CMPSC 497

Fall 2022 Programming Assignment 2 Report | Bert Yan | Due: 12/2/22

Task 1. Data Preprocessing

1(a) | Please calculate new features Remaining Balance = Bill Amount – Pay Amount and add them (6 features/columns in total) to the dataset. You may use excel or write code for the task. Please show the result of the first 3 records in your report.

We can easily accomplish this using a simple loop.

```
# First we Create and add Remaining balance (REM_AMT) = Bill Amt - Pay Amt
for i in range(1, 7):
    data['REM_AMT'+str(i)] = data['BILL_AMT'+str(i)] - data['PAY_AMT'+str(i)]
```

now when we check our dataset again:

```
# Now, check out new features
data.head()
```

output:

	ID	LIMIT_BAL	SEX	EDUCATION	MARRIAGE	AGE	PAY_0	PAY_2	PAY_3	PAY_4	...	PAY_AMT4	PAY_AMT5	PAY_AMT6	default payment next month	REM_AMT1	REM_AMT2	REM_AMT3	REM_AMT4	REM_AMT5	REM_AMT6
0	5771	180000	1	2	1	38	1	2	2	0	...	124	0	0	1	826	4412	4071	4947	5195	0
1	3999	20000	2	1	2	23	2	2	2	2	...	500	0	1000	1	6236	12559	11459	12457	13104	11744
2	397	100000	1	1	2	38	0	0	0	0	...	0	579	0	0	38308	41453	41491	43011	42432	28947
3	15467	220000	1	1	2	30	2	0	0	0	...	4000	4000	5000	0	79630	82604	85487	86889	87892	88815
4	16482	240000	2	4	1	59	0	0	0	0	...	10000	5000	20000	0	225561	225403	226564	231075	141440	112290

5 rows × 31 columns

```
#Next, we modify our dataset to only contain 18 categories: BILL_AMT, PAY_AMT,
and REM_AMT
categories = []

for i in range(1, 7):
    categories.append('BILL_AMT'+str(i))
    categories.append('PAY_AMT'+str(i))
    categories.append('REM_AMT'+str(i))

dataset = pd.DataFrame(data, columns=categories)
```

Now we have 18 numerical features to establish our project on.
6 BILL_AMT
6 PAY_AMT
6 REM_AMT

1(b) | As 18 attributes are used for clustering (which is quite many), please apply Principal Component Analysis (PCA) from sklearn to reduce the dimensionality of attributes to 3 (i.e., n_components=3). Note that normalizing the attributes by standardization before PCA is a common practice. Please perform the standardization in this task. Please show the result of the first 3 records in your report

First, normalize our attributes by ***standardization*** first, prior to PCA:

```
# Standarization for part b)
standardScaler = pp.StandardScaler()
dataset = pd.DataFrame(standardScaler.fit_transform(dataset), columns=categories)
dataset.head()
```

	BILL_AMT1	PAY_AMT1	REM_AMT1	BILL_AMT2	PAY_AMT2	REM_AMT2	BILL_AMT3	PAY_AMT3	REM_AMT3	BILL_AMT4	PAY_AMT4	REM_AMT4	BILL_AMT5	PAY_AMT5	REM_AMT5	BILL_AMT6	PAY_AMT6	REM_AMT6
0	-0.661096	-0.216712	-0.614507	-0.627844	-0.237323	-0.532575	-0.612107	-0.274410	-0.544000	-0.596510	-0.294516	-0.523682	-0.580101	-0.310517	-0.500643	-0.656632	-0.291441	-0.563376
1	-0.566391	-0.127640	-0.539944	-0.512951	-0.237323	-0.420515	-0.498639	-0.245236	-0.437813	-0.473376	-0.270893	-0.406337	-0.449185	-0.310517	-0.369521	-0.441494	-0.235534	-0.366951
2	-0.140504	-0.186903	-0.097912	-0.091372	-0.196410	-0.023087	-0.068645	-0.253405	-0.006167	-0.004104	-0.302306	0.071076	0.045858	-0.273432	0.116706	-0.167963	-0.291441	-0.079222
3	0.460074	-0.038568	0.471607	0.542549	-0.040939	0.542932	0.612279	-0.058520	0.626181	0.743475	-0.050996	0.756679	0.854974	-0.054315	0.870385	0.927107	-0.011905	0.922103
4	2.506150	0.160776	2.482897	2.626895	0.163668	2.507089	2.725092	0.279902	2.653862	3.088519	0.325970	3.009616	1.757894	0.009736	1.758154	1.576624	0.826702	1.314735

Here we are using sklearn.preprocessing's provided StandardScaler to accomplish this and to transform our dataframe to fit our standardization.

