbors because they are located all together in an area of constant density. On the other hand, the same cannot be said about a border-point (in blue and in yellow):

- a border-point like those shown in blue has some open sides, in the sense that its nearest neighbors are all in its own cluster, but they are not distributed all around 360 degrees, so its expansion radius will be greater than those of its nearest neighbors (that are mainly core-points);

- a border-point like those shown in yellow is like a hinge between two areas with different density; as a consequence of this, its expansion radius is affected by both low-density and high-density data-points (as these are its nearest neighbors), therefore it will be in general:

  - greater than the expansion rays of the high-density nearest neighbors;
  - lower than the expansion rays of the low-density nearest neighbors.

This concept makes sense, but pay attention to the fact that "greater than" and "lower than" terms have been used. How to quantify the differences among expansion rays? It is better to explain the procedure with a hand-made small data set (see Figure 3.16).

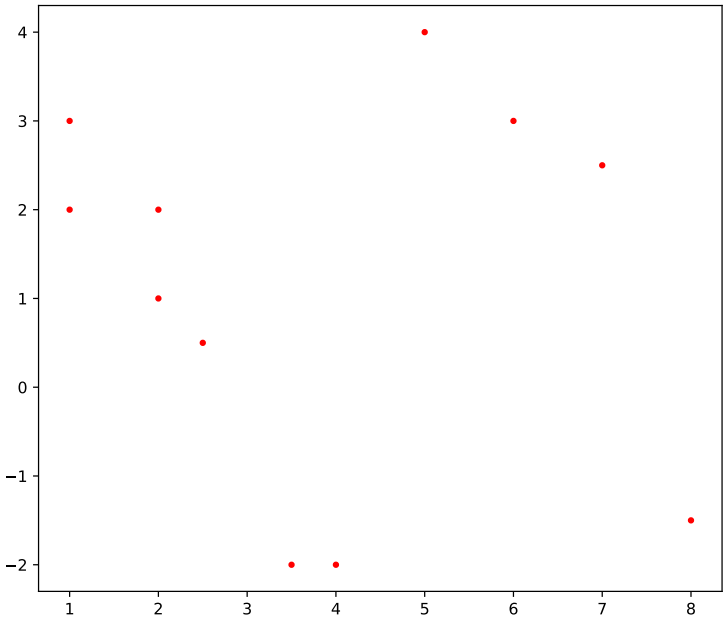

Figure 3.16: Unlabeled data

Suppose to set $k = 3$. First of all, the expansion rays are computing as depicted in Figure 3.17 (as usual the expansion radius of a data-point is the average of the distances from its first $k = 3$ nearest neighbors). Then, for each data-point, another expansion measure called *expansion relative* is computed as follows in Figure 3.18 and in the subsequent explanation.

data	dist-1	dist-2	dist-3
(1, 2)	1	1	1.41
(1, 3)	1	1.41	2.24
(2, 1)	0.71	1	1.41
(2, 2)	1	1	1.41
(2.5, 0.5)	0.71	1.58	2.12
(3.5, -2)	0.5	2.69	3.35
(4, -2)	0.5	2.91	3.61
(5, 4)	1.41	2.5	3.61
(6, 3)	1.12	1.41	4.12
(7, 2.5)	1.12	2.5	4.12
(8, -1.5)	4.03	4.12	4.53

AVG

exp_radius
1.14
1.55
1.04
1.14
1.47
2.18
2.34
2.51
2.22
2.58
4.23

Figure 3.17: Expansion radius computing

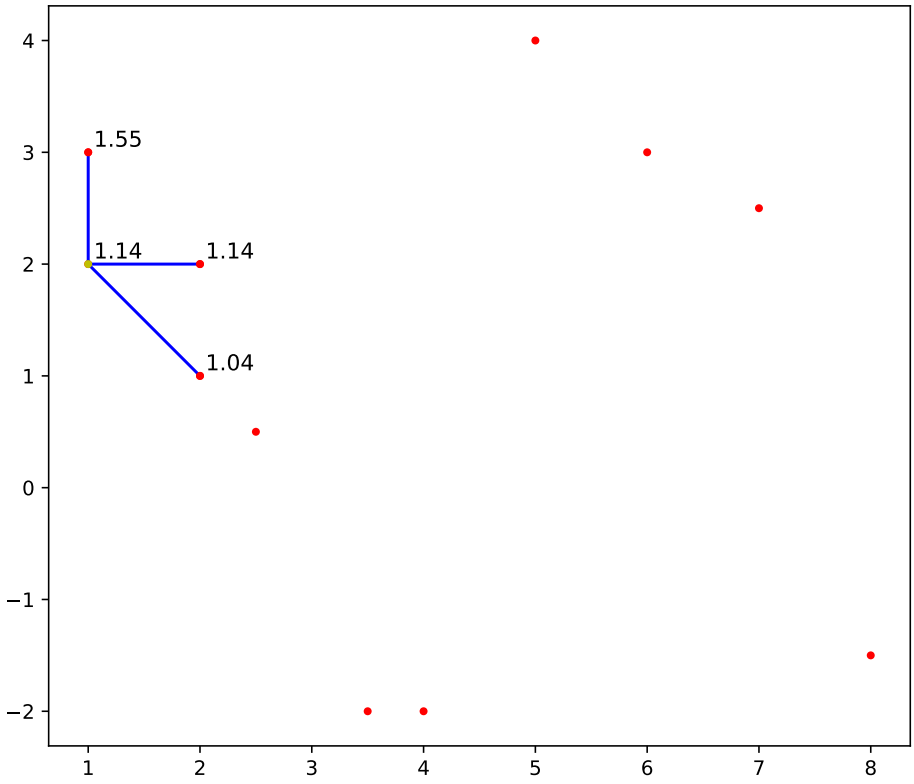

Figure 3.18: Expansion relative computing

Consider the data-point (1, 2) highlighted in golden yellow: its expansion radius is equal to 1.14. Watching its first $k = 3$ nearest neighbors (whose expansion rays are equals to 1.55, 1.04 and 1.14), for such data-point (1, 2):

$$exp\_radius = 1.14 \tag{3.1}$$

$$nn\_avg = \frac{(1.55 + 1.04 + 1.14)}{3} = 1.24 \tag{3.2}$$

$$exp\_relative = \frac{|exp\_radius - nn\_avg|}{exp\_radius} = \frac{|1.14 - 1.24|}{1.14} = 0.0921 \tag{3.3}$$

Finally, the expansion relative is calculated as indicated in the Equation 3.3. The same procedure is executed for each data-point in the data set, so at the end the outcome shows in Figure 3.19 is achieved.

exp_radius
1.14
1.55
1.04
1.14
1.47
2.18
2.34
2.51
2.22
2.58
4.23

exp_relative
0.0921
0.2868
0.2001
0.0921
0.2479
0.2591
0.3316
0.2105
0.0646
0.1564
0.4399

Figure 3.19: Expansion relative computing

In other words, the expansion relative compares a given expansion radius with the *average expansion radius* of the nearest neighbors (that substantially is an average of averages). So, given a data-point, the lower its expansion relative the more it is similar to its nearest neighbors. In a nutshell, the expansion relative could act as a discriminator between core-points and border-points. This is supported by the fact that the expansion relative of a data-point, as the name suggests, is a *relative measure*, in the sense that it is independent from the density around the data-point itself.

This is a very important concept. As discussed at the beginning of this section, the expansion radius is an absolute measure, so the previous *ert* parameter (which acts as discriminator between plotted/not plotted expansions) is strictly bound to the nature of the data set in question. As a consequence of this, it is hard to find a fixed value of *ert* that fits many situations. The expansion relative, instead, is a better discriminator because it sees each expansion radius in correlation with the others around: so a new threshold parameter - called t parameter - is introduced, and its filtering action is applied to the expansion relative in order to discard (from the plotting) hopefully the "real" border-points. For example, suppose to set $t = 0.25$ (see Figure 3.20).

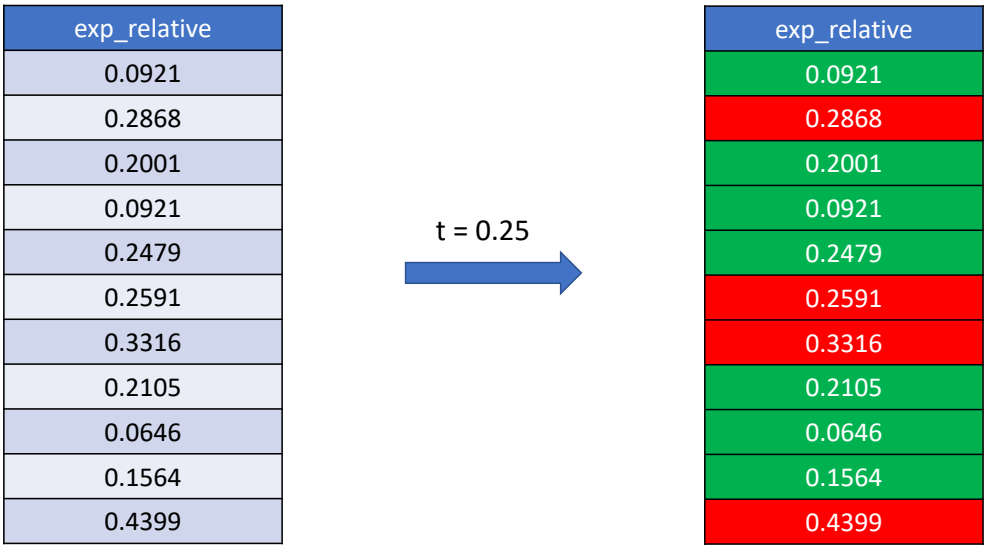

Figure 3.20: Expansion relative filtering

The data-points associated with the green cells (to the right in Figure 3.20) are supposed to be core-points, instead those associated with the red cells are supposed to be border-points. After making this distinction, only core-points will produce expansions, without the *ert* filtering action used in the version II, as depicted in Figure 3.21.

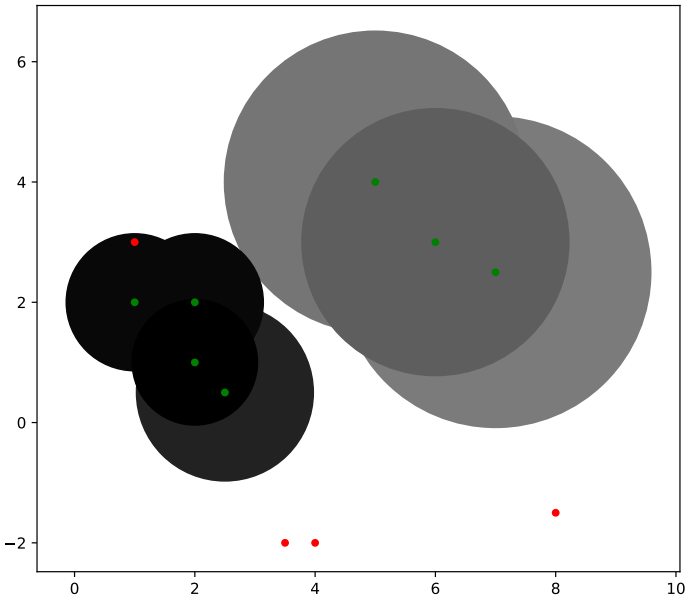

Figure 3.21: Expanded data with $k = 3$, $t = 0.25$

As you can see, with this choice of parameters, the three data-points in red are considered as border-points, so no expansion is plotted for them. Unfortunately this data set is too small (11 data-points) to see some relevant results, it has been useful only to explain how the classification phase works. Significant considerations will be done in the analysis of weakness subsection.

3.3.1.3 The segmentation phase

The segmentation phase has not changed since the previous two versions. As usual the watershed algorithm takes as input the expansion image (in output from the classification phase) and produces the segmentation image from which the cluster labels will be deduced.

3.3.2 Analysis of weaknesses

Also in version III, the problem of finding appropriate values for the algorithm parameters (this time k and t) persists. Fortunately, after a certain number of tests with clusters of different types, it has been observed that the pair ($k = 10$, $t = 0.1$) is fine almost always: the advantage of having a pair of fixed parameters comes from the transition from the absolute *ert* to the relative t threshold. So, for the moment it could be assumed that this is no longer a problem. The real problem encountered is another, as discussed below.

The following Figures 3.22-3.30 are a list of successful outcomes (unlabeled data to the left and expanded data to the right) achieved with the ExpansionClustering version III setting ($k = 10$, $t = 0.1$). Like for the version II, segmentation images are not reported to avoid overloading the representation and, anyway, the expansion images shown to the right are already explanatory about the segments which will be found.

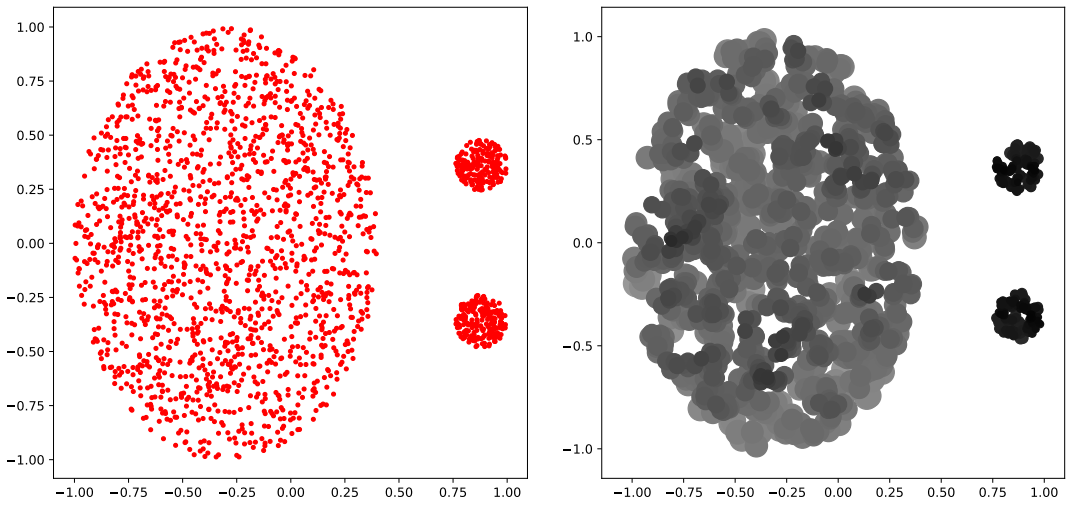

Figure 3.22: Center-based clusters ($k = 10$, $t = 0.1$)

