Space Breakdown Method

A new approach for density-based clustering

Authors Name/s per 1st Affiliation (*Author*)
line 1 (of *Affiliation*): dept. name of organization
line 2-name of organization, acronyms acceptable
line 3-City, Country
line 4-e-mail address if desired

Authors Name/s per 2nd Affiliation (*Author*)
line 1 (of *Affiliation*): dept. name of organization
line 2-name of organization, acronyms acceptable
line 3-City, Country
line 4-e-mail address if desired

***Abstract*— Overlapping clusters and different density clusters are recurrent phenomena of neuronal datasets because of how neurons fire. We propose a clustering method that is able to identify clusters with different densities, even with overlapping shapes. The experiments performed on synthetic data show that on imbalanced and overlapping cluster the proposed approach has similar or better results than some of the more popular algorithms, but the algorithm does not scale well with higher dimensions.**

Keywords—clustering; density; grid; spike sorting; machine learning; overlapping clusters; different density;

I. Introduction

Spike sorting in neuroscience addresses the challenging problem of clustering the recorded spikes in an area based on the neuron that fired them.

Depending on their role, neurons fire at different rates. Some neurons are excitatory and have a higher rate of firing, while others are inhibitory and fire less often [1].

Due to multiple possible reasons like electrode drift, neurons have different shaped spikes when firing in quick succession or the spikes of different neurons can have similar features making it difficult to decide which neuron they belong to.

There are other issues in spike sorting, but we tackle in this paper the ones described above. Another issue is the large volume of data of spikes. We assume the spikes are detected correctly and the right features were chosen [2]. These problems are translated into and solved as an imbalanced clustering problem. Because of this, the approach is applicable to clustering in general, not only to spike sorting.

Our aim is to be able to identify the correct number of neurons recorded and assign the spikes to the neuron that produced them. But due to the noisy nature of data, it is acceptable to define more clusters than the actual number of neurons and to merge them in a postprocessing step. Noise might not be classified at all

II. Related Work

Two of the most used algorithms in clustering and spike sorting are DBSCAN and K-Means.

DBSCAN [3] is a density-based approach that considers points grouped together as clusters and marks points with few neighbours as noise. DBSCAN does not require the number of clusters as input and is able to find clusters of arbitrary shapes, but it struggles with datasets that have clusters of different densities.

K-Means [4] partitions the dataset into k clusters in which each point is claimed by the cluster with the nearest mean. One of the biggest disadvantages that it has is that it is hard to estimate the number of clusters in advance. Another problem is that since it is based on distance, it has trouble classifying clusters of different densities.

III. The Space breakdown Method

A. Dataset description

The neuroscience issues described in the introduction boil down to a set of spikes that should be split into clusters having different densities and an unknown amount of overlap including the possibility of them to be embedded. A spike is a point in an N dimensional space where N is the number of features selected to represent the spike. Our objective is to develop a clustering method that can handle large amounts of data points, yet able to identify the number of different density clusters and sort the points to their respective cluster. Discovering more clusters than the actual number of neurons is acceptable as long as the number of misclassified points is kept to a minimum. A point that was not assigned to a cluster, even though it could have been part of one, is not considered misclassified.

The datasets in neuroscience can have tens of thousands of points. They can contain dense clusters of excitatory neurons which can extend on a larger area and usually overlap. There also are smaller less dense clusters of inhibitory neurons nearby. Due to how sparse they are and their proximity to the denser clusters, it is difficult to distinguish them from noise.

Most traditional clustering algorithms are not efficient in dealing with different density clusters or with slightly or totally overlapping clusters. Through this approach we have tried to identify the number of clusters and their general shape when dealing with large datasets of points which correspond to different density clusters with a gaussian distribution that can overlap.

The proposed process takes in a dataset of points as input and outputs an array of labels for each of the points, representing the cluster they belong to.

