Data Mining – Final Term Project – Option 2

Wellington Cunha

NJIT ID: 31548454

NJIT UC ID: wc44

[wc44@njit.edu](mailto:Wc44@njit.edu)

For this project, we selected Python to be the tool/language. So, the first step was to load the pip packages required:

|  |
| --- |
| **import** numpy **as** np  **import** pandas **as** pd  **import** math  **import** matplotlib**.**pyplot **as** plt  **import** matplotlib**.**patches **as** mpatches  **from** mpl\_toolkits **import** mplot3d  **import** warnings  warnings**.**filterwarnings**(**"ignore"**)**  **from** IPython**.**display **import** clear\_output |

We start by generating a set with 500 3-D points, as required, using numpy, to be using while developing the functions to be used in the main goal of the project. We convert it to Pandas DataFrame to take advantage of several functions embedded on a DataFrame for processing data.

|  |
| --- |
| test **=** np**.**random**.**randint**(**0**,** 100**,** size**=(**500**,** 3**))**  test **=** pd**.**DataFrame**(**test**,** columns **=** **[**"x"**,** "y"**,** "z"**])**  test**.**head**()** |

Here is the evidence of the code block above execution:

Graphical user interface, text, application

Description automatically generated

We plot the data to check how it looks like

|  |
| --- |
| fig **=** plt**.**figure**(**figsize **=** **(**16**,** 8**))**  ax **=** plt**.**axes**(**projection **=**"3d"**)**  ax**.**xaxis**.**pane**.**fill **=** **False**  ax**.**yaxis**.**pane**.**fill **=** **False**  ax**.**zaxis**.**pane**.**fill **=** **False**  my\_cmap **=** plt**.**get\_cmap**(**"gist\_rainbow\_r"**)**  ax**.**scatter3D**(**test**[**"x"**],** test**[**"y"**],** test**[**"z"**])** |

Chart, scatter chart

Description automatically generated

We now create some functions that will be used throughout the project. Let's start with a function that calculates the Euclidean distance. This function receives:

* **from\_data\_point**: a 3-D vector containing the initial data point from where the distance will be calculated
* **to\_data\_point**: a 3-D vector containing the final data point to where the distance will be calculated

and then it returns the distance between the two data points

**Note**: we are calling from and to just to differentiate the two arguments, as the distance on both ways are the same

|  |
| --- |
| **def** calculate\_euclidean\_distance**(**from\_data\_point**,** to\_data\_point**):**  **return** math**.**sqrt**(**  **(**from\_data\_point**[**0**]** **-** to\_data\_point**[**0**])** **\*\*** 2 **+**  **(**from\_data\_point**[**1**]** **-** to\_data\_point**[**1**])** **\*\*** 2 **+**  **(**from\_data\_point**[**2**]** **-** to\_data\_point**[**2**])** **\*\*** 2  **)**  calculate\_euclidean\_distance**(**test**.**iloc**[:**1**].**values**.**flatten**(),** **[**0**,** 0**,** 0**])** |

Text

Description automatically generated

Let's test for all of the 500 points generated, using the center of the space that generated the points.

|  |
| --- |
| test**.**apply**(lambda** row**:** calculate\_euclidean\_distance**(**row**.**values**.**flatten**(),** **[**50**,** 50**,** 50**]),** axis **=** 1**)** |

Graphical user interface, text

Description automatically generated

The second function we create is to calculate the center of a cluster. This function receives:

* **cluster**: a list of 3-D vectors

and will return the cluster centroid

|  |
| --- |
| **def** calculate\_centroid**(**cluster**):**  centroid **=** **[**  **sum([**x**[**0**]** **for** x **in** cluster**])** **/** **len(**cluster**),**  **sum([**y**[**1**]** **for** y **in** cluster**])** **/** **len(**cluster**),**  **sum([**z**[**2**]** **for** z **in** cluster**])** **/** **len(**cluster**)**  **]**    **return** centroid    calculate\_centroid**([[**0**,** 0**,** 0**],** **[**100**,** 100**,** 100**]])** |

Graphical user interface

Description automatically generated with medium confidence

Let's test this function for the DataFrame with our data points

|  |
| --- |
| calculate\_centroid**(**test**.**values**.**tolist**())** |



The third function that we create is to calculate the distance between two clusters. This function receives:

* **from\_data\_points**: a list containing 3-D vector for the initial data points from where the distance will be calculated
* **to\_data\_points**: a list containing 3-D vectors for the final points to where the distance will be calculated
* **distance\_type**: a parameter determining the type of distance (min, max, avg and center) that will be used for the calculation

and returns the distance between two sets of datapoints (even "sets" with only one datapoint), according to the type specified

|  |
| --- |
| from\_data\_points **=** **[[**0**,** 0**,** 0**],** **[**15**,** 15**,** 15**],** **[**100**,** 100**,** 100**]]**  to\_data\_points **=** **[[**10**,** 10**,** 10**],** **[**1**,** 1**,** 1**],** **[**50**,** 50**,** 50**]]**  **def** find\_cluster\_distance**(**from\_data\_points**,** to\_data\_points**,** **type** **=** "min"**):**  distance **=** 0.00  **if** **type** **==** "center"**:**  distance **=** calculate\_euclidean\_distance**(**  calculate\_centroid**(**from\_data\_points**),**  calculate\_centroid**(**to\_data\_points**)**  **)**  **else:**  li **=** **[]**  **for** each\_from **in** from\_data\_points**:**  **for** each\_to **in** to\_data\_points**:**  li**.**append**(**calculate\_euclidean\_distance**(**each\_from**,** each\_to**))**  **if** **len(**li**)** **==** 0**:**  distance **=** 0  **elif** **type** **==** "min"**:**  distance **=** **min(**li**)**  **elif** **type** **==** "max"**:**  distance **=** **max(**li**)**  **elif** **type** **==** "avg"**:**  distance **=** **sum(**li**)** **/** **len(**li**)**  **return** distance  **print(**"Distance between centroids:"**,** find\_cluster\_distance**(**from\_data\_points**,** to\_data\_points**,** **type=**"center"**))**  **print(**"Minimum distance:"**,** find\_cluster\_distance**(**from\_data\_points**,** to\_data\_points**,** **type=**"min"**))**  **print(**"Maximum distance:"**,** find\_cluster\_distance**(**from\_data\_points**,** to\_data\_points**,** **type=**"max"**))**  **print(**"Average distance:"**,** find\_cluster\_distance**(**from\_data\_points**,** to\_data\_points**,** **type=**"avg"**))** |

Text

Description automatically generated

Let's test this function for the DataFrame with our data points

|  |
| --- |
| **print(**find\_cluster\_distance**(**test**.**values**.**tolist**(),** test**.**values**.**tolist**(),** **type** **=** "center"**))**  **print(**find\_cluster\_distance**(**test**.**values**.**tolist**(),** test**.**values**.**tolist**(),** **type** **=** "min"**))**  **print(**find\_cluster\_distance**(**test**.**values**.**tolist**(),** test**.**values**.**tolist**(),** **type** **=** "max"**))**  **print(**find\_cluster\_distance**(**test**.**values**.**tolist**(),** test**.**values**.**tolist**(),** **type** **=** "avg"**))** |