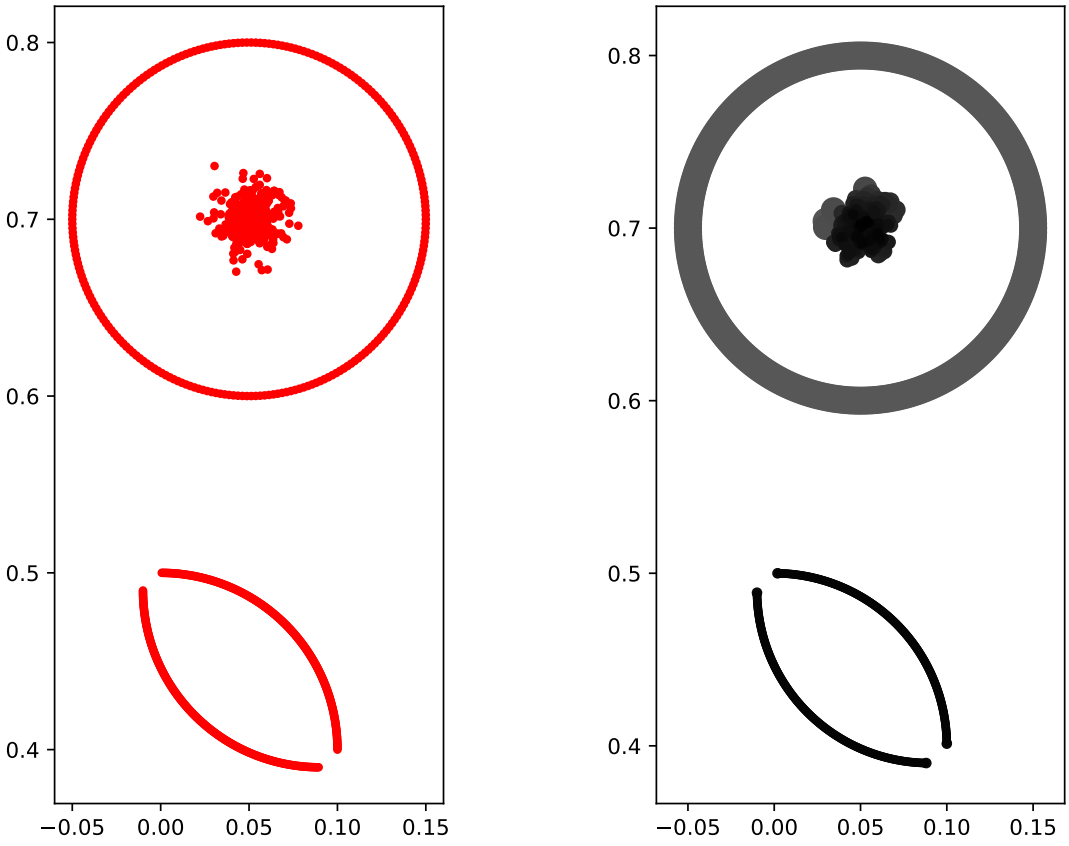

Figure 3.23: Contiguity-based clusters ($k = 10$, $t = 0.1$)

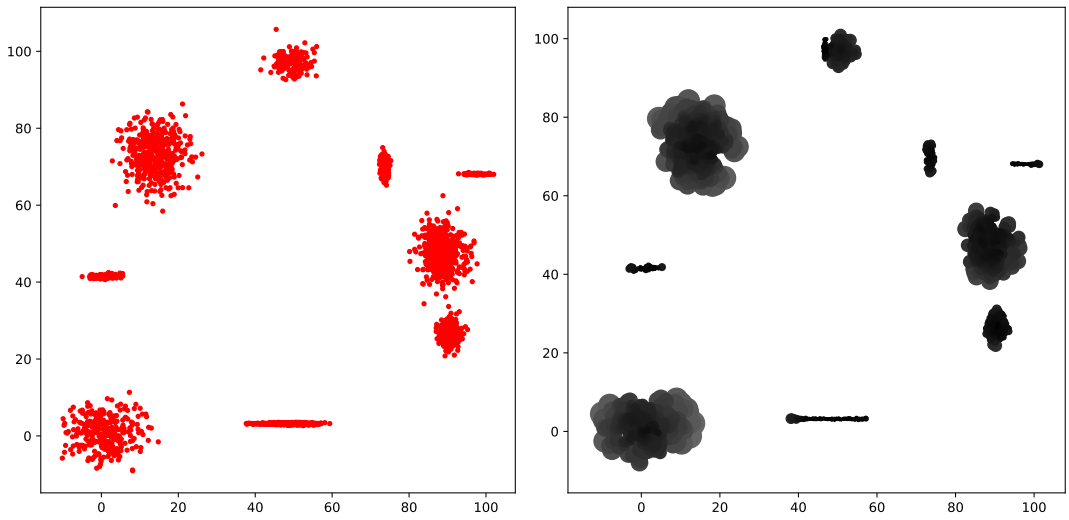

Figure 3.24: Well separated clusters ($k = 10$, $t = 0.1$)

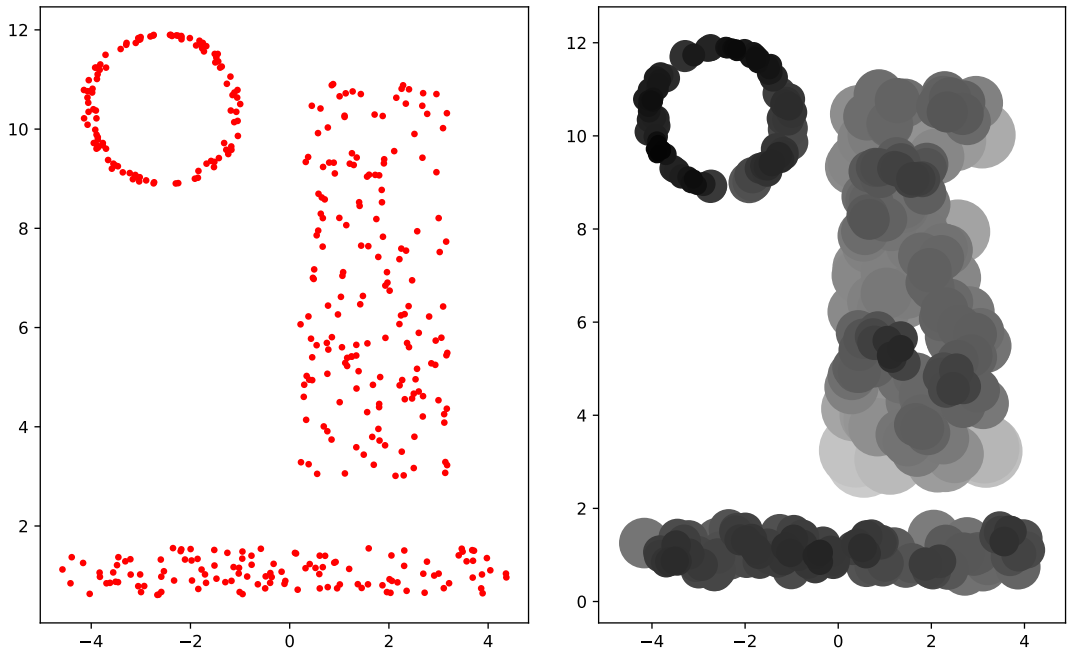

Figure 3.25: Density-based and contiguity-based clusters ($k = 10$, $t = 0.1$)

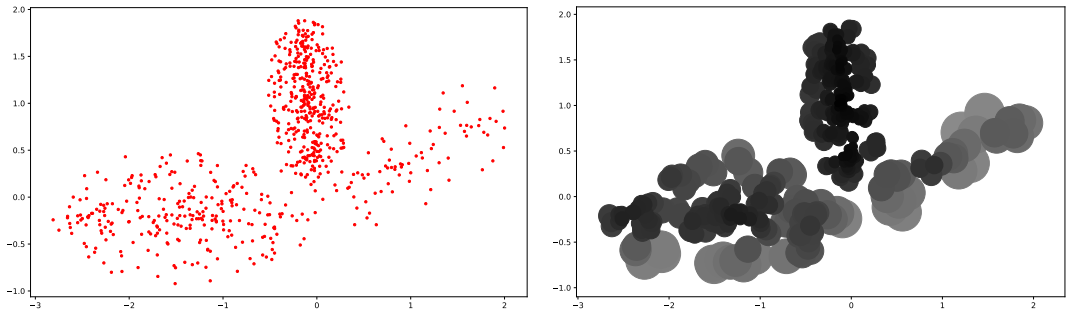

Figure 3.26: Density-based clusters ($k = 10$, $t = 0.1$)

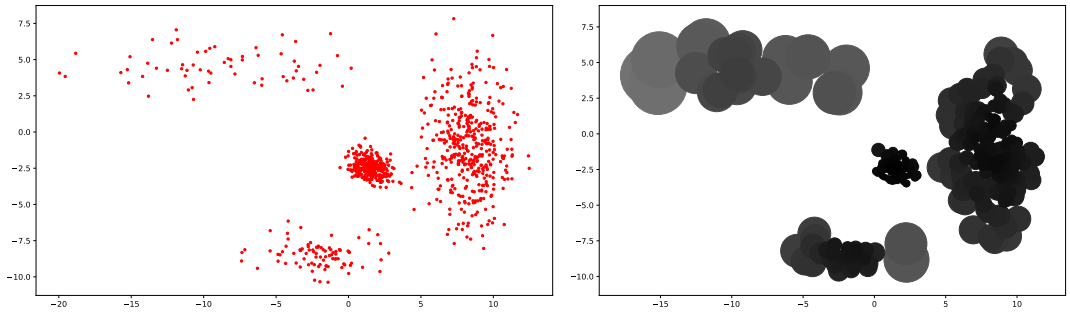

Figure 3.27: Density-based clusters ($k = 10$, $t = 0.1$)

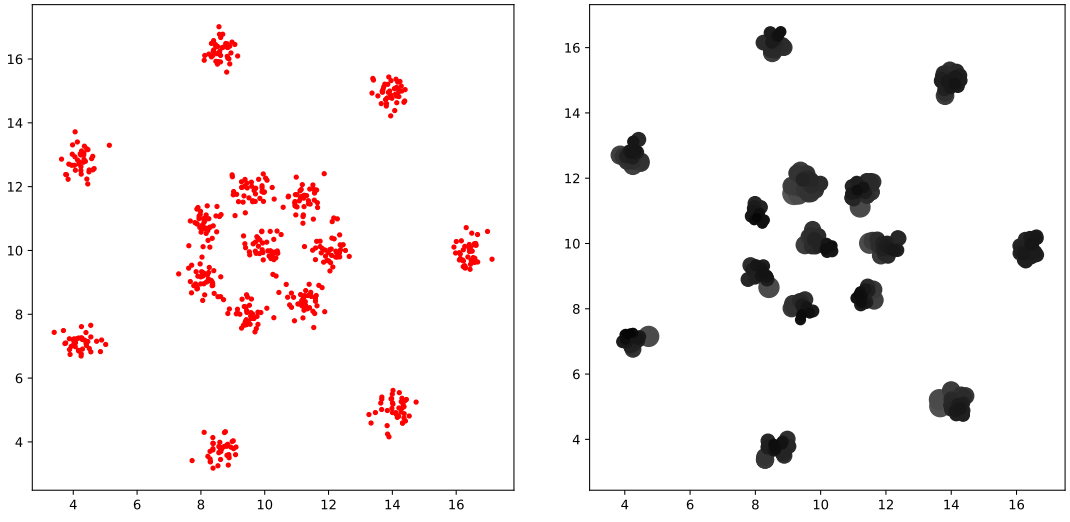

Figure 3.28: Center-based clusters ($k = 10$, $t = 0.1$)

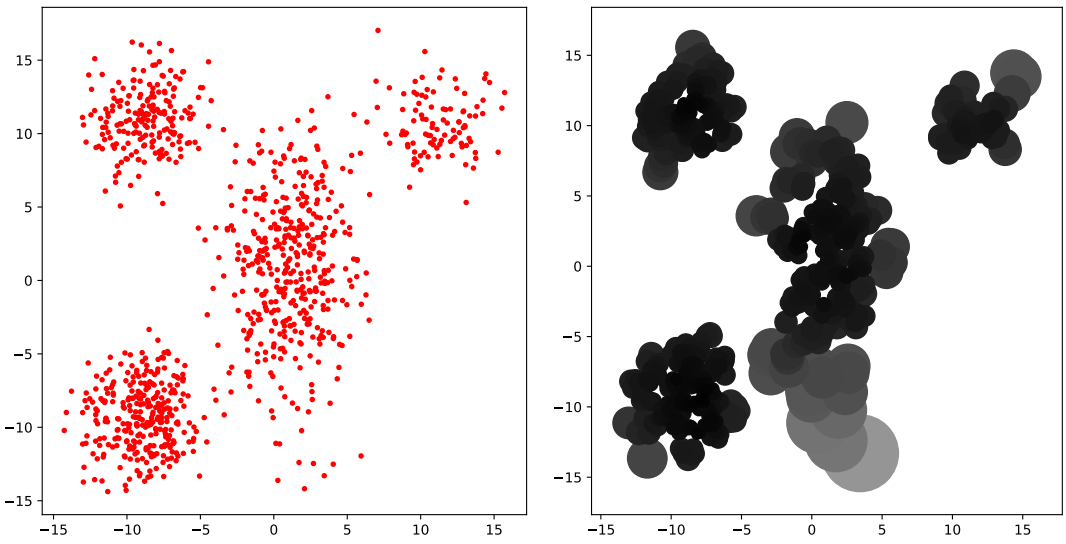

Figure 3.29: Center-based clusters ($k = 10$, $t = 0.1$)

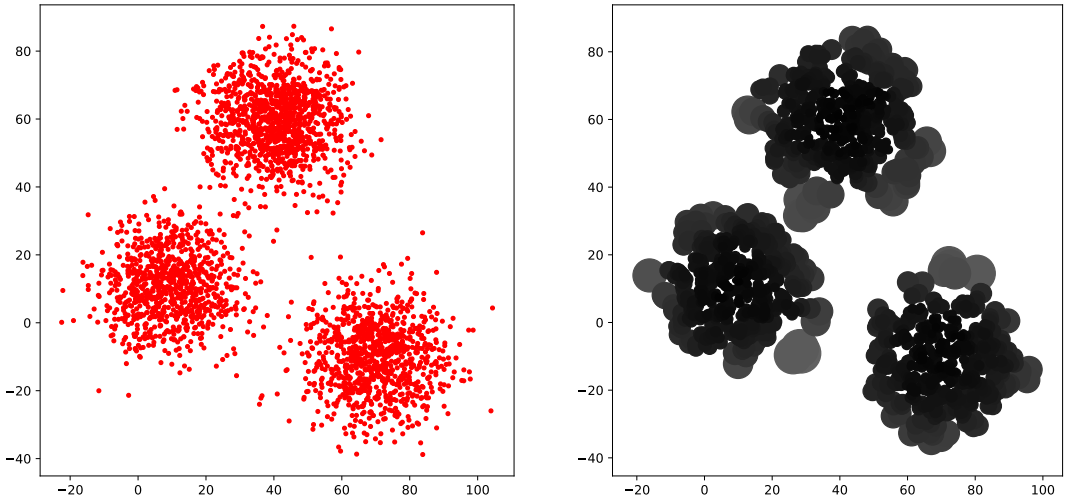

Figure 3.30: Center-based clusters ($k = 10$, $t = 0.1$)

As you can see, a significant improvement was achieved compared to version II, since the choice of parameters is indicated in a more "stable" way and a wider collection of data set types is clustered correctly. Unfortunately, the clustering problem is really difficult to solve in its entirety, in the sense that there are still some unsolved data sets that not even the version III is able to deal with (see Figures 3.31-3.32).

