Megha Murthy
Coding Language: Python

Final Project Option 2: Unsupervised Data Mining (Clustering)

This project is broken into two parts and the coding language used to complete this project is Python. The first part involves finding and removing any outliers in a set of 500 points in a two-dimensional Euclidean space and finding the Euclidean distance between any two points. The second part implements the hierarchical agglomerative clustering algorithm on the set created from part 1 that contains no outliers in k (user-specified) clusters. This will be repeated on a total of three different datasets.

Part 1: Identifying and Removing Outliers from a Set

Importing Libraries

For Part 1, which involves finding outliers in a generated dataset containing 500 points, I will be using the random library to randomly generate data points, the math library to calculate the Euclidean distance, and the Pandas library to store values in a data frame.

```
1  #import libraries
2  import numpy as np
3  import pandas as pd
4  from random import *
5  import math
```

Generating Random Data Points

Data points were randomly generated using the python random package and stored into a Pandas data frame.

```
#generate set of x,y coordinates
x = []
y = []
data = {}
seed(1)
for i in range(500):
    x_val = random()*15
    y_val = random()*15
    x.append(x_val)
    y.append(y_val)
    data = {"x":x,"y":y}

S = pd.DataFrame(data)
print(S)
```

In this code, the seed method stores the randomized generated points so that the same randomized values can be executed multiple times. Changing the seed number will result in

	x	y
0	2.015464	12.711506
1	11.456619	3.826035
2	7.431526	6.742366
3	9.773895	11.830850
4	1.407894	0.425212
..	...	...
495	13.887193	14.530279
496	4.013005	8.108040
497	6.603769	11.397828
498	12.635785	3.428402
499	4.118470	10.593923

[500 rows x 2 columns]

	x	y
0	14.340514	14.217412
1	0.848271	1.273080
2	12.532483	11.039550
3	10.045956	4.622047
4	9.089162	9.102026
..	...	...
495	9.675642	12.747197
496	2.330731	13.830559
497	0.432267	1.784681
498	1.554250	11.625383
499	7.569319	13.821165

[500 rows x 2 columns]

	x	y
0	3.569469	8.163438
1	5.549327	9.058801
2	9.385805	0.982933
3	0.197520	12.562036
4	3.890310	3.514964
..	...	...
495	12.494645	3.006853
496	8.914663	13.991878
497	12.725159	2.691856
498	14.451756	12.600639
499	2.518775	3.988373

[500 rows x 2 columns]

different randomized datasets, as shown in the figures below when changing seed value from 1 to 2 to 3. The random numbers were multiplied by 15 so that the randomized points fall into a 15x15 coordinate system. I did this so I can easily check any calculations manually to see if the values the code outputs are accurate. They were then appended to a list and then stored into a dictionary, which was then converted to a data frame.

Seed (1) Seed (2) Seed (3)

Calculating Euclidean Distance

Since there are a total of 500 points, I asked for user input to pick any two points or index values in the data frame to use for the calculations to calculate the Euclidean distance. However, we have to check for the validity of the user input, which is the purpose of the function in the screenshot below.

```
#definition to check user validity
def checkPoint(point):
    while point.isnumeric() == False:
        point = input("Please enter an index value (integer) for the first point between 0 to 499:")
    if point.isnumeric():
        point = int(point)
        while point < 0 or point > 499:
            point = input("Please enter an index value (integer) for the first point between 0 to 499:")
            point = int(point)
        return int(point)
```

The next set of code asks the user for the index of two points and checks the user input.

```
#ask user for index of first point
point1 = input("Please enter an index value (integer) for the first point between 0 to 499:")
#check point
point1 = checkPoint(point1)

#ask user for index to second point
point2 = input("Please enter a second index value(integer) for the second point between 0 to 499:")
point2 = checkPoint(point2)

