


Spring 2024

An Unsupervised Machine Learning Algorithm for Clustering Low Dimensional Data Points in Euclidean Grid Space

Josef Lazar
Bard College

Follow this and additional works at: https://digitalcommons.bard.edu/senproj_s2024

 Part of the [Analysis Commons](#), [Applied Mathematics Commons](#), [Applied Statistics Commons](#), [Artificial Intelligence and Robotics Commons](#), [Categorical Data Analysis Commons](#), [Data Science Commons](#), [Discrete Mathematics and Combinatorics Commons](#), [Numerical Analysis and Scientific Computing Commons](#), [Other Statistics and Probability Commons](#), and the [Set Theory Commons](#)



This work is licensed under a [Creative Commons Attribution-Noncommercial-No Derivative Works 4.0 License](#).

Recommended Citation

Lazar, Josef, "An Unsupervised Machine Learning Algorithm for Clustering Low Dimensional Data Points in Euclidean Grid Space" (2024). *Senior Projects Spring 2024*. 164.

https://digitalcommons.bard.edu/senproj_s2024/164

This Open Access is brought to you for free and open access by the Bard Undergraduate Senior Projects at Bard Digital Commons. It has been accepted for inclusion in Senior Projects Spring 2024 by an authorized administrator of Bard Digital Commons. For more information, please contact digitalcommons@bard.edu.

An Unsupervised Machine Learning Algorithm for Clustering Low Dimensional Data Points in Euclidean Grid Space

A Senior Project submitted to
The Division of Science, Mathematics, and Computing
of
Bard College

by
Josef Lazar

Annandale-on-Hudson, New York
May, 2024

Abstract

Clustering algorithms provide a useful method for classifying data. The majority of well known clustering algorithms are designed to find globular clusters, however this is not always desirable. In this senior project I present a new clustering algorithm, GBCN (Grid Box Clustering with Noise), which applies a box grid to points in Euclidean space to identify areas of high point density. Points within the grid space that are in adjacent boxes are classified into the same cluster. Conversely, if a path from one point to another can only be completed by traversing an empty grid box, then they are classified into separate clusters. GBCN requires two hyperparameters, one to determine the size of the grid and the other to adjust noise sensitivity. I provide algorithms and evaluation metrics to help the user determine appropriate hyperparameter values. I performed experiments on synthetic and real world data sets using GBCN and other clustering algorithms to evaluate GBCN's effectiveness and efficiency. The results of these experiments demonstrate that GBCN can effectively identify both globular and density-based clusters when given the right hyperparameter values, and that these hyperparameter values can be discovered using evaluation metrics.

Contents

Abstract	iii
Dedication	vii
Acknowledgments	ix
1 Introduction	1
1.1 Definitions of Clusters	2
1.2 Background Information on Algorithms	5
1.2.1 Algorithm 1: k -Means Clustering	6
1.2.2 Algorithm 2: k -Medians Clustering	7
1.2.3 Algorithm 3: DBSCAN	8
1.2.4 Algorithm 4: k -Nearest Neighbors	9
2 Grid Box Clustering	11
2.1 Constructing Grid Boxes	13
2.2 Adding Data Points into Grid Boxes	18
2.3 Clustering Points by Grid Boxes	20
3 Handling Noise in Data	25
3.1 Noise Reduction	25
3.2 Algorithms for Reclustering Noise	27
3.2.1 Noise Reclustering Algorithm 1: Nearest Neighbors With Euclidean Distance	27
3.2.2 Noise Reclustering Algorithm 2: Nearest Neighbors With Manhattan Dis-	
tance	29
3.2.3 Noise Reclustering Algorithm 3: Nearest Neighbors With Box Distance .	29

4	Tuning Hyperparameters	33
4.1	Clustering Evaluation Algorithms	34
4.1.1	Clustering Evaluation Algorithm 1: Silhouette Coefficient	34
4.1.2	Clustering Evaluation Algorithm 2: Calinski Harabasz Index	35
4.1.3	Clustering Evaluation Algorithm 3: Davies-Bouldin Index	36
4.1.4	Clustering Evaluation Algorithm 4: DBCV	37
4.1.5	Clustering Evaluation Algorithm 5: Rand Index	39
4.2	Using Evaluation Metrics to Optimize Hyperparameters	40
5	Results	43
5.1	Experiments With GBCN on Synthetic Data Sets	43
5.2	Experiments With k -Means Clustering and DBSCAN on Synthetic Data Sets	47
5.3	Using Evaluation Metrics to Optimize Synthetic Data Clustering	52
5.4	Using Evaluation Metrics to Optimize Real World Data Clustering	53
6	Ideas Worth Exploring Further	57
6.1	Adding Sophistication to hyperparameter optimization	57
6.2	Changing definition of neighboring grids	58
6.3	Optimization for High Dimensional Data Sets	58
	Bibliography	63

Dedication

I would like to dedicate this project to my family who has lovingly supported me along this journey and in life. Specifically I'd like to thank my mom and dad, my sister Mara, and my grandparents Mimi, Jack, and Arlene who have always been there for me. Thank you for shaping me into the person that I am today.

Acknowledgments

I would like to thank my advisors Rose Sloan and Ethan Bloch who have generously devoted much time to this project and to my education throughout my undergraduate career. Their guidance and attention and encouragement made this project an amazing experience and pushed me to try harder. Thank you. I would also like to thank Lauren Rose, Kerri-Ann Norton, Sven Anderson, Chuck Doran, Stefan Mendez-Diez, and Valerie Barr who I have had the privilege of learning from and who have been mentors to me. Finally I want to acknowledge and thank Daniel Rose-Levine and Santanu Antu, who have been among my closest friends at Bard, and who have taught me much of the math that I know today. I can't express how grateful I am for all of you.

1

Introduction

Clustering, or cluster analysis, is a method of data analysis that attempts to sort data into unordered classes, such that data points in each class share some commonality that separates them from the points in other classes. Clustering algorithms are commonly used and studied in the fields of statistics and machine learning, but are also broadly applied for their practical uses across many scientific fields and other areas of inquiry that require data analysis.

Machine learning as a field exists within the intersection of artificial intelligence and statistics. Generally, machine learning algorithms take a large data set as input and make predictions about it, or generate similar data. The artificial intelligence aspect of it refers to the fact that it can learn from data and make human-like insights. The statistical aspect refers to the fact that it uses mathematics to interpret and organize data. In fact, many algorithms, such as k -nearest neighbors and k -means clustering, that are today considered canon in machine learning algorithms, originated in statistics.

Broadly speaking, machine learning algorithms can be divided into two categories: **supervised**, and **unsupervised**, with semi-supervised algorithms and other variations laying on a spectrum in between. Supervised algorithms are trained on inputted data, which contains the information, known as **labels**, which we are trying to predict. The idea is that after seeing enough examples, it will know how to predict labels on new data points. An example of this

would be the Linear Regression algorithm which fits a line to a data set, and then uses the line to predict labels (the y values) on new data points (inputted x values) with missing labels by plugging them into the line equation. An algorithm that fits the line to the data, or more generally makes a function for predicting labels, is a **machine learning algorithms**. An outputted line equation, or more generally a function that has been tuned to a training data set to be able to predict labels of similar data points, is called a **model**.

Unsupervised algorithms on the other hand learn patterns exclusively on unlabeled data, and in the case of clustering, the data they are learning on is the data they are making predictions about. When this is the case, an implementation of the algorithm is a finished model. Still, their performance can be evaluated by running them on labeled data sets and comparing the labels predicted by the algorithm to the labels assigned in the data set (more on this in Chapter 4).

Many algorithms have parameters, called **hyperparameters**, that are not determined during the learning process, but rather are chosen by the user. Both supervised and unsupervised algorithms can have them. By examining how an unsupervised algorithm performs, given various hyperparameter values, on a data set with known labels, the user can learn to make more informed decision about which hyperparameter values to select when running the algorithm on unlabeled data.

A visualization of the distinction between supervised and unsupervised algorithms can be seen in Figure 1.0.1 from [14]. In the box on the right we see that the algorithm was given the label (color) of each data point, and based on this information was able to draw a line according to which it can classify new points whose labels are not known. In the box on the left we see that the algorithm was only given the location of the points, not their labels, and used the relationship of the points to each other to create its own classification labels.

1.1 Definitions of Clusters

A **cluster**, in machine learning, is a grouping of unlabeled data points. It is useful in situations where true labels of data points are not known. Researchers who collect data may seek

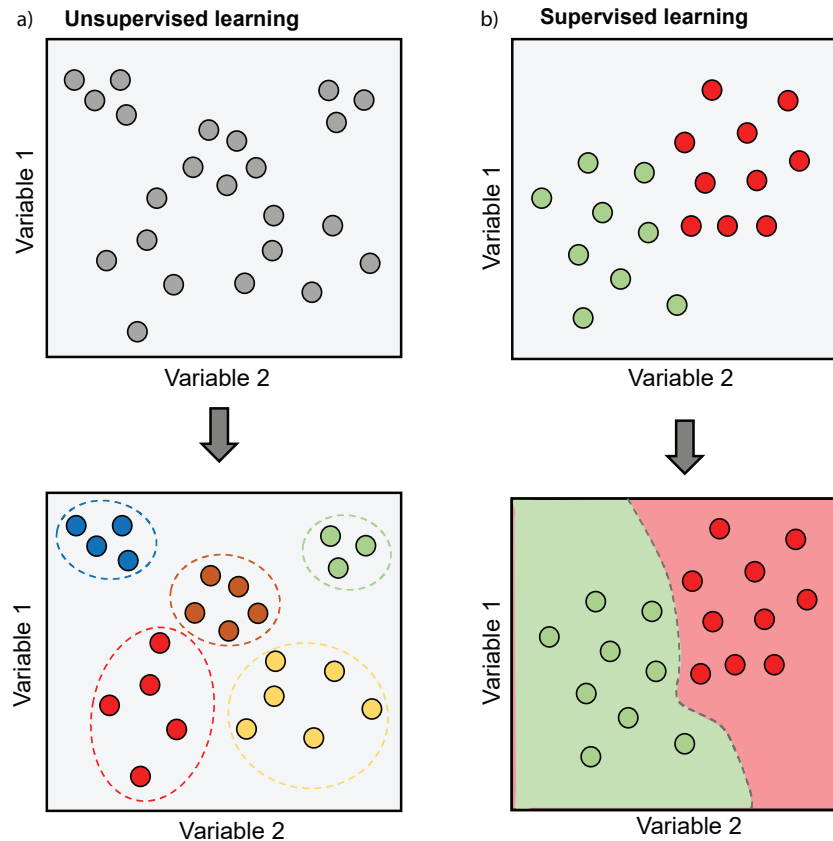


Figure 1.0.1: Example of how supervised and Unsupervised learning works, adopted from page 5 of [14]

mathematical tools for automated classification and analysis of their data. For example a sociological inquiry into a population may ask about peoples' income, demographic information, religious beliefs, and political views. People, represented as data points that occur near each other may share some interesting properties, that are not shared by some other cluster of data points further away. Based on this information, clustering algorithms could be used to divide the population into political interest groups with distinct beliefs and priorities. This information could be used by clever political strategists to create policy that reallocates resources from clusters within the population whose support is not needed to the ones whose support is needed to.

Along different areas of research, different notions of clusters may be of interest. In machine learning **globular clusters** are thought of as having a circular shape and often, but not always,

having increased density towards the center. This definition is inspired by the astronomical definition of a globular cluster, which is a spheroidal conglomeration of stars that is bound together by gravity. The stars seen in Figure 1.1.1 form an astronomical globular cluster. A data set containing their locations would form a globular cluster by the machine learning definition.



Figure 1.1.1: Messier 2, identified by NASA as a globular cluster. A data set containing the locations of its stars would also be a globular cluster, in the machine learning sense of the world

Density-Based clusters are another common notion of clusters. They do not take shape into consideration, instead separating clusters in the data by areas of low point density. In Figure 1.1.2 from [6] we see a map of New York City, where yellow areas indicate a lot of taxi cab pick-ups on January 16th, 2016. The data set that this figure was derived from included

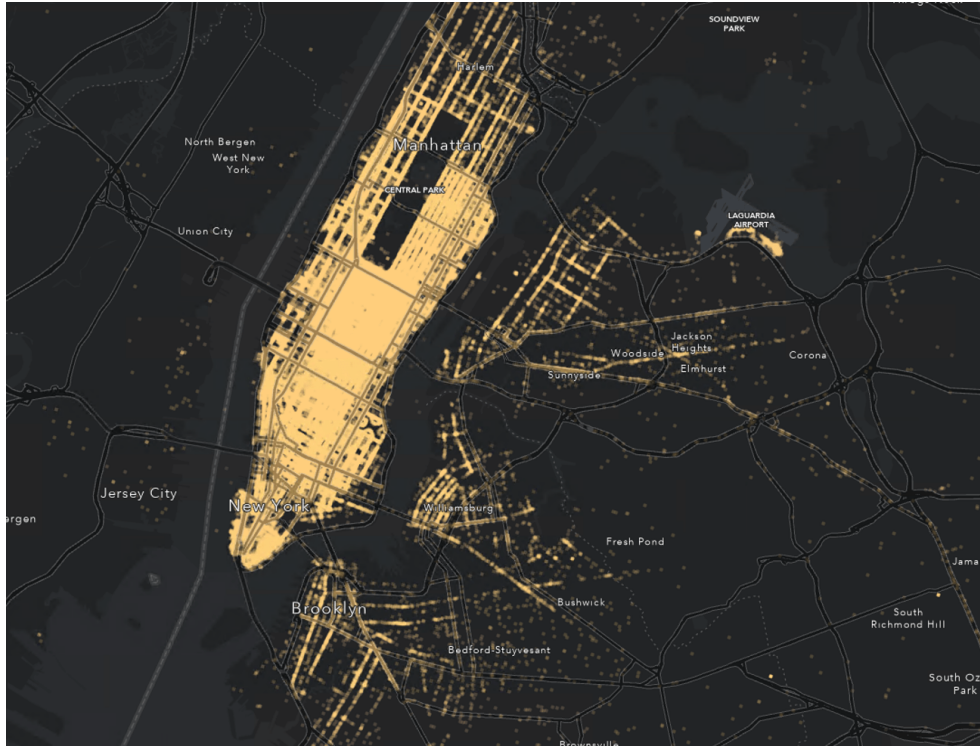


Figure 1.1.2: Map of New York City, where yellow areas indicate a high amount of taxi cab pick-ups on January 16th, 2016. The data set that this map was derived from included the times at which the pick-ups occurred

the times at which the pickups occurred, so some dense yellow areas could be classified into multiple clusters if there were distinct times of increased taxi pick-up activity separated by times of low taxi pick-up activity. The New York City Department of Transportation may be interested in identifying times and areas of high taxi pick-up activity to help it enact laws for traffic management planning. In this case the clusters will rarely be circular. Thus an algorithm that identifies clusters as continuous areas of high point density may prove to be more useful.

1.2 Background Information on Algorithms

Some noteworthy algorithms include k -means clustering, k -medians clustering, DBSCAN, and k -nearest neighbors. I give a review of these algorithms to give the reader a general overview of the variation of strategies that can be used to generate clusters and labels, and to show how algorithms can be modified, optimized, and used for both supervised and unsupervised tasks.

This review also gives relevant background information on the subject as a whole and on some of the ideas that inspired this project.

1.2.1 *Algorithm 1: k -Means Clustering*

This unsupervised clustering algorithm sorts data into k clusters, trying to minimize the distance between points in a cluster and the center of the cluster. Canon in cluster analysis, its simple and efficient yet powerful design has made k -means clustering an easy go-to choice for tasks involving clustering across many application. The original idea behind the algorithm is attributed to Polish mathematician Hugo Dyonizy Steinhaus, who published a paper [23] in 1956 in French, where he explained it and, sceptical of official government reporting, used it to estimate the number of German soldiers killed during World War II. The word k -means is thought to have first been used by James MacQueen, who independently discovered the algorithm and published it in an article [13] in 1967. For more history on the origins of k -means clustering see [8].

The algorithm takes hyperparameter k as input and begins by randomly choosing k points, “centroids,” in the space that the data set exists in. For each point in the data set it calculates its distance from the centroids and assigns it to be in the cluster of the centroid that it is closest to. Then it calculates the mean location of every cluster and moves the centroids to that location. It once again iterates over the points, reclustering them based on the new centroids. This process is repeated until points no longer change clusters between iterations, or until some threshold amount of iterations has been reached. A visual example of k -means clustering can be seen in Figure 1.2.1 from [27].

k -means clustering attempts to minimize square distance between points and their centroids. Because of this it is excellent at finding globular clusters. It often struggles, however, with identifying density-based clusters, whose points do not necessarily bare any relation to the centroid of the cluster they were assigned to.