B. Solution overview

Since the data are points in an N dimensional space, we propose to start by normalizing the dataset to bring every point of the dataset in the range [0, PN] on all N dimensions. The partitioning number (PN) represents the number of chunks each axis is split in. Each chunk has the length of 1 on each dimension. In a 2D space the chunks are squares, in a 3D space they are cubes, etc. An N dimensional density array stores the number of points from the original dataset belonging to each of these unit chunks. This array will have PN elements in each dimension resulting in PN^N chunks. On this array, we look for the possible cluster centroids. We start by finding the elements that have a larger value than a given threshold (`centroidThreshold`) which eliminates the possibility of noise chunks to be considered as cluster centroids. A good value for this threshold would be half of the total amount of points divided by the number of chunks created. Out of the chunks that passed that threshold, the chunks that are bigger than all their neighbors, i.e. the local maxima are considered to be the candidates for the centroids of all the clusters in the dataset. After finding the centroid candidates, we apply a breadth-first search (BFS) to expand each of these centroids to find the rest of the cluster by having an expansion queue which starts with the centroid and adding valid neighbors to the queue and the cluster. This results in each chunk receiving a label. These labels are gathered in a label array having the same size as the density array. The last step is to assign each point in the dataset the label of the chunk it belongs to.

We have illustrated the flow of the approach in Figure 1

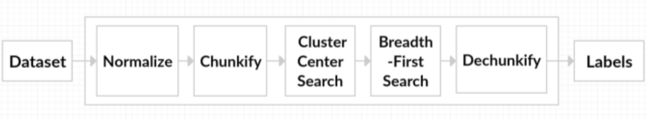

Figure 1 – Pipeline of the approach

The approach has a time complexity of O(n) for the normalize, chunkify and dechunkify algorithms and $O(PN^N)$ for the cluster centroid search and expansion algorithms.

C. Detailed algorithm

The required parameters of the approach are the dataset and the hyperparameters: `PN` and `centroidThreshold`.

The dataset is a matrix where the lines represent a point in space and the N columns (dimensions) represent the coordinates of the point.

The pseudocode for the proposed Space Breakdown Method (SBM):

```
SBM(dataset, pn, threshold, eLimit)
 X = normalize(dataset, pn)
 densityArray = chunkify(X, pn)
 cc = findCentroids(densityArray, threshold)
 initialize labelsArray with zeros
 for indexOfCenter= 0 to len(cc):
   center = cc[indexOfCenter]
   if labelsArray[center] == 0
     labelsArray=expand(densityArray
               labelsArray,indexOfCenter+1,cc)
  labels = dechunkify(X, labelsArray, pn)
 return labels
```

At first, we normalize the dataset to make the chunking process easier by applying a min-max normalization to bring all the coordinates in the interval [0-1] using minmax normalization.

$$X' = \frac{X - min(X)}{max(X) - min(X)}$$

Then we multiply each point with PN to bring the dataset in the 0-PN range on each dimension.

The second step represents the core idea of the algorithm which is aggregating points based on their location and reducing the number of elements we work with. It stems from the objective of working with datasets with large number of points. We breakdown the N dimensional space into chunks that count the number of points located in them, thus making the number of points less relevant and we only work with the number of chunks we create which can be much smaller than the number of points in the original dataset.

As inputs we have the normalized dataset of points X, and PN. As a result of this step we get a density array, that represents the chunkified dataset.

```
1  chunkify(X,  pn)
2    initialize densityArray with zeros
3        for point in X:
4          location = floor(point)
5      densityArray[location]+=1
6    return densityArray
```

We transform the dataset into an N dimensional density array with PN elements on each dimension (line 2). Since each point now has the coordinates between 0-PN, by flooring the values of the coordinates on each dimension (line 4), we obtain indexes on all dimensions locating to the unit chunk that the point belongs to (line 5).