```

```
Please enter an index value (integer) for the first point between 0 to 499:hello
Please enter an index value (integer) for the first point between 0 to 499:56
Please enter a second index value(integer) for the second point between 0 to 499:679
Please enter an index value (integer) for the first point between 0 to 499:67
```

The Euclidean Distance for any two points (x, y) in the Euclidean n - spaces can be measured by:

$$d(x, y) = \sqrt{\sum_{i=0}^{n} (x_i - y_i)^2}$$

The function below calculates the Euclidean distance by using the points specified from the user input and following the formula above.

```
#create function for calculating euclid_dist
def euclid_dist(point1,point2,S):
    x1 = S.x[point1]
    y1 = S.y[point1]
    x2 = S.x[point2]
    y2 = S.y[point2]
    diff_x = x2 - x1
    diff_y = y2 - y1
    euclid = math.sqrt(math.pow(diff_x,2) + math.pow(diff_y,2))
    return euclid
```

```
euclid_dist(point1,point2,S)
```

```
2.200624377584338
```

Seed (1)

Similarly, the Euclidean distance for the same index values for the second and third dataset are

```
euclid_dist(point1,point2,S)
```

```
9.104168810226096
```

```
euclid_dist(point1,point2,S)
```

```
5.298218750630282
```

shown below.

Seed (2) Seed (3)

Finding and Removing Outliers

Using the datasets generated above, we can find any outliers using statistics and print them out to the console. An outlier is defined, in statistics, by any value that falls outside a 1.5*IQR (interquartile range) below the 1st quartile and above the third quartile.

```
#using Statistics to find outliers
def outlier(S):
    q1 = S.quantile(0.25)
    q3 = S.quantile(0.75)
    q_range = q3-q1
    outliers = S[((S<(q1-1.5*q_range)) | (S>(q3+1.5*q_range)))]
    return outliers
```

In the function defined above, the lower and upper quantiles can be calculated using the quantile function. The interquartile range is calculated by finding the difference between the third quartile and first quartile values. The “outliers” variable stores and checks to see if any values are outside the specified range. The code below checks for any outliers and returns “No outliers” if there are no outliers, and if there are outliers, prints out the number of outliers and other information about the outliers.

```python
#Check for Outliers
S_out = outlier(S)
if S_out.isnull().values.any() == True:
    print("No outliers found")
    S_prime = S
    print("Dataframe without Outliers:\n", S_prime)
else:
    print("Number of outliers: "+ str(len(S_out)))
    print("Max outlier value: " + str(S_out.max()))
    print("Min outlier value: "+ str(S_out.min()))
    S_prime = S_out.dropna()
    print("Dataframe without Outliers:\n", S_prime)


```

```
No outliers found
Dataframe without Outliers:
              x          y
0      2.015464  12.711506
1     11.456619   3.826035
2      7.431526   6.742366
3      9.773895  11.830850
4      1.407894   0.425212
..          ...        ...
495   13.887193  14.530279
496    4.013005   8.108040
497    6.603769  11.397828
498   12.635785   3.428402
499    4.118470  10.593923

[500 rows x 2 columns]
```

Seed (1)

Similarly, for the second and third dataset, the results from checking for outliers is shown below.

```
No outliers found
Dataframe without Outliers:
               x          y
0     14.340514  14.217412
1      0.848271   1.273080
2     12.532483  11.039550
3     10.045956   4.622047
4      9.089162   9.102026
..          ...        ...
495    9.675642  12.747197
496    2.330731  13.830559
497    0.432267   1.784681
498    1.554250  11.625383
499    7.569319  13.821165

[500 rows x 2 columns]
```

Seed (2)

```
No outliers found
Dataframe without Outliers:
               x          y
0      3.569469   8.163438
1      5.549327   9.058801
2      9.385805   0.982933
3      0.197520  12.562036
4      3.890310   3.514964
..          ...        ...
495   12.494645   3.006853
496    8.914663  13.991878
497   12.725159   2.691856
498   14.451756  12.600639
499    2.518775   3.988373

[500 rows x 2 columns]
```

Seed (3)

Part 2: Hierarchical Agglomerative Clustering Algorithm

Part 2 utilizes the output from Part 1 to create a hierarchical agglomerative clustering algorithm into k (user specified) clusters.