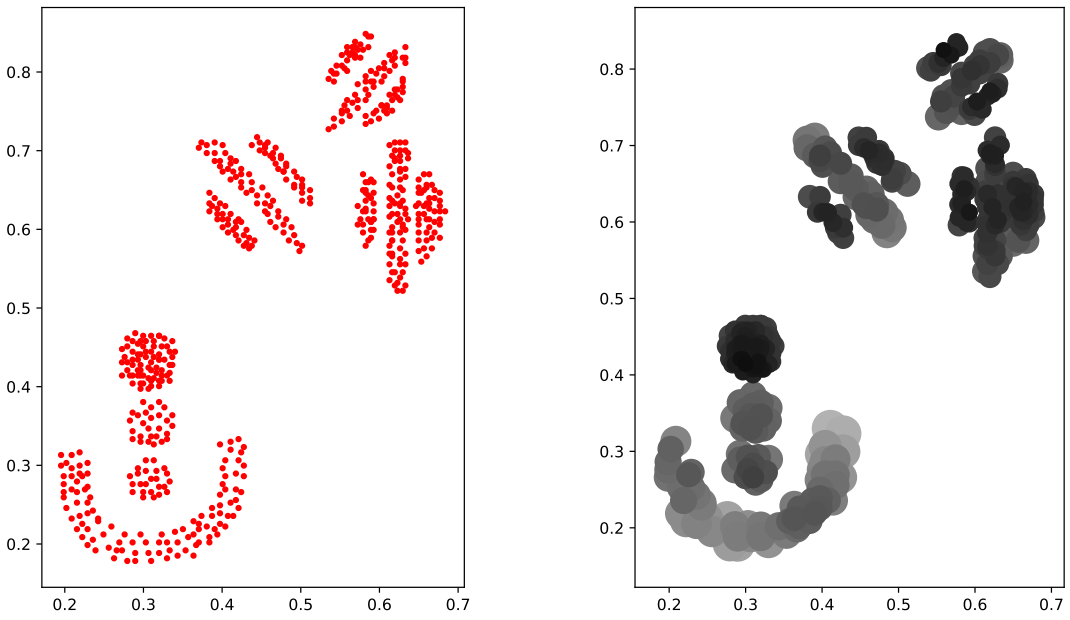

Figure 3.31: Contiguity-based clusters ($k = 10$, $t = 0.1$)

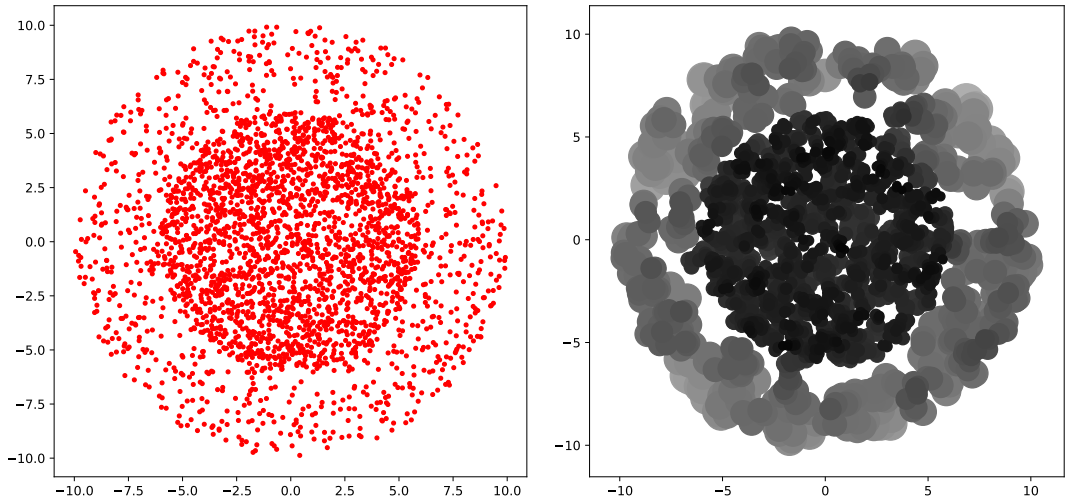

Figure 3.32: Density-based clusters ($k = 10$, $t = 0.1$)

What is the problem here? The version III fails with these data sets because, unfortunately, the different clusters are *too near to each other*. In fact, it has been observed that in general things go right in all those cases where different clusters are "sufficiently distant" to each other, independently from the type of the clusters themselves. This is the limit. If different clusters are too near to

each other, then it is hard to detect core-points and border-points correctly. As a consequence of this, the data expansions will be wrong and dirty the expansion image, so the subsequent segmentation will fail. Please consider that the key for a successful segmentation is to make sure that there are enough empty spaces between different clusters. If this is not true, the clustering result is not the desired one.

Furthermore, there may be some problems related to the possible presence of noise, as depicted in Figure 3.33.

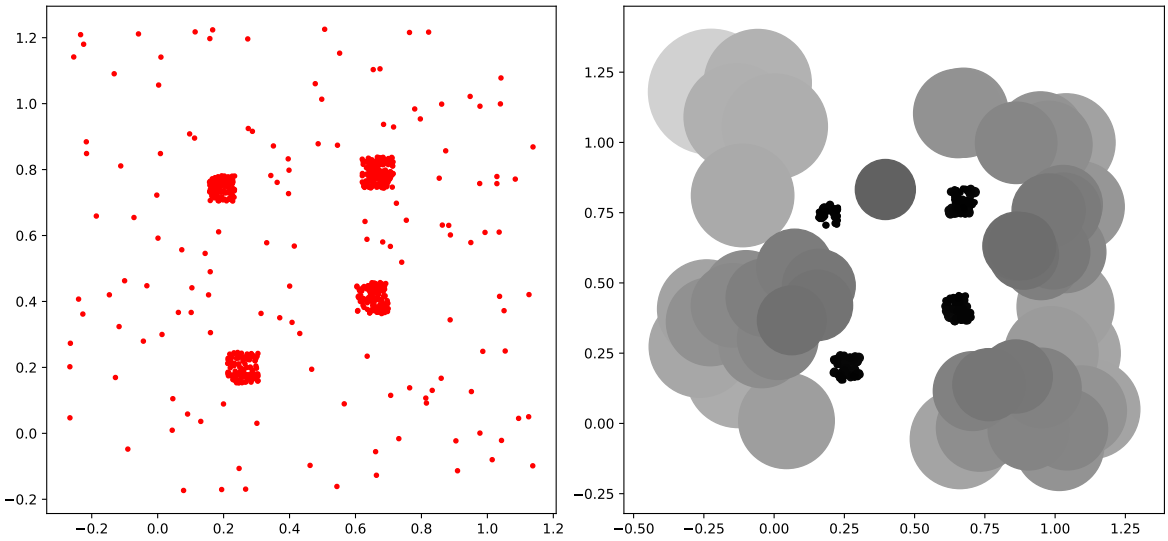

Figure 3.33: Data set with noise ($k = 10$, $t = 0.1$)

As expected, the four dense clusters were identified correctly, but all the noise-points around generated unwanted data expansions, so finally in total 7 clusters were found instead of 4. At a high level, you would need a pre-processing action which discards the noise-points even before the algorithm is executed.

In light of these issues, it was realized that an approach based on data expansions is limiting. The ExpansionClustering version III fits a fair number of data sets without worrying about setting parameters, but it is still not enough. Something more robust is needed. Ideally, if the ability of the human eye could be implemented somehow within an algorithm, then the clustering problem would be solved. Instead, metaphorically speaking, all the clustering algorithms known so far have been conceived "thinking like a machine" and not "imitating human

beings". That said, in order to solve the clustering problem, maybe it would be better to use an instrument more suited to imitate human behaviours: the *artificial neural network*.

3.3.3 Pseudo-code

Like for the version II, in the following pseudo-code listings only the new features are discussed since there are steps (e.g. the segmentation phase) exactly the same as in the previous two versions of the ExpansionClustering algorithm.

Listing 3.14: EC version III (part 1)

```
1  data = read_data() # 2D array of (x,y) data-points
2  n = len(data) # number of data-points
3  k = 10
4  t = 0.2
```

In lines 3-4 of Listing 3.14 the algorithm parameters are set (ref. classification phase).

3.3.3.1 The expansion phase

Listing 3.15: EC version III (part 2)

```
5  # EXPANSION PHASE ----------------------------------------
6  nn_distances = nearest_neighbors(data, k)
7  exp_rays = [avg(row[:]) for row in nn_distances]
8  exp_min = min(exp_rays)
9  exp_max = max(exp_rays)
10 exp_colors = (exp_rays - exp_min) / (exp_max - exp_min)
```

All the steps in this phase (see Listing 3.15) are the same as in version II, except for the *ert* filter which here is missing (ref. ExpansionClustering version II → pseudo-code).

3.3.3.2 The classification phase

Listing 3.16: EC version III (part 3)

```
11 for data_point in data:
12     exp_radius = exp_rays[data_point]
13
14     nn_exp_rays = []
15     for neighbor in nn_of(data_point):
16         nn_exp_rays.append(exp_rays[neighbor])
17     nn_avg = avg(nn_exp_rays)
18
19     exp_relative[data_point] = abs(exp_radius - nn_avg)/exp_radius
```

The lines 11-19 of Listing 3.16 implements the behaviour described in the classification phase subsection. In particular, for each data-point:

- its `exp_radius` is retrieved (see line 12 of Listing 3.16);
- its `nn_avg` is computed by looking at the first $k = 10$ nearest neighbors (see lines 14-17 of Listing 3.16);
- its `exp_relative` is computed by using the previous `exp_radius` and `nn_avg` and by using the formula (see line 19 of Listing 3.16) already discussed.

At the end of this for loop the `exp_relative` (n x 1) array will be available.

Listing 3.17: EC version III (part 4)

```
20 exp_relative.filter(x -> x <= t)
21 exp_image = plot_data_expansions(exp_rays, exp_colors, exp_relative)
```

In line 20 of Listing 3.17 the filter is applied by using the `t` parameter and then in line 21 of Listing 3.17 the `exp_image` is produced: this time the `plot_data_expansions` function need to know also the `exp_relative` array to know which expansions must be plotted.

3.3.3.3 The segmentation phase

Listing 3.18: EC version III (part 5)

```
22 # SEGMENTATION PHASE ----------------------------------------
23 seg_image = watershed(exp_image)
24 assigned_labels = assign_labels(data, seg_image)
25 real_labels # (n x 1) array known a priori
26 error = error_measurement(real_labels, assigned_labels)
```

No change is present for the segmentation phase (see Listing 3.18) compared to the version I.

Chapter 4

Proposed clustering algorithms: BridgeClustering

This chapter is dedicated to the last algorithm conceived to figure out the clustering problem. Its name is BridgeClustering since it delimits different clusters by connecting each data-point with some of its nearest neighbors, like bridges do. Moreover, as anticipated in the ExpansionClustering version III $\rightarrow$ analysis of weaknesses subsection, this algorithm makes use of the *artificial neural network* (*ann* in short) learning tool. So, there is an "electronic brain" which basically learns - drawing from various data sets - how clustering should be done, maybe imitating human behavior. Following is the detail of these features, as usual organizing the contents in three sections: graphic illustration, analysis of the weaknesses and pseudo-code.

4.1 Graphic illustration

Before talking about the BridgeClustering algorithm phases, it is necessary to go into the details of the two key concepts (bridges and ann, as mentioned above) behind the algorithm itself and justify their use. This deepening takes places in the subsequent "The use of bridges" and "The use of artificial neural network" sections, followed by the usual sections related to the algorithm phases.

4.1.1 The use of bridges

The BridgeClustering algorithm saw light after a long series of design sketches, not all reported in this thesis. An interesting sketch - not based on data expansion - was to link each data-point in the data set with its k nearest neighbors and to see what happened: relevant findings were gained by setting $k = 4$. Check out to the following Figures 4.1-4.5, considering that:

- for each data-point, the connection lines between it and its $k = 4$ nearest neighbors represent its bridges (hence the name BridgeClustering);

- the color of data-points and bridges is the same within the same connection area, different otherwise (it's easier to see than to say).