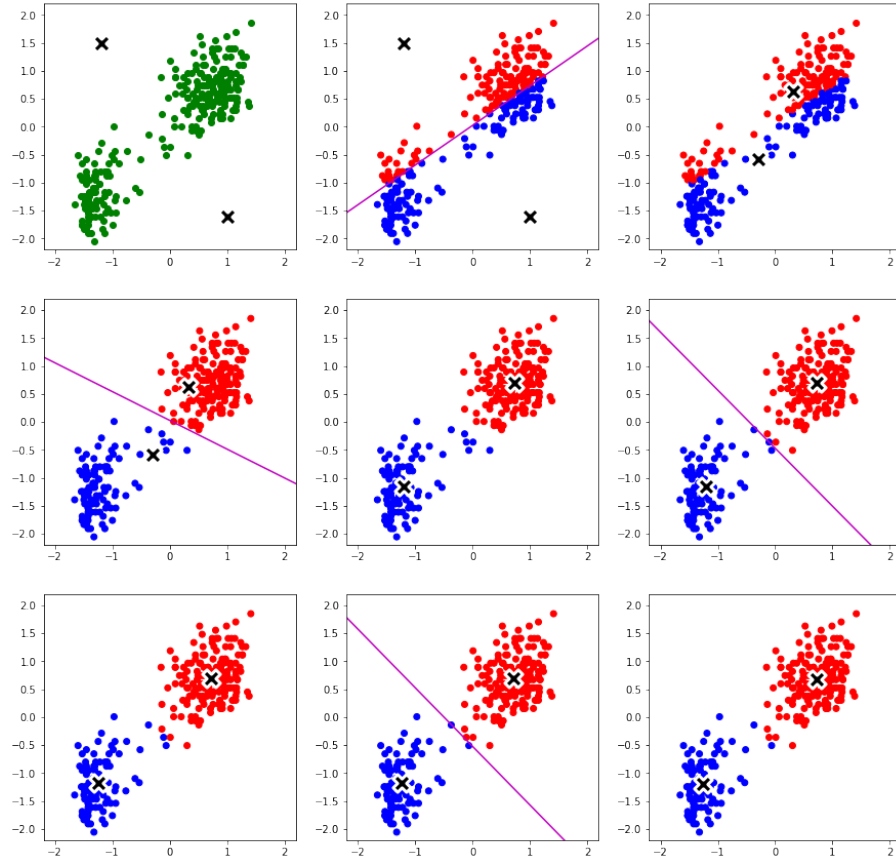


Figure 1.2.1: A data set in two dimensional Euclidean space being clustered with k -means clustering for $k = 2$ in 4 iterations

1.2.2 Algorithm 2: k -Medians Clustering

This unsupervised clustering algorithm is similar to k -means clustering. It differs in that it updates the value of a centroid by making its i th coordinate equal to the median value of the i th coordinate of all points in its cluster. This way the coordinates of the centroid will have always come from the data set (or be equal to the average of two values in the data set). This can make k -medians the more reliable option for discrete and binary data. k -medians shows how small changes to an existing algorithm to make it more suitable for different applications. Unlike k -means clustering, k -medians optimizes for minimal Manhattan distance between points and their centroids. This too causes for strong performance in globular cluster identification but weak performance in density-based cluster identification.

1.2.3 Algorithm 3: DBSCAN

Proposed in 1996 by Martin Ester, Hans-Peter Kriegel, Jörg Sander, and Xiaowei Xu [4], DBSCAN (Density-Based Spatial Clustering of Applications with Noise) is an unsupervised clustering algorithm that was specifically designed for finding clusters using a density-based understanding of what a cluster is. In this way it differs from the previous two algorithms. It also differs from them in the fact that it recognizes that some data points may not be of interest (fluke events, minor events that contradict a larger general trend, mistakes in data collection, etc.) and has the ability to classify them separately as noise points, rather than as members of a cluster.

DBSCAN takes two hyperparameters as input: ε and `MinPts`. It then separates all points in the data set into three categories. Firstly, core points, which have at least `MinPts` points within distance ε . Secondly, border points, which have fewer than `MinPts` points within distance ε , but which have at least one core point within distance ε . Thirdly, noise points, which fewer than `MinPts` points within distance ε and no core points within distance ε . All core points that are within ε distance of each other get classified into the same cluster. Border points are classified into the cluster of one of the core points within distance ε . A visualization of this can be seen in Figure 1.2.2 from [21].

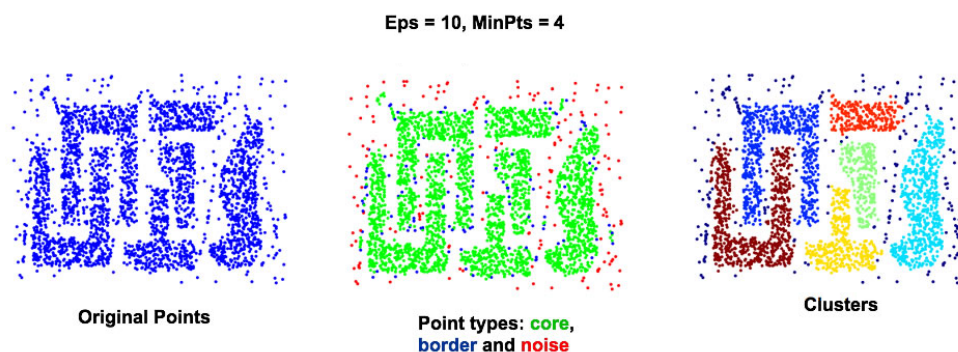


Figure 1.2.2: The picture on the left shows a data set. The picture in the middle shows how DBSCAN would divide it into core points, border points, and noise points, given $\varepsilon = 10$ and `MinPts` = 4. The picture on the right shows that subsequent clustering that DBSCAN would generate

1.2.4 Algorithm 4: *k*-Nearest Neighbors

Proposed in 1951 by Marcelo Beckmann, Nelson F. F. Ebecken, and Beatriz S. L. Pires de Lima [1], the *k*-NN (*k*-nearest neighbors) algorithm is a supervised learning classifier, which means that it takes a data set with known labels as input, and infers information from this data set to predict labels for a second inputted data set where labels are not known. In the context of this project we can think of the data set with labels as having been clustered, where points that have the same label are in the same cluster.

The *k*-NN algorithm classifies the unlabeled data in the following way. It iterates over each unlabeled point. For each one it computes the distance between it and all the labeled data points. It takes the *k* closest labeled data points and among them finds the most frequent label. This label is then assigned to the unlabeled point.

The `suvs_data` data set from [24] contains the following information about consumers: user ID, gender, age, estimated salary, and whether or not they purchased as SUV. The last bit of information is the label in this case. Using code from [11] we can apply *k*-NN with $k = 5$ to the age and estimated salary information, to create the model seen in Figure 1.2.3 which predicts if someone has bought an SUV. If then given a data set of consumers about whom we know their age and estimated salary, but don't know if they're purchased as SUV, we can plot the points into the model, and if a point is in the red region then most of the 5 nearest points in the labeled data set will have not purchased SUVs, if in the green then most of the 5 nearest points in the labeled data set will have purchased an SUV.

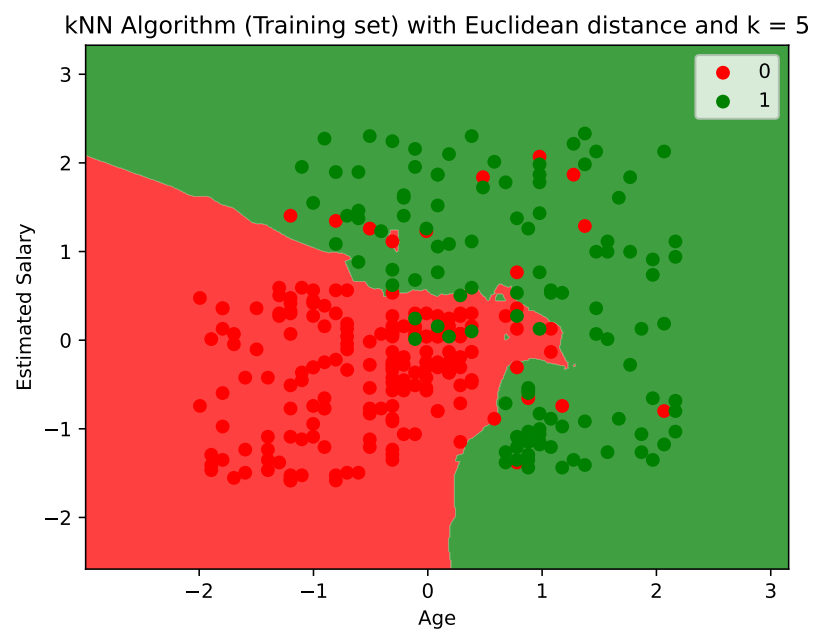


Figure 1.2.3: k -NN model with $k = 5$ predicting if consumers have purchased an SUV based on their age and estimated salary

2

Grid Box Clustering

Even though k -means clustering, k -median clustering, DBSCAN, and k -nearest neighbors are commonly used, they may not always be the best clustering algorithms for all tasks. That is why I decided to explore a different algorithm, called GBCN (Grid Box Clustering with Noise). This chapter is the first of two chapters explaining how the GBCN algorithm works. I explain here how grid boxes are constructed, how points are added to them, and finally how grid boxes and points get clustered. In the next chapter I finish explaining GBCN by discussing how it processes noise. I start this chapter with a simple explanation of how GBCN clusters grid boxes and continue till the end with a deeper and more technical explanation.

GBCN is an unsupervised clustering algorithm. It clusters data sets whose data points all exist in n -dimensional Euclidean space, for any $n \in \mathbb{N}$. The algorithm begins by applying a box grid to the Euclidean space. That is, if the space is one-dimensional, then the points lay on a line, and the line is cut into segments of equal length, and the program figures out which points are in which segments. Similarly, if the space is two-dimensional, then it divides it into rectangular regions, and if it is three-dimensional, then it is cut into rectangular boxes. More generally, an n -dimensional space will be partitioned into n -dimensional rectangular boxes. Each of these segments (lines, rectangles, rectangular boxes, etc.) will be referred to as a **grid box**. A visualization of this idea, up to three dimensions, can be seen in Figure 2.0.1 .

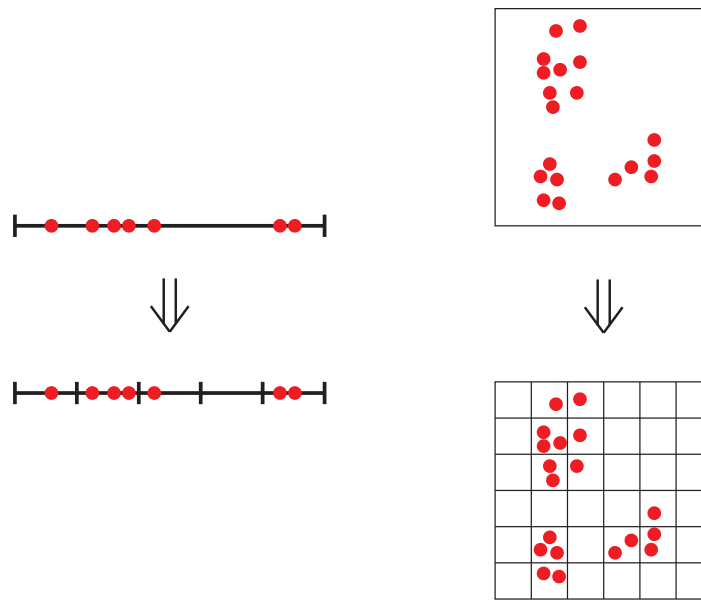


Figure 2.0.1: Square grid being applied to points in \mathbb{R}^1 and \mathbb{R}^2

Once the data points have been sorted into grid boxes, the clustering process can begin. Boxes are separated into **non-empty** ones containing at least one point, and **empty** ones. Two grid boxes are considered **neighbors** if they share a corner, edge, surface, etc. As seen in Figure 2.0.2, in one dimension, grid boxes can only share a corner; in two dimensions, grid boxes can share a corner or an edge; in three dimensions, grid boxes can share a corner, an edge, or a surface; and so on. If two boxes are neighboring, or connected through non-empty boxes, then they get classified as being in the same box cluster. Conversely, if you can not traverse from one box to another, without entering an empty box, then they will be classified as being in different box clusters. All the points in a box cluster are then labeled as being in the same cluster of points.

A critical question arises: What size should the grid boxes be? The simple answer to this question is that there is no correct answer that works every time. A size that is optimal for one data set could perform very poorly for another data set. Later (see Chapter 4) we will discuss methods for approximating the correct size for a given data set, but for now the basic idea is that a good size will be small enough that there are empty grid boxes between points that are in

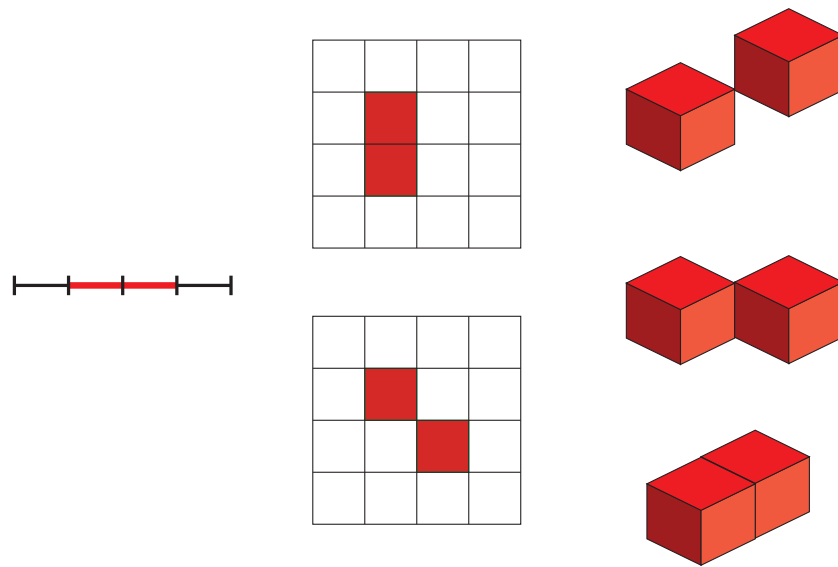


Figure 2.0.2: Examples of how grid boxes can neighbor each other in 1, 2, and 3 dimensional grid spaces

different clusters, but big enough that points within one cluster should either share grid boxes or be in neighboring grid boxes.

2.1 Constructing Grid Boxes

GBCN has been implemented in Python. It begins by making a `Grid` class. The dimension of the data set it will be applied to, the partitioning of the grid space, and the noise reduction sensitivity (see Chapter 3) are determined during initialization by the constructor:

```
def __init__(self, width, dimension, noise_reduction)
```

The `Grid` class has attribute `grid`, which is a d -dimensional list, where d is the inputted dimension. The contents of grid box $b = (b_1, b_2, \dots, b_d)$ can be retrieved with

```
grid[b1][b2]...[bd]
```

Note that there is a bijection between grid boxes and elements in lists at `grid`'s lowest level. The first $(d - 1)$ layers of `grid`'s lists store references to other lists. Lists at the bottom of the tree at layer d initially contain zeros until they are later replaced with lists of points that belong

to the corresponding box. If an element of a list at the d th layer is a non-list value (including the initial integer value zero) then it indicates that the corresponding grid box is empty.

Grid's width input tells it how many times to partition each axis. `width` can be a positive `int` or it can be a list of length d whose elements are positive `ints`. If `width` is an integer, then each axis of the grid space will be cut into that integer amount of partitions. See Figure 2.1.1 for a code snippet example. If the inputted `width` is a list of integers, then after checking that the length of `width` is equal to `dimension`, it cuts each dimension of `grid` into its corresponding index in `width` amount of grid boxes. See Figure 2.1.2 for a code snippet example.

```
>>> example_grid1 = Grid(4, 3, 0)
>>> print(example_grid1.grid)
[[[0, 0, 0, 0], [0, 0, 0, 0], [0, 0, 0, 0], [0, 0, 0, 0]],
 [[0, 0, 0, 0], [0, 0, 0, 0], [0, 0, 0, 0], [0, 0, 0, 0]],
 [[0, 0, 0, 0], [0, 0, 0, 0], [0, 0, 0, 0], [0, 0, 0, 0]],
 [[0, 0, 0, 0], [0, 0, 0, 0], [0, 0, 0, 0], [0, 0, 0, 0]]]
```

Figure 2.1.1: Visualization of Grid's attribute `grid`, where the `width` input was an integer. It has 3 dimensions, each of which are partitioned into 4 grids

```
>>> example_grid2 = Grid([5, 6, 4], 3, 0)
>>> print(example_grid2.grid)
[[[0, 0, 0, 0], [0, 0, 0, 0], [0, 0, 0, 0], [0, 0, 0, 0], [0, 0, 0, 0], [0, 0, 0, 0]],
 [[0, 0, 0, 0], [0, 0, 0, 0], [0, 0, 0, 0], [0, 0, 0, 0], [0, 0, 0, 0], [0, 0, 0, 0]],
 [[0, 0, 0, 0], [0, 0, 0, 0], [0, 0, 0, 0], [0, 0, 0, 0], [0, 0, 0, 0], [0, 0, 0, 0]],
 [[0, 0, 0, 0], [0, 0, 0, 0], [0, 0, 0, 0], [0, 0, 0, 0], [0, 0, 0, 0], [0, 0, 0, 0]],
 [[0, 0, 0, 0], [0, 0, 0, 0], [0, 0, 0, 0], [0, 0, 0, 0], [0, 0, 0, 0], [0, 0, 0, 0]]]
```

Figure 2.1.2: Visualization of Grid's attribute `grid`, where the `width` input was a list of integers. It has 3 dimensions, `grid` has length 5, `grid[i]` has length 6, and `grid[i][j]` has length 4

The creation of the `grid` list is done recursively using the `add_dim_len_n(grid, n)` function seen below. It takes an incomplete grid (a grid that does not yet have the inputted amount of dimensions) as its first input, and the desired amount of partitions in the dimension to be added as its second input. Using the fact that every element at the bottom of the grid should be an integer (or a float), it recursively iterates through the entire grid, replacing every `int/float` with a list whose length is the second input and whose elements are all zero.

```
def add_dim_len_n(grid, n):
    local_grid = grid.copy()
```

```

#if at the lowest layer of the grid
if is_int_or_float(local_grid[0]):
    #add a new layer containing n boxes and return
    for i in range(len(local_grid)):
        local_grid[i] = [0] * n
    return local_grid

#if not yet at the lowest layer
elif is_list(local_grid[0]):
    #recursively run this function one layer lower, then return
    for i in range(len(grid)):
        local_grid[i] = add_dim_len_n(local_grid[i], n)
    return local_grid

#one of the two previous conditions should have been satisfied
#if not, check for errors
else:
    raise Exception(
        "every data type within the grid should be a list, int, or float")

```

This function is run $d - 1$ times where the input for `grid` is initially `[0, 0, ..., w]` or `[0, 0, ..., w[0]]` and subsequently the `grid` that the previous iteration returned. The input for `n` is `width` (if `width` is an integer) or `width[i]` (if `width` is a list of integers), where `i` indicates the iteration.