Importing Libraries

There are several packages that can be used to implement the hierarchical agglomerative clustering algorithm taught in class.

```
#import relevant libraries
import matplotlib.pyplot as plt
from sklearn.cluster import AgglomerativeClustering
from scipy.cluster import hierarchy
import seaborn as sns
from sklearn.cluster import KMeans
from sklearn.metrics import silhouette_score
```

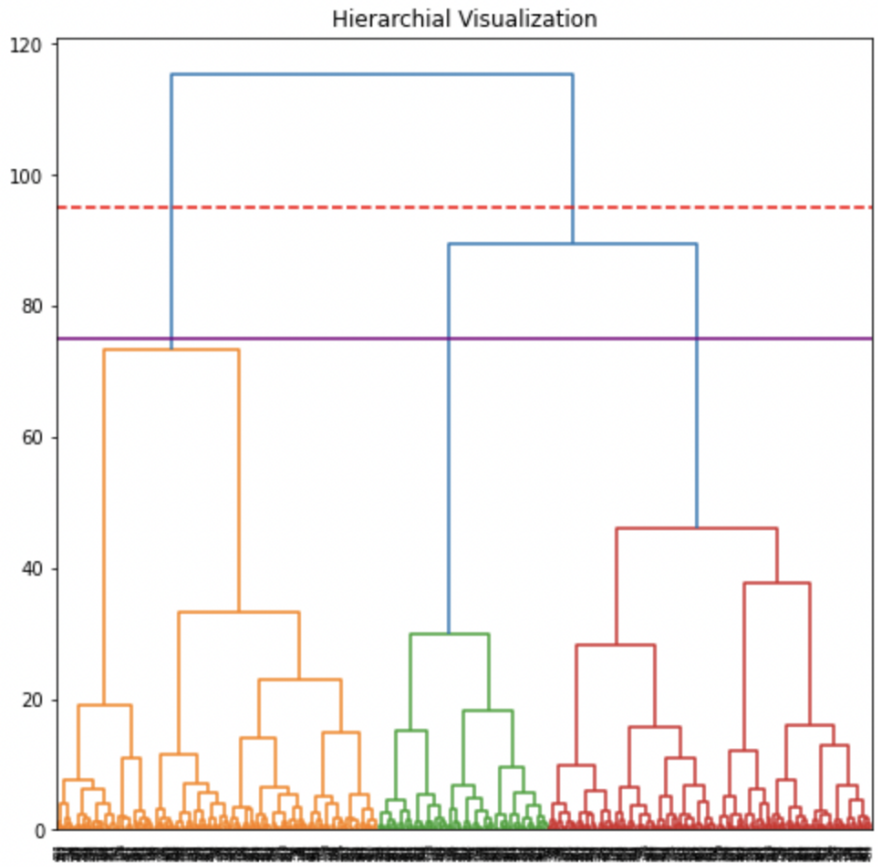
Matplotlib and seaborn are both used for visualization purposes and sklearn and scipy are used for clustering algorithms.

Visualizing Hierarchy of Data

A dendrogram can be plotted to visualize the hierarchical structure of the clusters in our dataset containing no outliers. This can be done using the hierarchy library.