Secondly, we can apply PCA to reduce our dimensionality to 3:

```
pca = PCA(n_components=3)
standardData = pd.DataFrame(pca.fit_transform(dataset))
standardData.head()
```

And as requested, we can observe the first 3 records as follows:

	0	1	2
0	-2.098892	-0.332516	-0.076863
1	-1.650147	-0.298458	-0.108874
2	-0.213882	-0.537101	-0.109666

Note here our dataset is new variable: standardData.

1(c) | Before you do the clustering, please perform normalization (into [0,1]) on the three attributes used for clustering. Describe the normalization scheme you adopt and show the result of the first 3 records in your report.

As reasons were listed in previous programming assignment, we continue to adopt the ***MinMaxScaler*** scheme for our normalization to conform our data into the range of [0,1] of our 3 attributes.

```
normScaler = pp.MinMaxScaler()
normalData = pd.DataFrame(normScaler.fit_transform(standardData))
normalData.head()
```

	0	1	2
0	0.076566	0.035675	0.371407
1	0.090043	0.036034	0.370660
2	0.133176	0.033519	0.370642

Dataset from this point on will be “normalData” (standardized and normalized)

Task 2. (Your own) Implementation of the Agglomerative Hierarchical Clustering (MIN) algorithm

2(a) | Represent the data points in a cluster using a data structure or alternative method. The data structure should encode the hierarchical structure of data points in the cluster. Discuss/show your design and illustrate it by examples in the report. If you do not use a data structure in your design, explain your idea and show your design (with illustration by examples) to represent a cluster in your implementation.

Firstly, for our agglomerative hierarchical clustering algorithm, we need to use normalData.values so that we have a format closer to that of a sklearn’s dataset, while we are not using any sklearn to accomplish our method, we do need a format of our data points to be handleable.

In our “preparation” function in the agglomerative class:

```
    def preparation(self):
        # create distance matrix for each data
        if self.__metric == 'euclidean':
            self.__distance_matrix = euclidean_distances(self.__pointer_data)
        elif self.__metric == 'manhattan':
```

```
            self.__distance_matrix = manhattan_distances(self.__pointer_data)
        elif self.__metric == 'cosine':
            self.__distance_matrix = cosine_distances(self.__pointer_data)

        n_data = len(self.__pointer_data)
        self.__clusters = [0] * len(self.__pointer_data)
        # create singleton cluster
        for i in range(0, n_data):
            self.__memory.append([i])
```

we are representing the data points in the following manner:

```
        n_data = len(self.__pointer_data)
        self.__clusters = [0] * len(self.__pointer_data)
        # create singleton cluster
        for i in range(0, n_data):
            self.__memory.append([i])
```

This way, we can append our created cluster to this data structure following our metric calculations with the distance matrix.

2(b) | Proximity matrix is essential to Agglomerative Hierarchical Clustering algorithm. In the report, explain your choice of the similarity or dissimilarity measure in the proximity matrix and show/explain the module you code to calculate/construct the matrix.

Since we are asked to create a clustering algorithm that is single linkage, therefore we will adopt ***Euclidean*** as our distance calculation metric.

$$d(\mathbf{p}, \mathbf{q}) = \sqrt{\sum_{i=1}^{n} (q_i - p_i)^2}$$

Here is the code in a separate testing code:

```
import math

def distance(p, q):
    return math.sqrt(sum([(pi - qi)**2 for pi, qi in zip(p, q)]))

def single_link(ci, cj):
    return min([distance(vi, vj) for vi in ci for vj in cj])
```

Given that custom codes here is no more advantage than the provided Euclidean calculation from sklearn, either is acceptable.