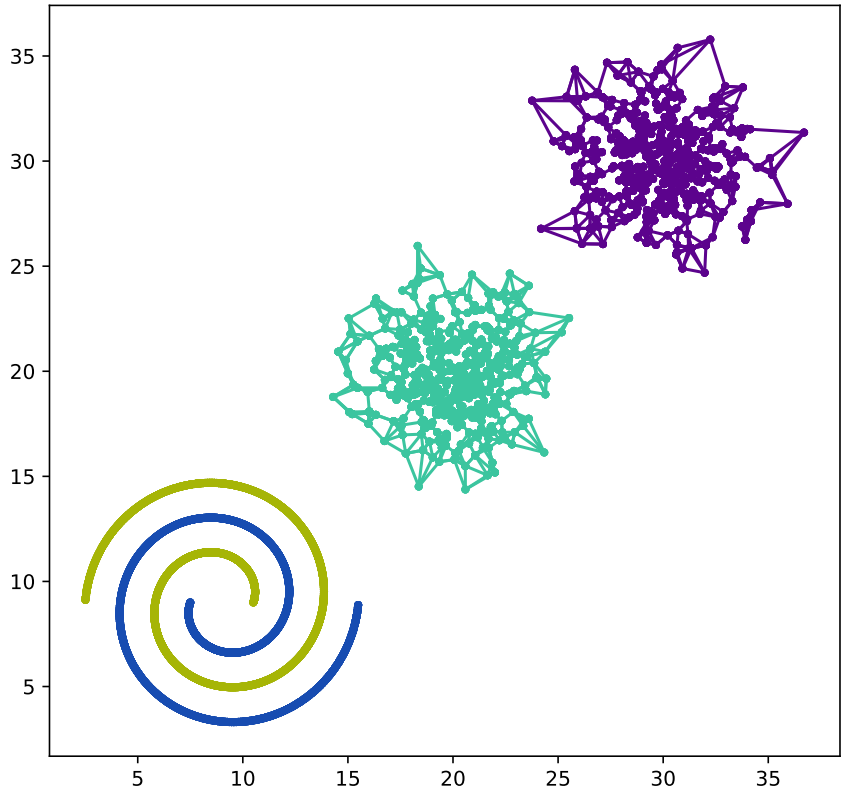

Figure 4.1: Center-based and contiguity-based clusters

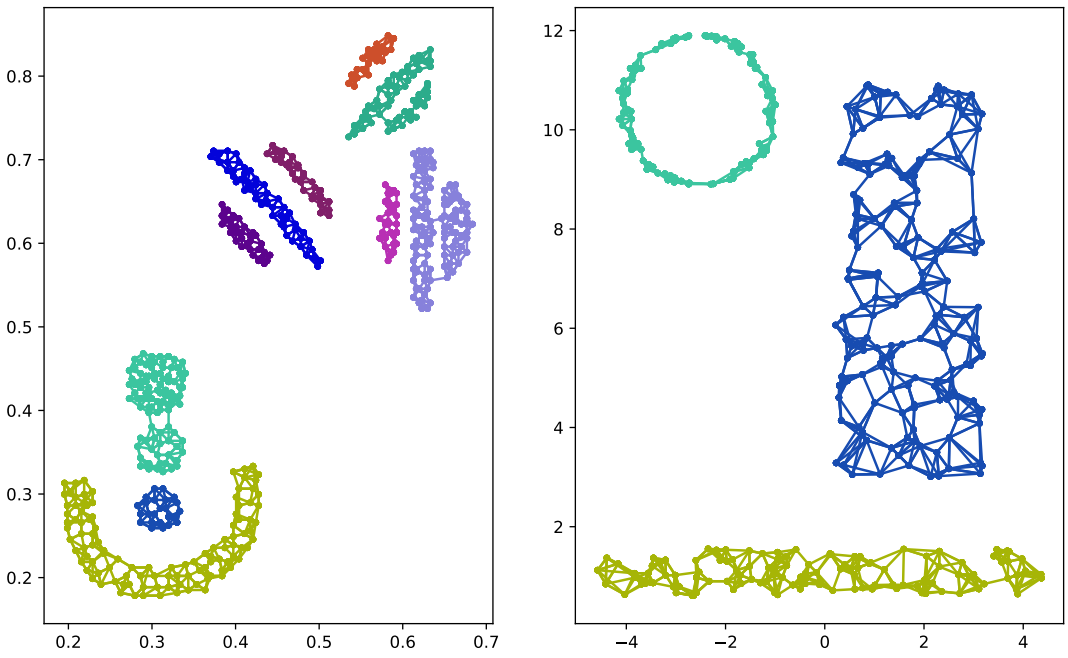

Figure 4.2: Contiguity-based clusters

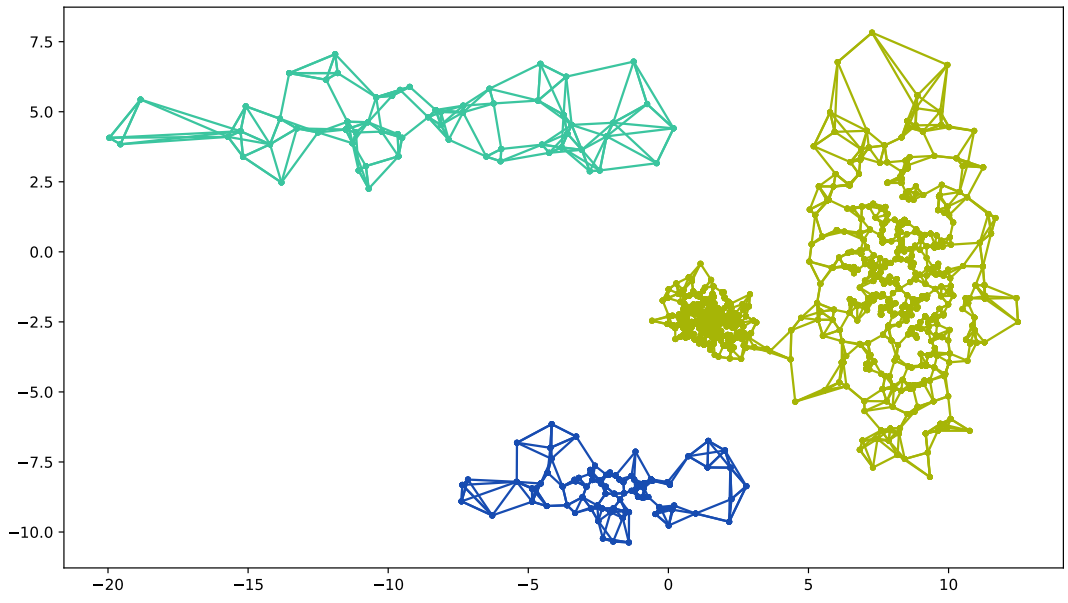

Figure 4.3: Density-based clusters

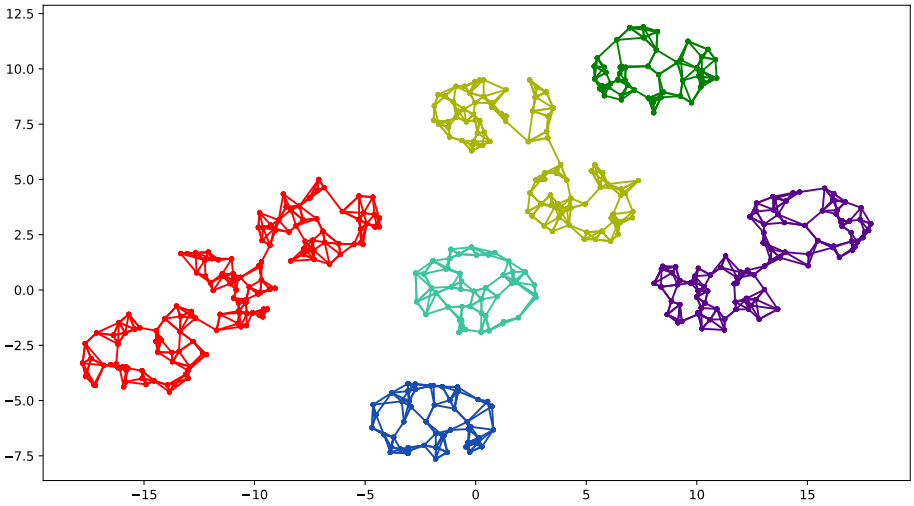

Figure 4.4: Center-based clusters

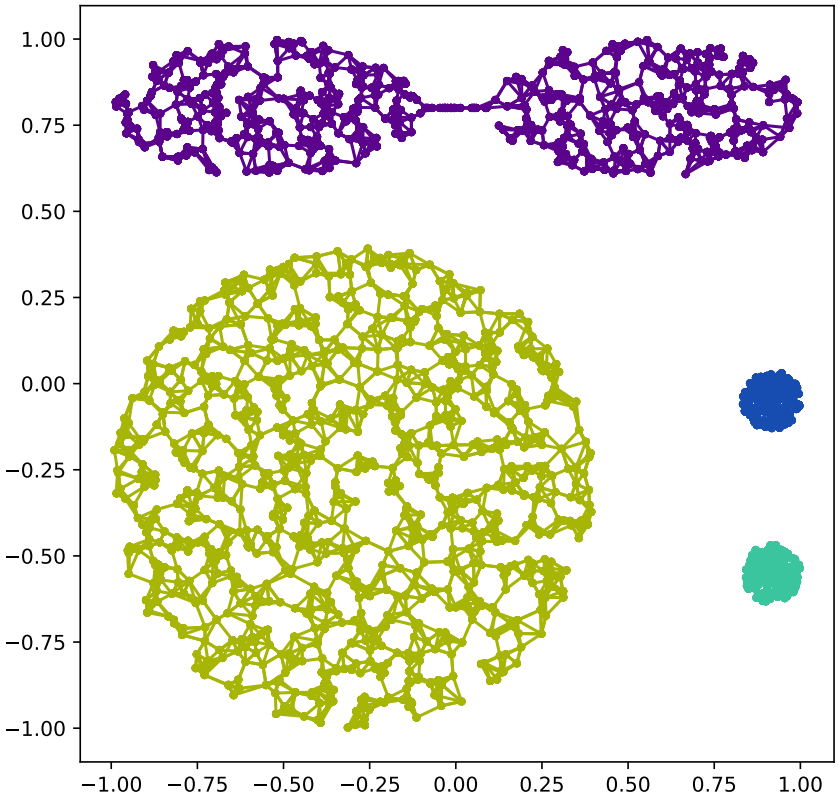

Figure 4.5: Center-based and contiguity-based clusters

As you can see, for these data sets the clustering achieved is pretty good since it almost coincides with the right one. Furthermore, these results were obtained with an approach not based on a graphic processing which involves expansion phase or segmentation phase as before, basically it is much simple: each data-point is linked with its $k = 4$ nearest neighbors. And this affair is very interesting. Definitely, in order to obtain the right clustering, it would be enough to prevent the creation of "harmful" bridges, that is bridges which put in the same cluster data-points that should not be together (you can easily see them in the previous figures). As a consequence of this, how can you expect, the solution envisages that some data-points must not make any bridge.
Got to this point, a literary distinction is useful:

- a *cross-point* is a data-point whose first $k = 4$ nearest neighbors belong to its own cluster;
- an *island-point* is a data-point that has at least one neighbor within its first k = 4 nearest which does not belong to its own cluster.

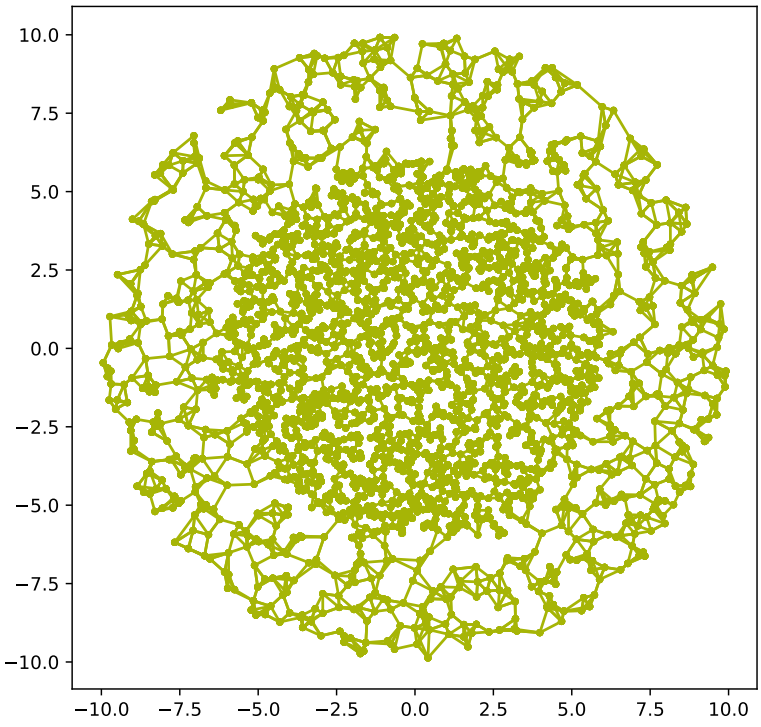

Figure 4.6: Data set with many island-points

Please observe that, depending on the data set in question, the island-points can be few (as in the previous Figures 4.1-4.5) or many (as in Figure 4.6). In the second case, unfortunately there are so many island-points that only a single cluster is identified, although there are two. Figure 4.7 shows the target clustering obtained by preventing the island-points (black data-points) from make any bridge, while the cross-points (colored data-points) make a bridge to each of its first $k = 4$ neighbors, in order from the nearest one.

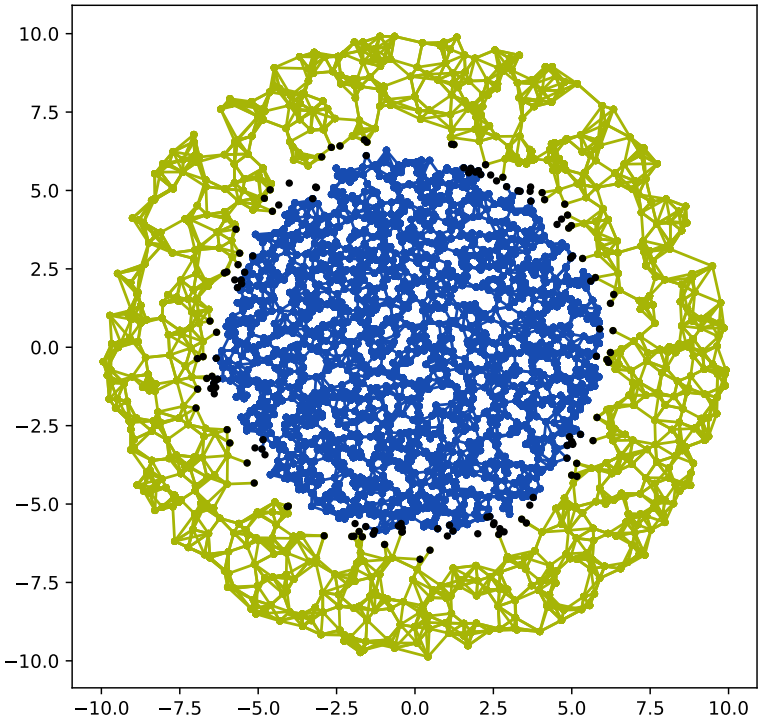

Figure 4.7: Data set with many island-points - target clustering

In short, the task to solve is the classification of data-points in cross-points and island-points. As you remind, this problem is the same already encountered for the ExpansionClustering version III (ref. to analysis of weaknesses subsection). In fact, the previous core-points and border-points here correspond, more or less, to cross-points and island-points respectively. So, before talking about the classification task, why use bridges instead of data expansions? First of all because the bridge-based approach is simpler, then because it has been observed that on average data expansions are more cumbersome than bridges, especially

when different clusters are very close together. In other words, it is easier to have a "harmful" data expansion than a "harmful" bridge ("harmful" in the sense of the above).

Definitely, after having realized that 4 is a pretty good value for k, the challenge of the BridgeClustering algorithm consists in identify the island-points and prevent them to make bridges. How to do this?

4.1.2 The use of artificial neural network

Instead of trying an approach based on some criterion requiring parameters, this time it was thought to let the machine itself learn to distinguish cross-points from island-points. This was implemented thought the use of an artificial neural network that:

- in the input layer has 20 neurons that must be supplied with information regarding a given data-point (details are given below);
- has 4 hidden layers each of which has 100 neurons;
- in the output layer has 1 neuron whose value is equal to 0 if the data-point in question is predicted as an island-point, equal to 1 if as a cross-point.

But what kind of information, for each data-point, should be given to the input layer? This is a tough and important question, in fact if you provide the network with not relevant information, then the network does not learn at all.

In order to answer the question, bear in mind that the basic principle that animates all the algorithms proposed in this thesis is that the nature of each data-point is closely correlated to the positions of the data-points around it. In other words, the network could jokingly say to a given data-point: "tell me who your nearest neighbors are and I'll tell you who you are". So, given a data-point to classify as cross/island-point, the choice was to provide the input layer with the Cartesian coordinates (relative to the data-point itself) of its $m = 10$ nearest neighbors. Since the position of each neighbor is identified by 2 coordinates in the Cartesian plane, a total of (2 x m) = (2 x 10) = 20 input neurons is required. Please keep in mind that both the value of the previous k parameter and of this

m parameter were derived after carrying out several tests with the ann. The great advantage of the BridgeClustering algorithm is that they are fixed.

Before going into the details of the algorithm, pay attention to a very important concept: as mentioned in the introductory chapter, clustering is an unsupervised problem since there is no prior information that can correct the task outcome (differently from the classification or regression problems). The approach adopted by the BridgeClustering algorithm breaks this assumption, in fact here there is an ann which learns by drawing from a certain number of training sets whose labels are therefore known a priori. In a nutshell, the algorithm itself provides a supervised method for an unsupervised problem. The goal is to extract from very different training sets a general pattern which can adapt to various situations.

As a supervised method, the BridgeClustering algorithm consists of three phases described in the subsequent subsections: the pre-processing phase, the training phase and the predicting phase.