Theorem 2.1.1. *The time and space complexity of constructing a d -dimensional grid, whose width w is an integer, is $\Theta(w^d)$.*

Proof. For the base case suppose $d = 1$. Then the grid is a one dimensional list of length w , and the time and space complexity of constructing it is $\Theta(w) = \Theta(w^d)$. For the inductive step let $d \in \mathbb{N}$ and suppose that the time and space complexity of a d -dimensional grid is $\Theta(w^d)$. Run the function `add_dim_len_n` on the grid to increase it to a $(d + 1)$ -dimensional grid. The

function will recurse w times to get from the first layer to the second layer, w^2 times to get from the second layer to the third layer, all the way up to w^{d-1} times to get from the $(d-1)$ th layer to the d th layer. In total

$$w^1 + w^2 + \dots + w^{d-1} = \sum_{i=1}^{d-1} w^i$$

recursions will occur. Note

$$w^{d-1} < w^1 + w^2 + \dots + w^{d-1} < 2w^{d-1},$$

hence $w^1 + w^2 + \dots + w^{d-1} = \Theta(w^{d-1})$, so $\Theta(w^{d-1})$ will be the time and space complexity cost of recursing through the grid.

When `add_dim_len_n` gets to the bottom layer it does not recurse, instead it creates w lists of length w that it adds to the grid. Since it gets to this layer w^{d-1} times, the time and space cost will be $\Theta(w^{d-1}w^2) = \Theta(w^{d+1})$. Combine the cost of getting the initial d -dimensional grid, the cost of recursing through it, and the cost of adding the new layer to find that a $(d+1)$ -dimensional grid has time and space complexity $\Theta(w^d + w^{d-1} + w^{d+1}) = \Theta(w^{d+1})$. \square

Theorem 2.1.2. *The time and space complexity of constructing a d -dimensional grid, whose width $\vec{w} = (w_1, w_2, \dots, w_d)$ is a list, is*

$$\Theta\left(\prod_{i=1}^d w_i\right) \text{ and } O\left(\max_{w_i \in \vec{w}} (w_i)^d\right),$$

assuming $w_i \geq 2$ for all $i \in \{1, 2, \dots, d\}$.

Proof. For the base case suppose $d = 1$. Then the grid is a one dimensional list of length w_1 , and the time and space complexity of constructing it is

$$\Theta(w_1) = \Theta\left(\prod_{i=1}^d w_i\right).$$

For the inductive step let $d \in \mathbb{N}$ and suppose that the time and space complexity of a d -dimensional grid is $\Theta\left(\prod_{i=1}^d w_i\right)$. Run the function `add_dim_len_n` on the grid to increase it to a $(d+1)$ -dimensional grid. The function will recurse w_1 times to get from the first layer to the

second layer, then $w_1 \cdot w_2$ times to get from the second layer to the third, all the way up to $w_1 \cdot w_2 \cdot \dots \cdot w_{d-1}$ times to get from the $(d-1)$ th layer to the d th layer. In total

$$(w_1) + (w_1 \cdot w_2) + \dots + (w_1 \cdot w_2 \cdot \dots \cdot w_{d-1}) = \sum_{i=1}^{d-1} \left[\prod_{j=1}^i w_j \right]$$

recursions will occur. Note

$$(w_1 \cdot w_2 \cdot \dots \cdot w_{d-1}) < (w_1) + (w_1 \cdot w_2) + \dots + (w_1 \cdot w_2 \cdot \dots \cdot w_{d-1}).$$

Note that we assumed $w_i \geq 2$ for all $i \in \{1, 2, \dots, d\}$. From this we can deduce the following

$$\begin{aligned} 2(w_1 \cdot w_2 \cdot \dots \cdot w_{d-1}) &= (w_1 \cdot w_2 \cdot \dots \cdot w_{d-1}) + (w_1 \cdot w_2 \cdot \dots \cdot w_{d-1}) \\ &\geq (w_1 \cdot w_2 \cdot \dots \cdot w_{d-1}) + 2(w_1 \cdot w_2 \cdot \dots \cdot w_{d-2}) \\ &= (w_1 \cdot w_2 \cdot \dots \cdot w_{d-1}) + (w_1 \cdot w_2 \cdot \dots \cdot w_{d-2}) + (w_1 \cdot w_2 \cdot \dots \cdot w_{d-2}) \\ &\geq (w_1 \cdot w_2 \cdot \dots \cdot w_{d-1}) + (w_1 \cdot w_2 \cdot \dots \cdot w_{d-2}) + 2(w_1 \cdot w_2 \cdot \dots \cdot w_{d-3}) \\ &\vdots \\ &\geq (w_1 \cdot w_2 \cdot \dots \cdot w_{d-1}) + \dots + (w_1 \cdot w_2) + (w_1) \end{aligned}$$

Hence, we've proven that

$$(w_1 \cdot w_2 \cdot \dots \cdot w_{d-1}) < (w_1) + (w_1 \cdot w_2) + \dots + (w_1 \cdot w_2 \cdot \dots \cdot w_{d-1}) \leq 2(w_1 \cdot w_2 \cdot \dots \cdot w_{d-1}),$$

so

$$(w_1) + (w_1 \cdot w_2) + \dots + (w_1 \cdot w_2 \cdot \dots \cdot w_{d-1}) = \Theta(w_1 \cdot w_2 \cdot \dots \cdot w_{d-1}) = \Theta\left(\prod_{i=1}^{d-1} w_i\right)$$

will be the time and space complexity cost of recursing through the grid.

When `add_dim_len_n` gets to the bottom layer it does not recurse, instead it creates w_d lists of length w_{d+1} that it adds to the grid. Since it gets to this layer $w_1 \cdot w_2 \cdot \dots \cdot w_{d-1}$ times, the time and space cost will be

$$\Theta((w_1 \cdot w_2 \cdot \dots \cdot w_{d-1}) \cdot w_d \cdot w_{d+1}) = \Theta(w_1 \cdot w_2 \cdot \dots \cdot w_{d+1}) = \Theta\left(\prod_{i=1}^{d+1} w_i\right).$$

Combine the cost of getting the initial d -dimensional grid, the cost of recursing through it, and the cost of adding the new layer to find that a $(d + 1)$ -dimensional grid has time and space complexity

$$\Theta \left(\left(\prod_{i=1}^d w_i \right) + \left(\prod_{i=1}^{d-1} w_i \right) + \left(\prod_{i=1}^{d+1} w_i \right) \right).$$

Using the same logic that we used to prove that $2(w_1 \cdot w_2 \cdot \dots \cdot w_{d-1}) \geq (w_1 \cdot w_2 \cdot \dots \cdot w_{d-1}) + \dots + (w_1 \cdot w_2) + (w_1)$, we can prove that this complexity is equal to

$$\Theta \left(\prod_{i=1}^{d+1} w_i \right).$$

Note there is some $i \in \{1, 2, \dots, d\}$ such that $w_i \geq w_j$ for all $j \in \{1, 2, \dots, d\}$. Then

$$(w_i)^d \geq \prod_{j=1}^{d+1} w_j,$$

so

$$O \left(\max_{w_i \in \vec{w}} (w_i)^d \right)$$

is a valid upper bound time and space complexity of constructing a d -dimensional grid. \square

In both the case where the width is an integer, and where the width is a list of integers, its time and space complexity will grow exponentially as the number of dimensions increase. This growth is prohibitive, and as a result my computer begins to run out of its 16GB of RAM when computing 9-dimensional grids with 8 or more partitions per axis. The majority of this space is being taken up by references to empty grid boxes. In Section 6.3 I discuss ways that this problem could be improved. I show there, that in theory, the time and space complexity could be reduced to $O(nd + b \log(b))$, where n is the size of the data set, where d is the number of dimensions that the data points exist in, and where b is the number of non-empty boxes in the grid.

2.2 Adding Data Points into Grid Boxes

The `Grid` class has a `data_points` attribute that stores the data set. Data sets can be added to a `Grid` object using its `add_points` method, which takes a data set as a two dimensional list

as input. After a data set is inputted, it finds the most extreme points along each axis, saving their values into the `Grid` object's list attributes `max_coordinates` and `min_coordinates`.

From the user's `width` input, the grid knows how many partitions each dimension should be cut into. Once it has updated `max_coordinates` and `min_coordinates`, the length of a grid box along axis i can be calculated by taking the difference between `max_coordinates[i]` and `min_coordinates[i]` and dividing it by the number of partitions on axis i . These box widths are stored as elements of `Grid` class's `box_widths` attribute.

Using this information `add_points` can proceed to iterate over every point and, with the help of the `find_grid_box` method, determine which grid box in `grid` it belongs in. The `find_grid_box` method takes a point and a dimension as input. It returns the box coordinate, along the inputted dimension, that the point belongs in. An example of this can be seen in Figure 2.2.1. Given a point $\vec{p} = (p_1, p_2, \dots, p_d)$, its box coordinate BC along axis i , which has w_i partitions, can be found using the following equation:

$$BC(\vec{p}, i) = \left\lfloor \frac{p_i - \min_i}{\max_i - \min_i} \cdot w_i \right\rfloor.$$

For points that are exactly on a partitioning line, they are assigned to the grid box with the higher coordinate number. The one exception to this rule are points along the highest edge: those are put into the lower box, as doing otherwise would cause an index error.

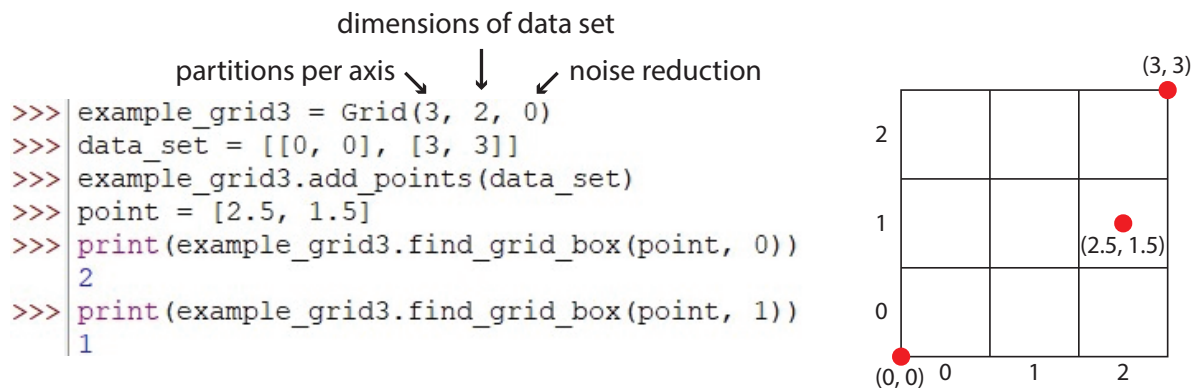


Figure 2.2.1: Demonstration of the `find_grid_box` method: We can see that it correctly identified point $(2.5, 1.5)$ as being in the box with index $(2, 1)$

Once `add_points` has determined a point's box coordinate, it adds it to the grid using the `add_point` method. The `add_point` method recursively iterates through a copy of the grid and through the remaining box coordinates until both have run out, at which point the base case has been reached and the point can be added to the current subgrid. The `add_point` method also updates `Grid`'s self explanatory `box_coordinate_to_points` and `point_to_box_coordinate` dictionary attributes.

For all points $\vec{p} \in \mathbb{R}^d$, it is the case that the box index of \vec{p} along any given axis can be calculated in $\Theta(1)$ time, so the entire box coordinate of \vec{p} can be calculated in $\Theta(d)$ time. The time complexity of placing all points in a data set into a grid is therefore $\Theta(dn)$, where n is the number of points the given data set. Keep in mind though, that the value of d must be small, as it is an exponent of complexity.

2.3 Clustering Points by Grid Boxes

Once a grid has been made, and the points have been added to it, the points can be clustered. The `cluster_boxes` method updates the `box_clusters` attribute, which is a list where every element is a **box cluster**, i.e. a list of box coordinates corresponding to boxes in the same cluster. It starts by getting a list of non-empty grid boxes by getting the keys of the `box_coordinate_to_points` dictionary attribute.

Next it filters out the noise boxes (for more on that see Chapter 3). After that it iterates over the remaining non-empty boxes. At the start of each iteration it pops a non-empty box, creates a new cluster of boxes which initially contains just this popped box, and finds a list of its neighboring boxes by using the `find_all_neighbors` method. The neighboring boxes get added to a stack of neighbors. To find all the boxes that should be in this cluster, a nested iteration – that terminates when the stack is empty – occurs. Every time a neighbor in the stack is found to be among the unvisited non-empty boxes, it gets added to the cluster, removed from the non-empty boxes list, and its neighbors are added to the stack of neighbors. When a stack becomes

empty, the resulting box cluster is appended to the `box_clusters` attribute. Pseudo-code can be seen below, and a visual example of this algorithm can be seen in Figure 2.3.1.

```

while len(non_empty_boxes) > 0: #iterates over all non-empty, non-noise boxes

    box = non_empty_boxes.pop()

    cluster = [box]

    #stores boxes that are potentially in the same cluster

    neighbors = self.find_all_neighbors(box)

    #repeats until all boxes in cluster have been found

    while len(neighbors) > 0:

        neighbor = neighbors.pop()

        #have we found a neighbor who belongs into the cluster?

        if self.box_non_empty(neighbor) and (not neighbor in cluster) and

            (not neighbor in self.noise_boxes):

            cluster.append(neighbor)

            non_empty_boxes.remove(str(neighbor))

        self.box_clusters.append(cluster)

return

```

The `find_all_neighbors` function takes some box coordinate $\vec{a} = (a_1, a_2, \dots, a_d)$ where $a_i \in \{0, 1, \dots, w_i - 1\}$ as input and returns a list containing

$$\left\{ \vec{b} \mid b_i \in \{a_i - 1, a_i, a_i + 1\} \text{ and } 0 \leq b_i < w_i \text{ holds for all natural } i \leq d \right\}.$$

The Python implementation of this method uses a recursive helper method that takes a box coordinate and a dimension to alter as input. The base case is that the inputted dimension is equal to the length of the box coordinates minus one, in which case it returns the following three neighbors: the inputted box coordinate, the inputted box coordinate with the last index incremented by one, and finally the inputted box coordinate with the last index decremented by one. In the recursive case it changes the inputted box coordinate at index `dimension` by $-1, 0,$

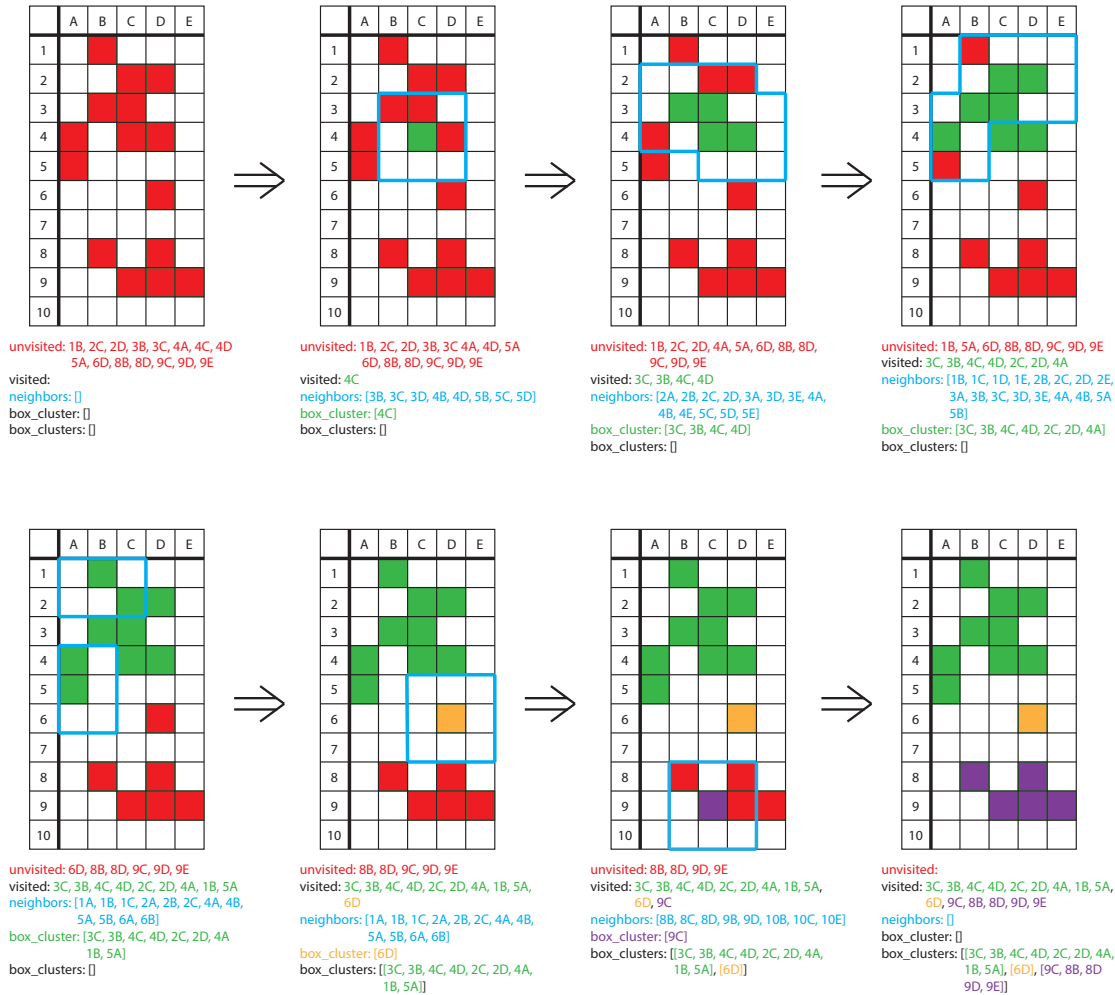


Figure 2.3.1: Demonstration of how the `cluster_boxes` method works on a toy data set. The last few steps are skipped over

and 1, and recursively calls itself on each of the altered box coordinates, with the value of the dimension increased by one.