In order to calculate thee distance amongst the clusters or between data and clusters, we use the following function to achieve the calculation:

```
    def cluster_distance(self, a, b):
        distance = 0
        if self.__linkage == 'single': # distance between two nearest data in
cluster
            distance = self.__distance_matrix[a[0]][b[0]]
            for i in a:
                for j in b:
                    if distance > self.__distance_matrix[i][j]:
            for i in a:
                for j in b:
                    distance += self.__distance_matrix[i][j]
            distance = distance/(len(a) * len(b))
        return distance
```

And as mentioned in previous part, we use the following distance matrix for each data point:

```
        if self.__metric == 'euclidean':
            self.__distance_matrix = euclidean_distances(self.__pointer_data)
        elif self.__metric == 'manhattan':
            self.__distance_matrix = manhattan_distances(self.__pointer_data)
        elif self.__metric == 'cosine':
            self.__distance_matrix = cosine_distances(self.__pointer_data)
```

2(c) | Find two closest clusters based on the MIN/Single Link scheme. In the report, show and explain the module of your implementation code for it

In order to find the two closest clusters based on the min/single linkage scheme, we developed the following function to accomplish this:

```
    def clustering(self):
        # iteration to produce n_clusters
        for iteration in range(len(self.__pointer_data) - self.__n_clusters):
            cluster_a = 0
            cluster_b = 1
            if (len(self.__memory[0])==1) and (len(self.__memory[1])==1):
                # distance between singleton cluster
                min_distance = self.__distance_matrix[0][1]
            else:
                # distance between multi-elements cluster
                min_distance = self.cluster_distance(self.__memory[0],
self.__memory[1])

            # find two nearest data or clusters to be combined into one cluster
```

```
            for i in range(len(self.__memory)):
                for j in range(len(self.__memory)):
                    if (i != j) and (i < j):
                        if (len(self.__memory[i])==1) and
(len(self.__memory[j])==1):
                            # distance between singleton cluster
                            dist = self.__distance_matrix[i][j]
                        else:
                            # distance between multi-elements cluster
                            dist = self.cluster_distance(self.__memory[i],
self.__memory[j])
                        if dist < min_distance:
                            cluster_a = i
                            cluster_b = j
                            min_distance = dist
            temp = self.__memory.pop(cluster_b)
            for i in range(len(temp)):
                self.__memory[cluster_a].append(temp[i])
```

Please refer to the enclosed comments for navigation.

2(d) |Merge two clusters into a new cluster and update the proximity matrix. In the report, show/explain the module(s) of your code for implementation of the above operations. Also use an example (based on the data structure of your design in Task 2(a) to illustrate the merged cluster

As part of the function demonstrated above, the section of codes here showcases how we are merging two clusters into a new cluster and updating the proximity matrix.

Here, for example, using the data structure from Task 2(a), we note that self._memory is used as the structure to host the clustering and the calculations of the distances 1) distance between singleton cluster, 2) distance between multi-element cluster. Finally, we use a temp variable to house the cluster that is void of cluster_b, which is no longer following our distance calculations, we will append new cluster information to self._memory[cluster_a].

```
            # find two nearest data or clusters to be combined into one cluster
            for i in range(len(self.__memory)):
                for j in range(len(self.__memory)):
                    if (i != j) and (i < j):
                        if (len(self.__memory[i])==1) and
(len(self.__memory[j])==1):
                            # distance between singleton cluster
                            dist = self.__distance_matrix[i][j]
                        else:
```

```
                            # distance between multi-elements cluster
                            dist = self.cluster_distance(self.__memory[i],
self.__memory[j])
                        if dist < min_distance:
                            cluster_a = i
                            cluster_b = j
                            min_distance = dist
            temp = self.__memory.pop(cluster_b)
            for i in range(len(temp)):
                self.__memory[cluster_a].append(temp[i])
```

2(e) |In this task, you are asked to implement your own version of the Agglomerative Hierarchical Clustering (MIN) algorithm in python. Given a dataset and a parameter n_clusters, your code should return the specified number of clusters, each of which represented in the data structure of your design. Additionally, a label (i.e., 1, 2, ..., n_clusters) should be assigned to every data point in the dataset. In this task, please use the first 100 data points from Task 1 and n_clusters = 5 to generate your clustering result

After introducing the above functions and the init function as well as a couple of helper functions (please refer to the Jupyter notebook for detailed documentation), now we can showcase our algorithm running to produce 5 clusters with the first 100 data points from Task 1 (normalData):

```
# Results: running custom method on first 100 data points

model = agglomerative(metric='euclidean', linkage='single', n_clusters=5)
predict = model.fit_predict(normalData.values[:100])
print("Labels prediction: \n", predict)

clusters = model.get_clusters()
print("\nWe have the following clusters: \n", clusters)
```

```
Labels prediction:
 [0, 0, 0, 0, 1, 0, 0, 2, 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, 3, 4, 0, 0, 0, 0, 0, 3, 0, 0, 4, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0]

We have the following clusters:
 [[0, 45, 30, 74, 82, 10, 15, 53, 81, 52, 16, 55, 32, 37, 59, 90, 18, 71, 33, 19, 76, 40, 47, 17, 58, 20, 64, 98, 6, 25, 65, 67, 69, 75, 77, 43, 51, 31, 41, 49, 57, 87, 84, 88, 1, 13, 62, 94, 60, 93, 61, 91, 66, 70,
36, 44, 68, 72, 21, 24, 26, 34, 39, 2, 11, 28, 14, 54, 73, 78, 12, 50, 8, 48, 92, 83, 97, 95, 99, 56, 85, 38, 46, 35, 42, 22, 27, 9, 63, 96, 3, 23, 29, 5], [4], [7], [79, 86], [80, 89]]
```