4.1.3 The pre-processing phase

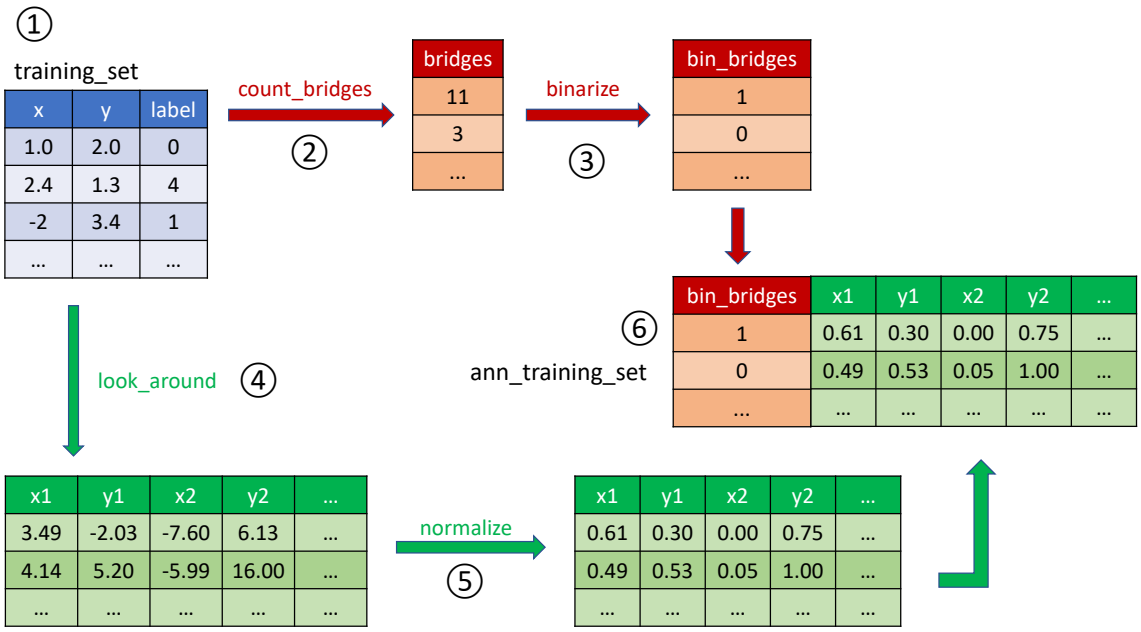

Figure 4.8: The pre-processing phase

The goal of the pre-processing phase is to transform a given training set - with schema $(x, y, label)$ - in such a way that it is compliant with the input of the ann. In order to understand how it works, look at the block diagram in Figure 4.8 which represents into details the steps of the pre-processing phase:

1. A given `training_set` is available in blue (top left): its schema presents the two coordinates (x, y) of a data-point as attributes and the associated clustering label; the latter is precisely the prior information discussed previously.

2. The `count_bridges` function (red arrow no.2) is the first to be run on such `training_set`. For each labeled data-point, this function scans its neighbors ordered from the nearest onwards and counts them until they have the same label as the data-point in question. In short, the purpose is to count, for each data-point, how many of its sorted nearest neighbors belong to its own cluster. This calculation is a measure of how many bridges each data-point should make.

3. If the counts of point 2 were not processed, then it would certainly fall into

the phenomenon of *over-fitting*: in simple terms, the ann would over-adapt to the `training_set` and consequently it would not be able to produce a general model. That said, the `binarize` function (red arrow no.3) filters each value of the `bridges` array at the threshold $k = 4$: if the current value is less than 4 then 0 is returned, otherwise 1 is returned. Hence, this binary label (which will be the label for the subsequent `ann_training_set`) associates the value 0 for island-points and the value 1 for cross-points. This is the purpose of the `bin_bridges` array. It has been observed that this action allows the ann to generalize better.

4. The next two steps (in green) are more complex and they serve to produce the values for the input layer of the ann. The `look_around` function (green arrow no.4) scans all the data-points in the `training_set` and, for each of them, it computes the Cartesian coordinates $(x1, x2), (x2, y2), ..., (x10, y10)$ of its first $m = 10$ nearest neighbors in relation to the position of the data-point in question. For example, if the current data-point is (1, 3) has as first nearest neighbor the data-point (4, -5), then for it:

$$x1 = 4 - 1 = 3 \tag{4.1}$$

$$y1 = -5 - 3 = -8. \tag{4.2}$$

   Such procedure is executed for each data-point in the `training_set` in blue (top left).

5. It is well known that the use of normalization for the input layer of an ann is recommended for a more efficient learning. This is what the `normalize` function does (green arrow no.5). In particular, all the values of each row in the green table at the bottom left are normalized in the range [0, 1].

6. The last step brings together the values just computed and the binary labels (`bin_bridges`) of point 3 to constitute the `ann_training_set`, as clearly shown in Figure 4.8.

These steps must be done for each training set available, so at the end all the associated ann training sets need to be concatenated to create an unique file

that will represent the whole training set for the artificial neural network.

4.1.4 The training phase

This is the phase in which the ann is trained drawing from the whole ann training set discussed in the pre-processing phase. Without going into the details of the topic, learning takes place through the use of the *back-propagation* technique which, at each training iteration (called epoch), tends to minimize the label error. The output of the training phase is a (hopefully general) model in the form of weight coefficients. For the rest, there are no relevant implementation explanations to be given for this part.

4.1.5 The predicting phase

The predicting phase is the one you are most interested in, since it consists in labeling an unlabeled data set. Its flow has some parts in common with the pre-processing phase as represented in Figure 4.9.

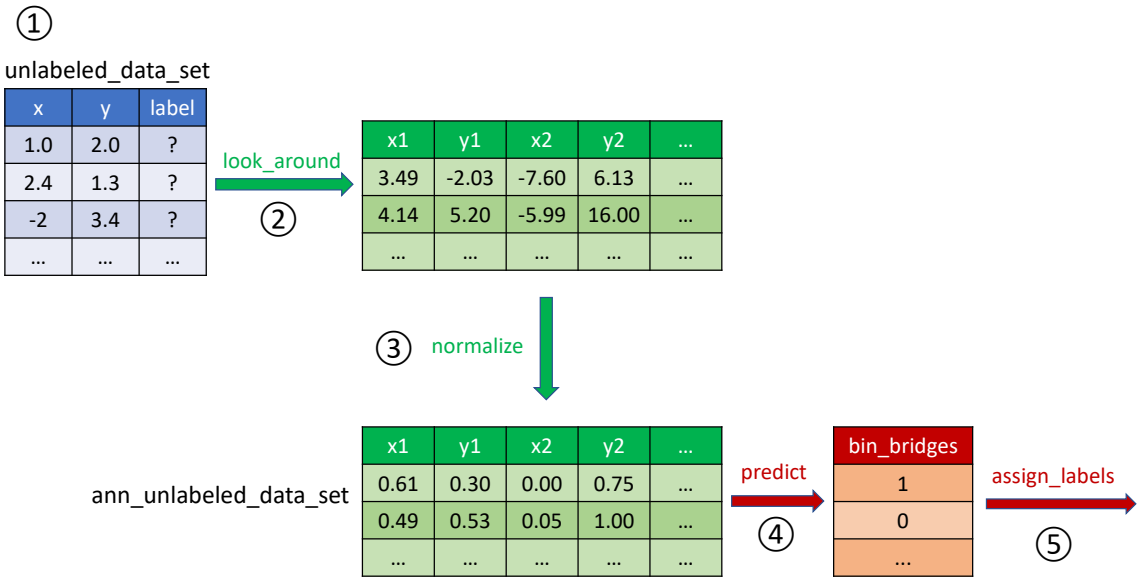

Figure 4.9: The predicting phase

Also for the predicting phase it is better to illustrate its steps by using an explanatory block diagram:

1. At the start, there is an `unlabeled_data_set` in blue (top left); as you can see, labels are not yet known.

2. The `look_around` function (green arrow no.2) is the same as that one previously described in the pre-processing phase.

3. The `normalize` function (green arrow no.3) is the same as that one previously described in the pre-processing phase.

4. Once the `ann_unlabeled_data_set` is computed, the prediction can take place: each row of the green table (associated to an unlabeled data-point) is passed in input to the ann, then its predicted binary label is registered in the `bin_bridges` array in red. This predictive action is performed for all the rows.

5. The last step transforms the ann output into the required clustering labels. The `bin_bridges` array indicates that if a data-point is predicted as cross-point (i.e. the predicted value is 1, so it must make $k = 4$ bridges around it) or as island-point (i.e. the predicted value is 0, so it does not make any bridges). Finally, already knowing how many bridges each data-point makes, labels are assigned. In particular:

   - the label of a cross-point is the same as that of those cross-points inside its connection area and different from the other labels outside;
   - the label of an island-point is the same as that of its first nearest cross-point.

   This is what the `assign_labels` function (red arrow no.5) does.

4.2 Analysis of weaknesses

Choosing appropriate training sets for a successful learning of the artificial neural network is crucial. Typically, it is better that they are very different from each other so that the network can generalize its behaviour, by drawing from clusters of a different nature. The following Figures 4.10-4.17 are a grid

containing the 40 data sets chosen for the training of the ann mentioned in "The use of artificial neural network" section.

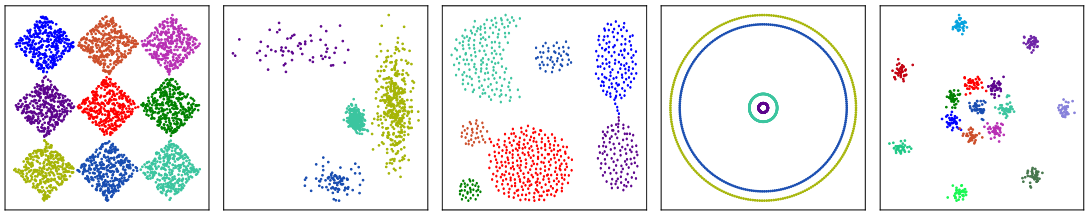

Figure 4.10: Training sets 1-5

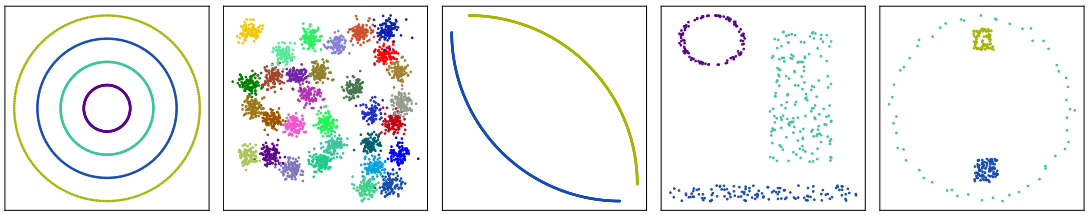

Figure 4.11: Training sets 6-10

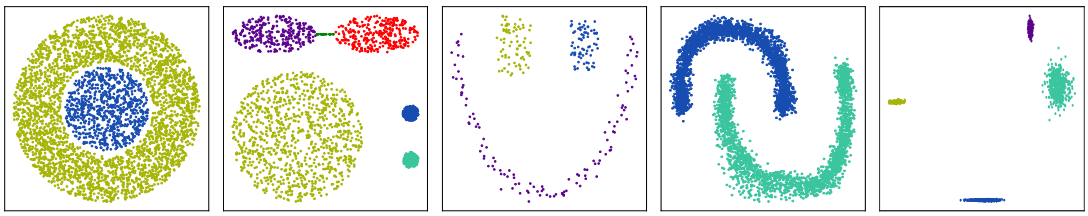

Figure 4.12: Training sets 11-15

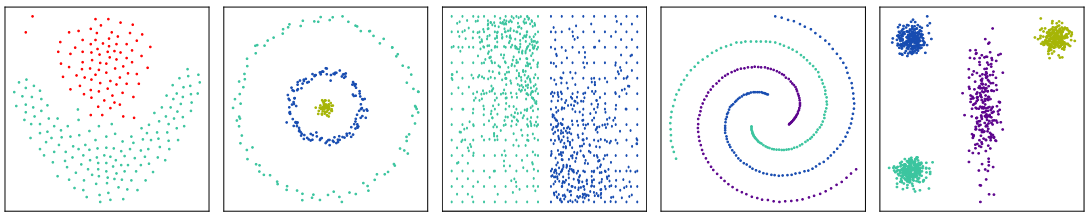

Figure 4.13: Training sets 16-20

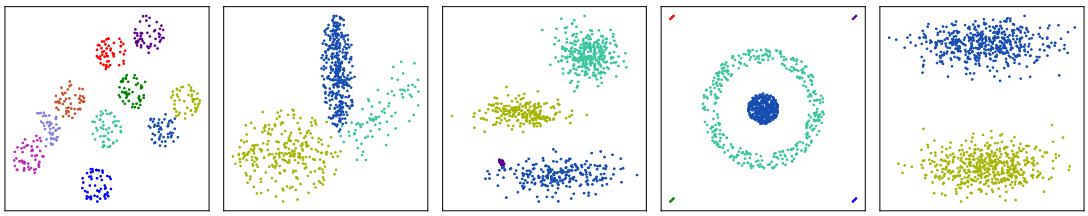

Figure 4.14: Training sets 21-25

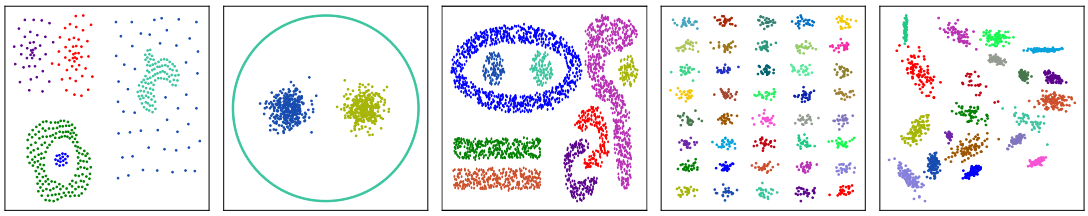

Figure 4.15: Training sets 26-30

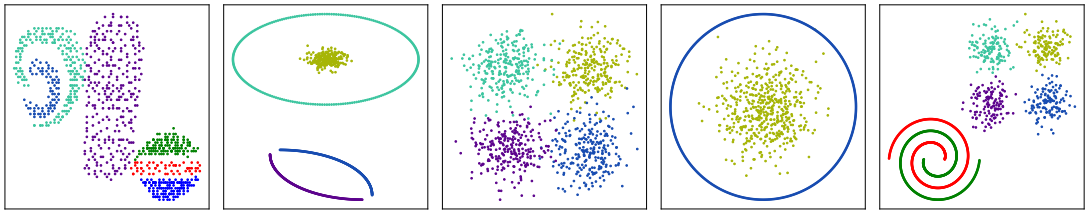

Figure 4.16: Training sets 31-35

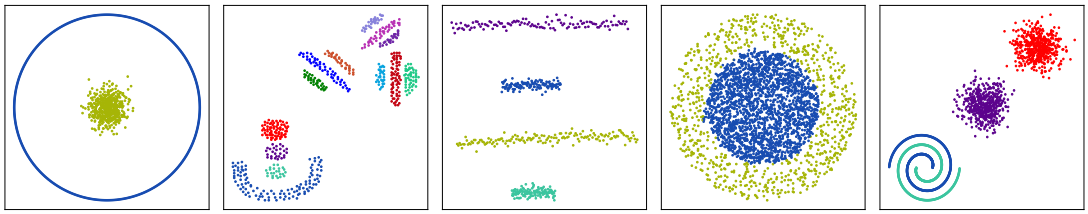

Figure 4.17: Training sets 36-40

The goal of the training phase is, substantially, to adjust - epoch by epoch for each training set in input - the weight coefficients of the network. At the end, these coefficients will constitute the model used to make predictions for unlabeled data sets. Figures 4.18-4.22 show how the BridgeClustering algorithm performs with unlabeled data sets *not included* in the previous training set grid:

this is the real case. The data-points predicted as island-points by the ann are depicted in black.