The third step is to find the possible cluster centroids. We need the density array calculated in the previous step and a chosen threshold as inputs. The result is a list of the cluster centroids.

```
findClusterCentroids(densityArray,threshold)
   clusterCentroids = []
   for location in densityArray
     if ( densityArray[location] >= threshold
        and isMaxima(densityArray, location))
          add location to clusterCentroids
   return clusterCentroids
```

We iterate through all the PN^N elements in the density array (line 3) and check a couple of conditions. Location represents an array with N elements containing the coordinates with which we access a chunk in the `densityArray`. Firstly, for a chunk to be considered a centroid candidate, the value of the chunk must be greater than an input threshold (line 4). This reduces the possibility of noise chunks to create clusters. Secondly, the chunk must be a local maximum (line 5).

To be a local maximum, all of the neighbours of a chunk must have a value less than or equal to the chunk. We consider neighbours all chunks with a location that varies by at most 1 on any axis.

```
1  isMaxima(densityArray, location):
2    neighbours = getNeighbours(location)
3    for nb in neighbours:
4
if(densityArray[nb]>densityArray[location]):
5        return False
6    return True
```

GetNeighbors returns a list of locations of all the neighbors of the current location while making sure that the locations exist in the array meaning that on all dimensions, the values are between [0 - PN).

```
1 getNeighbours(location):
2  offsets= all N digit combinations of -1,0,1
3  remove the 0 on all dimensions offset
4  neighbours = []
5  for every offset in offsets
6    newLoc = p + offset
7      if newLoc>=0 and newLoc<PN
8        neighbours append newLoc
9    return neighbours
```

Once the cluster centroids candidates have been found, the algorithm goes through each candidate(SBM line 6) and, if the candidate was not included in another cluster yet(SBM line 8), it expands the cluster using the BFS approach.

The parameters for expand are: density array (`dnstyA`), the labels array we got so far (`lblsA`), the label with which we are marking the current cluster (`crntLbl`) and the list of the cluster centers (`cc`) to solve conflicts. The label of a cluster is the index of that clusters center in the `cc` array +1, this implies that the labels start from 1 and that label 0 represents unclustered. The output is an updated labels array which contains the expanded cluster.

```
 expand(dnstyA,lblsA,crntLbl,cc)
1  startC = cc[crntLbl]
2  init visited, shaped as dnstyA, with false
3  expansionQueue = [startC]
4  lblsA[start] = crntLbl
5  visited[start] = true
6  dropoff = calcDropoff(array,startC)
7  while expansionQueue not empty
8   chunk = expansionQueue.pop()
9   shape = shape of dnstyA
10  nbrs = getNeighbours(chunk, shape)
11  for nbr in nbrs
12   distance = distance(startC,nbr)
13   minVal = dropoff*sqrt(distance)
14   if not visited[nbr] and \
15         minVal<dnstyA[nbr]<=dnstyA[chunk]
16     visited[nbr] = true
17     if lblsA[nbr] == 0
18       expansionQueue.push(nbr)
19       lblsA[nbr] = crntLbl
20     else //nbr was discovered already
21       expand = solveConflict(
22                dnstyA,lblsA,nbr,crntLbl,cc)
23       if expand == true
24          expansionQueue.push(nbr)
25 return lblsA
```

Expand was developed in an empirical way starting from the base idea of BFS of expanding to the all the neighbours first. One of the first things we do is calculate the drop-off of the cluster which is a number that represents how quickly we should stop expanding. A high value means that the cluster is going to be discovered in only a few chunks and the expansion should stop soon. Distance from the center also play a role in the expansion and by multiplying the square root of the distance with the drop-off value, we obtain a number which can be used as the minimum required amount of points for a chunk to be considered part of the cluster.

There exists the possibility that the chunk, to which we want to extend, is already assigned to a cluster (line 20). If so, we solve the dispute (line 21) and check if we should continue expanding that way or stop.