In case the screenshot is not clear, here is the copied version:

Labels prediction:

[0, 0, 0, 0, 1, 0, 0, 2, 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, 3, 4, 0, 0, 0, 0, 0, 3, 0, 0, 4, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0]

We have the following clusters:

[[0, 45, 30, 74, 82, 10, 15, 53, 81, 52, 16, 55, 32, 37, 59, 90, 18, 71, 33, 19, 76, 40, 47, 17, 58, 20, 64, 98, 6, 25, 65, 67, 69, 75, 77, 43, 51, 31, 41, 49, 57, 87, 84, 88, 1, 13, 62, 94, 60, 93, 61, 91, 66, 70, 36, 44, 68, 72, 21, 24, 26, 34, 39, 2, 11, 28, 14, 54, 73, 78, 12, 50, 8, 48, 92, 83, 97, 95, 99, 56, 85, 38, 46, 35, 42, 22, 27, 9, 63, 96, 3, 23, 29, 5], [4], [7], [79, 86], [80, 89]]

We have labeled the generated clustering to 0, 1, 2, 3, 4 to denote the five clusters. And in the clusters generated, we find one particular large cluster with 4 small cluster:

Cluster 1: [0, 45, 30, 74, 82, 10, 15, 53, 81, 52, 16, 55, 32, 37, 59, 90, 18, 71, 33, 19, 76, 40, 47, 17, 58, 20, 64, 98, 6, 25, 65, 67, 69, 75, 77, 43, 51, 31, 41, 49, 57, 87, 84, 88, 1, 13, 62, 94, 60, 93, 61, 91, 66, 70, 36, 44, 68, 72, 21, 24, 26, 34, 39, 2, 11, 28, 14, 54, 73, 78, 12, 50, 8, 48, 92, 83, 97, 95, 99, 56, 85, 38, 46, 35, 42, 22, 27, 9, 63, 96, 3, 23, 29, 5]

Cluster 2: [4]

Cluster 3; [7]

Cluster 4: [79, 86]

Cluster 5: [80, 89]

Task 3. Hierarchical Clustering

3(a) | Perform the agglomerative hierarchical clustering (MIN) to cluster the data obtained from Task 1. Our goal is to extract five largest clusters for further analysis later. To visually decide those clusters, please draw dendrograms (with sufficient levels) for inspection. Based on the decision made from your inspection, re-run the clustering to obtain the clustering. You may use the parameters distance_threshold or n_clusters in sklearn.cluster.AgglomerativeClustering to decide the clustering, but note that n_cluster = 5 does not necessarily give you the five largest clusters.

i) Explanation with illustration and Justification

```
from sklearn.cluster import AgglomerativeClustering

# Hierarchical clustering using average linkage
min = AgglomerativeClustering(n_clusters=5, linkage='single')
```

```
# Fit & predict
min_pred = min.fit_predict(normalData.values)
```

Upon attempting to examining the dendrogram, we observes the following error:

Canceled future for execute_request message before replies were done

The Kernel crashed while executing code in the the current cell or a previous cell. Please review the code in the cell(s) to identify a possible cause of the failure. Click here for more info. View Jupyter log for further details.

Investigating this issue I found that using single linkage for the sklearn's AgglomerativeClustering would cause the issue described here:

cluster.hierarchy.linkage with method='single' causes infinite recursion. #7512 (https://github.com/scipy/scipy/issues/7512)

also:

Single linkage clustering fails for large datasets #11513

(https://github.com/scikit-learn/scikit-learn/issues/11513)

To replicate this issue:

Steps/Code to Reproduce

```
import numpy as np
import sklearn.cluster

data = np.random.normal(size=(64000,2))
clusterer = sklearn.cluster. AgglomerativeClustering(linkage='single').fit(data)
```

Expected Results

clusterer is assigned a trained single linkage clustering instance.