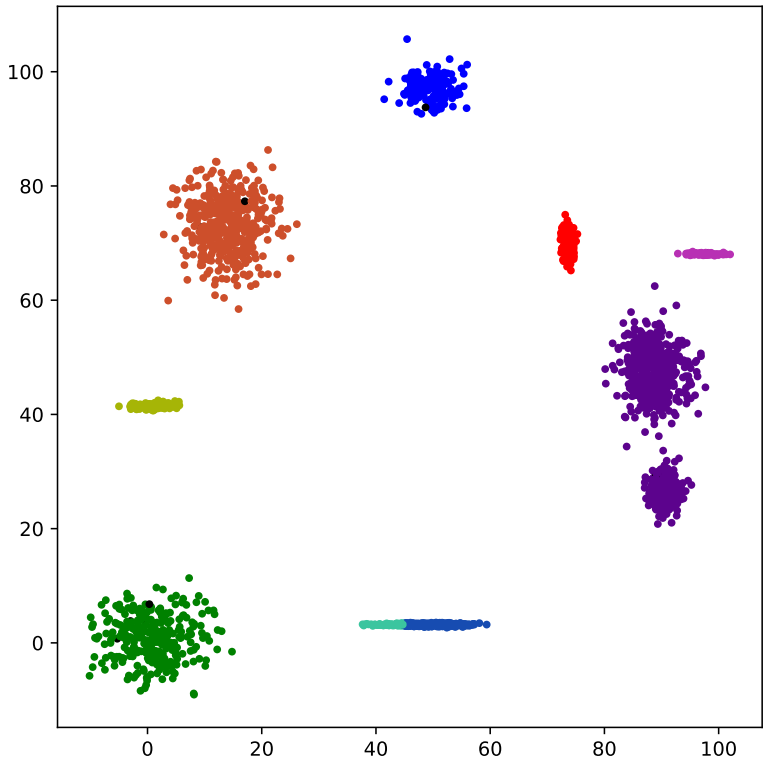

Figure 4.18: Well separated clusters

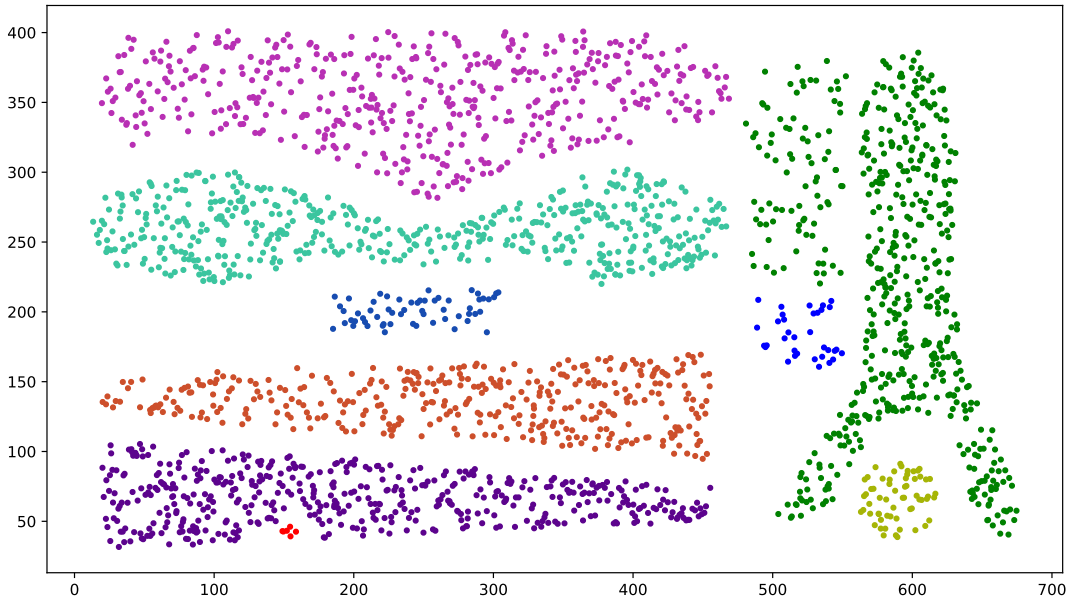

Figure 4.19: Density-based and contiguity-based clusters

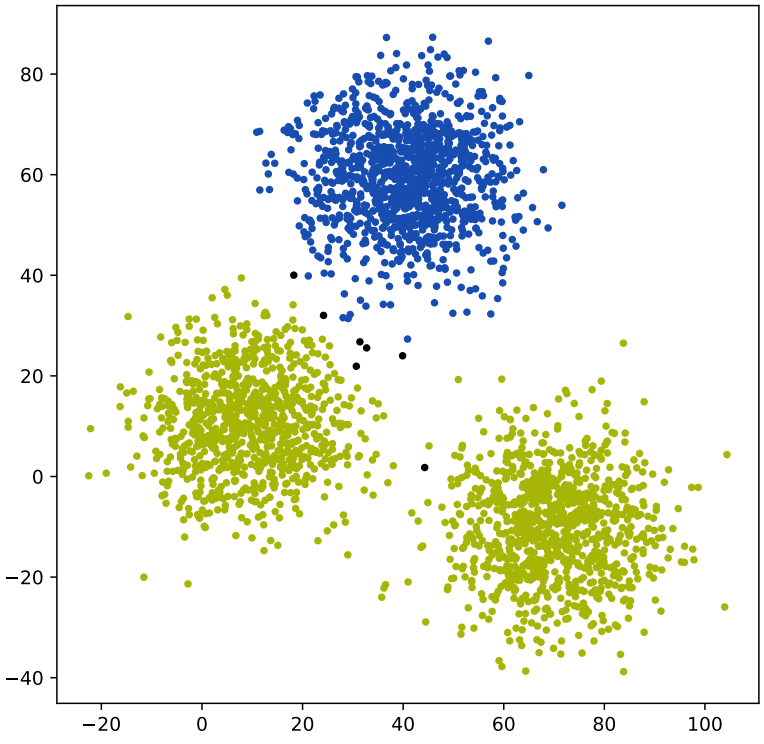

Figure 4.20: Center-based clusters

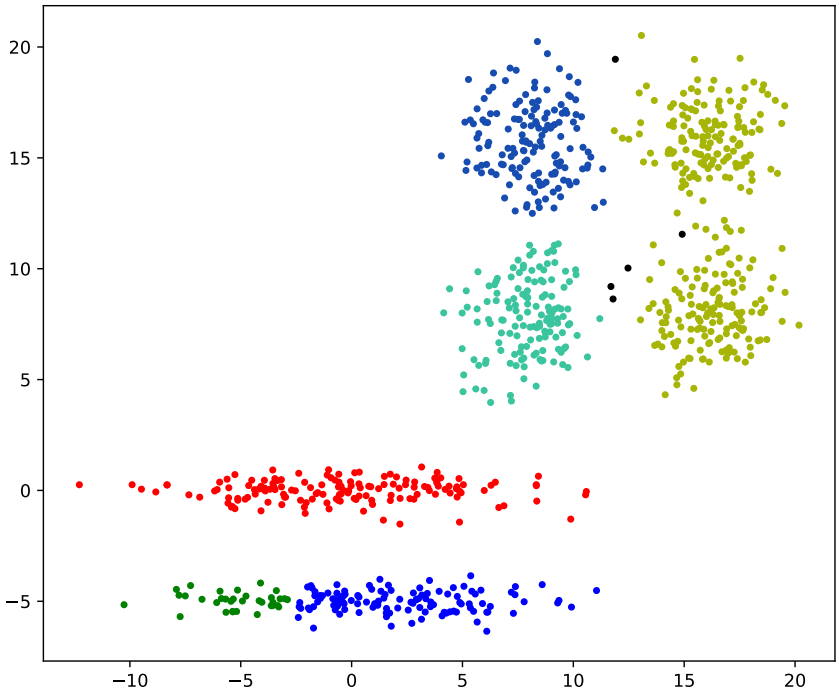

Figure 4.21: Center-based and contiguity-based clusters

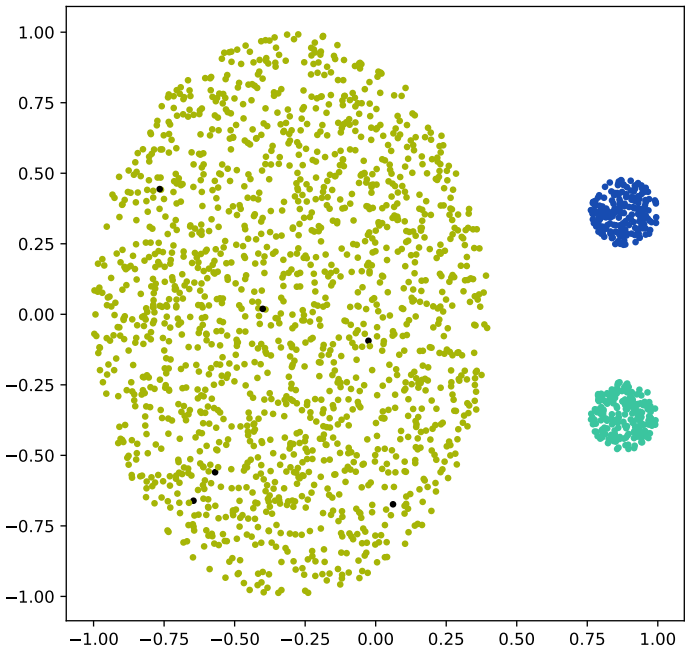

Figure 4.22: Well separated and center-based clusters

As you can see, sometimes island-points are predicted more or less correctly since in the middle of different clusters (see Figure 4.20, Figure 4.21) but sometimes they are predicted wrongly (see Figure 4.22). Definitely, even if this is still not the right clustering, of course these results show that it is gradually being achieved. But the most reassuring fact is given by the next Figures 4.23-4.25 showing how the BridgeClustering algorithm performs with unlabeled data sets *included* in the previous training set grid.

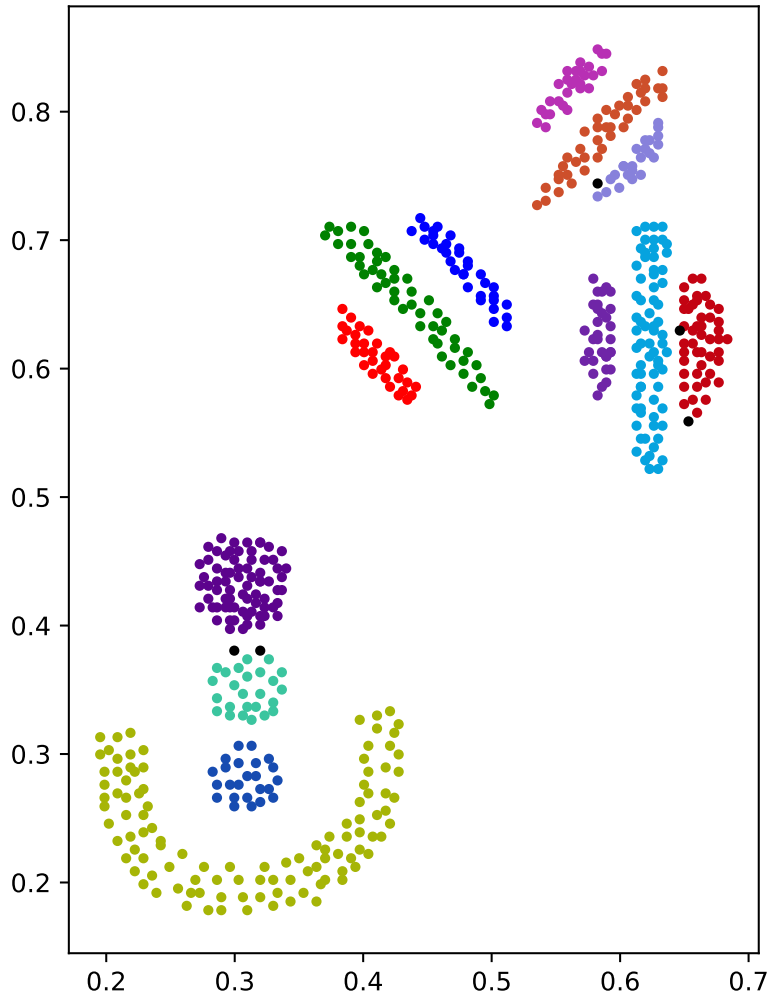

Figure 4.23: Contiguity-based clusters

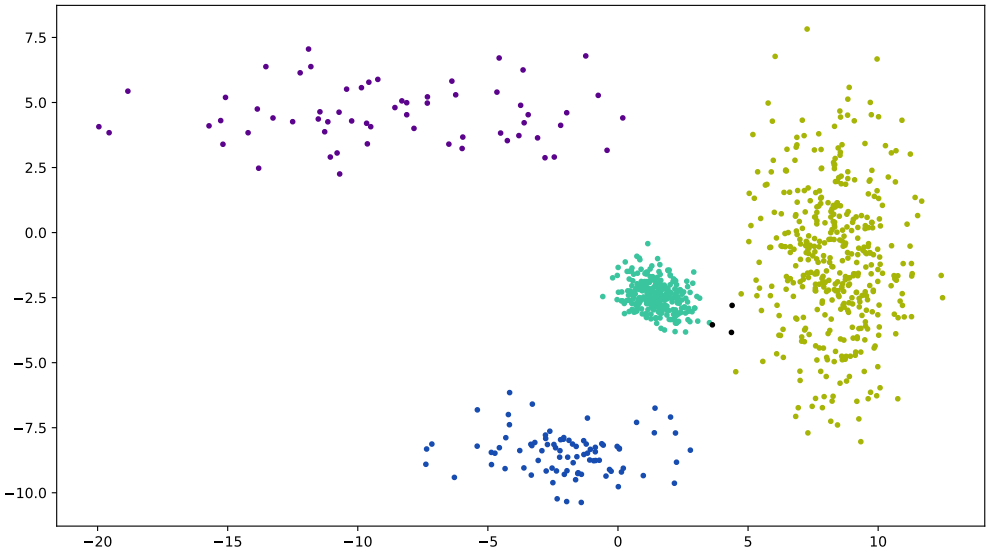

Figure 4.24: Density-based clusters

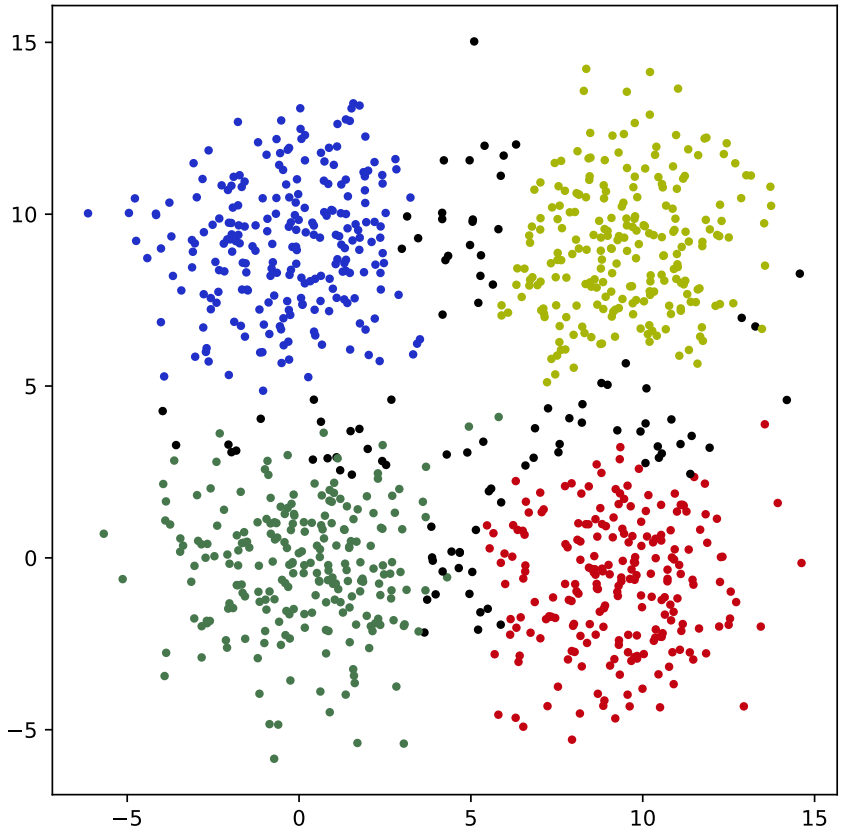

Figure 4.25: Center-based clusters

This time the right clustering is obtained successfully. However, the first thing that comes to mind is that this is due to over-fitting: the ann has over-adapted to the above 40 training sets, so it is able to cluster them correctly. Over-fitting or not, in reality the discussion here is deeper. The initial obstacle of using a learning-based approach was to find (for each data-point in a training set) useful information to provide at the ann input layer. Regardless of this information, the main fear was that, for example, a cross-point and an island-point could produce the same (or very similar) ann input in the pre-processing phase: what should the network have learned in this case? In short, the problem is that if two data-points produce very similar ann inputs but their predicted label should be different (i.e. cross-point and island-point), then a contradiction arises for the network learning. Fortunately, such inconvenience has not happened for any outcome, in fact a cross-point and an island-point that had the same ann input were not observed. This means that an artificial neural network potentially able to learn clustering could exist.