Graphical user interface

Description automatically generated with low confidence

**Outlier detection**

For outlier detection, we implement the **Distance-Based Outlier Detection** using the **Nested Loop Method** as described in chapter 12.4.1 of **Data Mining: Concepts and Techniques** coursebook. We construct a function that receives:

* **data\_points**: a list of 3-D vectors containing all datapoints
* **d\_distance**: the distance threshold to be used
* **p\_fraction**: the fraction of neighbors to be considered in the radius

and returns a list with all datapoints and a flag (True/False) of that datapoint being a potential outlier

|  |
| --- |
| test\_list **=** test**.**copy**().**values**.**tolist**()**  **def** DB\_outlier\_detection**(**data\_points**,** d\_distance **=** 100**,** p\_fraction **=** 0.1**):**  li **=** **[]**  **for** i **in** **range(**0**,** **len(**data\_points**)):**  count **=** 0  is\_outlier **=** **True**  **for** j **in** **range(**0**,** **len(**data\_points**)):**  distance **=** find\_cluster\_distance**([**data\_points**[**i**]],** **[**data\_points**[**j**]],** **type** **=** "min"**)**  **if** **(**i **!=** j**)** **&** **(**distance **<=** d\_distance**):**  count **+=** 1  **if** count **>=** p\_fraction **\*** **len(**data\_points**):**  is\_outlier **=** **False**  li**.**append**([**data\_points**[**i**],** is\_outlier**])**  **return** li  outliers **=** pd**.**DataFrame**(**DB\_outlier\_detection**(**test\_list**),** columns **=** **[**"coordinates"**,** "potential\_outlier"**])**  outliers |

Graphical user interface, application, Word

Description automatically generated

We run again, but now reducing the d (distance) and increasing p\_fraction parameters to generate some outliers. In order to visually inspect the results, we converted the list into a Pandas DataFrame and then transform and add columns in order to plot the results

|  |
| --- |
| outliers **=** DB\_outlier\_detection**(**test\_list**,** d\_distance **=** 50**,** p\_fraction **=** 0.15**)**  outliers **=** pd**.**DataFrame**(**outliers**,** columns **=** **[**"coordinates"**,** "potential\_outlier"**])**  outliers**[**"coordinates\_str"**]** **=** **[**','**.**join**(map(str,** l**))** **for** l **in** outliers**[**"coordinates"**]]**  outliers**[[**"x"**,** "y"**,** "z"**]]** **=** outliers**[**"coordinates\_str"**].str.**split**(**","**,** expand**=True)**  outliers**[[**"x"**,** "y"**,** "z"**]]** **=** outliers**[[**"x"**,** "y"**,** "z"**]].**apply**(**pd**.**to\_numeric**)**  outliers **=** outliers**[[**"x"**,** "y"**,** "z"**,** "potential\_outlier"**]]**  outliers**[**"color"**]** **=** outliers**[**"potential\_outlier"**].**apply**(lambda** row**:** "Red" **if** row **else** "Blue"**)**  outliers**.**tail**()** |

Graphical user interface, application, Word

Description automatically generated

Now we can plot the results

|  |
| --- |
| fig **=** plt**.**figure**(**figsize **=** **(**16**,** 8**))**  ax **=** plt**.**axes**(**projection **=**"3d"**)**  ax**.**xaxis**.**pane**.**fill **=** **False**  ax**.**yaxis**.**pane**.**fill **=** **False**  ax**.**zaxis**.**pane**.**fill **=** **False**  ax**.**scatter3D**(**outliers**[**"x"**],** outliers**[**"y"**],** outliers**[**"z"**],** facecolors **=** outliers**[**"color"**])**  outlier\_label **=** **(**  "Potential Outliers: " **+**  **str(len(**outliers**[**outliers**[**"potential\_outlier"**]** **==** **True]))** **+**  " out of " **+**  **str(len(**outliers**))**  **)**  red\_patch **=** mpatches**.**Patch**(**color**=**"red"**,** label**=**outlier\_label**)**  ax**.**legend**(**handles**=[**red\_patch**],** bbox\_to\_anchor**=(**1.5**,** .5**),** loc**=**"center right"**)** |

Chart, scatter chart

Description automatically generated

We have attempted several combinations for distance and fraction of neighbors and came to the conclusion that the best minimum distance to be used for this dataset was 50 while the fractions between 0.11 and 0.15 were the ones that best "tagged" potential outliers. We are going to use d = 50 and p = 0.15 for now on

Finally, we create a function to remove the outliers of a dataset (Pandas DataFrame). This function receives:

* **data\_points**: a Pandas DataFrame contaning 3 columns (x, y and z), each one with one of the coordinates for the data points
* **d\_distance**: the distance threshold to be used
* **p\_fraction**: the fraction of neighbors to be considered in the radius

and returns the Pandas DataFrame without the outliers according to the parameters informed

|  |
| --- |
| **def** remove\_outliers**(**data\_points**,** d\_distance **=** 100**,** p\_fraction **=** 0.1**):**  datapoints\_list **=** data\_points**.**copy**().**values**.**tolist**()**  outliers **=** DB\_outlier\_detection**(**datapoints\_list**,** d\_distance **=** 50**,** p\_fraction **=** 0.15**)**  outliers **=** pd**.**DataFrame**(**outliers**,** columns **=** **[**"coordinates"**,** "potential\_outlier"**])**  outliers**[**"coordinates\_str"**]** **=** **[**','**.**join**(map(str,** l**))** **for** l **in** outliers**[**"coordinates"**]]**  outliers**[[**"x"**,** "y"**,** "z"**]]** **=** outliers**[**"coordinates\_str"**].str.**split**(**","**,** expand**=True)**  outliers**[[**"x"**,** "y"**,** "z"**]]** **=** outliers**[[**"x"**,** "y"**,** "z"**]].**apply**(**pd**.**to\_numeric**)**  outliers **=** outliers**[[**"x"**,** "y"**,** "z"**,** "potential\_outlier"**]]**  outliers**.**merge**(**data\_points**)**  outliers **=** outliers**[**outliers**[**"potential\_outlier"**]** **==** **False]**    **return** outliers    test\_without\_outliers **=** remove\_outliers**(**test**.**copy**(),** d\_distance **=** 50**,** p\_fraction **=** 0.15**)**  test\_without\_outliers |

Graphical user interface, text, application

Description automatically generated

As we can see above, the 42 outliers previously detected were removed from the final DataFrame

**K-means**

Now we implement the K-means. But before that, we implement a function to find the closest coordinate of a Data Point. We use this to assign the data point to the closest cluster. This function receives:

* **coordinates**: the 3-D coordinates of a data point
* **clusters**: a list of 3-D coordinates of clusters
* **type**: the type of distance (center, min, max or avg) to be used when calculating the distance and determining the nearest cluster

and outputs the item from the list of clusters that is closest according to "type" criteria defined

|  |
| --- |
| **def** find\_closest\_cluster**(**coordinates**,** clusters**,** **type** **=** "min"**):**  li **=** **[]**  **for** each\_cluster **in** clusters**:**  distance **=** find\_cluster\_distance**(**coordinates**,** **[**each\_cluster**],** **type** **=** "min"**)**  li**.**append**(**distance**)**    **return** clusters**[**li**.**index**(min(**li**))]**    coordinates **=** test\_without\_outliers**.**copy**().**iloc**[:**1**][[**"x"**,** "y"**,** "z"**]].**values**.**tolist**()**  **print(**coordinates**)**  find\_closest\_cluster**(**coordinates**,** **[[**0**,** 0**,** 0**],** **[**100**,** 100**,** 100**]])** |