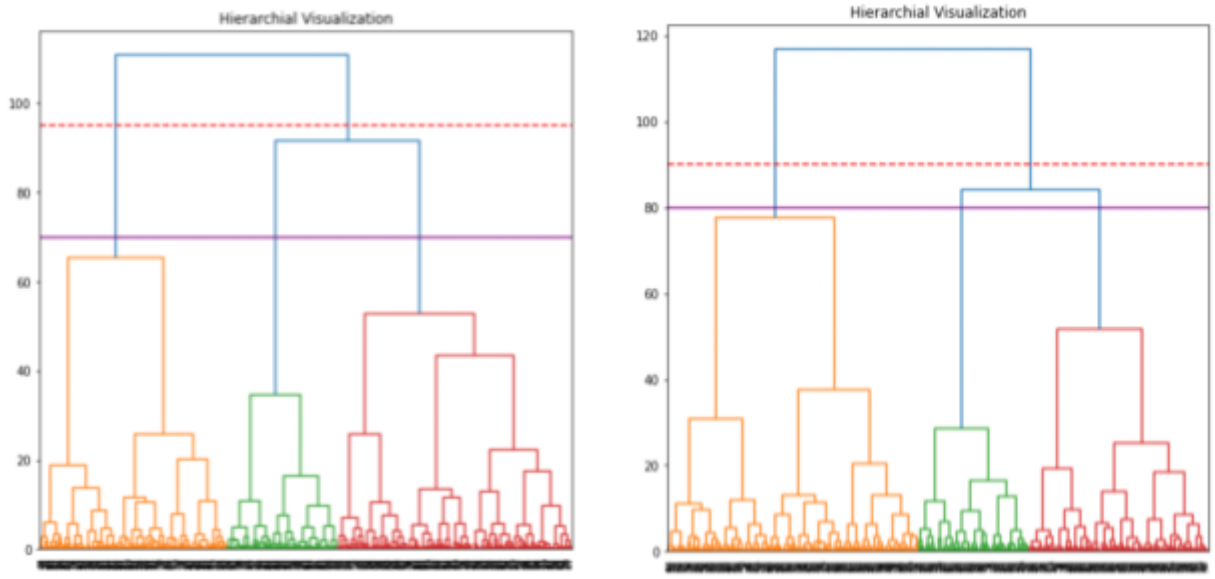
```
clusters = hierarchy.linkage(S_prime, method="ward")

plt.figure(figsize =(8, 8))
plt.title('Hierarchial Visualization')
Dendrogram = hierarchy.dendrogram(clusters)

# first biggest distance between clusters
plt.axhline(100, color='red', linestyle='--');
# second biggest distance between clusters
plt.axhline(80, color='purple');
```

In the code above, the clusters variable creates clusters and the dendrogram variable plots the clusters that are created from the linkage function.



Seed (1)

The lines in the plot denote the biggest distance between the clusters. Essentially, this hierarchical structure creates a tree-like structure to show similarity between groups. In this case, it groups points together based on distance between the data points.



Similarly, a dendrogram can be created for the second and third dataset as well as shown below.

Seed (2) Seed (3)

KMeans Clustering

KMeans Clustering is another clustering method that accounts more for the average distance between data points to group points together. A silhouette score can be used to evaluate how well the data points are grouped together or clustered together. The value for the silhouette score normally ranges from -1 to 1, with 1 representing well separated clusters and -1 representing values that may be in the wrong cluster.

```
kmodel = KMeans(n_clusters=k, random_state=1)
kmodel.fit_predict(S_prime)
score = silhouette_score(S_prime, kmodel.labels_, metric='euclidean')
print('Silhouette Score: %.3f' % score)
```

```
Silhouette Score: 0.387
```

Seed (1)

Like the other clustering algorithms, the code above creates clusters based on user input and calculates the silhouette score based on the optimum number of clusters. For a k value of 3 above, the silhouette score is 0.387. We can also visually check for the optimal number of clusters using the elbow method. The optimal number of clusters is located at the elbow of the plot.

We can find the silhouette score for the other two datasets as well.