Moreover, consider that the choice of which information to provide at the ann input layer was not trivial. For example, before deciding to use the Cartesian coordinates of the nearest neighbors, their polar coordinates had been used. Essentially, for each neighbor, instead of x and y there were $\rho = \sqrt{x^2 + y^2}$ and $\phi = atan2(y, x)$ in the ann input, where the $atan2$ function computes the arctangent of all four quadrants. Unfortunately, several tests showed that this was not a good input choice. That said, final considerations about the BridgeClustering algorithm are reported in the last conclusive chapter.

4.3 Pseudo-code

The following listings are the pseudo-code which helps to understand the steps of the BridgeClustering algorithm at a lower level.

4.3.0.1 The pre-processing phase

Listing 4.1: BC (part 1)

```
1  # PRE-PROCESSING PHASE ------------------------------------
2  training_set = read_training_set()

3  n = len(training_set) # number of data-points
```

This is the initial step in which the `training_set` in line 2 of Listing 4.1 is read somehow (e.g. from a file or from the network). It is a (n x 3) array containing, for each row i, a 2D data-point plus its label.

Listing 4.2: BC (part 2)

```
4  bridges = count_bridges(training_set)
5
6  def count_bridges(data):
7    bridges = []
8    for data_point in data:
9      count = 0
10     for neighbor in data_point.get_neighbors():
11       if (data_point.get_label() != neighbor.get_label()):
12         break
13       count++
14     bridges.append(count)
15   return bridges
```

The `count_bridges` function in line 4 of Listing 4.2 was already discussed in the graphic illustration (ref. to pre-processing phase → `count_bridges`). In short it says, for each data-point, how many of its neighbors are on its own cluster, ordered from the nearest onwards, as easily understandable in its implementation (see lines 6-15 of Listing 4.2).

Listing 4.3: BC (part 3)

```
16 bin_bridges = binarize(bridges, k=4)
17
18 def binarize(bridges, k):
19   bin_bridges = []
```

```
20    for b in bridges:
21      bin_bridges.append(0 if (b < k) else 1)
22    return bin_bridges
```

The `binarize` function in line 16 of Listing 4.3 returns a binary array where, for each cell, there is 0 if the related value is less than `k=4`, otherwise there is 1 (ref. to pre-processing phase → `binarize`). In lines 18-22 of Listing 4.3 a possible implementation is reported.

Listing 4.4: BC (part 4)

```
23  ann_input = look_around(training_set, m=10)
24
25  def look_around(data, m):
26    x_array = empty array of dim (n x m)
27    y_array = empty array of dim (n x m)
28    for i, data_point in data:
29      x = data_point.get_x()
30      y = data_point.get_y()
31      for j, neighbor in data_point.get_neighbors(m):
32        x_array[i][j] = neighbor.get_x() - x
33        y_array[i][j] = neighbor.get_y() - y
34    return merge(x_array, y_array)
```

Then, the values for the ann input layer are computed by the `look_around` function in line 23 of Listing 4.4 (ref. to pre-processing phase → `look_around`). As you can see, its implementation is more cumbersome than the previous ones. The `m` parameter says how many nearest neighbors to consider and for each `data_point` (in the `data` array parameter) the relative Cartesian coordinates of its nearest neighbors are computed and set in the `x_array` (see line 26 of Listing 4.4) and `y_array` (see line 27 of Listing 4.4) arrays: so, each row of these arrays correspond to a `data_point`. Note, moreover, that the `get_neighbors` function in line 31 of Listing 4.4 can accept a parameter indicating how many neighbors it must return, ordered as usual. Finally, the `merge` function in line 34 of Listing 4.4 performs the join between the `x_array` and `y_array` array as described in

Figure 4.8 by the green table at the bottom left. Its implementation is not reported because it is not so relevant in this context (too low level details).

Listing 4.5: BC (part 5)

```
35 ann_input = normalize(ann_input)
36 ann_training_set = bin_bridges + ann_input
```

In lines 35 of Listing 4.5 normalization is performed. Finally, the concatenation in line 36 of Listing 4.5 symbolizes the union of the `bin_bridges` and `ann_input` arrays in order to produce the `ann_training_set`, as clearly shown in Figure 4.8.

4.3.0.2 The training phase

Listing 4.6: BC (part 6)

```
37 # TRAINING PHASE ----------------------------------------
38 ann = ArtificialNeuralNetwork()
39 ann.fit(ann_training_set)
```

As expected, the training phase counts much less lines than the pre-processing phase because the ann management is delegated to external libraries. So, in line 38 of Listing 4.6 a new instance of the `ArtificialNeuralNetwork` object is created, then in line 39 of Listing 4.6 the training takes place, passing the `ann_training_set` previously calculated as parameter to the `fit` method. Unfortunately this phase may require several minutes, depending on how large is the `ann_training_set` array.

4.3.0.3 The predicting phase

Listing 4.7: BC (part 7)

```
40 # PREDICTING PHASE ----------------------------------------
41 unlabeled_data_set = read_unlabeled_data_set()
```

The `unlabeled_data_set` in 41 of Listing 4.7 is a (n x 2) array containing, for each row i, an unlabeled 2D data-point.

Listing 4.8: BC (part 8)

```
42 ann_input = look_around(unlabeled_data_set, m=10)
43 ann_input = normalize(ann_input)
```

The steps in lines 42-43 of Listing 4.8 are identical to those in the pre-processing phase. At this point, the `ann_input` is ready for the artificial neural network.

Listing 4.9: BC (part 9)

```
44 bin_bridges = ann.predict(ann_input)
```

The `predict` method in line 44 of Listing 4.9 represents the heart of the predicting phase: it forwards each row of the `ann_input` multi-dimensional array in the network to obtain the binary label, so finally it returns the `bin_bridges` array on which the last following operation is based.

Listing 4.10: BC (part 10)

```
45 assign_labels(unlabeled_data_set, bin_bridges, k=4)
46
47 def assign_labels(data, bin_bridges, k):
48   labels = empty array of dim (n x 1)
49   current_label = 0
50   for i, data_point in data:
51     label(data_point, current_label, bin_bridges, k)
52     current_label++
53   label_islands(data, bin_bridges, k)
```

The `assign_labels` function in line 45 of Listing 4.10 accepts as parameters the `unlabeled_data_set` to label, the `bin_bridges` array to know what are the cross/island-points and the `k` parameter (set to 4) saying how many bridges a cross-point must make. The goal of such function is to assign the final label to each data-point, knowing where there are bridges and where not. In line 48 of Listing 4.10 the `labels` (n x 1) array is defined. In line 49 of Listing 4.10 the `current_label` variable is defined and initialized to 0, in fact final labels are integers greater than or equal to 0. The increase of `current_label` in line 52 of

Listing 4.10 corresponds to a cluster change. Finally, in line 53 of Listing 4.10 the `label_islands` function provides labels for island-points, by looking at their first cross-points nearest as explained in the predicting phase → point 5. The high level implementation of this function is non reported because it consists only in details not relevant for this thesis. As you can see, all the intelligence of this phase lies in the `label` recursive function, invoked for each data-point in line 51 of Listing 4.10. The high level implementation of such function is in Listing 4.11.

Listing 4.11: BC (part 11)

```
54 # Recursive function
55 def label(data_point, current_label, bin_bridges, k):
56   data_point.set_label(current_label)
57   if (bin_bridges[index_of(data_point)] == 0):
58     return
59   for neighbor in data_point.get_neighbors(k):
60     label(neighbor, current_label, bin_bridges, k)
```

At each level of recursion, the `label` function takes as parameters the current `data_point` to process, the `current_label` to assign, the `bin_bridges` array and the `k` parameter: please note that the last two parameters do not change for all the invocations. The first thing to do (see line 56 of Listing 4.11) is to assign the `current_label` to the current `data_point`. Then, in line 57 of Listing 4.11 there is the stop condition of the recursion: if the current `data_point` has a value equal to 0 in the `bin_bridges` array (the `index_of` function is used), then it is predicted as island-point, so the function must return. In fact in the next for loop (see line 59 of Listing 4.11) only cross-points must be iterated. Finally, if the current `data_point` is a cross-point then the `label` function is invoked recursively (see line 60 of Listing 4.11) for each of its first `k` nearest neighbors. As just seen, the `get_neighbors` function can accept a parameter indicating how many neighbors it must return, ordered as usual.

Chapter 5

Evaluation

This chapter reports some cluster evaluations made on several data sets by using the ExpansionClustering version III and the BridgeClustering algorithms. In particular - for each data set and each algorithm - two evaluation indices were computed by comparing (time after time) the obtained cluster result with the desired one:

- the *rand index* (RI), a float number in the [0,1] range:
  - $RI = 0$ means that the two compared cluster results are "totally different";
  - $RI = 1$ means that they are "totally equal";
- the *adjusted rand index* (ARI), a float number in the [-1,1] range:
  - $ARI \leq 0$ means that the two compared cluster results are "totally different"
  - $ARI = 1$ means that they are "totally equal";

Please note that RI is less reliable than ARI and typically it tends to be higher than the latter. Substantially, without going into the details of their definitions, look at both indices but keep in mind that ARI gets more importance. Below, Figures 5.1-5.14 indicate such evaluations for ExpansionClustering version III (to the left) and for BridgeClustering (to the right), by reporting both RI and ARI values.