To calculate the drop-off, we need the density array and the chunk for which we want to calculate the drop-off. Using a similar function to the root mean square, we get an idea of how well the chunk fits with its neighbours. If a cluster has the drop-off of the center high, it means that the variation between the cluster center and its neighbours is high which means that the cluster does not extend on a large area.

```
1 calcDropoff(dnstyA,chunk)
2  dropoffSum = 0
3  nbrs = getNeighbours(chunk,shape of dnstyA)
4  for nbr in nbrs
5    diff = dnstyA[chunk] - dnstyA[nbr]
6    dropoffSum += (diff^2)/dnstyA[chunk]
7  return sqrt(dropoffSum)
```

We only expand to chunks that have a value less than or equal to the one of the chunk we expanded from. This is because we expect the data to have a gaussian distribution meaning if we go further away from the center, we cannot get higher values. `MinVal` increases with distance because although by going further away from the center, we expect to have less points that belong to the cluster, the chunks become more and more likely to be noise chunks and it is important to us to not consider them part of the cluster.

Once the conditions for expansion to a chunk were met, we need to check if the chunk was previously claimed by another cluster (expand line 17). If there is a conflict (expand line 20), we solve it by checking which of the two clusters (current cluster vs old cluster) has a greater pull on the conflict chunk.

To solve a conflict, we need the density array, labels array, the location of the conflict chunk, the label with which we were expanding (`crntLbl`) and the list of cluster centroids. The result is an updated labels array and a boolean signifying whether or not we should keep expanding in that direction.

```
1 solveConflict(dnstyA,lblsA,loc,crntLbl,cc)
2  crntClstr = cc[crntLbl]
3  oldLbl = lblsA[loc]
4  oldClstr = cc[crntLbl]
5  if loc == oldClstr or
7            dnstyA[loc] == dnstyA[oldClstr]
6    in lblsA replace the oldLbl with crntLbl
7    return false
8  d1 = distance(loc, crntClstr)
9  d2 = distance(loc, oldClstr)
10 drop1 = calcDropoff(dnstyA,crntClstr) * d1
11 drop2 = calcDropoff(dnstyA,oldClstr) * d2
12 str1= dnstyA[crntClstr]/dnstyA[loc] - drop1
13 str2= dnstyA[crntClstr]/dnstyA[loc] - drop1
14 if str1 > str2
15   lblsA[loc] = crntLbl
16   return true
17 else
18   return false
```

The calculation of the pull of a cluster results from an empirical formula based on the ratio between the cluster centroid and the conflict chunk having, having a positive effect on the pull, distance from the centroid of the cluster to the conflict point and the drop-off of the centroid, having a negative impact.

There are 3 possible outcomes of the conflict:

- The current cluster has a greater pull in which case we relabel the chunk and continue expanding (lines 15,16).
- The old cluster has a greater pull (line 18) and we stop the expansion.
- The old cluster was actually a subsection of the current cluster which happened to have a cluster centroid candidate and was expanded first (lines 6,7).

Once all the cluster candidates have been considered for expansion, the last step is what we call the dechunkification of the labels array back into the labels of the dataset. We need the original dataset X, the labels array resulted from the BFS and the partitioning number PN, to output the final point labels.

```
dechunkify(X, labelsArray, pn)
 initialize pointLabels with zeros
 for index from 0 to length(X):
   point = X[index]
   location = floor(point)
   pointLabels[index]=labelsArray[location]
 return pointLabels
```

PointLabels is an array having the length equal to the number of points in the dataset.

IV. Validation

For validation, we compared DBSCAN and K-Means to our algorithm on 4 datasets. The datasets have been chosen so that they have different types of densities and different amounts of overlapping. The 4 datasets are: S1, S2 and Unbalanced (U) provided by the University of Eastern Finland [7] and the fourth one, named Unbalance Overlapping (UO), was generated by us.