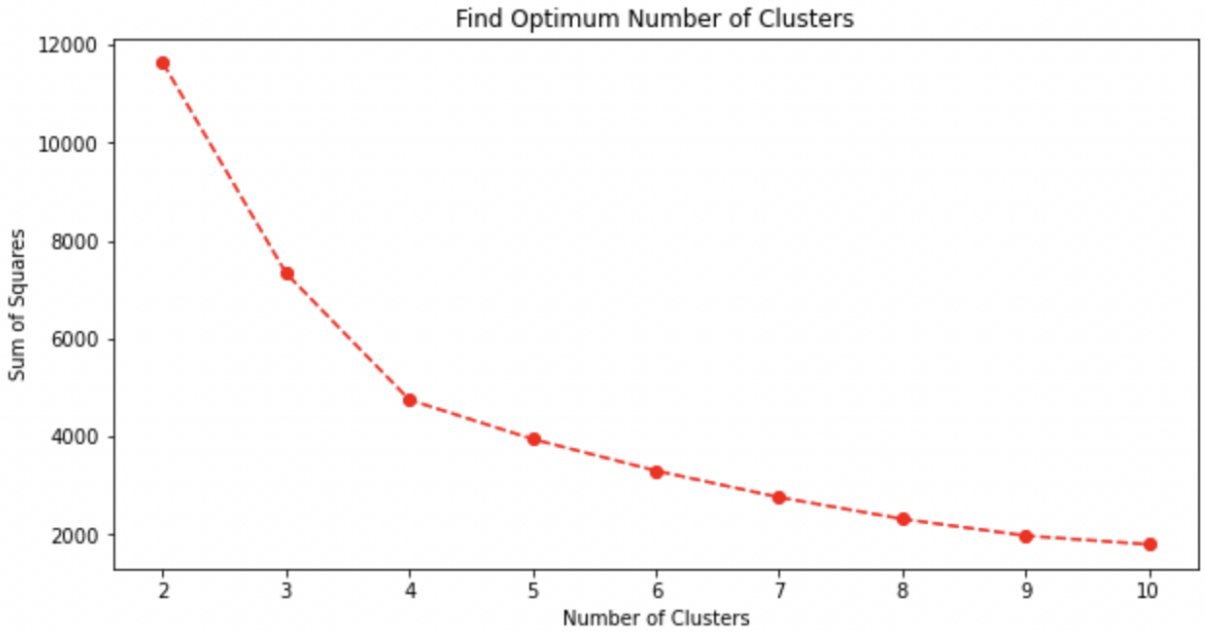
```
Silhouette Score: 0.381
```

```
Silhouette Score: 0.400
```

Seed (2) Seed (3)

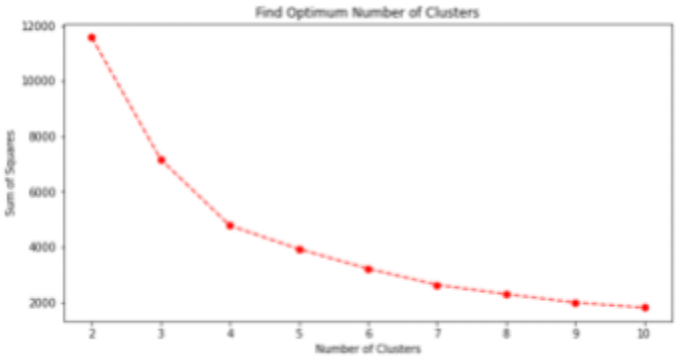
```
numClust = list(range(2,11))
dist = []
for i in numClust:
    kmeans = KMeans(n_clusters=i)
    kmeans.fit(S_prime)
    dist.append(kmeans.inertia_)

plt.figure(figsize=(10, 5))
plt.plot(numClust, dist, 'ro--')
plt.xlabel('Number of Clusters')
plt.ylabel('Sum of Squares')
plt.title('Find Optimum Number of Clusters')
```

The code above shows a loop through potential k values for the number of clusters and calculating the minimum sum of squares to plot the points.