The `cluster_points` method updates the `clustered_points` list attribute, which is a list where every element is a **point cluster**, i.e. a list of points in the same cluster. It begins by running the `cluster_boxes` method to sort the boxes into box clusters. Next, it iterates over box clusters, finding all the points in their boxes and adding them to a point cluster. For each box cluster a corresponding point cluster is made, which gets appended to the `clustered_points` attribute.

Theorem 2.3.1. *Let d be the dimension of a data set with n points, and let b be the number of non-empty boxes in the grid. The time complexity of the box clustering algorithm is $O(n3^d)$ and $\Theta(b3^d)$.*

Proof. To assess the time complexity of the box clustering algorithm, note that it iterates over every non-empty grid box once. Each time it iterates over a box, it computes all of its neighbors. Every box has $3^d - 1$ neighbors, so the time complexity of finding a box's neighbors is $\Theta(3^d)$. This is done for each non-empty grid box, so the time complexity of the box clustering algorithm is $\Theta(b3^d)$. The amount of non-empty grid boxes is at most the number of points in a given data set, so $O(n3^d)$ also serves as a valid upper bound time complexity. \square

Like with constructing grids, the time complexity grows exponentially as the number of dimensions increases. For large dimensions a majority of the computed neighbors will be empty grid boxes. For 10-dimensional data sets, each grid box will have more than 50,000 neighbors. For 20-dimensional data sets, each grid box will have more than 3 billion neighbors. In Chapter 6.3 I propose an alternative method for finding neighbors which would in theory reduce the time complexity of finding a grid box's neighbors to $\Theta(b^{\log_x(3)})$ for some $x \geq 3$, where b is the number of non-empty grid boxes. This would also reduce the time complexity of finding all non-empty grid box's neighbors to $\Theta(b^{1+\log_x(3)})$.

Given our definition of neighbors, one may justifiably ask how far apart can two points in neighboring grid boxes be from each other, which is answered in the following theorem.

Theorem 2.3.2. *Let $\vec{p} = (p_1, p_2, \dots, p_d)$, $\vec{q} = (q_1, q_2, \dots, q_d)$ be data points in a d -dimensional data set, whose width $\vec{w} = (w_1, w_2, \dots, w_d)$ is a list of integer, and whose minimum and maximum coordinates along any axis i are \min_i and \max_i respectively. If \vec{p} and \vec{q} are in neighboring grid boxes, then their Euclidean distance is at most*

$$2 \cdot \sqrt{\sum_{i=1}^d \left(\frac{\max_i - \min_i}{w_i} \right)^2},$$

and their Manhattan distance is at most

$$2 \sum_{i=1}^d \frac{\max_i - \min_i}{w_i}.$$

Proof. Note that the length of a grid box in this data set along some axis $i \in \{1, 2, \dots, d\}$ is

$$\frac{\max_i - \min_i}{w_i}.$$

Also note that by both the Euclidean and Manhattan definition of distance, points \vec{p} and \vec{q} will be furthest away from each other if their grid boxes, say $\vec{b}_{\vec{p}}$ and $\vec{b}_{\vec{q}}$ respectively, only share a vertex, and if \vec{p} and \vec{q} are in the vertices of their boxes that are opposite the vertex where $\vec{b}_{\vec{p}}$ and $\vec{b}_{\vec{q}}$ touch. It follows for any axis $i \in \{1, 2, \dots, d\}$ the distance (Euclidean and Manhattan) between p_i and q_i will be

$$|p_i - q_i| \leq \frac{\max_i - \min_i}{w_i} + \frac{\max_i - \min_i}{w_i} = 2 \cdot \frac{\max_i - \min_i}{w_i}.$$

It follows that the Euclidean distance d_E between \vec{p} and \vec{q} is

$$d_E(\vec{p}, \vec{q}) \leq \sqrt{\sum_{i=1}^d (p_i - q_i)^2} = \sqrt{\sum_{i=1}^d \left(2 \cdot \frac{\max_i - \min_i}{w_i}\right)^2} = 2 \cdot \sqrt{\sum_{i=1}^d \left(\frac{\max_i - \min_i}{w_i}\right)^2},$$

and the Manhattan distance d_M between \vec{p} and \vec{q} is

$$d_M(\vec{p}, \vec{q}) \leq \sum_{i=1}^d |p_i - q_i| = 2 \sum_{i=1}^d \frac{\max_i - \min_i}{w_i}.$$

□

3

Handling Noise in Data

Many data sets contain unhelpful bits of information, known as **noise**. This can include fluke events, minor events that contradict a larger general trend, errors in data collection, and other bits of data that can not be understood or interpreted correctly. In the context of the GBCN algorithm, noise would include points that don't belong to any cluster, as well as individual points, or small groups of points that are far away from the cluster that an ideal clustering would put them in.

3.1 Noise Reduction

The GBCN algorithm has a built in technique for filtering out noise. Given some noise reduction hyperparameter $k \in \mathbb{N}$, it treats all grid boxes with k or fewer points in them as if they were empty. This can eliminate minor outlying points, or for higher values of k , even remove a cluttered background of noisy points. For clusters which are deemed (by the user or by true labels) distinct, but which are connected by a narrow bridge of points, a good value of k will disrupt such a bridge, but preserve a majority of the points in the clusters. When noise reduction is applied, many points that otherwise would have been given a useful classification are at risk of being incorrectly classified as noise. To compensate for this, there are algorithms for reclustering noise points back into clusters.

A visual explanation of how GBCN's noise reduction works can be seen in Figure 3.1.1. It demonstrates how sensitive GBCN is to hyperparameters. In the figure we see that dividing each axis into 6 partitions with no noise reduction produces an ideal clustering, but reducing the partition count by just one produces a useless clustering where all points are put into one cluster. Still, this sensitive example is a lucky case, as there may be data sets where without using noise reduction there would not be a partition count which produced an ideal clustering. Note that the clusterings where there were 5 partitions and k was set to 1 or 2, and where there were 6 partitions and k was set to 1, produced clusterings with the right number of clusters in the right locations. Their only flaw was that they incorrectly classified some points as noise. We will see in the next section that with noise reclustering techniques, these clusterings could be used to produce ideal classification. Because of this, we can conclude that reclustering gives the user more flexibility when choosing the width and noise reduction hyperparameters, and broadens the domain of data sets that GBCN can give a useful clustering of.

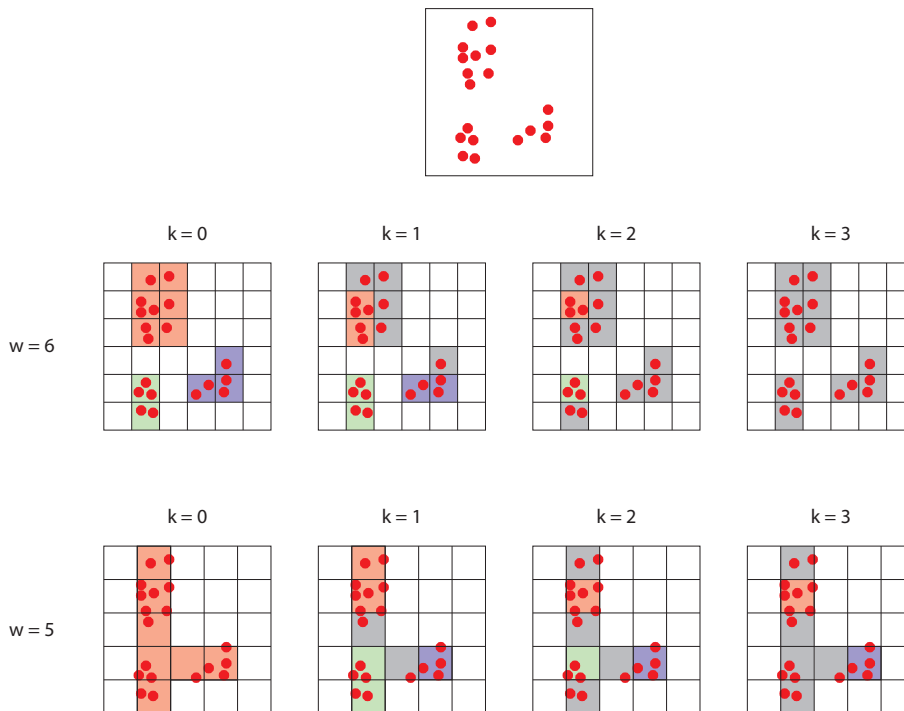


Figure 3.1.1: Visual explanation of how different grid box widths (labeled w) and noise sensitivities (labeled k) produce different clusterings

In the Python implementation of the algorithm, the user assigns k to the `noise_reduction` parameter in `Grid`'s constructor. When points are clustered using the `cluster_points` method, grid boxes with fewer than k points are appended to `Grid`'s `noise_boxes` attribute, rather than to a list in `box_clusters` and to `clustered_boxes`. Likewise, the points in boxes that have been deemed noise are appended to the `noise_points` attribute rather than to a list in `point_clusters` and to `clustered_points`.

3.2 Algorithms for Reclustering Noise

In this section I give three algorithms for reclustering noise points in GBCN. They are all variants on each other, and take inspiration from k -NN. The first two reclustering algorithms, when applied to the data set in Figure 3.1.1, would have successfully reclustered the noise points in the clusterings which had three clusters. The last reclustering algorithm would have consistently done so in the $w = 6$ and $k = 1$ cases, and would sometimes get it right in the other two cases.

3.2.1 Noise Reclustering Algorithm 1: Nearest Neighbors With Euclidean Distance

The first noise reclustering algorithm takes a point and an integer k as input (not the same k as we use for noise reduction), and finds the k points that are closest to the inputted point in terms of Euclidean distance. The cluster whose points appear the most frequently among the k nearest points is the one that the inputted point gets labeled as. Within the implemented code this is handled by `Grid`'s `reassign_noise_with_knn_euclidean_distance` method, which iterates over every point in `noise_points`, and uses the `calculate_cluster_id_with_knn_euclidean` method to find which cluster it belongs to. Each time a point is iterated over, a two element list, with a reference to the point as the first element, and the cluster id as the second element is created and appended to a list of point cluster pairs. This list of point cluster pairs is then iterated over and the `assign_noise_point_to_cluster` method is used to assign the point to the cluster it was determined to belong to.

Initially I intended to find the cluster and assign the point to it in one iteration, but ultimately decided against it, because for nearest neighbors I only wanted to consider points that were originally included in a cluster, not noise points from previous iterations whose cluster had already been determined. This double iteration method insures that the algorithm is deterministic, as the order in which noise points are iterated over has no impact on the result.

The `calculate_cluster_id_with_knn_euclidean` function is a method in the `Grid` class that takes a point and a k value as input and returns the index of a cluster in `point_clusters` that the inputted point should be assigned to. It finds this cluster by iterating over every clustered point, using the `math.dist` function to calculate the Euclidean distance between it and the inputted point. As it iterates it updates a list of the k nearest points. It then finds the cluster whose points are most represented among the k nearest points and returns that clusters index. In the case of a tie, it returns the cluster with the lower index.

The `assign_noise_point_to_cluster` function is a method in the `Grid` class that takes a point and a point cluster id as input and makes all the necessary internal changes so that the inputted point becomes a part of the cluster whose id was inputted. It begins by checking that the inputted point is in `noise_points`, raising an exception if it is not. Then it removes it from `noise_points`, appends it to `clustered_points`, and to `point_clusters[point_cluster_id]`.

Note, the `reassign_noise_with_knn_euclidean_distance` method iterates over every noise point twice – a constant number of times, and for each noise point it iterates over every clustered point once. The complexity of calculating the distance between a noise points and a clustered point is dependent on the dimension of the space that the points exist in. Hence, the time complexity of reclustering noise points using this nearest neighbors with Euclidean distance algorithm is $\Theta(d \cdot n_{\text{noise}} \cdot n_{\text{clustered}})$.

3.2.2 Noise Reclustering Algorithm 2: Nearest Neighbors With Manhattan Distance

This second algorithm for reclustering noise points is similar to the previous one in Subsection 3.2.1, but instead of Euclidean distance, Manhattan distance is used. **Manhattan distance** is defined as the sum of differences between two points along each axis.

In the implemented code, this is done by `Grid`'s `reassign_noise_with_knn_manhattan_distance` method, which utilizes the `calculate_cluster_id_with_knn_manhattan` method. Instead of the `math.dist` function it uses `Grid`'s `manhattan_distance` method, which takes two points as input and sums their differences along each axis. The time complexity of this algorithm is also $\Theta(d \cdot n_{\text{noise}} \cdot n_{\text{clustered}})$.

3.2.3 Noise Reclustering Algorithm 3: Nearest Neighbors With Box Distance

This last reclustering algorithm is similar to the previous two, but instead of analyzing the distance between points, it analyzes the distance between grid boxes. More specifically, **Box distance** is defined as the minimum number of boxes required to traverse (diagonally, or orthogonally) to get from one box to another. In the implemented code, this is done by `Grid`'s `reassign_noise_with_knn_box_distance` method, which iterates over every grid box in `noise_boxes`, and uses the `calculate_cluster_id_with_knn_box_distance` method to find which box cluster it belongs to. Each time a grid box is iterated over, a two element list, with a reference to the box as the first element, and the cluster id as the second element is created and appended to a list of box cluster pairs. This list of box cluster pairs is then iterated over and the `assign_noise_box_to_cluster` method is used to assign the box to the cluster it was determined to belong to.

The `calculate_cluster_id_with_knn_box_distance` function is a method in the `Grid` class that takes a grid box and a k value as input and returns the index of a box cluster in `box_clusters` that the inputted grid box should be assigned to. It finds this cluster by iterating over every clustered grid box, using `Grid`'s `box_distance` method to calculate the box distance between it and the inputted box. As it iterates it updates a list of the k nearest boxes.

It then finds the box cluster whose grid boxes occur most frequently among the k nearest grid boxes and returns that box cluster index. In the case of a tie, it returns the box cluster with the lower index.

The `box_distance` method takes two grid boxes as input and returns the minimum number of grid boxes that must be traversed to get from one to the other. As demonstrated in Figure 3.2.1, for d dimensional boxes \vec{p} and \vec{q} , this number is

$$\max_{0 < i \leq d} (|p_i - q_i|),$$

i.e. the difference between \vec{p} and \vec{q} along the axis where they differ the most.