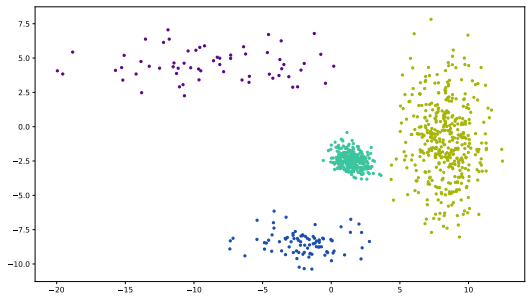

Figure 5.1: $RI = 1$, $ARI = 1$

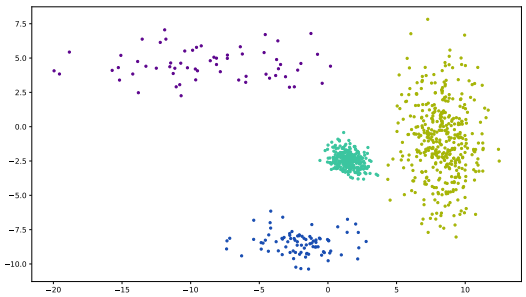

Figure 5.2: $RI = 1$, $ARI = 1$

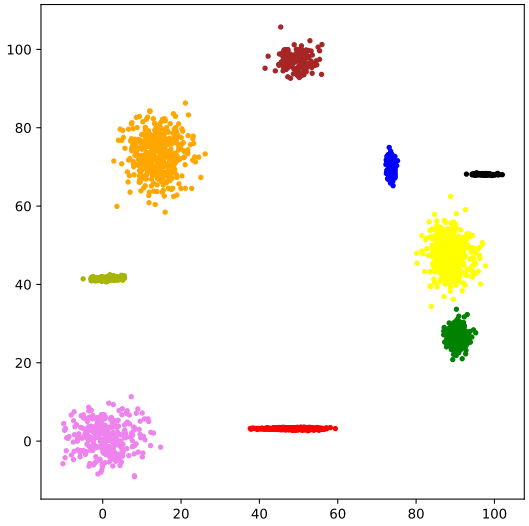

Figure 5.3: $RI = 1$, $ARI = 1$

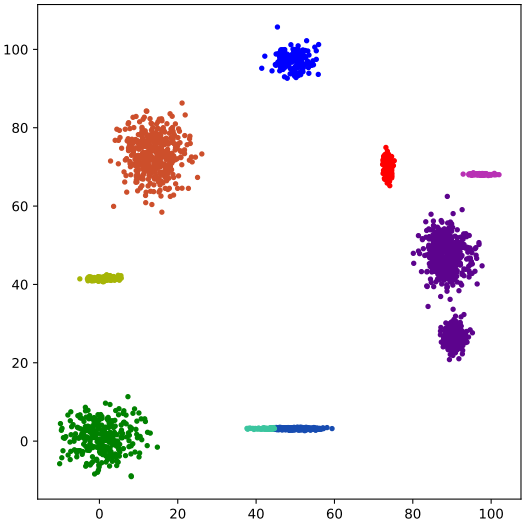

Figure 5.4: $RI = 0.957$, $ARI = 0.836$

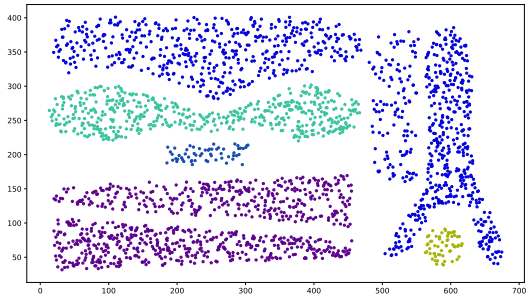

Figure 5.5: $RI = 0.842$, $ARI = 0.591$

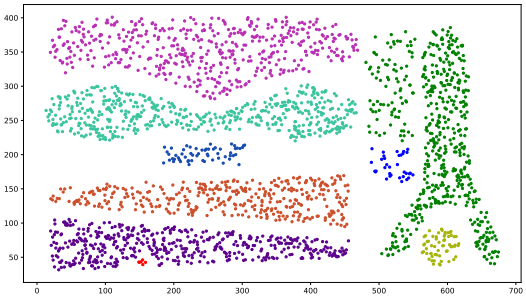

Figure 5.6: $RI = 0.986$, $ARI = 0.951$

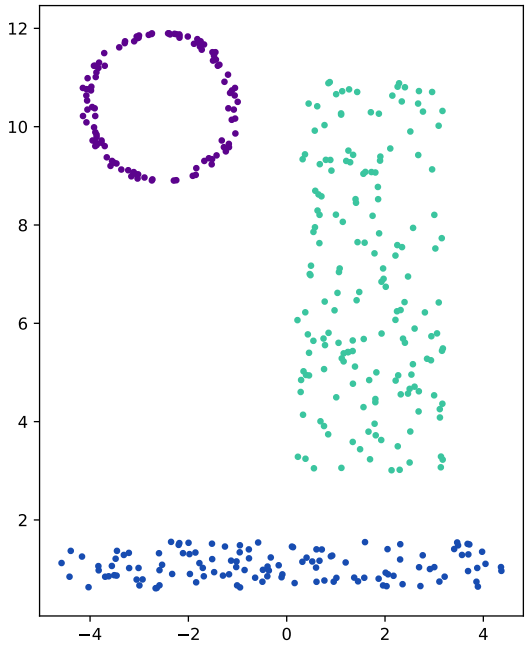

Figure 5.7: $RI = 1$, $ARI = 1$

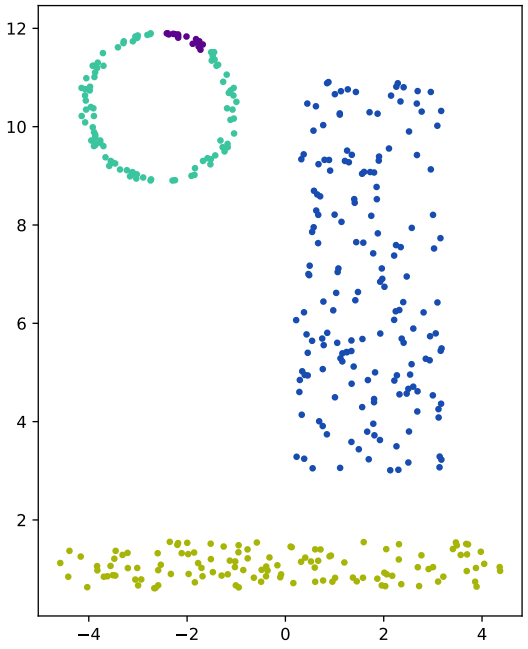

Figure 5.8: $RI = 0.981$, $ARI = 0.958$

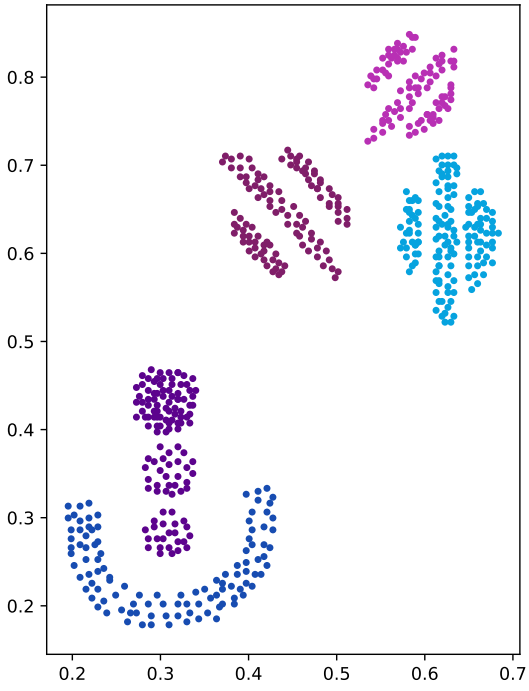

Figure 5.9: $RI = 0.893$, $ARI = 0.590$

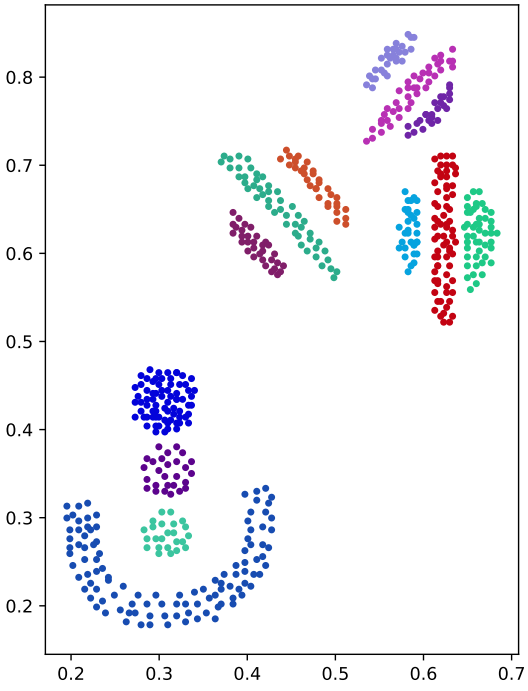

Figure 5.10: $RI = 1$, $ARI = 1$

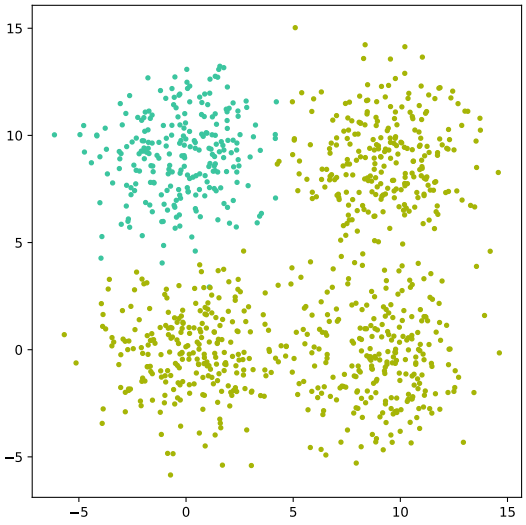

Figure 5.11: $RI = 0.607$, $ARI = 0.306$

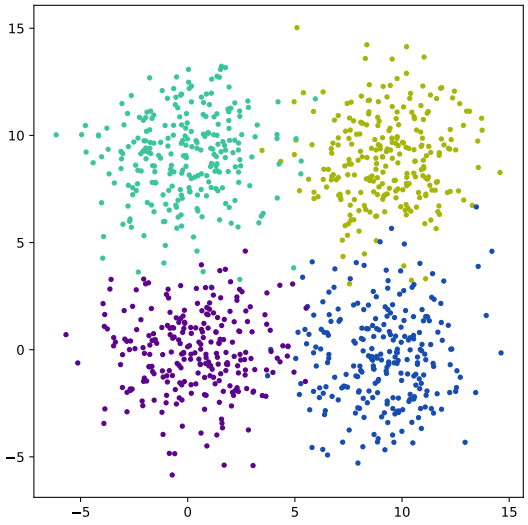

Figure 5.12: $RI = 1$, $ARI = 1$

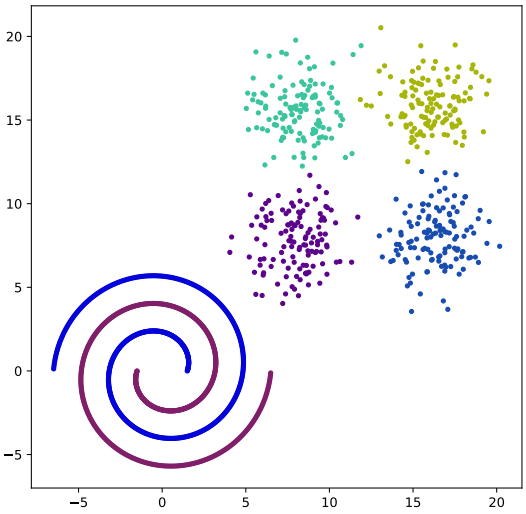

Figure 5.13: $RI = 1$, $ARI = 1$

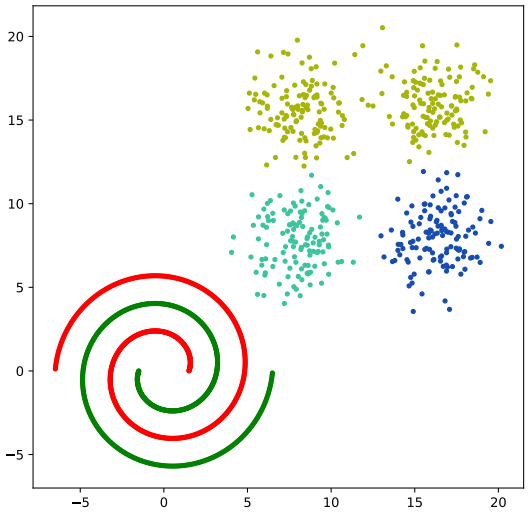

Figure 5.14: $RI = 0.988$, $ARI = 0.963$

These outcomes reiterate what has been explained in the previous descriptive chapters. Consider that, even if it seems that more or less the two solutions are "equivalent" (where one fails the other succeeds), however the BridgeClustering algorithm could be better since potentially improvable as illustrated in the next conclusive chapter.

Chapter 6

Future projects

This chapter summarizes all the work done for this thesis and looks at possible future projects in order to improve it. As said at the beginning, the goal was to find an algorithm that solves the clustering problem in the best possible way. Unfortunately, the clustering problem is really complex. The very first intuition was to design an algorithm based on data expansion; this gave birth to the ExpansionClustering algorithm in all its versions. Subsequently, after having ascertained the advantages and the limitations of such algorithm, it was decided to change approach completely by moving towards machine learning, in particular artificial neural networks. Finally, this idea led to the BridgeClustering algorithm.

It has seen that the BridgeClustering algorithm frees us from setting parameters, but also that the obtained results are still not the right ones, even if almost. What are the drawbacks to be remedied?

- First of all, the *number of training sets* used is not enough to cover the entire variety of clusters. Typically, the more training sets you have, the better.

- The *inner structure* chosen for the ann may not be optimal. In particular, starting from scratch, how many hidden layers would be better to place? And how many neurons should each layer have? Different combinations have been tried (e.g. 10 layers with 50 neurons each, 20 layers with 20 neurons each, pyramidal inner structures, etc) and at the end the best one

was selected. Definitely, these questions are not trivial. Given that artificial neural networks is a very vast topic, only experience and specialization in this field can bring significant improvements.

- Even if you cannot tell exactly, most likely the trained network is suffering from some *overfitting*. Anyway, in order to definitely avoid overfitting it would be better to adopt sophisticated techniques like *dropout* or *regularization*.

- The eventual presence of *noise* could be a problem (see Figures 6.1-6.2).

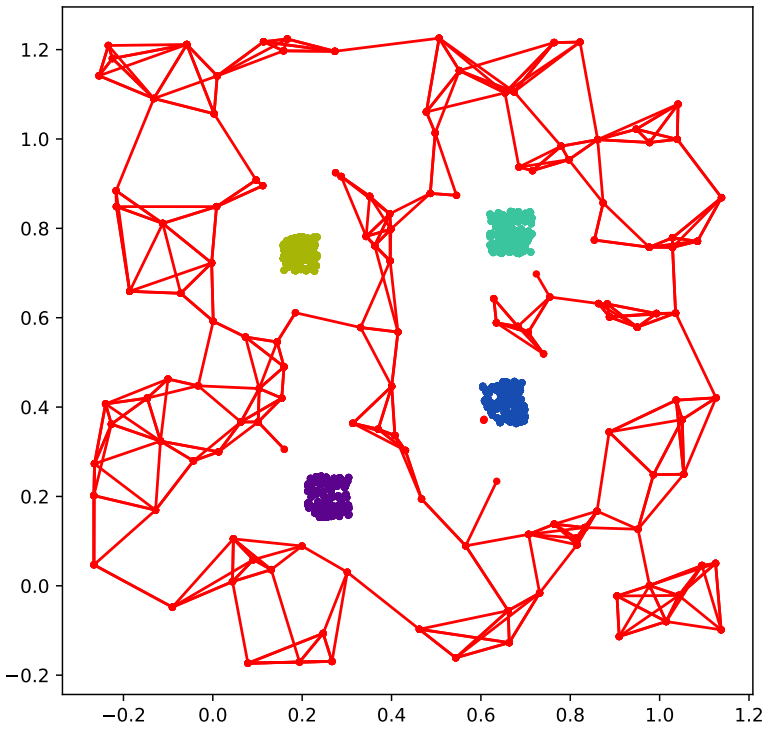

Figure 6.1: Data set with noise (in red)

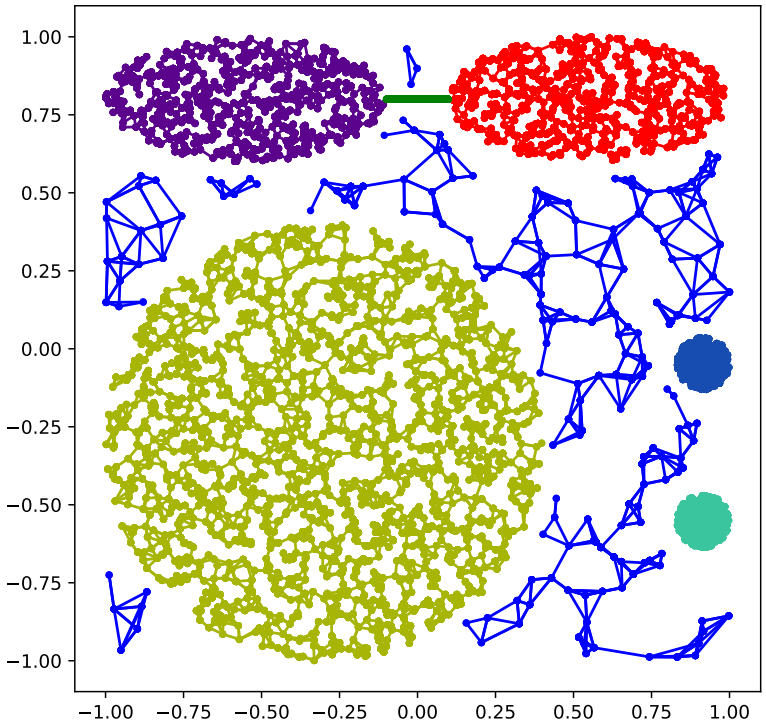

Figure 6.2: Data set with noise (in medium-blue)

As you can see, noise cannot always be connected by bridges in its entirety. Therefore, it would be useful to enrich the pre-processing with a noise rejection action before even performing the bridge counting.

These are the key points to be deepened in order to make BridgeClustering an algorithm usable in the real world. Investing in a solution based on machine learning is strongly motivated by the fact that, nowadays, a high number of problems are solved brilliantly by cutting-edge learning techniques. This happens because programmers face increasingly complex problems, so complex that they would not be solvable without requiring the machine to learn from the man himself. This is the new computer science. And, inevitably, the technologies derived from it have implications of an economic, social and moral nature. Definitely, the principle is to support the man and not to replace him completely, since technologies are always means and never ends.

References

[1] GitHub data set benchmark:
https://github.com/deric/clustering-benchmark/

[2] Image segmentation theory (chapters 9, 10, 11):
http://web.ipac.caltech.edu/staff/fmasci/home/astro_refs/
Digital_Image_Processing_2ndEd.pdf