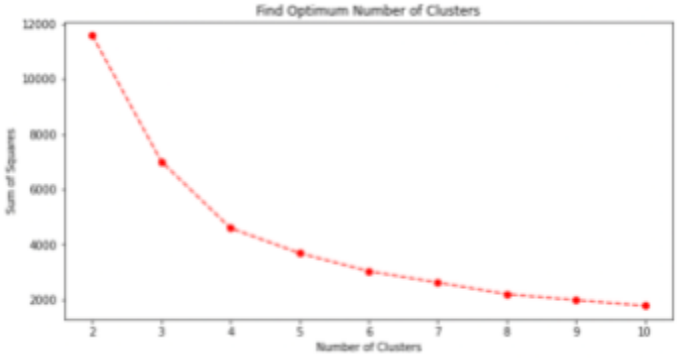


Seed (1)

Based on the elbow plot above, we can see that k = 3 is the ideal number of clusters for this dataset (and the other two datasets as well). K = 3 also has the highest silhouette score. We can plot an elbow plot for the other two datasets as well.



Seed (2)



Seed (3)

Agglomerative Clusters

We can create a clustering model from the data containing no outliers using the Scikit-learn cluster package.

```
#ask users to enter number of clusters
k = input("Please input the number of clusters you want:")
while k.isnumeric() == False:
    k = input("Please input a numerical value for the number of clusters want:")

k = int(k)
```

```
Please input the number of clusters you want:3
```

The code above asks the user for how many clusters they would like to work with and checks to see if their input is numerical. Now that we know 3 is the optimal value based on the elbow plot

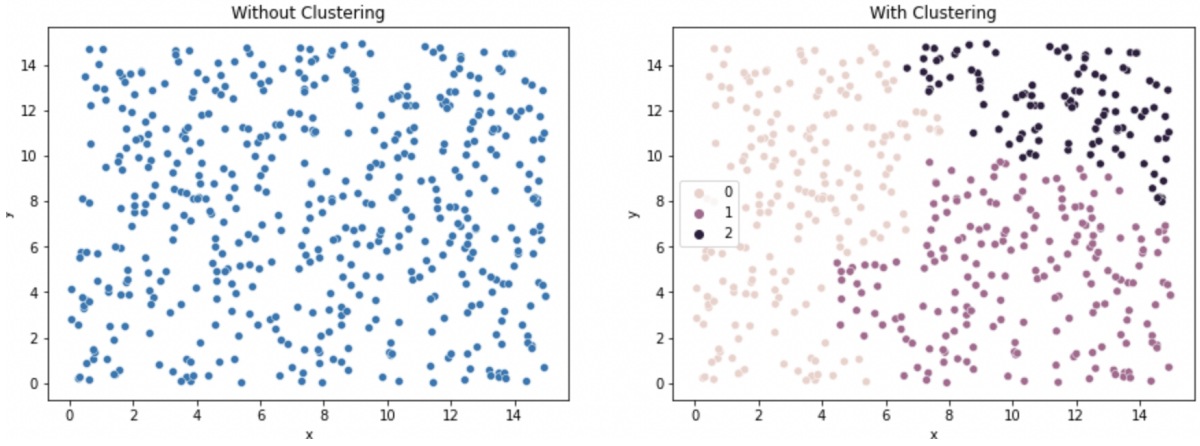
for the first data set, we can just input the number of clusters as 3 as a basis to work with the data.

```
#create clustering model
clust = AgglomerativeClustering(n_clusters = k, linkage="ward")
clust.fit(S_prime)
labels = clust.labels_
```

Clusters can be created using the user input from the code above. The labels help to differentiate between the different clusters in the actual scatterplot.

```
fig, axes = plt.subplots(nrows=1, ncols=2, figsize=(15,5))
sns.scatterplot(ax=axes[0],data=S_prime,x=S_prime.x,y=S_prime.y).set_title('Without Clustering')
sns.scatterplot(ax=axes[1],data=S_prime,x=S_prime.x,y=S_prime.y, hue=labels).set_title('With Clustering');
```