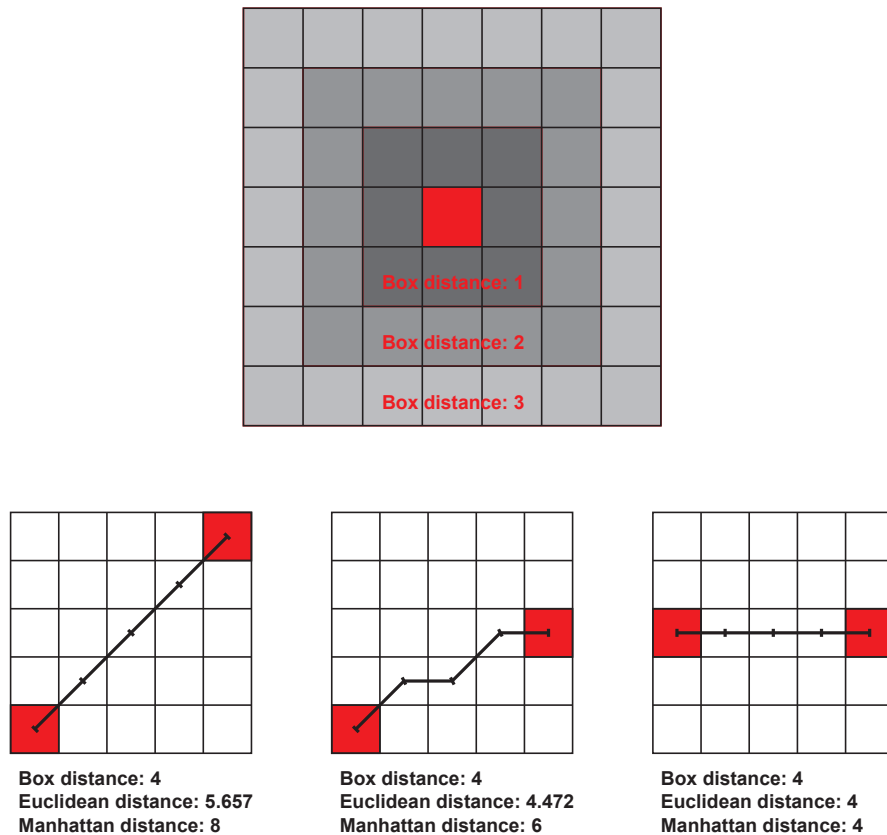


Figure 3.2.1: The upper picture shows three levels of Box distance from a central box. The three lower pictures show segmented paths from one box to another. Each segment increases the box distance by one. Though the examples in the three lower pictures have different Euclidean and Manhattan distances, their Box distances are equal

The `assign_noise_box_to_cluster` function is a method in the `Grid` class that takes a grid box and a box cluster id (`box_cluster_index`) as input and makes all the necessary internal changes so that the inputted grid box becomes a part of the box cluster whose id was inputted, and so that all the points in the inputted box are reassigned to the point cluster corresponding with the inputted box cluster. It begins by checking that the inputted grid box is in `noise_boxes`, raising an exception if it is not. Then it removes it from `noise_boxes`, appends it to `clustered_boxes`, and to `box_clusters[box_cluster_index]`. Lastly it iterates over all points in the inputted grid box, assigning them with the `assign_noise_point_to_cluster` method to the point cluster that corresponds to the inputted box cluster id.

Note, the `reassign_noise_with_knn_box_distance` function iterates over every noise box twice – a constant number of times, and for each noise box it iterates over every clustered box once. The complexity of calculating the distance between noise boxes and clustered boxes is dependent of the dimension of the space that the boxes exist in. Lastly noise points are iterated over and reassigned to point clusters. Hence, the time complexity of this method is $\Theta(d \cdot b_{\text{noise}} \cdot b_{\text{clustered}} + n_{\text{noise}})$. Depending on how many points there are per non-empty grid box, this time complexity can be significantly faster than the previous two algorithms. In the worst case scenario there is a distinct box for every point, so $O(d \cdot n_{\text{noise}} \cdot n_{\text{clustered}})$ is a valid upper bound time complexity.

4

Tuning Hyperparameters

Many machine learning algorithms work by having two algorithms operating in a symbiotic relationship. One generating results, the other evaluating the performance of those results (sometimes referred to as a validation algorithm). The algorithm producing results changes its parameters to improve the score generated by the evaluation metric. To give a less technical analogy, it's like a student looking over the teacher's feedback on the previous test to figure out how to get a higher score on the next test, an algorithm can learn and improve itself by reciprocating to the feedback that it's getting by the evaluation algorithm. GBCN is a result-generating algorithm. The hyperparameters that we are interested in optimizing are the number of partitions along its axes, and the noise reduction coefficient. The evaluating algorithms used in this project are standard ones (perhaps with the exception of DBCV, discussed in Subsection 4.1.4). Their use allows for stability and standardisation in evaluation, as well as easy comparison to other clustering algorithms' performances.

Evaluation is done very differently with supervised and unsupervised machine learning algorithms. With supervised algorithms there are known ground truths, so the performance can be measured very precisely and objectively. The goal is usually to build a model that can correctly classify data where labels are not known. Often the challenge in those cases is to make an algorithm that can fit, but not overfit, to data. With unsupervised algorithms there are no ground

truths so there is a lot more ambiguity. How does one decide if a clustering is good or bad? Often there is no objectively correct answer. Different clusterings will provide different insights into a given data set. Some may be useful. Some may not be. It is important to note that an unsupervised result-generating algorithm can be evaluated by both an **extrinsic evaluation metric** which does not take ground truths into account, as well as by an **intrinsic evaluation metric** – if ground truths are known – which does take ground truths into account. Though it is uncommon for data sets with known ground truths to be clustered by unsupervised algorithms and evaluated by an intrinsic evaluation metric, this method can be useful, as it can provide insights on hyperparameter optimization. Hyperparameter combinations on an extrinsic evaluation metric of labeled data can be exhaustively tested to see what the best possible result that an algorithm can generate is, when given optimal hyperparameters.

Different extrinsic evaluation algorithms will have different ideas about what a good clustering looks like. They typically consider a combination of information when calculating a score, such as how far are clusters' points from the mean of their cluster. How close are points within a cluster to other points of the same cluster, and how far away are they from points in other clusters. Some evaluation algorithms focus on the shape of a clusters, while others may focus on other things, such as the density. Many have biases which could, for example, cause them to favor classifications with many small clusters, or, conversely, classifications with few big clusters. Because of this wide variety, it is important for the data analyst to have some familiarity with their data set, to know what kind of clusters they are looking for, and to know the biases of the evaluation algorithm that they are using.

4.1 Clustering Evaluation Algorithms

4.1.1 Clustering Evaluation Algorithm 1: Silhouette Coefficient

Proposed in 1986 by Peter Rousseeuw [22], the silhouette coefficient is an extrinsic cluster evaluation algorithm that takes a clustering of points in some metric space as input and returns a score between -1 and 1 (inclusive). It positively ranks clusterings where clusters' points

are closer to other points of the same cluster, then to points of the nearest other cluster. If the silhouette score is negative, then the clustering is very poor. A score close to 0 indicates overlapping clusters. Positive silhouette scores are considered better than random, and 0.5 is often used as a threshold above which a clustering is considered strong.

The silhouette coefficient requires that the clustering have at least 2 distinct clusters. It calculates a score for each sample in the data set, and returns the mean of the sample scores as the final silhouette score of the inputted clustering. Let DS be a clustered data set with n_{DS} points. For each point $\vec{p} \in DS$, define $a_{\vec{p}}$ to be the mean distance between \vec{p} and all other points in its cluster, and define $b_{\vec{p}}$ to be the mean distance between \vec{p} and all points in the nearest cluster that \vec{p} is not a part of (nearest, as in, the cluster that results in the lowest $b_{\vec{p}}$ value). Then the silhouette score $s_{\vec{p}}$ of \vec{p} is defined as

$$s_{\vec{p}} = \frac{b_{\vec{p}} - a_{\vec{p}}}{\max a_{\vec{p}}, b_{\vec{p}}}.$$

Thus, the silhouette score s_{DS} of DS is defined as

$$s_{DS} = \frac{\sum_{\vec{p} \in DS} s_{\vec{p}}}{n_{DS}}.$$

Because of its reliance on the mean distance between points within a cluster, the silhouette coefficient will generally give high scores to globular clusters, such as the ones that k -means clustering (explained in Subsection 1.2.1) generates, but may give low scores to good clusterings which are not convex. This algorithm is implemented in the scikit-learn library [17].

4.1.2 Clustering Evaluation Algorithm 2: Calinski Harabasz Index

Proposed in 1974 by Tadeusz Caliński and Jerzy Harabasz [2], the Calinski Harabasz Index, also known as the Variance Ratio Criterion, is an extrinsic cluster evaluation algorithm that takes a clustering of points in some metric space as input and returns a score between 0 and infinity (not inclusive), where the higher the score the better. It positively ranks clusterings that have high dispersion between their clusters and low dispersion within their clusters. More specifically it calculates the score by taking the ratio between the two.

Let DS be a clustered data set with n_{DS} points. Define k to be the number of distinct clusters in DS and define $C = \{C_1, C_2, \dots, C_k\}$ to be the set of clusters in DS , where C_q is the set of all points in cluster q for all $q \in \{1, 2, \dots, k\}$. Furthermore, define n_q to be the number of points in cluster q , define \vec{c}_q to be the center of cluster q (the mean location of all $\vec{p} \in C_q$), and \vec{c}_{DS} to be the center of DS (the mean location of all $\vec{p} \in DS$). Then the value of DS 's within-cluster dispersion W is defined by

$$W = \sum_{q=1}^k \left[\sum_{\vec{p} \in C_q} \|\vec{p} - \vec{c}_q\|^2 \right],$$

i.e. for each point, find the sum of squared differences between its coordinates and the coordinates of the center of its cluster. Then W will be the sum of these values for every point. Next note that the value of DS 's between-cluster dispersion B is defined by

$$B = \sum_{q=1}^k n_q \|\vec{c}_q - \vec{c}_{DS}\|^2,$$

i.e for each cluster center, find the sum of squared differences between its coordinates and the coordinates of the center of the data set and multiply it by the number of clusters. Then B will be the sum of these values for every cluster. The Calinski Harabasz score s_{DS} of DS is defined as

$$s_{DS} = \frac{B(n_{DS} - k)}{W(k - 1)}.$$

Like the silhouette coefficient, Calinski Harabasz index is good at evaluating algorithms that generate globular clusters, but struggles to give good scores to non-convex density based clusterings. This algorithm is implemented in the scikit-learn library [17].

4.1.3 Clustering Evaluation Algorithm 3: Davies-Bouldin Index

Proposed in 1979 by David L. Davies and Donald W. Bouldin [3], the Davies-Bouldin Index is an extrinsic cluster evaluation algorithm that takes a clustering of points in Euclidean space as input and returns a score between 0 and infinity (including 0), where a lower score indicates a better clustering. It computes the score by taking the average similarity of every cluster to its most similar cluster, where similarity is the ration of within-cluster distances to between-cluster

distances. Thus, clusters whose points are relatively close to their cluster mean and far from other clusters means will get good scores.

Let DS be a clustered data set. Define k to be the number of distinct clusters in DS and define $C = \{C_1, C_2, \dots, C_k\}$ to be the set of clusters in DS , where C_q is the set of all points in cluster q for all $q \in \{1, 2, \dots, k\}$. Furthermore, define \vec{c}_q to be the center of cluster q (the mean location of all $\vec{p} \in C_q$), define s_q to be the mean distance between \vec{p}_q and \vec{c}_q for all $p_q \in C_q$, and define d_{ij} to be the Euclidean distance between \vec{c}_i and \vec{c}_j . Next let $R : \{1, 2, \dots, k\}^2 \rightarrow \mathbb{R}$ be a function of similarity between clusters defined by

$$R(i, j) = \frac{s_i + s_j}{d_{ij}}.$$

Then the the Davies-Bouldin score s_{DS} of DS is defined as

$$s_{DS} = \frac{1}{k} \sum_{i=1}^k \max_{j \neq i} R(i, j).$$

As with the previous two evaluation metrics, Davies-Bouldin favors clusterings that contain globular clusters and is not especially useful for evaluating non-convex clusters. This algorithm is implemented in the scikit-learn library [17].

4.1.4 Clustering Evaluation Algorithm 4: DBCV

Proposed in 2014 by Davoud Moulavi, Pablo A. Jaskowiak, Ricardo J. G. B. Campello, Arthur Zimek, and Jörg Sander [15], DBCV (Density-Based Cluster Validation), is an extrinsic cluster evaluation algorithm that takes a distance function d , and a clustering of points in \mathbb{R}^n for some natural n as input and returns a score between -1 and 1 , where the higher the score the better. For each cluster, DBCV finds the area where its internal density is the lowest, and the area between it and another cluster where density is the highest. Clusterings where clusters' lowest internal density is relatively high, and where the highest between cluster densities are relatively low, will perform well.

Let DS be a clustered data set with n_{DS} points. Define k to be the number of distinct clusters in DS and define $C = \{C_1, C_2, \dots, C_k\}$ to be the set of clusters in DS , where C_q is the set of all

points in cluster q for all $q \in \{1, 2, \dots, k\}$. Furthermore, define n_q to be the number of points in cluster q , and define $KNN(\vec{p}, i)$ to be the distance between \vec{p} and its i th nearest neighbor. Let $a_{ptscoredist} : DS \rightarrow \mathbb{R}$ be a function, which takes a point $\vec{p} \in C_q$ as input and returns a value that is inversely related to the point's density in relation to all other $n_q - 1$ points in C_q , be defined by

$$a_{ptscoredist}(\vec{p}) = \left(\frac{\sum_{i=2}^{n_q} \left(\frac{1}{KNN(\vec{p}, i)} \right)^d}{n_q - 1} \right)^{-\frac{1}{d}}.$$

Let $d_{mreach} : DS^2 \rightarrow \mathbb{R}$ be a function, which takes two points as input and returns their mutual reachability distance, be defined by

$$d_{mreach}(\vec{p}_i, \vec{p}_j) = \max \{ a_{ptscoredist}(\vec{p}_i), a_{ptscoredist}(\vec{p}_j), d(\vec{p}_i, \vec{p}_j) \}.$$

Let $G_{MRD}(P)$ be a complete graph with points in any inputted data set P as vertices, and with the mutual reachability distance between the respective pair of points as the weight of each edge. Let $G_{MST}(P)$ be a minimum spanning tree of $G_{MRD}(P)$. Short for Density Sparseness of a Cluster, define $DSC(C_q)$ to be the maximum edge weight of the internal edges of $G_{MST}(C_q)$ for all clusters $C_q \in C$. Short for Density Separation of a Pair of Clusters, define $DSPC(C_i, C_j)$ to be the minimum reachability distance between the internal nodes of $G_{MST}(C_i)$ and $G_{MST}(C_j)$. Next let $s : C \rightarrow \mathbb{R}$ be a function, which gives a score to each cluster that, be defined by

$$s(C_i) = \frac{\min_{1 \leq j \leq k, j \neq i} (DSPC(C_i, C_j)) - DSC(C_i)}{\max \left(\min_{1 \leq j \leq k, j \neq i} (DSPC(C_i, C_j)), DSC(C_i) \right)}.$$

Then the DBCV score s_{DS} of DS is defined as

$$s_{DS} = \sum_{i=1}^{i=k} \frac{|C_i|}{|DS|} s(C_i).$$

This algorithm was implemented by Christopher Jenness, who kindly made it publicly available [12] via his GitHub profile. The most current version contains a bug, but there is an older version from August 25th 2018 which works correctly. Unfortunately this implementation has very slow run times for large data sets. The specifics are shown in Table 4.1.1. These run times

n_{DS}	time	
256	2s	
512	7s	
1024	38s	0.6min
2048	180s	3.0min
4096	1003s	16.7min

Table 4.1.1: DBCV run times for n_{DS} sized data sets with intel(r) core(tm) i7-10510u cpu @ 1.80ghz processor

make DBCV too slow to be used for exhaustive hyperparameter optimization tests. Another Python implementation of this algorithm was created and made publicly available [25] by Felipe Alves Siqueira. This version implements more speed optimization techniques and gives the user the option to trade precision for better processing times. Unfortunately I was not able to get it to run. Siqueira’s implementation is based on the original authors’ code [10], which was written in MATLAB.

4.1.5 Clustering Evaluation Algorithm 5: Rand Index

Proposed in 1985 by Lawrence Hubert and Phipps Arabie [7], Rand Index is a intrinsic cluster evaluation algorithm that takes a list of true labels and a list of predicted labels as input and returns a score between 0 and 1 (inclusive), where higher scores indicate more similarity between the two lists. It treats label permutations as equivalent, so predicted labels $[0, 0, 0, 1, 1]$ and $[1, 1, 1, 0, 0]$ would both receive the same score.

Let DS be a data set with n_{DS} points. Let $T = \{T_1, T_2, \dots, T_n\}$ be the set of ground truth clusters in DS , where T_q is the set of all points in the ground truth cluster q for all $q \in \{1, 2, \dots, n\}$. Let $P = \{P_1, P_2, \dots, P_m\}$ be the set of predicted clusters in DS , where P_q is the set of all points in the predicted cluster q for all $q \in \{1, 2, \dots, m\}$.