Text

Description automatically generated with medium confidence

Finally, we create the function that implement the K-means algorithm. This function receives:

* **data\_points**: the data points to be clustered as a Pandas DataFrame
* **k**: the number of clusters to be generated

and returns the Pandas DataFrame with three additional columns (cx, cz and cy) containing the coordinates of the centroid of the cluster the data point is assigned to. It also prints the number of iterations required to converge

**Note**: the function picks the initial centroids randomly

|  |
| --- |
| test\_without\_outliers **=** remove\_outliers**(**test**.**copy**(),** d\_distance **=** 50**,** p\_fraction **=** 0.15**)**  **def** k\_means**(**data\_points**,** k**=**10**):**  current\_centroids **=** np**.**zeros**((**k**,** 3**)).**tolist**()**  centroids **=** np**.**random**.**randint**(**0**,** 100**,** size**=(**k**,** 3**)).**tolist**()**  counter **=** 1  **while** current\_centroids **!=** centroids**:**  current\_centroids **=** centroids  # Assign data points to centroids  data\_points**[**"nearest"**]** **=** data\_points**.**apply**(**  **lambda** row**:** find\_closest\_cluster**(**  **[[**row**.**x**,** row**.**y**,** row**.**z**]],**  centroids**,**  **type** **=** "min"**),**  axis **=** 1**)**  data\_points**[**"cluster"**]** **=** data\_points**[**"nearest"**].**astype**(str)**  # Re-calculate the centroids  centroids **=** data\_points**[[**"cluster"**,** "x"**,** "y"**,** "z"**]].**groupby**([**"cluster"**]).**mean**()**  centroids **=** centroids**[[**"x"**,** "y"**,** "z"**]].**values**.**tolist**()**  counter **+=** 1  **print(**f"Number of iterations needed to converge to {k} clusters:"**,** counter**)**    data\_points**[**"cluster"**]** **=** data\_points**[**"cluster"**].str.**replace**(**"\[|\]"**,** ""**,** regex**=True)**  data\_points**[[**"cx"**,** "cy"**,** "cz"**]]** **=** data\_points**[**"cluster"**].str.**split**(**","**,** expand**=True)**  data\_points**[[**"cx"**,** "cy"**,** "cz"**]]** **=** data\_points**[[**"cx"**,** "cy"**,** "cz"**]].**apply**(**pd**.**to\_numeric**)**  **return** data\_points**.**drop**([**"nearest"**,** "cluster"**],** axis**=**1**)**  k\_means**(**test\_without\_outliers**.**copy**(),** 10**)** |

Graphical user interface, table

Description automatically generated with medium confidence

To check how K-means accommodated the data points into clusters, let's first see how many datapoints are contained in each cluster

|  |
| --- |
| test\_kmeans **=** k\_means**(**test\_without\_outliers**.**copy**(),** 10**)**  test\_kmeans**[[**"cx"**,** "cy"**,** "cz"**,** "potential\_outlier"**]].**groupby**([**"cx"**,** "cy"**,** "cz"**]).**count**()** |

Graphical user interface, text, application

Description automatically generated

And we also can plot it

|  |
| --- |
| fig **=** plt**.**figure**(**figsize **=** **(**16**,** 8**))**  ax **=** plt**.**axes**(**projection **=**"3d"**)**  ax**.**xaxis**.**pane**.**fill **=** **False**  ax**.**yaxis**.**pane**.**fill **=** **False**  ax**.**zaxis**.**pane**.**fill **=** **False**  my\_cmap **=** plt**.**get\_cmap**(**"gist\_rainbow\_r"**)**  ax**.**scatter3D**(**test\_kmeans**[**"x"**],** test\_kmeans**[**"y"**],** test\_kmeans**[**"z"**],**  cmap **=** my\_cmap**,** c **=** **(**test\_kmeans**[**"cy"**]** **+** test\_kmeans**[**"cz"**]** **+** test\_kmeans**[**"cx"**]))** |

Chart, scatter chart

Description automatically generated

**Silhouette coefficient**

Next, we implement the silhouette coefficient algorithm. To implement and test it, we used the example provided during the respective lecture to make sure the calculations are matching. So, first we create a Pandas DataFrame with those data points and the respective cluster centroids coordinates as per the lecture

|  |
| --- |
| c1 **=** **[[**5**,** 16**,** 0**],** **[**8**,** 13**,** 0**],** **[**8**,** 10**,** 0**],** **[**5**,** 9**,** 0**]]**  c1 **=** pd**.**DataFrame**(**c1**,** columns**=[**"x"**,** "y"**,** "z"**],** index**=[**1**,** 2**,** 3**,** 4**])**  c1**[**"cluster"**]** **=** c1**[[**"x"**,** "y"**,** "z"**]].**apply**(lambda** row**:** calculate\_centroid**(**c1**.**values**.**tolist**()),** axis **=** 1**)**  c1**[**"coordinates\_str"**]** **=** **[**','**.**join**(map(str,** l**))** **for** l **in** c1**[**"cluster"**]]**  c1**[[**"cx"**,** "cy"**,** "cz"**]]** **=** c1**[**"coordinates\_str"**].str.**split**(**","**,** expand**=True)**  c1**[[**"x"**,** "y"**,** "z"**]]** **=** c1**[[**"x"**,** "y"**,** "z"**]].**apply**(**pd**.**to\_numeric**)**  c1 **=** c1**[[**"x"**,** "y"**,** "z"**,** "cx"**,** "cy"**,** "cz"**]]**  c2 **=** **[[**14**,** 16**,** 0**],** **[**18**,** 17**,** 0**]]**  c2 **=** pd**.**DataFrame**(**c2**,** columns**=[**"x"**,** "y"**,** "z"**],** index**=[**5**,** 6**])**  c2**[**"cluster"**]** **=** c2**[[**"x"**,** "y"**,** "z"**]].**apply**(lambda** row**:** calculate\_centroid**(**c2**.**values**.**tolist**()),** axis **=** 1**)**  c2**[**"coordinates\_str"**]** **=** **[**','**.**join**(map(str,** l**))** **for** l **in** c2**[**"cluster"**]]**  c2**[[**"cx"**,** "cy"**,** "cz"**]]** **=** c2**[**"coordinates\_str"**].str.**split**(**","**,** expand**=True)**  c2**[[**"x"**,** "y"**,** "z"**]]** **=** c2**[[**"x"**,** "y"**,** "z"**]].**apply**(**pd**.**to\_numeric**)**  c2 **=** c2**[[**"x"**,** "y"**,** "z"**,** "cx"**,** "cy"**,** "cz"**]]**  c3 **=** **[[**10**,** 3**,** 0**],** **[**12**,** 5**,** 0**],** **[**16**,** 7**,** 0**],** **[**16**,** 11**,** 0**]]**  c3 **=** pd**.**DataFrame**(**c3**,** columns**=[**"x"**,** "y"**,** "z"**],** index**=[**7**,** 8**,** 9**,** 10**])**  c3**[**"cluster"**]** **=** c3**[[**"x"**,** "y"**,** "z"**]].**apply**(lambda** row**:** calculate\_centroid**(**c3**.**values**.**tolist**()),** axis **=** 1**)**  c3**[**"coordinates\_str"**]** **=** **[**','**.**join**(map(str,** l**))** **for** l **in** c3**[**"cluster"**]]**  c3**[[**"cx"**,** "cy"**,** "cz"**]]** **=** c3**[**"coordinates\_str"**].str.**split**(**","**,** expand**=True)**  c3**[[**"x"**,** "y"**,** "z"**]]** **=** c3**[[**"x"**,** "y"**,** "z"**]].**apply**(**pd**.**to\_numeric**)**  c3 **=** c3**[[**"x"**,** "y"**,** "z"**,** "cx"**,** "cy"**,** "cz"**]]**  all\_clusters **=** pd**.**concat**([**c1**,** c2**,** c3**])**  all\_clusters |