The datasets are synthetically generated, have two-dimensions and have the following characteristics:

- S1 (Appendix, figure 1a) contains 5000 points distributed (Gaussian distribution) equally in 15 clusters (333 points each)
- S2 (Appendix, figure 1b) contains 5000 points distributed (Gaussian distribution) equally in 15 clusters (333 points each)
- U (Appendix, figure 1c) contains 6500 points in 8 clusters, the 3 clusters on the right contain 2000 points each while the other 5 on the left side 100 each
- UO (Appendix Figure 1d) contains 4300 points with the following distribution:
    - ❖ 500 points with the cluster center at [-2, 0] and a standard deviation of 0.8 (blue)
    - ❖ 50 points with the cluster center at [-2, 3] and a standard deviation of 0.3 (cyan)
    - ❖ 1000 points with the cluster center at [3, -2] and a standard deviation of 1 (magenta)
    - ❖ 1250 points with the cluster center at [5, 6] and a standard deviation of 1 (yellow)
    - ❖ 250 points with cluster center at [4, -1] and a standard deviation of 0.1 (red)
    - ❖ 1250 points with cluster center at [1, -2] and a standard deviation of 0.2 (green)

Because we are interested in the points that we do assign a cluster to, actually belong to that cluster and because SBM does consider a significant portion of the points as noise, we will compare the three algorithms twice using two metrics. The first test will be including all the points (ALL) and the second one will be comparing only the classified points meaning no noise points (NNP).

The two metrics are: Adjusted Rand Index (ARI) [5] and Adjusted Mutual Information (AMI) [6]. The Rand Index (RI) metric looks at the similarity of two clusters taking pairs of points, looking if they are in the same or different clusters in the predicted and the given labels and divide it by the sum of the agreements and disagreements. Then this value is adjusted for chance.

The Adjusted Mutual Information metric is adjustment for chance of the Mutual Information metric. Mutual Information looks at the similarity of two labels.

TABLE I. ADJUSTED RAND INDEX (ALL)

	S1	S2	U	UO
K-Means	**99.49%**	**95.75%**	**100.00%**	66.17%
DBSCAN	96.96%	84.74%	99.99%	56.69%

SBM	66.31%	53.60%	98.17%	**84.56%**

TABLE II. ADJUSTED RAND INDEX (NNP)

	S1	S2	U	UO
K-Means	99.49%	95.75%	**100.00%**	66.17%
DBSCAN	99.51%	**97.37%**	99.99%	57.40%
SBM	**100%**	96.46%	99.99%	**92.57%**

TABLE III. ADJUSTED MUTUAL INFORMATION (ALL)

	S1	S2	U	UO
K-Means	**99.46%**	**96.36%**	**100.00%**	72.99%
DBSCAN	96.06%	88.61%	99.81%	63.13%
SBM	83.34%	78.07%	93.29%	**79.35%**

TABLE IV. ADJUSTED MUTUAL INFORMATION (NNP)

	S1	S2	U	UO
K-Means	99.46%	96.36%	**100.00%**	72.99%
DBSCAN	99.51%	**98.03%**	99.87%	63.48%
SBM	**100%**	96.68%	99.75%	**91.17%**

The parameters used for DBSCAN are:

- min points = log(dataset size) for all datasets
- eps = 27000, 45000, 18000, 0.5 (S1, S2, U, UO)

The values of k for K-Means were: 15 for S1, 15 for S2, 8 for U and 6 for UO

On UO, we can see that the score of both K-Means and DBSCAN fall heavily, and we can easily see that they do not cluster correctly (Appendix, figures 2 and 3), while SBM is able to find all of the clusters, but not all of the points of each cluster (Appendix, figure 4). We can also observe that SBM slit the original yellow cluster in two, this is not a problem because a manual reviewer could merge those two clusters where as he could not split the clusters merged by K-Means (original blue and cyan are now one green cluster) and DBSCAN (original green, red and pink are now one red cluster).

SBM does not perform that well on S1 and S2 and that is because of what is considered noise. We can also observe that by removing the points that were classified as noise, K-Means performs the same since it does not label noise, DBSCAN performs better and SBM has a massive improvement being close or even better than K-Means and DBSCAN.