Define a to be the number of point pairs \vec{p}_i, \vec{p}_j such that $\vec{p}_i, \vec{p}_j \in T_{q_1}$ and $\vec{p}_i, \vec{p}_j \in P_{q_2}$ for some $q_1, q_2 \in \mathbb{N}$. Define b to be the number of point pairs \vec{p}_i, \vec{p}_j such that $\vec{p}_i \in T_{q_1}$ and $\vec{p}_j \notin T_{q_1}$ hold, and $\vec{p}_i \in P_{q_2}$ and $\vec{p}_j \notin P_{q_2}$ hold for some $q_1, q_2 \in \mathbb{N}$. Lastly define c to be the total number of

possible pairs of points in DS . Then

$$c = \binom{n_{DS}}{w} = \frac{n_{DS}!}{2! \cdot (n_{DS} - 2)!} = \frac{n_{DS} \cdot (n_{DS} - 1) \cdot (n_{DS} - 2)!}{2 \cdot (n_{DS} - 2)!} = \frac{(n_{DS})^2 - n_{DS}}{2}.$$

It follows that the Rand score s_{DS} of DS is defined as

$$s_{DS} = \frac{a + b}{c}.$$

Note that if we think of a as being the number of true positives, and think of b as being the number of true negatives, then it follows that c is the number of true positives, plus true negatives, plus false positives, plus false negatives. Then the definition of rand score,

$$s_{DS} = \frac{a + b}{c} = \frac{TP + TN}{TP + TN + FP + FN},$$

is equivalent to the definition of the accuracy evaluation metric. Therefore, the key insight that Rand Index makes is its definition of what a true positive and what a true negative look like in a cluster. This algorithm is implemented in the scikit-learn library [17].

4.2 Using Evaluation Metrics to Optimize Hyperparameters

GBCN is highly sensitive to hyperparameters. As seen previously in Figure 3.1.1, small changes to the diameters of grid boxes and to noise sensitivity can make a big difference in the way the data points get clustered. Each clustering that was generated in that figure would receive a different score when scored by an evaluation algorithm. When the right evaluation algorithm is chosen, the most desirable clustering should receive the highest score. In addition to quantifying how good a clustering is, an evaluation metric can serve as a heuristic device to compare the performances of different hyperparameter values. A method then for finding good clusterings is to take different hyperparameter values to generate different clusterings, and choosing the one that ranks higher than all others.

There are currently four functions implemented in my code that do exhaustive hyperparameter evaluations:

```
optimize_parameters_for_extrinsic_metric
```

```
optimize_parameters_for_extrinsic_metric_one_width
```

```
optimize_parameters_for_intrinsic_metric_one_width
```

```
optimize_parameters_for_DBCV_one_width
```

They all take as input a data set, its dimension, a minimum grid box width, a maximum grid box width, a minimum noise reduction, a maximum noise reduction, and a figure title. In addition to that, the first three take an evaluation metric as input, and `optimize_parameters_for_intrinsic_metric_one_width` takes true labels as input. `optimize_parameters_for_DBCV_one_width` does not take an evaluation metric, because it was uniquely built to calculate the score of a clustering using DBCV, which, you will recall from Chapter 4.1.4, is not implemented in any standard machine learning libraries. A noteworthy difference between `optimize_parameters_for_extrinsic_metric` and the other three functions is that the former computes every possible combination of grid box widths within the inputted range, of which there are $(w_{\max} - w_{\min})^d$, while the latter three simply iterate from the inputted minimum grid box width till the maximum grid box width, using each number in between as the amount of partitions on each axis for a total of $w_{\max} - w_{\min}$ width hyperparameter combinations. For each width value that the functions test, they test every noise sensitivity within the inputted range. Therefore, though more thorough, `optimize_parameters_for_extrinsic_metric` performs $\Theta\left((w_{\max} - w_{\min})^d (nr_{\max} - nr_{\min})\right)$ clusterings, while the other three are exponentially faster, performing just $\Theta((w_{\max} - w_{\min})(nr_{\max} - nr_{\min}))$ clusterings. If the inputted data set was two dimensional, then the functions produce a figure of the best ranked clustering that they found.

5

Results

In this chapter I evaluate the performance of GBCN. I compare it to k -means clustering and DBSCAN. I run it on synthetic data sets from the scikit-learn package [17], as well as a real world data set from the UCI Machine Learning Repository [16]. I use optimization techniques to find good hyperparameter values.

5.1 Experiments With GBCN on Synthetic Data Sets

I used the following three lines of code to make my first three synthetic data sets from scikit-learn

```
noisy_circles = datasets.make_circles(n_samples=1500, factor=0.4, noise=0.05)
noisy_moons = datasets.make_moons(n_samples=1500, noise=0.05)
blobs = datasets.make_blobs(n_samples=1500, random_state=30)
```

They are two dimensional and a plot of them can be seen in Figure 5.1.1.

With the width hyperparameter set to 25 and the noise reduction hyperparameter set to 0, GBCN gives a clustering of `noisy_circles` that separates the two ring shaped groups of points. I consider this clustering to be the one that best reflects the shape of the data. See a plot of it in Figure 5.1.2. The silhouette, Calinski Harabasz, Davies-Bouldin, and DBCV scores of this clustering are documented in Table 5.1.1 and can also be seen in the figure. Note that the distance between the means of the two clusters is extremely small, and thus it makes sense that

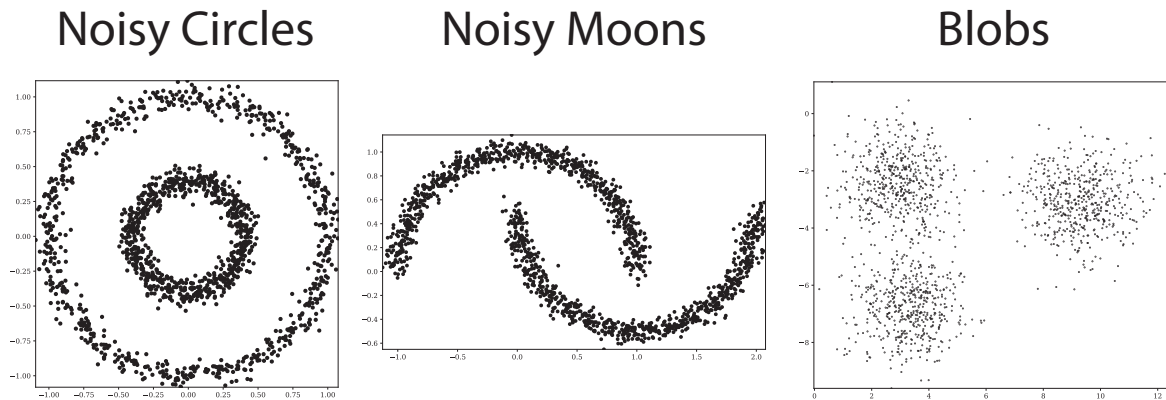


Figure 5.1.1: `noisy_circles`, `noisy_moons`, and `blobs` data set generated by the scikit-learn library

the Davies-Bouldin, and Calinski Harabasz scores are so low. The silhouette score does not take into account the mean location of clusters, and thus its score, while not great, is a bit better. The DBCV score on the other hand gives the clustering a very positive ranking. This makes sense given that the density of points at any given area within a cluster is relatively high, and given that the point density of areas between clusters is very low.

With the width hyperparameter set to $[15, 25]$ and the noise reduction hyperparameter set to 0, GBCN gives a clustering of `noisy_moons` that separates the two moon shaped groups of points. In this case too I consider this clustering to be the one that best describes shape of the data. See a plot of this clustering in Figure 5.1.3. We can see in Table 5.1.1 that the evaluation metric scores of `noisy_moons` are universally better than those of `noisy_circles`. While the Calinski Harabasz and Davies-Bouldin scores in `noisy_circles` suggested that the clustering was bad, the scores in `noisy_moons` are within a range that deems them suitable to serve as heuristics, i.e. alone it is difficult to tell how good they deem the clustering, but their comparison to scores of other clusterings could be used to find the better clustering. `noisy_moons`'s silhouette score suggests that the clustering is better than random, but not much more than that. In contrast, the DBCV score is strong.

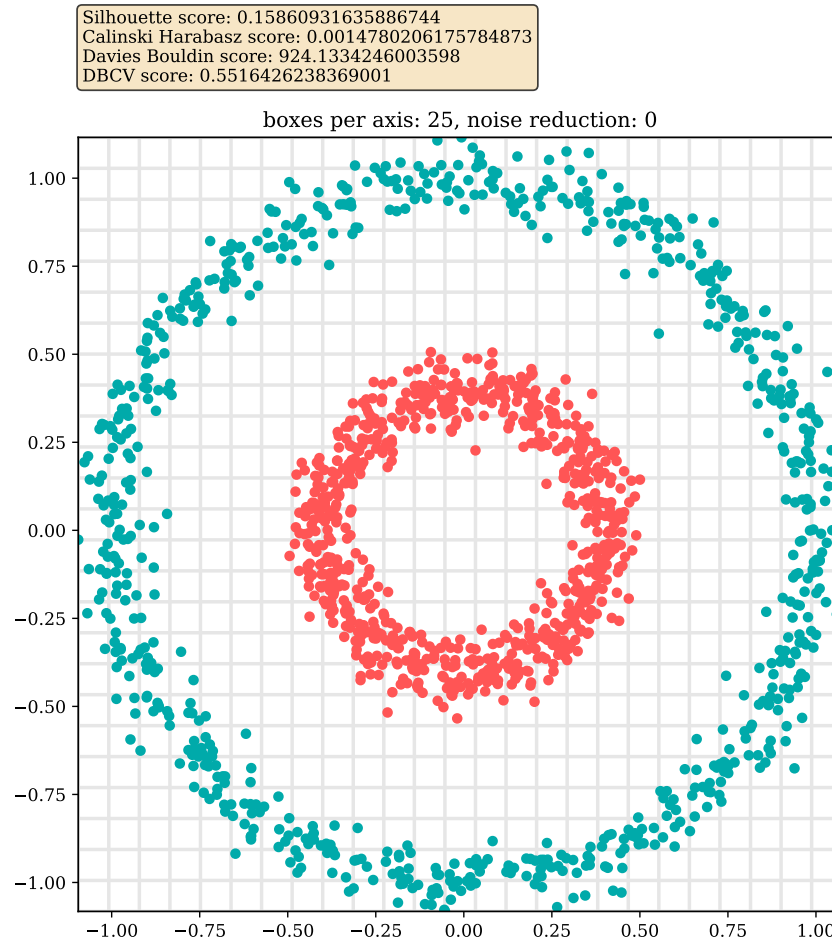


Figure 5.1.2: Applying GBCN to `noisy_circles`. Each axis is partitioned 25 times to create the grid lines marked in light gray. This choice of hyperparameters allows GBCN to generate a clustering that accurately reflects the shape of the data

When noise reduction and width are both set to 15, applying GBCN to `blobs` produces the clustering seen in Figure 5.1.4. Running

```
reassign_noise_with_knn_euclidean(1)
```

subsequently produces the clustering seen in Figure 5.1.5. I consider this to be one of the clusterings that best reflects the shape of the data. I say ‘one of’ because there is some ambiguity about the correct clustering of the points in the region where the blue and green clusters overlap. Using other (see Section 3.2) reassignment methods with a variety of k values produced very similar clusterings, with very similar scores. Some differed slightly in how they reassigned points in the intersection of the overlapping clusters, yet they were similar enough that I would consider

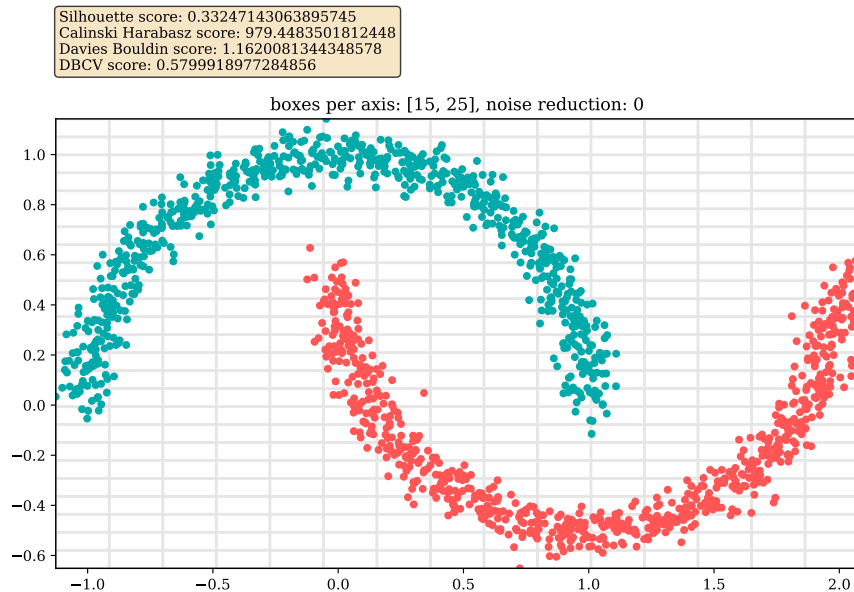


Figure 5.1.3: Applying GBCN to `noisy_moons`. The horizontal axis is partitioned 15 times and the vertical axis is partitioned 25 times to create the grid lines marked in light gray. This choice of hyperparameters allows GBCN to generate a clustering that accurately reflects the shape of the data

them all to be among the clusterings that best reflect the shape of the data. The evaluation metric scores (see Table 5.1.1) of this clustering are in stark contrast with the scores of the previous two clusterings. This is not too surprising, as the `blobs` data set is globular, and we see that the evaluators that reward globular clustering gave it significantly higher scores than they gave the previous data sets. This result shows that GBCN is capable of generating globular clusterings, and suggests that there are evaluation metrics which could be used to help find the hyperparameters that enable it to do so.

	Silhouette	Calinski Harabasz	Davies-Bouldin	DBCV
<code>noisy_circles</code>	0.158609	0.001478	924.133425	0.551643
<code>noisy_moons</code>	0.332471	979.448350	1.162008	0.579992
<code>blobs</code>	0.645515	4736.678512	0.497095	-0.668201

Table 5.1.1: Evaluation scores of the clusterings of the `noisy_circles`, `noisy_moons`, and `blobs` data sets seen in Figure 5.1.2, Figure 5.1.3, and Figure 5.1.5

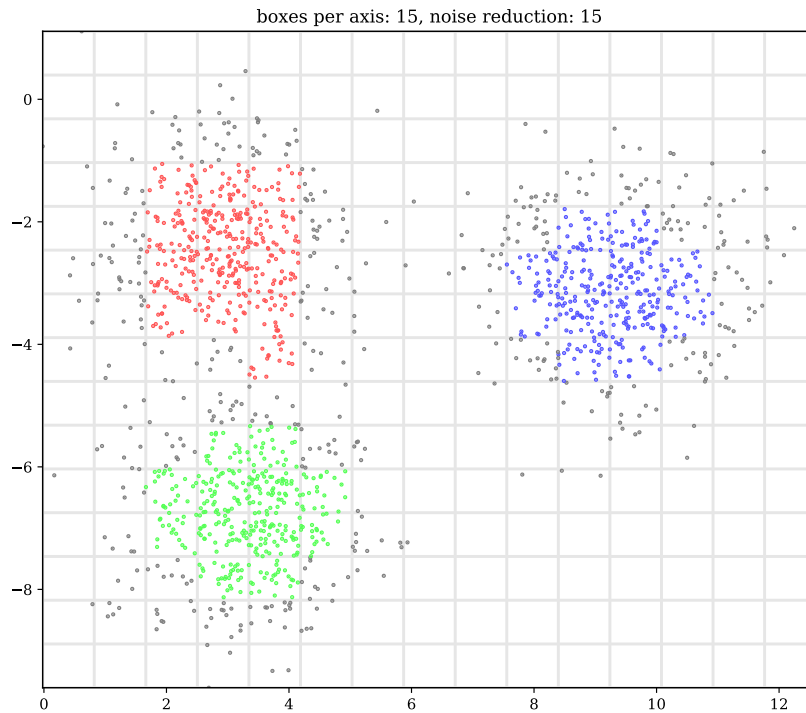


Figure 5.1.4: Applying GBCN to blobs. Each axis is partitioned 15 times, to create the grid lines marked in light gray. Noise sensitivity was set to be such that boxes with 15 or fewer points were labeled as noise. This combination of grid and noise sensitivity was able to separate clusters whose edges overlapped, while preserving each cluster’s dense central area

5.2 Experiments With k -Means Clustering and DBSCAN on Synthetic Data Sets

In this section I use k -means clustering and DBSCAN to cluster the synthetic data sets and measure their performance using the evaluation metrics. The effectiveness of these algorithms can serve as a benchmark to compare GBCN to. The clusterings produced by k -means are shown in Figure 5.2.1. The clusterings produced by DBSCAN are shown in Figure 5.2.2. The evaluation scores of these clusterings are documented in Table 5.2.1 and can also be seen in the figures.