Graphical user interface, text, application

Description automatically generated

Now we define the calculate\_silhouette function. It receives:

* **data\_points**: a Pandas DataFrame containing the coordinates of each data point (on columns x, y and z) and the coordinates of their cluster's respective centroids (on columns cx, cz and cy)

and returns a Pandas DataFrame with one additional column containing the silhouette coefficient of that data point

|  |
| --- |
| **def** calculate\_silhouette**(**data\_points**):**  intra **=** **[]**  **for** i**,** each\_point **in** data\_points**.**iterrows**():**  neighbors **=** data\_points  neighbors **=** neighbors**.**drop**(**i**)**  neighbors **=** neighbors**[(**neighbors**[**"cx"**]** **==** each\_point**[**"cx"**])** **&**  **(**neighbors**[**"cy"**]** **==** each\_point**[**"cy"**])** **&**  **(**neighbors**[**"cz"**]** **==** each\_point**[**"cz"**])]**  intra\_cluster **=** find\_cluster\_distance**(**  **[**each\_point**.**values**.**tolist**()],** neighbors**.**values**.**tolist**(),** **type** **=** "avg"**)**  other\_clusters **=** data\_points  other\_clusters **=** other\_clusters**[~((**other\_clusters**[**"cx"**]** **==** each\_point**[**"cx"**])** **&**  **(**other\_clusters**[**"cy"**]** **==** each\_point**[**"cy"**])** **&**  **(**other\_clusters**[**"cz"**]** **==** each\_point**[**"cz"**]))]**  other\_clusters **=** **(**other\_clusters**[[**"cx"**,** "cy"**,** "cz"**]]).**drop\_duplicates**()**  inter **=** **[]**  **for** j**,** each\_non\_neighbors **in** other\_clusters**.**iterrows**():**  non\_neighbors **=** data\_points  non\_neighbors **=** non\_neighbors**[(**non\_neighbors**[**"cx"**]** **==** each\_non\_neighbors**[**"cx"**])** **&**  **(**non\_neighbors**[**"cy"**]** **==** each\_non\_neighbors**[**"cy"**])** **&**  **(**non\_neighbors**[**"cz"**]** **==** each\_non\_neighbors**[**"cz"**])]**  non\_neighbors **=** non\_neighbors**[[**"x"**,** "y"**,** "z"**]]**  inter\_cluster **=** find\_cluster\_distance**(**  **[**each\_point**.**values**.**tolist**()],** non\_neighbors**.**values**.**tolist**(),** **type** **=** "avg"**)**  inter**.**append**(**inter\_cluster**)**  b1 **=** **min(**inter**)**  s1 **=** **(**b1 **-** intra\_cluster**)** **/** **max(**intra\_cluster**,** b1**)**  intra**.**append**({**  "index"**:** i**,**  "silhouette\_coeficient"**:** s1  **}**  **)**  **return** pd**.**concat**([**data\_points**,** pd**.**DataFrame**(**intra**).**set\_index**(**"index"**)],** axis**=**1**)**  calculate\_silhouette**(**all\_clusters**.**copy**())** |

Text

Description automatically generated

Let's check with the last cluster we generated using K-means algorithm

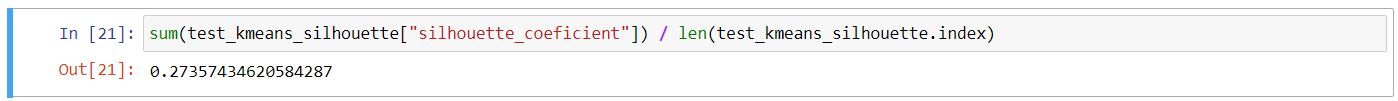
|  |
| --- |
| test\_kmeans\_silhouette **=** calculate\_silhouette**(**test\_kmeans**.**copy**())**  test\_kmeans\_silhouette**.**tail**()** |

Graphical user interface, text, application

Description automatically generated

We can also get the average silhouette for the whole clustering:

|  |
| --- |
| **sum(**test\_kmeans\_silhouette**[**"silhouette\_coeficient"**])** **/** **len(**test\_kmeans\_silhouette**.**index**)** |



As an exercise, let's calculate the coefficient for a range of K values between 5 and 15, to check which one of them has the higher value (towards 1)

|  |
| --- |
| li **=** **[]**  initial\_k**,** final\_k **=** 5**,** 15  **for** k **in** **range(**initial\_k**,** final\_k **+** 1**):**  check\_best\_k **=** k\_means**(**test\_without\_outliers**.**copy**(),** k**)**  k\_silhouette **=** calculate\_silhouette**(**check\_best\_k**.**copy**())**  row **=** **{**  "k"**:** k**,**  "silhouette"**:** **sum(**k\_silhouette**[**"silhouette\_coeficient"**])** **/** **len(**k\_silhouette**.**index**)**  **}**  li**.**append**(**row**)** |

Text

Description automatically generated with medium confidence

|  |
| --- |
| silhouette\_coefs **=** pd**.**DataFrame**(**li**)**  silhouette\_coefs**.**sort\_values**(**by**=[**"silhouette"**],** ascending**=False).**head**()** |

Graphical user interface, text, application, email

Description automatically generated

We can see above that the best K for the current data points is 6. However, as we will be generating the data points randomly when comparing it with hierarchical agglomerative clustering, we will keep using K=10. We could develop a function to check the best K and then use it, but as it would deman long time to run, especially on agglomerative clustering, we are not doing it (even because we should already compare with different distance types for agglomerative)

We also plot the values on a chart:

|  |
| --- |
| fig**,** ax **=** plt**.**subplots**(**figsize**=(**12**,** 4**))**  x **=** silhouette\_coefs**[**"k"**]**  y **=** silhouette\_coefs**[**"silhouette"**]**  plt**.**plot**(**x**,** y**)**  plt**.**xlabel**(**"K (Number of Clusters)"**,** size**=**12**)**  plt**.**ylabel**(**"Silhouette Coefficient"**,** size**=**12**)**  plt**.**title**(**"K (Number of Clusters) vs Silhouetter Coefficient"**,** size**=**15**)**  **for** index **in** **range(len(**x**)):**  ax**.**text**(**x**[**index**],** y**[**index**],** **round(**y**[**index**],** 3**),** size**=**12**)**  plt**.**xticks**(**x**,** size**=**12**)**  plt**.**grid**()**  plt**.**show**()** |

Chart, line chart

Description automatically generated

**Hierarchical Agglomerative Clustering**

Now we implement the Hierarchical Agglomerative Clustering algorithm. We have tested several implementations, but as we need to keep calculating the distance between every cluster after each agglomeration, it was taking a lot of time to process for 500 (minus the outliers) data points. One way of solving that was to store the distances into a Pandas DataFrame.