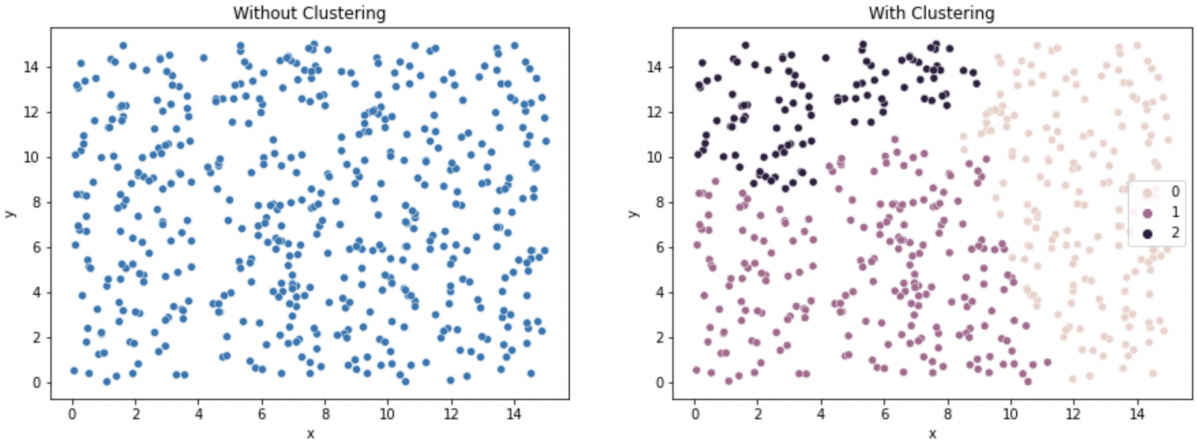
In the code above, a subplot containing 1 row and 2 columns are created to compare the plots without clustering and with the user specified clustering.



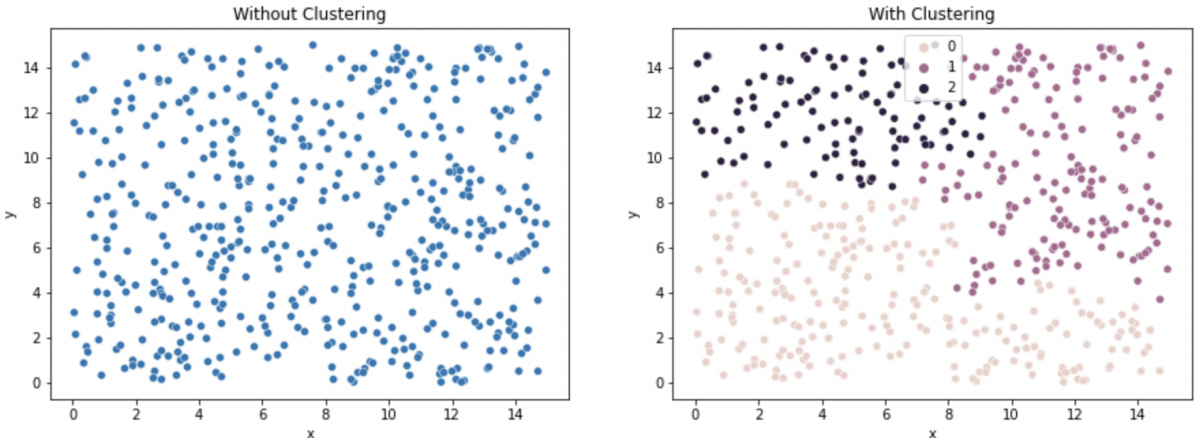
Seed (1)

The plots above show the difference between the clustering algorithm, which groups points together based on proximity, and without any clustering.

Similarly, we can create a plot to show the difference using clustering and without clustering for the other two datasets as well.



Seed (2)



Seed (3)