The clusterings on the `noisy_circles` and `noisy_moons` data sets produced by k -means clustering received substantially better silhouette, Calinski Harabasz, and Davies-Bouldin scores

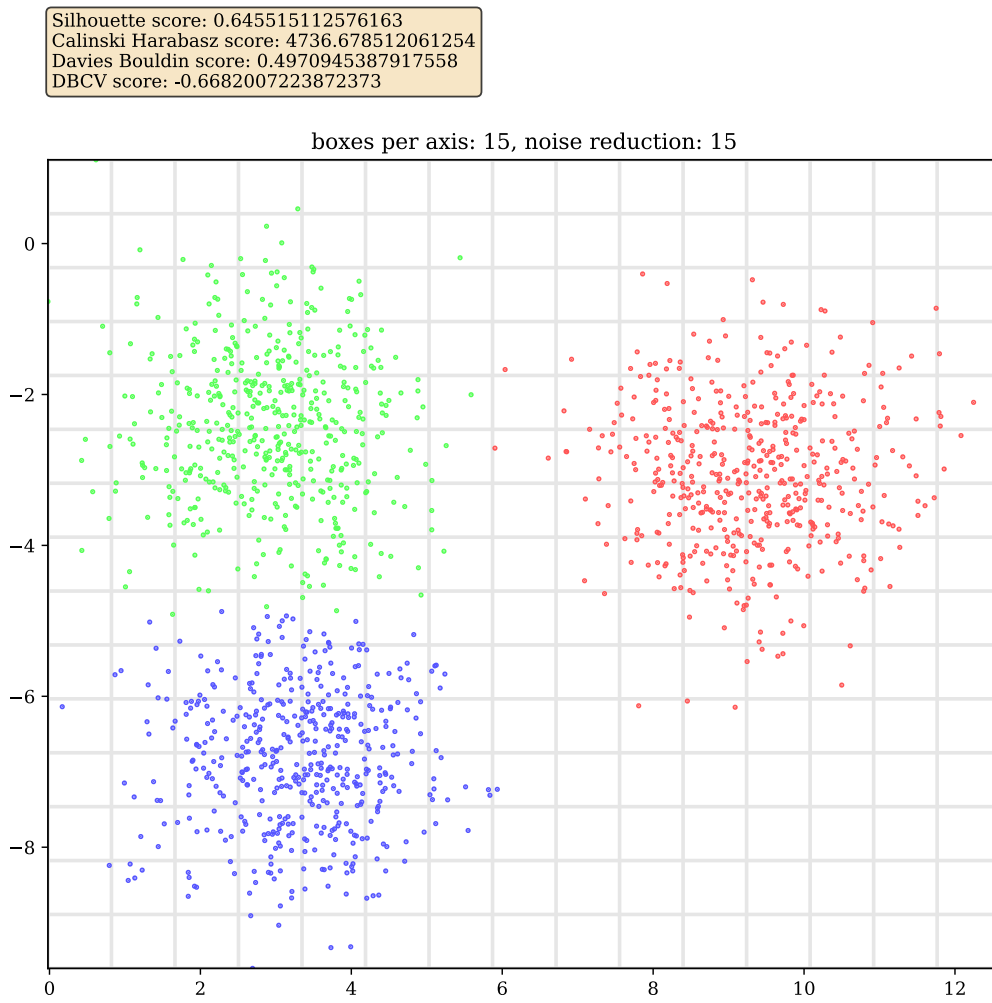


Figure 5.1.5: This clustering demonstrates how noise reclustering can be used to produce clusterings that accurately reflect the shape of data

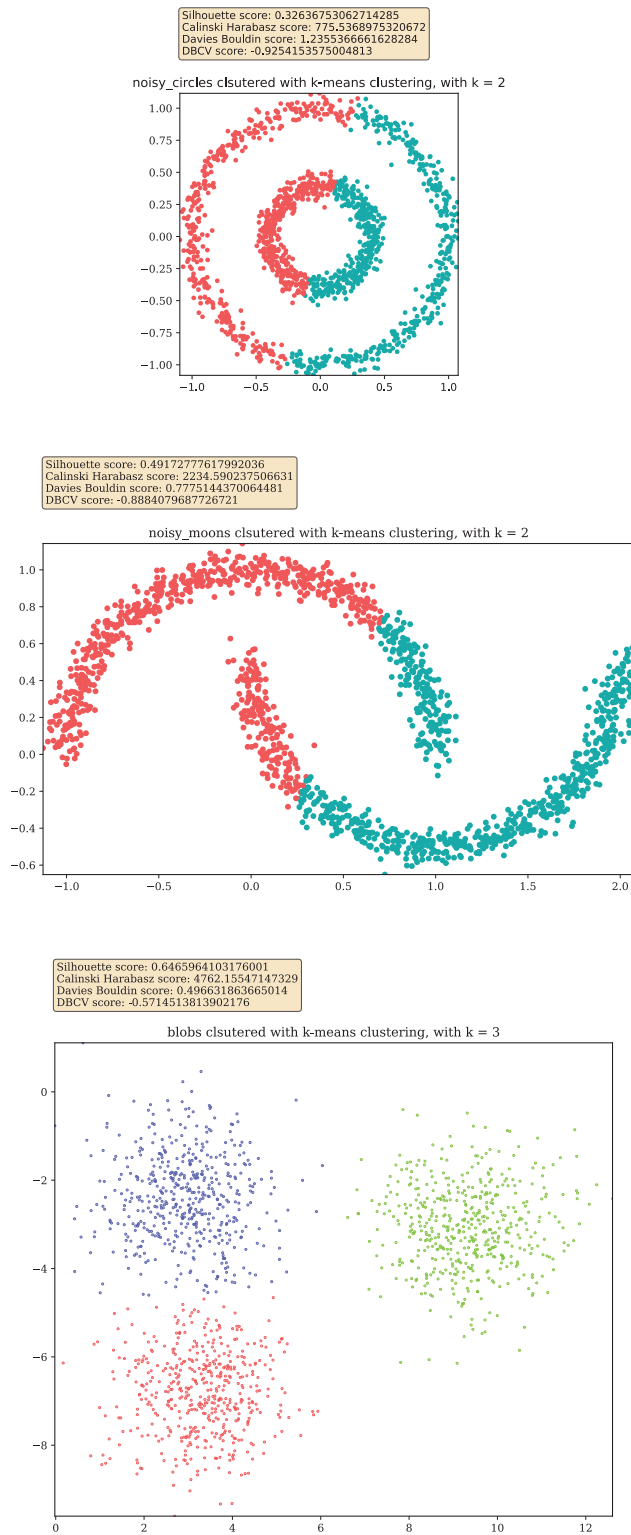


Figure 5.2.1: Clusterings of the synthetic data sets produced by k -means clustering

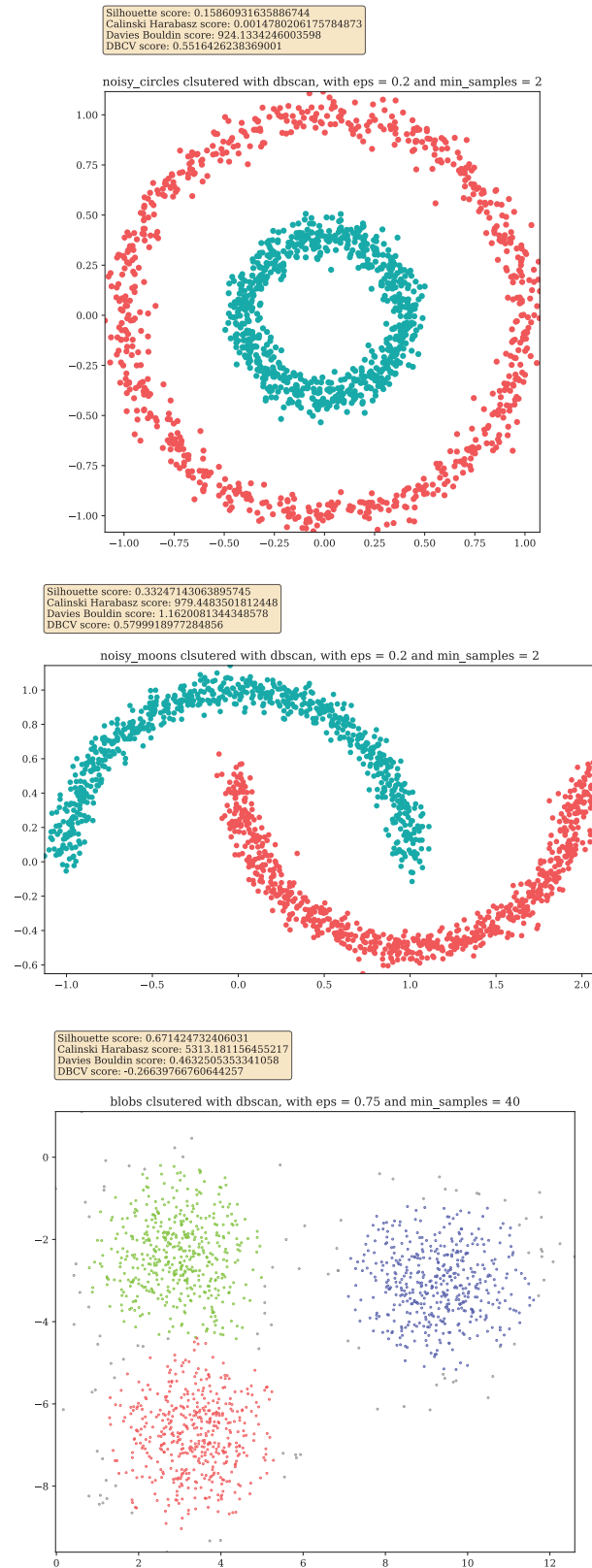


Figure 5.2.2: Clusterings of the synthetic data sets produced by DBSCAN

	Algorithm	Silhouette	Calinski Harabasz	Davies-Bouldin	DBCV
<code>noisy_circles</code>	GBCN	0.158609	0.001478	924.133425	0.551643
<code>noisy_moons</code>	GBCN	0.332471	979.448350	1.162008	0.579992
<code>blobs</code>	GBCN	0.645515	4736.678512	0.497095	-0.668201
<code>noisy_circles</code>	k -means	0.326368	775.536898	1.235537	-0.925415
<code>noisy_moons</code>	k -means	0.491728	2234.590238	0.777514	-0.888408
<code>blobs</code>	k -means	0.646596	4762.155471	0.496639	-0.571451
<code>noisy_circles</code>	DBSCAN	0.158609	0.001478	924.133425	0.551643
<code>noisy_moons</code>	DBSCAN	0.332471	979.448350	1.162008	0.579992
<code>blobs</code>	DBSCAN	0.671425	5313.181156	0.463251	-0.266398

Table 5.2.1: Evaluation metric scores of synthetic data clusterings using GBCN, k -means clustering, and DBSCAN

than the ones produced by GBCN. Yet we see that these clusterings do not provide any useful insights into the shape of the data. Hence, the validity of these metrics' scores, on non-globular shaped data sets, should be viewed skeptically. DBCV, on the other hand, correctly gave the clusterings very low scores.

The clusterings on the `noisy_circles` and `noisy_moons` data sets produced by DBSCAN were the same as the ones produced by GBCN, and thus received the same scores. The best hyperparameters I could find for clustering `blobs` with DBSCAN, while attempting to classify all points into three clusters, labeled most points along the edges of clusters as noise. When calculating the scores for this clustering, I did not include the noise points. Because of this its scores will be higher than if noise was reclustered, or if noise points were counted as their own cluster.

GBCN, k -means, and DBSCAN all produced similar clusterings of `blobs`, and GBCN and DBSCAN outperformed k -means clustering on the other two data sets. The effectiveness of GBCN and DBSCAN was similar, but unlike DBSCAN, GBCN has built in noise reclustering algorithms enabling it to produce accurate clusterings of `blobs`, which classify all points. Also note from the original paper [4] that the time complexity of DBSCAN is $O(n \cdot \log(n))$ on average, but $O(n^2)$ in the worst case. In Section 6.3 I prove that the time complexity of GBCN can be reduced to $O(nd + b \log(b) + b^{1+\log_x(3)})$ for some real $x \geq 3$, which is faster on average when $d < \log(n)$, and which is faster in the worst case when $d < n$.

Note that GBCN’s clusterings of `noisy_circles` and `noisy_moons` accurately reflect the shape of the data, and that the DBCV evaluator gave them both an unambiguously positive score. Also note that the clusterings of `noisy_circles` and `noisy_moons` that k -means produced did not accurately reflect the shape of the data, and that the DBCV evaluator gave these clusterings very low scores. This serves as strong evidence that DBCV can accurately evaluate density-based clusters, and suggests that the combination of GBCN and DBCV has a lot of potential for finding density based clusters. This may be an area worth exploring further.

5.3 Using Evaluation Metrics to Optimize Synthetic Data Clustering

In this section I describe an experiment I conducted using a hyperparameter optimization functions discussed in Section 4.2. I ran

```
optimize_parameters_for_extrinsic_metric_one_width
```

three times for each of the three synthetic data sets, optimizing for silhouette coefficient, Calinski Harabasz index, and for Davies-Bouldin index. I set the minimum width to be 1, the maximum width to be 100, the minimum noise reduction to be 0, and the maximum noise reduction to be 10. For noise reclustering I used

```
reassign_noise_with_knn_euclidean(1)
```

The results of this hyperparameter search is in Table 5.3.1.

	Evaluation Metric	Width	Noise Red.	Description
<code>noisy_circles</code>	Silhouette	46	4	18 clusters, far from optimal
<code>noisy_circles</code>	Calinski Harabasz	72	2	50+ clusters, far from optimal
<code>noisy_circles</code>	Davies-Bouldin	23	6	16 clusters, far from optimal
<code>noisy_moons</code>	Silhouette	52	8	2 clusters, good but not optimal
<code>noisy_moons</code>	Calinski Harabasz	51	4	35 clusters, far from optimal
<code>noisy_moons</code>	Davies-Bouldin	76	4	13 clusters, far from optimal
<code>blobs</code>	Silhouette	26	8	optimal
<code>blobs</code>	Calinski Harabasz	26	8	optimal
<code>blobs</code>	Davies-Bouldin	59	0	3 big, many small clusters, good

Table 5.3.1: Using evaluation metrics to find hyperparameters of synthetic data sets. Optimal in this case means that the clustering accurately described the shape of the data

The silhouette, Calinski Harabasz, and Davies-Bouldin metrics did not prove useful for optimizing GBCN hyperparameters to cluster `noisy_circles` or `noisy_moons`. The only arguably good evaluator was silhouette score for `noisy_moons`. It set width and noise reduction to be very large values, causing the majority of points to be classified as noise, leaving just two small clusters for noise to be reclustered to. The noise points were then reclustered, for the most part in accordance with the clustering seen in Figure 5.1.3. The exception to this were the interior tails of the moons, which were classified as being part of the other moon. In Table 5.3.1 I call this clustering good because it has the correct amount of clusters, and about 75% of the points were labeled correctly. None the less, this seems more like a fluke, then a reproducible phenomenon.

We can see in Table 5.3.1 that the silhouette, Calinski Harabasz, and Davies-Bouldin metrics proved very useful when trying to optimize GBCN hyperparameters for clustering `blobs`. Two of them were able to create clusterings that accurately reflected the shape of the data, and the third, Davies-Bouldin, was able to make a reasonably good clustering in which the three clusters were correctly identified, but where the points along the edges of blobs were often excluded and formed their own little clusters. This may reflect a bias for larger quantities of small clusters in the Davies-Bouldin evaluator.

5.4 Using Evaluation Metrics to Optimize Real World Data Clustering

I use the following lines of code to extract the Iris [5] data set from the UC Irvine Machine Learning Repository

```
from ucimlrepo import fetch_ucirepo
iris = fetch_ucirepo(id=53)
```

Iris is a famous machine learning data set containing 150 data points, each representing an iris flower. The labels of data points are their species. There are three different species, which each appear 50 times. The data is 4-dimensional, where sepal length, sepal width, petal length, and petal width are the axis. A visualization of this data set can be seen in Figure 5.4.1 by [20].