So, first we initialize the distances DataFrame, considering each individual point as one cluster, and then we keep deleting distances for data points/clusters that were just agglomerated and then adding the distances for the new cluster in relation to any other cluster/data point. The function that initializes the distances receives:

* **initial\_data\_points**: a Pandas DataFrame containing all the initial data points

and outputs a Pandas DataFrame with the distances between each individual data point

|  |
| --- |
| test\_hierarchical **=** remove\_outliers**(**test**.**copy**(),** d\_distance **=** 50**,** p\_fraction **=** 0.15**)**  initial\_data\_points **=** test\_hierarchical**.**copy**()**  **def** initialize\_distances**(**initial\_data\_points**):**  distances **=** pd**.**DataFrame**()**  **for** i**,** each\_from **in** initial\_data\_points**.**iterrows**():**  initial\_data\_points**.**drop**(**index **=** i**,** inplace**=True)**  **for** j**,** each\_to **in** initial\_data\_points**.**iterrows**():**  distances **=** distances**.**append**(**  **{**  "cx1"**:** each\_from**[**"x"**],**  "cy1"**:** each\_from**[**"y"**],**  "cz1"**:** each\_from**[**"z"**],**  "cx2"**:** each\_to**[**"x"**],**  "cy2"**:** each\_to**[**"y"**],**  "cz2"**:** each\_to**[**"z"**]**  **},** ignore\_index **=** **True**  **)**  distances**[**"distance"**]** **=** distances**.**apply**(lambda** row**:** calculate\_euclidean\_distance**(**  row**[[**"cx1"**,** "cy1"**,** "cz1"**]].**values**.**flatten**(),**  row**[[**"cx2"**,** "cy2"**,** "cz2"**]].**values**.**flatten**()),** axis **=** 1**)**  **return** distances  distances **=** initialize\_distances**(**initial\_data\_points**.**copy**().**head**(**10**))**  distances**.**head**()** |

Graphical user interface, text, application

Description automatically generated

And now we implement the Hierarchical Agglomerative Cluster on a function that receives:

* **data\_points**: a Pandas DataFrame with all data points
* **k**: the number of clusters to be generated
* **distance\_type**: the distance type (min, max, avg or center) to be used when calculating the distance between two clusters
* **distances**: the distances Pandas DataFrame with the initial distances. If this argument is not provided, the function initializes the distances DataFrame

as in the K-means function, it returns the Pandas DataFrame with three additional columns (cx, cz and cy) containing the coordinates of the centroid of the cluster the data point is assigned to

Just for the sake of having an idea on how the processing is going, we also print the step it is performing (cleaning the screen before)

|  |
| --- |
| **def** hierarchical\_agglomerative**(**data\_points**,** k**=**10**,** distance\_type **=** "min"**,** distances **=** **None):**  # First we define each point belonging to its own "cluster"  data\_points**[**"cx"**]** **=** data\_points**[**"x"**]**  data\_points**[**"cy"**]** **=** data\_points**[**"y"**]**  data\_points**[**"cz"**]** **=** data\_points**[**"z"**]**  **if** distances **is** **None:**  distances **=** initialize\_distances**(**data\_points**.**copy**())**  # Agglomerate while the number of clusters is greater than k  **while** **len((**data\_points**[[**"cx"**,** "cy"**,** "cz"**]]).**drop\_duplicates**().**index**)** **>** k**:**  clear\_output**(**wait**=True)**  **print(**"Agglomerating to"**,**  **len((**data\_points**[[**"cx"**,** "cy"**,** "cz"**]]).**drop\_duplicates**().**index**)** **-** 1**,**  f"clusters using {distance\_type} distance"**)**    shortest\_distance\_index **=** distances**.**sort\_values**(**by**=[**"distance"**]).**index**[**0**]**    to\_merge **=** data\_points**[((**data\_points**[**"cx"**]** **==** distances**.**at**[**shortest\_distance\_index**,** "cx1"**])** **&**  **(**data\_points**[**"cy"**]** **==** distances**.**at**[**shortest\_distance\_index**,** "cy1"**])** **&**  **(**data\_points**[**"cz"**]** **==** distances**.**at**[**shortest\_distance\_index**,** "cz1"**]))** **|**  **((**data\_points**[**"cx"**]** **==** distances**.**at**[**shortest\_distance\_index**,** "cx2"**])** **&**  **(**data\_points**[**"cy"**]** **==** distances**.**at**[**shortest\_distance\_index**,** "cy2"**])** **&**  **(**data\_points**[**"cz"**]** **==** distances**.**at**[**shortest\_distance\_index**,** "cz2"**]))**  **]**    **for** i**,** each\_point\_to\_merge **in** to\_merge**.**iterrows**():**  distances\_to\_remove **=** distances**[**  **((**distances**[**"cx1"**]** **==** each\_point\_to\_merge**[**"cx"**])** **&**  **(**distances**[**"cy1"**]** **==** each\_point\_to\_merge**[**"cy"**])** **&**  **(**distances**[**"cz1"**]** **==** each\_point\_to\_merge**[**"cz"**]))** **|**  **((**distances**[**"cx2"**]** **==** each\_point\_to\_merge**[**"cx"**])** **&**  **(**distances**[**"cy2"**]** **==** each\_point\_to\_merge**[**"cy"**])** **&**  **(**distances**[**"cz2"**]** **==** each\_point\_to\_merge**[**"cz"**]))**  **]**  distances**.**drop**(**distances\_to\_remove**.**index**,** inplace **=** **True)**    data\_points**.**loc**[**to\_merge**.**index**,** **[**"cx"**]]** **=** to\_merge**[**"x"**].**mean**()**  data\_points**.**loc**[**to\_merge**.**index**,** **[**"cy"**]]** **=** to\_merge**[**"y"**].**mean**()**  data\_points**.**loc**[**to\_merge**.**index**,** **[**"cz"**]]** **=** to\_merge**[**"z"**].**mean**()**    other\_clusters **=** data\_points**.**copy**()**  other\_clusters**.**drop**(**index**=**to\_merge**.**index**,** inplace**=True)**    other\_clusters **=** **(**other\_clusters**[[**"cx"**,** "cy"**,** "cz"**]]).**drop\_duplicates**()**  **for** j**,** each\_non\_neighbors **in** other\_clusters**.**iterrows**():**  non\_neighbors **=** data\_points**.**copy**()**  non\_neighbors **=** non\_neighbors**[(**non\_neighbors**[**"cx"**]** **==** each\_non\_neighbors**[**"cx"**])** **&**  **(**non\_neighbors**[**"cy"**]** **==** each\_non\_neighbors**[**"cy"**])** **&**  **(**non\_neighbors**[**"cz"**]** **==** each\_non\_neighbors**[**"cz"**])]**  cluster\_distance **=** find\_cluster\_distance**(**  to\_merge**[[**"x"**,** "y"**,** "z"**]].**values**.**tolist**(),** non\_neighbors**.**values**.**tolist**(),** **type** **=** distance\_type**)**  distances **=** distances**.**append**(**  **{**  "cx1"**:** to\_merge**[**"x"**].**mean**(),**  "cy1"**:** to\_merge**[**"y"**].**mean**(),**  "cz1"**:** to\_merge**[**"z"**].**mean**(),**  "cx2"**:** each\_non\_neighbors**[**"cx"**],**  "cy2"**:** each\_non\_neighbors**[**"cy"**],**  "cz2"**:** each\_non\_neighbors**[**"cz"**],**  "distance"**:** cluster\_distance  **},** ignore\_index **=** **True**  **)**  **return** data\_points  k **=** 2  test\_hierarchical **=** remove\_outliers**(**test**.**copy**(),** d\_distance **=** 50**,** p\_fraction **=** 0.15**)**  clusterized\_hierarchical **=** hierarchical\_agglomerative**(**test\_hierarchical**.**copy**().**head**(**10**),** k**,** distance\_type **=** "min"**)**  clusterized\_hierarchical |