We also tested the duration of the execution. The implementation of the approach was done using python (version 3.6) with the following libraries: NumPy (version 1.16), matplotlib (version 3.1.0), sklearn (version 0.21.2) [8], pandas (version 0.24)

It has been run on a laptop with Intel Core i7 4720HQ at 2.60GHz with 4 cores hyperthreaded, 16GB RAM at 1600MHz, NVIDIA GeForce GTX 950M with 2048 MB of VRAM,1 TB HDD with 5400 RPM with the following results (average of 100 runs):

TABLE V. TIME

	S1	S2	U	UO
K-Means	0.147s	0.208s	**0.078s**	0.100s
DBSCAN	**0.049s**	**0.062s**	0.183s	**0.091s**
SBM	0.127s	0.136s	0.117s	0.098s

From Table V we can observe that the SBM has a runtime somewhere in between K-Means and DBSCAN. We can also observe that the times across all datasets do not vary as much as they do for the other 2 algorithms.

V. DISCUSSION AND CONCLUSIONS

The algorithm did achieve our goal of handling large datasets in a reasonable amount of time by running a 80.000 point real life dataset in under 5 seconds, but this came at a cost in other parts. Due to the space and time complexity of PN^N, the algorithm is very sensitive to the PN parameter and the number of dimensions of a point. PN even though is much smaller than the expected number of points in a dataset, if the points have many dimensions, the algorithm suffers. For example a dataset with 1.000.000 points and 10 dimensions, even for a partition number as low as 5, 5^{10} chunks are created and at least 8.000.000 of them are empty. On the other hand, if the number of dimensions is low and the number of points in the dataset is much larger than the partitioning number and since PN and N are fixed, $O(PN^N)$ the complexity of the algorithm is O(n) meaning that adding more points to the dataset will only increase the time linearly.

The partitioning number should also be large enough so that the unit chunks can separate the clusters, but not large enough to make the centroid of a sparse cluster to be undetectable. On our datasets, PN=25 seemed the best value.

In conclusion, SBM can have similar or better performance compared to K-Means and DBSCAN when we are not interested in what is classified as noise but are interested in classifying correctly what the points that we do assign to clusters in datasets where clusters have different densities and overlap. Also, if the number of dimensions is low, it can perform the clustering in O(n)

REFERENCES

[1] G. Buzsaki, “Rhythms of the Brain”, 2009

[2] M. Lewicki, “A review of methods for spike sorting: the detection and classification of neural action potentials” in Network: Computation in Neural Systems, 1998

[3] Transylvanian Institute of Neuro-Science: http://tins.ro/

[4] MacQueen, J. Some methods for classification and analysis of multivariate observations. Proceedings of the Fifth Berkeley Symposium on Mathematical Statistics and Probability, Volume 1: Statistics, 281--297, University of California Press, Berkeley, Calif., 1967. https://projecteuclid.org/euclid.bsmsp/1200512992

[5] Hubert, L. & Arabie, P. Journal of Classification, 1985, 2: 193. https://doi.org/10.1007/BF01908075

[6] Vinh, Epps, & Bailey, “Information Theoretic Measures for Clusterings Comparison: Variants, Properties, Normalization and Correction for Chance”, JMLR, 2010

[7] Clustering basic benchmark datasets from the University of Eastern Finland: https://cs.joensuu.fi/sipu/datasets/

[8] Scikit-learn: Machine Learning in Python, Pedregosa et al., JMLR 12, pp. 2825-2830, 2011

Appendix

In the following figures are plotted the datasets that are present in the validation phase:

.