I begin the clustering by running

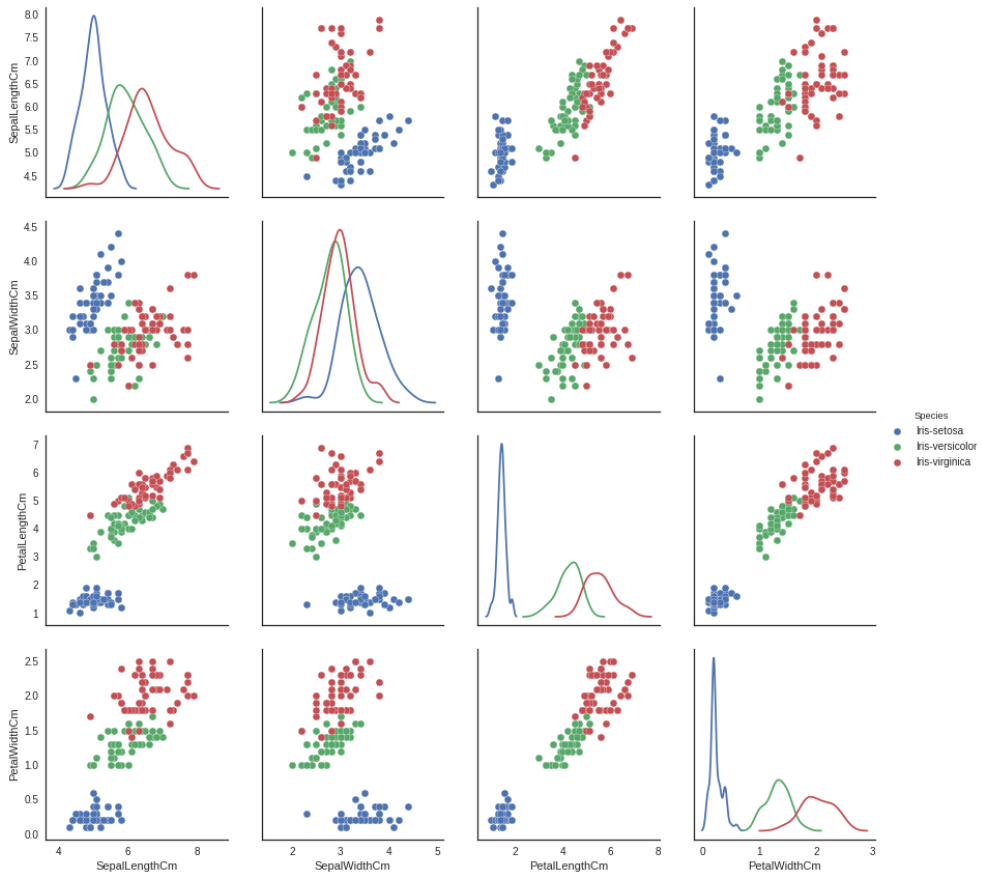


Figure 5.4.1: Visualization of Iris data by [20]

```
optimize_parameters_for_extrinsic_metric_one_width
```

on the Iris data three times, optimizing for silhouette coefficient, Calinski Harabasz index, and Davies-Bouldin index. The width range is from 1 to 50, and the noise reduction range is from 0 to 5. I subsequently run

```
optimize_parameters_for_intrinsic_metric_one_width
```

on the Iris data, optimizing for Rand Index. The results of these hyperparameter searches are documented in Table 5.4.1. We see there that when optimizing for silhouette score, GBCN was

	Width	Noise Red.	Description
Silhouette	4	4	makes 2 clusters, separates setosa species
Calinski Harabasz	50	0	separates points into 143 clusters
Davies-Bouldin	50	0	separates points into 143 clusters
Rand	10	1	0.9 score, 5 near homogeneous clusters

Table 5.4.1: Using evaluation metrics to find hyperparameters of Iris data set

able to separate the Iris setosa species, marked in blue in Figure 5.4.1, from the versicolor and virginica species, which it put into one cluster. Optimizing for the Calinski Harabasz and Davies-Bouldin evaluation metrics failed to generate any meaningful hyperparameter values. The use of the intrinsic Rand Index metric produced a very good clustering, proving that GBCN can cluster the Iris data when the right hyperparameters are given. A detailed overview of the clustering produced when optimizing for Rand is given in Table 5.4.2.

cluster index	setosa	versicolor	virginica	$\frac{T}{T+F}$
0	50	0	0	1
1	0	41	1	$\frac{41}{42} \approx 0.98$
2	0	5	38	$\frac{38}{43} \approx 0.88$
3	0	4	0	1
4	0	0	11	1

Table 5.4.2: The near homogeneous clustering of Iris data that GBCN produces when its width is equal to 10 and its noise reduction is equal to 1

In conclusion, I have shown here that GBCN is an effective clustering algorithm, capable of classifying both density-based and globular clusters when given the right hyperparameters. I have presented evaluation metrics and algorithms to assist the user in discovering useful hyperpa-

parameter values. GBCN has demonstrated some early success and a lot of potential in classifying both synthetic and real world data sets, encouraging further exploration. GBCN struggles with high dimensional data (read Chapter 6.3 to see this can be improved), but is highly efficient on low dimensional data sets. The key areas of further research in the field that I see are: improving and inventing new evaluation metrics for density-based notions of clusters, and discovering efficient and effective parameter tuning techniques for discrete and non-differentiable parameters.

6

Ideas Worth Exploring Further

This project has been a tremendous learning experience and has opened many new doors of inquiry. I would be thrilled to see these ideas analyzed and explored further and I encourage curious readers to reach out to me for collaboration. This section of the thesis is dedicated to analyzing areas of further exploration, what could have been done differently in my implementation of GBCN.

6.1 Adding Sophistication to hyperparameter optimization

Currently, the `optimize_parameters_for_extrinsic_metrics` function runs prohibitively slowly for large width and noise reduction ranges. The set of possible hyperparameter combinations is very large and there may be more efficient methods for approximating them than the currently implemented brute force methods.

An easy next step would be to start by trying a few hyperparameter value combinations that are evenly spread out across the space of possibilities. Areas around the best performing areas could then be recursively explored further. The amount of possible hyperparameter values would be decreased to a fraction of itself with each recursion, causing a hyperparameter solution to be found in logarithmic time.

The next possible avenue of exploration takes inspiration from how neural networks and how they optimize their parameters. In the case of GBCN, the hyperparameters are discrete, so

derivatives can not be taken over them. It may be possible though to reclaim the idea of moving in the direction of the steepest slope. We could have a starting set of hyperparameters and compute the evaluation metric scores that result from moving them in various directions. After exploring a few of them we move in the direction that results in the greatest increase to the evaluation score.

6.2 Changing definition of neighboring grids

Currently, two grid boxes are considered neighboring if all of their coordinates differ by no more than 1. An additional neighbor sensitivity hyperparameter ns could be added that would require that two boxes $\vec{p} = (p_1, p_2, \dots, p_d)$ and $\vec{q} = (q_1, q_2, \dots, q_d)$ have $p_i = q_i$ for at least ns distinct values $i \in \{1, 2, \dots, d\}$ for them to be considered neighbors.

As proven in Theorem 2.3.2, the maximum Euclidean distance that two neighboring points can have is

$$2 \cdot \sqrt{\sum_{i=1}^d \left(\frac{\max_i - \min_i}{w_i} \right)^2}.$$

Adding an $ns \geq 1$ value would reduce this maximum distance, but would also make it so that two points could in theory have distance ε for any $\varepsilon > 0$ and not be classified as neighbors. This would make the criteria for creating a cluster less sensitive and therefore could help with data sets where the Box Clustering Algorithm has trouble separating distinct clusters.

Lastly, changing the definition of neighboring boxes to be less sensitive could help with processing time. Currently, a d dimensional box will have $3^d - 1$ neighboring boxes, which is prohibitively large even for relatively low dimensions, so an enhancement to the current definition of neighboring could significantly improve the algorithm.

6.3 Optimization for High Dimensional Data Sets

The current implementation of GBCN has shown promising results on low dimensional data sets, but its large time and memory cost for high dimensional data has proven prohibitive. I

propose here an alternative way to implement the algorithm to significantly reduce its time and space complexity on high dimensional data sets.

Start by making an empty set `boxes` and a hash table `point_to_grid_box`. Iterate over all points in the data set. For each point calculate its grid box and add it to `boxes` and update `point_to_grid_box` so that inputting the point as a key returns the grid box as a value. This will take $\Theta(dn)$ time. Convert the `boxes` set to a list. Sort the grid boxes in `boxes` by the value of their first coordinate. Among boxes whose first i coordinates are the same, the $i + 1$ th coordinate will determine their order. This can be done in $O(b \log(b))$ time, where b is the length of `boxes`. Next, make a hash table `grid_box_to_point` where every box in `boxes` is a key, and its corresponding value is initially an empty set. Iterate over every point. For each one find its box in constant time using the `point_to_grid_box` hash table and do `grid_box_to_points[box].add(point)`. This will take $\Theta(n)$ time.

The data set, the `boxes` list, the `point_to_grid_box` hash table, and the `grid_box_to_points` hash table are all we need to have a usable grid. Hence the `Grid` class can be optimized to not have its d dimensional list `grid`. Thus a grid can theoretically be constructed in $O(nd + b \log(b))$ time, vastly improving the currently implemented algorithm, whose time and space complexity we proved is

$$\Theta\left(\prod_{i=1}^d w_i\right).$$

Note, the data set has space complexity $\Theta(n)$, the `boxes` list has space complexity $\Theta(b)$, the `point_to_grid_box` hash table has space complexity $\Theta(n)$, and the `grid_box_to_points` hash table has space complexity $\Theta(n + b)$, so the amount of space being used would be some small multiple of n plus some small multiple of b .

Finding grid boxes' non-empty neighbors can also be prohibitively expensive for large dimensions, but once again a faster algorithm can be constructed. To find the neighbors of some grid box $\vec{b} = (b_1, b_2, \dots, b_d)$, search through `boxes` to find the following six indexes: the first and last grid box whose first coordinate is $b_1 - 1$, the first and last grid box whose first coordinate is b_1 , and the first and last grid box whose first coordinate is $b_1 + 1$. Note, if it exists, then the

first grid box whose first coordinate is b_1 will have the same index – but incremented by one – as the last grid box whose first coordinate was $b_1 - 1$. The same holds for the last b_1 and the first $b_1 + 1$. Hence, only 4 searches have to be done, and since `boxes` is sorted, each search will take $O(\log(b))$ time. Next make three sublists, each containing grid boxes starting with $b_1 - 1$, b_1 , and $b_1 + 1$ respectively. Within each sublist repeat the process, searching for the indexes of boxes whose second index is the first or last instance of $b_2 - 1$, b_2 , or $b_2 + 1$. Repeat this process all the way down to b_d . The grid boxes you will be left with are all the grid boxes in `boxes` that are neighboring \vec{b} .

This neighbor finding algorithm is recursive, and can be described by the following recurrence relation

$$T(b) = 3T\left(\frac{b}{x}\right) + 4 \cdot \log_2(b),$$

where for each recursion x is defined such that $\frac{b}{x}$ will be the number of grid boxes that get passed down to one of the next recursions. Let $f : \mathbb{R} \rightarrow \mathbb{R}$ be defined by $f(b) = 4 \cdot \log_2(b)$. Assume $x \geq 3$. Given that 3 disjoint sublists are extracted from b grid boxes, and given that in general there will be grid boxes that are not included in any sublists, this is a fair assumption for us to make. Let $\varepsilon \in (0, 3)$. Let $y = \log_x(3 - \varepsilon)$. Note $y \in (0, 1)$. Then using L'Hôpital's rule we see that

$$\begin{aligned} \lim_{b \rightarrow \infty} \frac{\log_2(b)}{b^y} &= \lim_{b \rightarrow \infty} \frac{\frac{d}{db} [\log_2(b)]}{\frac{d}{db} [b^y]} \\ &= \lim_{b \rightarrow \infty} \frac{\frac{1}{\ln(2)b}}{y b^{y-1}} \\ &= \lim_{b \rightarrow \infty} \frac{1}{\ln(2)b^y b^{1-y}} \cdot \frac{b^{1-y}}{y} \\ &= \lim_{b \rightarrow \infty} \frac{1}{\ln(2)b^y y} \\ &= 0. \end{aligned}$$

Hence, we've shown that $\frac{\log_2(b)}{b^y}$ converges to 0, so it must be the case that there is some $B \in \mathbb{R}$ such that if $b \geq B$, then $\frac{\log_2(b)}{b^y} < 1$, and therefore $\log_2(b) < b^y$, which implies that $f(b) =$

$4 \log_2(b) < 4b^y = 4b^{\log_x(3-\epsilon)}$. Hence $f(b) = O(b^{\log_x(3-\epsilon)})$, which means, by the Master theorem [26], that the time complexity of finding a box's neighbor is $\Theta(b^{\log_x(3)})$ for some $x \geq 3$.

To pay part of the box clustering processing time cost upfront, make a hash table called `grid_box_to_neighbors`. You could set the hash table to have b slots and make its hash function return a grid box's index in `boxes`. This would guarantee that it has no collisions, eliminating the rare possibility of $O(n)$ look up times. The drawback is that finding a grid box's index in the sorted `boxes` list would take $O(\log(b))$ time, rather than the constant $O(1)$ time it takes a default hash function to compute a hash code. Once you've made your hash table, iterate through `boxes`. For each box calculate its neighbors and do `grid_box_to_neighbors[box] = neighbors`. This will take $\Theta(b^{1+\log_x(3)})$ time. Further optimizations that take advantage of knowing the neighbors of previously iterated boxes may also be worth exploring further.

Bibliography

- [1] Marcelo Beckmann, Nelson F. F. Ebecken, and Beatriz S. L. Pires de Lima, *A KNN Under-sampling Approach for Data Balancing*, Journal of Intelligent Learning Systems and Applications **7**, no. 4, 1951, available at <https://www.scirp.org/journal/paperinformation?paperid=60996>.
- [2] Tadeusz Caliński and Jerzy Harabasz, *A dendrite method for cluster analysis*, 1974, <https://doi.org/10.1080/03610927408827101>.
- [3] David L. Davies and Donald W. Bouldin, *A Cluster Separation Measure*, IEEE Transactions on Pattern Analysis and Machine Intelligence **PAMI-1** (1979), no. 2, 224-227, available at <https://ieeexplore.ieee.org/document/4766909>.
- [4] Martin Ester, Hans-Peter Kriegel, Jörg Sander, and Xiaowei Xu, *A Density-Based Algorithm for Discovering Clusters in Large Spatial Databases With Noise*, 1996, <https://dl.acm.org/doi/10.5555/3001460.3001507>.
- [5] R. A. Fisher, *Iris*, 1988, <https://doi.org/10.24432/C56C76>.
- [6] Cheng-Chia Huang, *Detect Spatial-Temporal Point Clusters by Incorporating Time into Density-based Clustering*, <https://www.esri.com/arcgis-blog/products/arcgis-pro/analytics/detect-spatial-temporal-point-clusters-by-incorporating-time-into-density-based-cluster>
- [7] Lawrence Hubert and Phipps Arabie, *Comparing partitions*, Journal of Classification **2** (1985), 193—218, available at <https://doi.org/10.1007/BF01908075>.
- [8] Anil K. Jain, *Data Clustering: 50 Years Beyond K-means*, Pattern Recognition Letters **31** (2010), no. 8, 651-666, available at https://link.springer.com/content/pdf/10.1007/978-3-540-87479-9_3.pdf.
- [9] Merriam-Webster, *cluster*, <https://www.merriam-webster.com/dictionary/cluster>.
- [10] Pablo Andretta Jaskowiak and Davoud Moulavi, *DBCv*, 2023, <https://github.com/pajaskowiak/dbcv/>.

- [11] Java Point, *K-Nearest Neighbor(KNN) Algorithm for Machine Learning*, <https://www.javatpoint.com/k-nearest-neighbor-algorithm-for-machine-learning>.
- [12] Christopher Jenness, *DBCv*, 2017, <https://github.com/christopherjenness/DBCv.git>.
- [13] James MacQueen, *Some Methods for Classification and Analysis of Multivariate Observations*, Proceedings of the 5th Berkeley Symposium on Mathematical Statistics and Probability **1: Statistics** (1967), 281–297, available at https://digitalassets.lib.berkeley.edu/math/ucb/text/math_s5_v1_article-17.pdf.
- [14] Juliano Morimoto and Fleur Ponton, *Virtual reality in biology: could we become virtual naturalists?*, Evolution: Education and Outreach **14** (2021), 5, available at <https://evolution-outreach.biomedcentral.com/articles/10.1186/s12052-021-00147-x>.
- [15] Davoud Moulavi, Pablo A. Jaskowiak, Ricardo J. G. B. Campello, Arthur Zimek, and Jörg Sander, *Density-Based Clustering Validation*, 2014, <https://doi.org/10.1137/1.9781611973440.96>.
- [16] Kolby Nottingham, Rachel Longjohn, and Markelle Kelly, *UC Irvine Machine Learning Repository*, 2023, <https://archive.ics.uci.edu/>.
- [17] F. Pedregosa, G. Varoquaux, A. Framfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay, *Scikit-learn: Machine Learning in Python* **12** (2011), 2825–2830 pp., available at <https://scikit-learn.org/stable/index.html>.
- [18] Piotta G. et al., *The largest of its kind*, <https://esahubble.org/images/potw1913a/>.
- [19] Erich Schubert, Jörg Sander, Martin Ester, Hans-Peter Kriegel, and Xiaowei Xu, *DBSCAN Revisited, Revisited: Why and How You Should (Still) Use DBSCAN*, 2017, <https://doi.org/10.1145/3068335>.
- [20] Kimberly Staudt, *Iris Data Visualization*, 2017, <https://www.kaggle.com/code/kstaud85/iris-data-visualization>.
- [21] Loren Rhodes, *DBSCAN and Misc. clustering topics*, 2021, <https://jcsites.juniata.edu/faculty/rhodes/ml/dbscan.htm>.
- [22] Peter J. Rousseeuw, *Silhouettes: A graphical aid to the interpretation and validation of cluster analysis*, 1986, https://www.sciencedirect.com/science/article/pii/0377042787901257?ref=pdf_download&fr=RR-2&rr=877fc92fee06236b.
- [23] Hugo D. Steinhaus, *Sur la division des corps matériels en parties (On the division of material bodies into parts)* **4** (1956), 801–804 pp., available at http://www.laurent-duval.eu/Documents/Steinhaus_H_1956_j-bull-acad-polon-sci_division_cmp-k-means.pdf.
- [24] Aniket Sharma, *suv_data*, <https://www.kaggle.com/datasets/iamaniket/suv-data/data>.
- [25] Felipe Alves Siqueira, *Fast Density-Based Clustering Validation (DBCv)*, 2024, <https://github.com/FelSiq/DBCv>.
- [26] Jessica Su, *CS 161 Lecture 3*, <https://web.stanford.edu/class/archive/cs/cs161/cs161.1168/lecture3.pdf>.
- [27] Banhu Yerra, *Centroid-based Clustering K-Means Algorithm*, 2018, https://github.com/mlBhanuYerra/machineLearning/blob/master/Clustering_K-means.ipynb.