Graphical user interface, text, application

Description automatically generated

Let's check how this small clustering looks like

|  |
| --- |
| clusterized\_hierarchical**[[**"cx"**,** "cy"**,** "cz"**,** "potential\_outlier"**]].**groupby**([**"cx"**,** "cy"**,** "cz"**]).**count**()** |

Graphical user interface, text, application

Description automatically generated

Now let's test the function but using distances DataFrame already calculated (this will be valuable when comparing for different distance types later) and for the four types of distance

|  |
| --- |
| distances **=** initialize\_distances**(**test\_hierarchical**.**copy**().**head**(**10**))**  distances**.**head**()** |

Graphical user interface, text, application

Description automatically generated

Starting with min distance

|  |
| --- |
| k **=** 2  test\_hierarchical **=** remove\_outliers**(**test**.**copy**(),** d\_distance **=** 50**,** p\_fraction **=** 0.15**)**  hierarchical\_agglomerative**(**test\_hierarchical**.**head**(**10**),** k**,** distance\_type **=** "min"**,**  distances **=** distances**.**copy**())** |

Graphical user interface, text, application

Description automatically generated

Using max distance

|  |
| --- |
| hierarchical\_agglomerative**(**test\_hierarchical**.**copy**().**head**(**10**),** 2**,** distance\_type **=** "max"**,**  distances **=** distances**.**copy**())** |

Graphical user interface, text, application

Description automatically generated

Using average distance

|  |
| --- |
| hierarchical\_agglomerative**(**test\_hierarchical**.**copy**().**head**(**10**),** 2**,** distance\_type **=** "avg"**,**  distances **=** distances**.**copy**())** |

Graphical user interface, application

Description automatically generated

And finally using the center distance

|  |
| --- |
| hierarchical\_agglomerative**(**test\_hierarchical**.**copy**().**head**(**10**),** 2**,** distance\_type **=** "center"**,**  distances **=** distances**.**copy**())** |

Graphical user interface, text, application

Description automatically generated

As a last test before we put all the pieces together, let's try with the whole dataset (already without the outliers)

|  |
| --- |
| test\_hierarchical **=** remove\_outliers**(**test**.**copy**(),** d\_distance **=** 50**,** p\_fraction **=** 0.15**)**  distances **=** initialize\_distances**(**test\_hierarchical**.**copy**())**  distances |

Graphical user interface, text, application

Description automatically generated

|  |
| --- |
| k **=** 10  test\_hierarchical **=** hierarchical\_agglomerative**(**  test\_hierarchical**.**copy**(),** k**,** distance\_type **=** "min"**,** distances **=** distances**.**copy**())**  test\_hierarchical |

Text

Description automatically generated with medium confidence

We can also plot it

|  |
| --- |
| fig **=** plt**.**figure**(**figsize **=** **(**16**,** 8**))**  ax **=** plt**.**axes**(**projection **=**"3d"**)**  ax**.**xaxis**.**pane**.**fill **=** **False**  ax**.**yaxis**.**pane**.**fill **=** **False**  ax**.**zaxis**.**pane**.**fill **=** **False**  my\_cmap **=** plt**.**get\_cmap**(**"gist\_rainbow\_r"**)**  ax**.**scatter3D**(**test\_hierarchical**[**"x"**],**  test\_hierarchical**[**"y"**],**  test\_hierarchical**[**"z"**],**  cmap **=** my\_cmap**,**  c **=** **(**test\_hierarchical**[**"cy"**]** **+** test\_hierarchical**[**"cz"**]** **+** test\_hierarchical**[**"cx"**]))** |

Chart, scatter chart

Description automatically generated

**Note**: the plot seems a little bit odd, but we will address the reasons later on conclusions

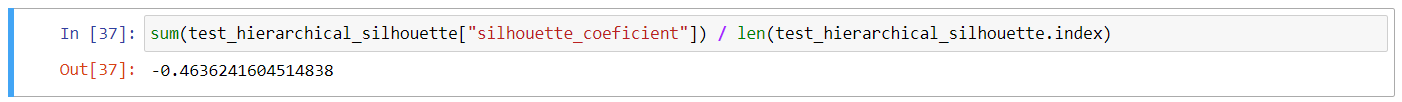
And calculate the silhouette coefficient

|  |
| --- |
| test\_hierarchical\_silhouette **=** calculate\_silhouette**(**test\_hierarchical**.**copy**())**  test\_hierarchical\_silhouette**.**tail**()** |

Graphical user interface, text, application

Description automatically generated

|  |
| --- |
| **sum(**test\_hierarchical\_silhouette**[**"silhouette\_coeficient"**])** **/** **len(**test\_hierarchical\_silhouette**.**index**)** |



**Putting everything together**

Let's generate a new set of data points to work with a fresh dataset

|  |
| --- |
| test **=** np**.**random**.**randint**(**0**,** 100**,** size**=(**500**,** 3**))**  test **=** pd**.**DataFrame**(**test**,** columns **=** **[**"x"**,** "y"**,** "z"**])** |



And start by creating a function to plot the data points, with or without clustering

|  |
| --- |
| **def** plot\_data\_points**(**data\_points**,** chart\_title **=** **None):**  fig **=** plt**.**figure**(**figsize **=** **(**16**,** 8**))**  ax **=** plt**.**axes**(**projection **=**"3d"**)**  ax**.**xaxis**.**pane**.**fill **=** **False**  ax**.**yaxis**.**pane**.**fill **=** **False**  ax**.**zaxis**.**pane**.**fill **=** **False**  my\_cmap **=** plt**.**get\_cmap**(**"gist\_rainbow\_r"**)**  **if** "cy" **in** data\_points**.**columns**:**  ax**.**scatter3D**(**data\_points**[**"x"**],** data\_points**[**"y"**],** data\_points**[**"z"**],**  cmap **=** my\_cmap**,** c **=** **(**data\_points**[**"cy"**]** **+** data\_points**[**"cz"**]** **+** data\_points**[**"cx"**]))**  **else:**  ax**.**scatter3D**(**data\_points**[**"x"**],** data\_points**[**"y"**],** data\_points**[**"z"**])**  **if** chart\_title **is** **not** **None:**  plt**.**title**(**chart\_title**,** size**=**15**)**    plot\_data\_points**(**test**.**copy**(),** "test title"**)** |