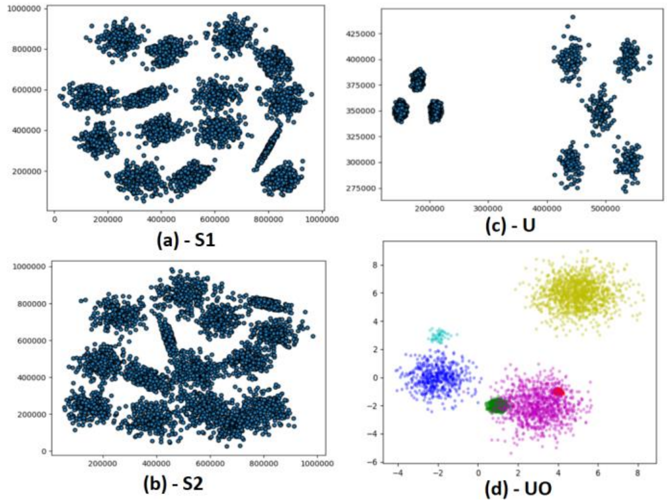

Figure 1 – Datasets

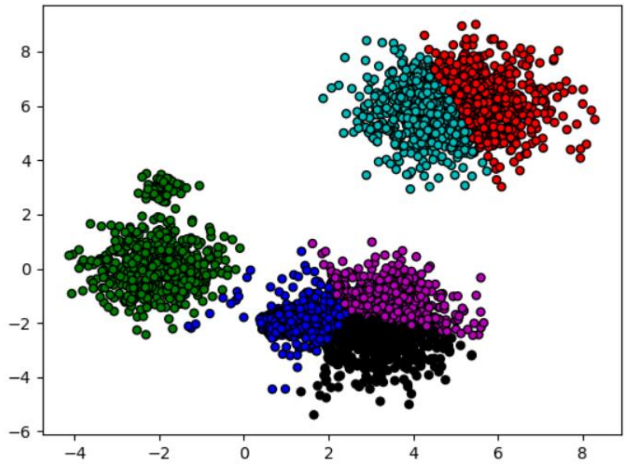

Figure 2 – K-Means on UO

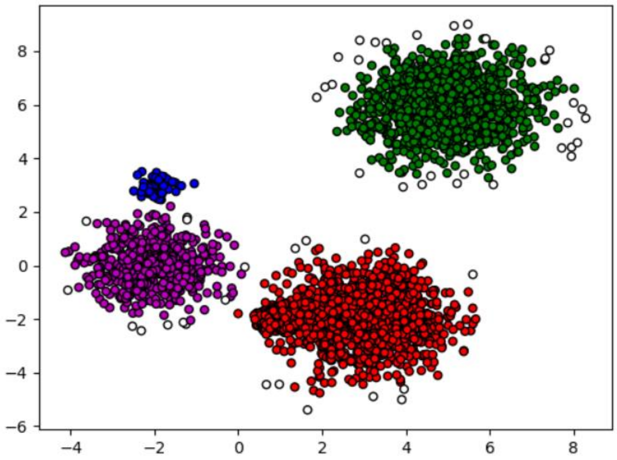

Figure 3 –DBSCAN on UO

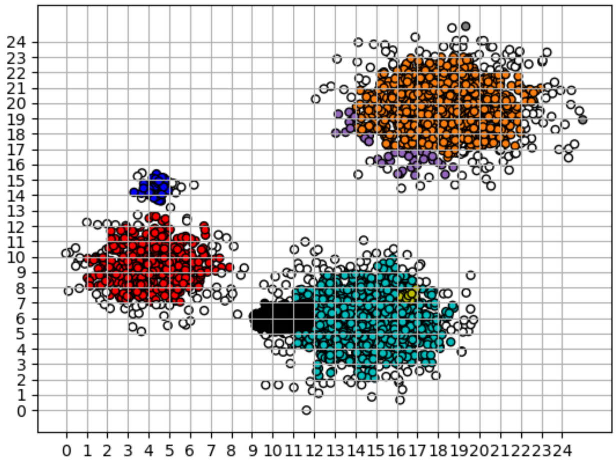

Figure 4 – SBM on UO