Chart, scatter chart

Description automatically generated

And another function to plot the outliers

|  |
| --- |
| **def** plot\_outliers**(**data\_points**,** d\_distance **=** 100**,** p\_fraction **=** 0.15**,** chart\_title **=** **None):**  outliers **=** DB\_outlier\_detection**(**data\_points**.**values**.**tolist**(),** d\_distance **=** 50**,** p\_fraction **=** 0.15**)**  outliers **=** pd**.**DataFrame**(**outliers**,** columns **=** **[**"coordinates"**,** "potential\_outlier"**])**  outliers**[**"coordinates\_str"**]** **=** **[**','**.**join**(map(str,** l**))** **for** l **in** outliers**[**"coordinates"**]]**  outliers**[[**"x"**,** "y"**,** "z"**]]** **=** outliers**[**"coordinates\_str"**].str.**split**(**","**,** expand**=True)**  outliers**[[**"x"**,** "y"**,** "z"**]]** **=** outliers**[[**"x"**,** "y"**,** "z"**]].**apply**(**pd**.**to\_numeric**)**  outliers **=** outliers**[[**"x"**,** "y"**,** "z"**,** "potential\_outlier"**]]**  outliers**[**"color"**]** **=** outliers**[**"potential\_outlier"**].**apply**(lambda** row**:** "Red" **if** row **else** "Blue"**)**    fig **=** plt**.**figure**(**figsize **=** **(**16**,** 8**))**  ax **=** plt**.**axes**(**projection **=**"3d"**)**  ax**.**xaxis**.**pane**.**fill **=** **False**  ax**.**yaxis**.**pane**.**fill **=** **False**  ax**.**zaxis**.**pane**.**fill **=** **False**  ax**.**scatter3D**(**outliers**[**"x"**],** outliers**[**"y"**],** outliers**[**"z"**],** facecolors **=** outliers**[**"color"**])**  outlier\_label **=** **(**  "Potential Outliers: " **+**  **str(len(**outliers**[**outliers**[**"potential\_outlier"**]** **==** **True]))** **+**  " out of " **+**  **str(len(**outliers**))**  **)**  red\_patch **=** mpatches**.**Patch**(**color**=**"red"**,** label**=**outlier\_label**)**  ax**.**legend**(**handles**=[**red\_patch**],** bbox\_to\_anchor**=(**1.5**,** .5**),** loc**=**"center right"**)**  **if** chart\_title **is** **not** **None:**  plt**.**title**(**chart\_title**,** size**=**15**)**    plot\_outliers**(**test**.**copy**(),** 50**,** .15**)** |

For the sake of clarity, we are only providing the lower piece of the screenshot

Chart, scatter chart

Description automatically generated

Finally, we create a function to perform all the activities developed:

1. Remove outliers
2. Generate K-means clustering and calculate its silhouette
3. Initialize the distances Pandas DataFrame
4. Generate the Hierarchical Agglomerative clustering using nearest, farthest, average and center measures and calculate their silhouette

This function receives:

* **data\_points**: a list of 3-D vectors containing all datapoints
* **d\_distance**: the distance threshold to be used for outliers detection
* **p\_fraction**: the fraction of neighbors to be considered in the radius for outliers detection
* **k**: the number of clusters to be generated

and generates a Pandas DataFrame comparing the silhoutte coefficient for the five clustering (K-means and Hierarchical Agglomerative using the four distance types). It also plots the following charts before outputs the Pandas DataFrame:

* The initial data points
* The data points with the outliers annotated
* All the five clustering results

|  |
| --- |
| **def** clustering\_analysis**(**data\_points**,** d\_distance **=** 100**,** p\_fraction **=** 0.15**,** k **=** 10**):**  plot\_data\_points**(**data\_points**,** "Initial Data Points"**)**  plot\_outliers**(**data\_points**,** d\_distance**,** p\_fraction**,** "Outliers Detection"**)**  data\_points **=** remove\_outliers**(**data\_points**.**copy**(),** d\_distance **=** d\_distance**,** p\_fraction **=** p\_fraction**)**  data\_points\_kmeans **=** k\_means**(**data\_points**.**copy**(),** k**)**  plot\_data\_points**(**data\_points\_kmeans**,** chart\_title **=** "K-Means clusterin"**)**  data\_points\_kmeans\_silhouette **=** calculate\_silhouette**(**data\_points\_kmeans**.**copy**())**  clear\_output**()**  **print(**"Initializing distances DataFrame"**)**  distances **=** initialize\_distances**(**data\_points**.**copy**())**  data\_points\_hierarchical\_min **=** hierarchical\_agglomerative**(**  data\_points**.**copy**(),** k**,** distance\_type **=** "min"**,** distances **=** distances**.**copy**())**  plot\_data\_points**(**  data\_points\_hierarchical\_min**.**copy**(),**  chart\_title **=** "Hierarchical Agglomerative using distance between nearest two points"**)**  data\_points\_hierarchical\_min\_silhouette **=** calculate\_silhouette**(**  data\_points\_hierarchical\_min**.**copy**())**  data\_points\_hierarchical\_max **=** hierarchical\_agglomerative**(**  data\_points**.**copy**(),** k**,** distance\_type **=** "max"**,** distances **=** distances**.**copy**())**  plot\_data\_points**(**  data\_points\_hierarchical\_max**,**  chart\_title **=** "Hierarchical Agglomerative using distance between farthest two points"**)**  data\_points\_hierarchical\_max\_silhouette **=** calculate\_silhouette**(**data\_points\_hierarchical\_max**.**copy**())**  data\_points\_hierarchical\_avg **=** hierarchical\_agglomerative**(**  data\_points**.**copy**(),** k**,** distance\_type **=** "avg"**,** distances **=** distances**.**copy**())**  plot\_data\_points**(**  data\_points\_hierarchical\_avg**,**  chart\_title **=** "Hierarchical Agglomerative using agerage distance between clusters"**)**  data\_points\_hierarchical\_avg\_silhouette **=** calculate\_silhouette**(**data\_points\_hierarchical\_avg**.**copy**())**  data\_points\_hierarchical\_center **=** hierarchical\_agglomerative**(**  data\_points**.**copy**(),** k**,** distance\_type **=** "center"**,** distances **=** distances**.**copy**())**  plot\_data\_points**(**  data\_points\_hierarchical\_center**,**  chart\_title **=** "Hierarchical Agglomerative using distance between the center of clusters"**)**  data\_points\_hierarchical\_center\_silhouette **=** calculate\_silhouette**(**data\_points\_hierarchical\_center**.**copy**())**  clear\_output**()**  result **=** **[]**  result**.**append**({**  "method"**:** "K-means"**,**  "silhouette"**:** **(sum(**data\_points\_kmeans\_silhouette**[**"silhouette\_coeficient"**])** **/**  **len(**data\_points\_kmeans\_silhouette**.**index**))**  **})**    result**.**append**({**  "method"**:** "Hierarchical Agglomerative using nearest distance between clusters"**,**  "silhouette"**:** **(sum(**data\_points\_hierarchical\_min\_silhouette**[**"silhouette\_coeficient"**])** **/**  **len(**data\_points\_hierarchical\_min\_silhouette**.**index**))**  **})**    result**.**append**({**  "method"**:** "Hierarchical Agglomerative using farthest distance between clusters"**,**  "silhouette"**:** **(sum(**data\_points\_hierarchical\_max\_silhouette**[**"silhouette\_coeficient"**])** **/**  **len(**data\_points\_hierarchical\_max\_silhouette**.**index**))**  **})**  result**.**append**({**  "method"**:** "Hierarchical Agglomerative using average distance between clusters"**,**  "silhouette"**:** **(sum(**data\_points\_hierarchical\_avg\_silhouette**[**"silhouette\_coeficient"**])** **/**  **len(**data\_points\_hierarchical\_avg\_silhouette**.**index**))**  **})**  result**.**append**({**  "method"**:** "Hierarchical Agglomerative using cluster center distance"**,**  "silhouette"**:** **(sum(**data\_points\_hierarchical\_center\_silhouette**[**"silhouette\_coeficient"**])** **/**  **len(**data\_points\_hierarchical\_center\_silhouette**.**index**))**  **})**  **return** pd**.**DataFrame**(**result**)** |

From now on, we conduct experiments using fresh new datasets and we also change the parameters (d, p and k) for each one of them

**Experiment #1**

For this experiment we used the same parameters we were using so far:

* d\_distance = 50
* p\_fraction = 0.15
* k = 10

|  |
| --- |
| experiment\_1 **=** np**.**random**.**randint**(**0**,** 100**,** size**=(**500**,** 3**))**  experiment\_1 **=** pd**.**DataFrame**(**experiment\_1**,** columns **=** **[**"x"**,** "y"**,** "z"**])**  experiment\_1 **=** clustering\_analysis**(**experiment\_1**.**copy**(),** d\_distance **=** 50**,** p\_fraction **=** 0.15**,** k **=** 10**)**  experiment\_1**.**sort\_values**(**by**=[**"silhouette"**],** ascending**=False)** |

Here are how the initial data points (still with the outliers) look like:

Chart, scatter chart

Description automatically generated

And here are the detected outliers (45 out of 500):

Chart, scatter chart

Description automatically generated

The best accommodation for this dataset was using K-means, with a silhouette coefficient of 0.271881:

Graphical user interface, text, application, Word

Description automatically generated

Here are all the clusterings generated ordered from the highest silhouette to the lowest:

Chart, scatter chart

Description automatically generated Chart, scatter chart, bubble chart

Description automatically generated Chart, scatter chart, bubble chart

Description automatically generated Chart, scatter chart, bubble chart

Description automatically generated Chart, scatter chart, bubble chart

Description automatically generated

**Experiment #2**

For this experiment we changed the parameters, as below:

* d\_distance = 40
* p\_fraction = 0.11
* k = 6

|  |
| --- |
| experiment\_2 **=** np**.**random**.**randint**(**0**,** 100**,** size**=(**500**,** 3**))**  experiment\_2 **=** pd**.**DataFrame**(**experiment\_2**,** columns **=** **[**"x"**,** "y"**,** "z"**])**  experiment\_2 **=** clustering\_analysis**(**experiment\_2**.**copy**(),** d\_distance **=** 40**,** p\_fraction **=** 0.11**,** k **=** 6**)**  experiment\_2**.**sort\_values**(**by**=[**"silhouette"**],** ascending**=False)** |

Here are how the initial data points (still with the outliers) look like:

Chart, scatter chart

Description automatically generated

And here are the detected outliers (35 out of 500):

Chart, scatter chart

Description automatically generated

The best accommodation for this dataset was (again) using K-means, with a silhouette coefficient of 0.303377:

Graphical user interface, application, Word

Description automatically generated

Here are all the clusterings generated ordered from the highest silhouette to the lowest:

Chart, scatter chart, bubble chart

Description automatically generated Chart, scatter chart

Description automatically generated Chart, scatter chart

Description automatically generated Chart, scatter chart

Description automatically generated Chart, scatter chart

Description automatically generated

**Experiment #3**

For this experiment we change again the parameters, as below:

* d\_distance = 60
* p\_fraction = 0.2
* k = 14

|  |
| --- |
| experiment\_3 **=** np**.**random**.**randint**(**0**,** 100**,** size**=(**500**,** 3**))**  experiment\_3 **=** pd**.**DataFrame**(**experiment\_3**,** columns **=** **[**"x"**,** "y"**,** "z"**])**  experiment\_3 **=** clustering\_analysis**(**experiment\_3**.**copy**(),** d\_distance **=** 60**,** p\_fraction **=** 0.2**,** k **=** 14**)**  experiment\_3**.**sort\_values**(**by**=[**"silhouette"**],** ascending**=False)** |

Here are how the initial data points (still with the outliers) look like:

Chart, scatter chart

Description automatically generated

And here are the detected outliers (41 out of 500):

Chart, scatter chart

Description automatically generated

The best accommodation for this dataset, one more time, was using K-means, with a silhouette coefficient of 0.256358. But now, the Hierarchical Agglomerative clustering using average distance was really close (0.256118 for silhouette coefficient):

Graphical user interface, application, Word

Description automatically generated

Here are all the clusterings generated ordered from the highest silhouette to the lowest:

Chart, scatter chart, bubble chart

Description automatically generated Chart, scatter chart

Description automatically generated Chart, scatter chart

Description automatically generated Chart, scatter chart, bubble chart

Description automatically generated Chart, scatter chart

Description automatically generated

**Conclusion**

As we can see not just during the experiments, but as well during the development of the program above, K-means is always better, for this type of data set, than Hierarchical Agglomerative. This is probably caused because K-means keeps re-assigning the data points to their closest clusters at each round, while Agglomerative keeps them into the first cluster they were assigned, with no "re-calibration".

Another factor we noticed is that the higher the K used in K-means, the higher the time taken to process and rounds to converge. This may not be noticed in the example above, with K only between 5 and 15 but during the development we have tested up to 50 clusters and the trend line for time and rounds always goes up with K.

On the other hand, Hierarchical Agglomerative has an opposite trend: the higher the K, the lower the time to conclude. This is caused mainly because K in this case acts somehow as an early-stop in agglomerating.

With these two observations, we can say that, although K-means always performed better than Hierarchical Agglomerative, the resources used may be a fact of decision and one may decide between the trade-off of K-means better clustering vs Hierarchical Agglomerative speed (for high number of K=clusters).

On Hierarchical Agglomerative, we came to the conclusion that the min distance is the worse for this type of dataset, because it usually keeps agglomerating to an existing cluster for this amount of data points. That is why on the plots almost all data points belong to one cluster while the rest of them form clusters of very few data points.

The other types of distances generated clusters with good silhouette coefficients (at least above 0), with one or another being better than others, depending on the data points randomly generated. One thing to notice is that for small number of data points, sometimes two of them and even the three of them converged to the same clustering. That never happened with min distance, though.

The Jupyter Notebook containing the codes, along with the databases, can be also found in GitHub: <https://github.com/wellingtoncunha/data_mining/tree/master/mid_term_project>.