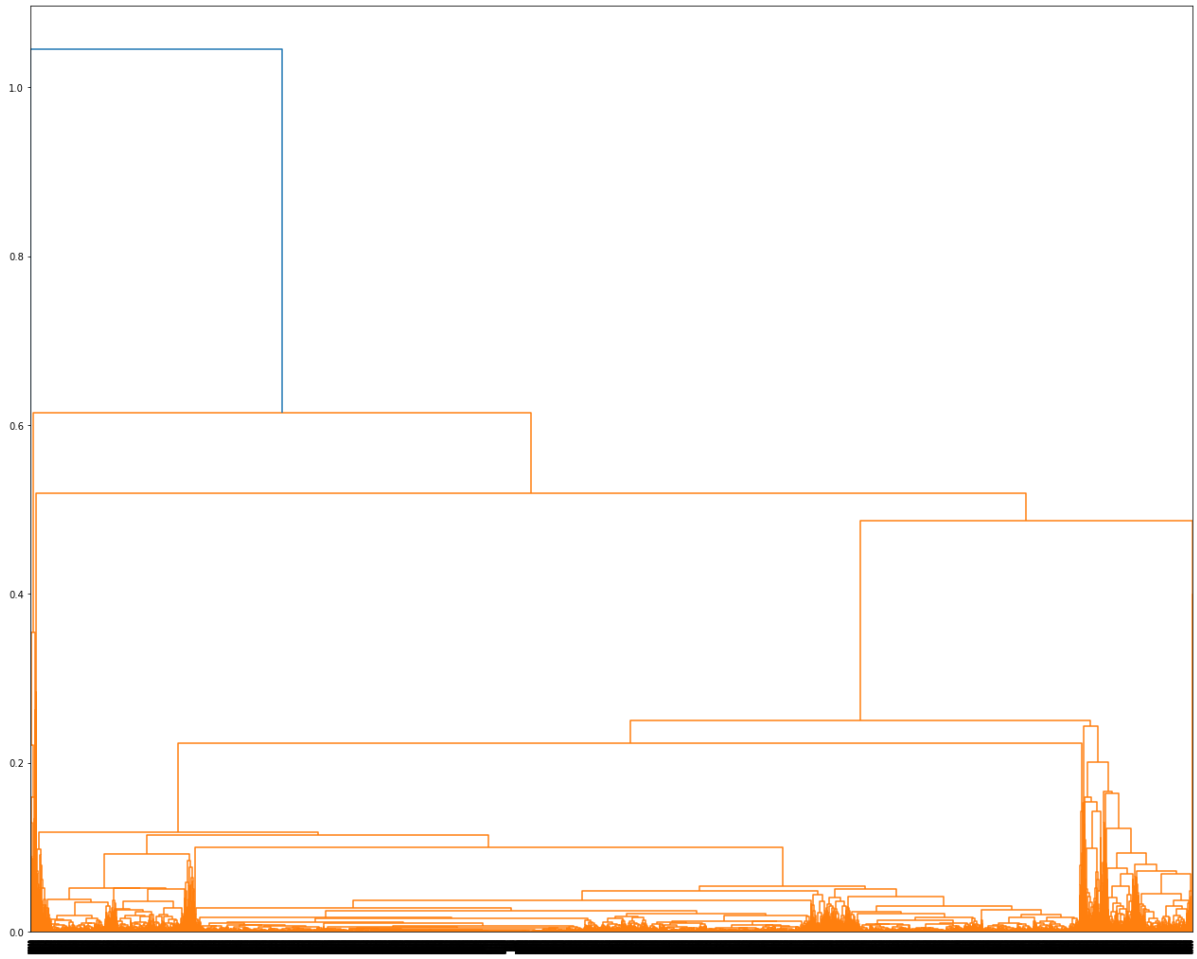
Actual Results

On my mac laptop this simply freezes the whole machine. On Linux a MemoryError with no traceback results. If the (lack of) error is not reproducible on your machine simply make data a larger array.

Single linkage clustering fails for sufficiently large data arrays. This is due to issues in the scipy single linkage clustering.

A potential workaround is given here: https://github.com/amueller/information-theoretic-ms, however due to the nature that we are not supposed to use outside source other than the what's already mentioned, I will leave it at that, and the solution given is quite elegant, using MST to help and contribute to cut down the complexity.

In light of this issue, we will take a look at the dendrogram formed for the average linkage since we know that single linkage is fast, and can perform well on non-globular data, but it performs poorly in the presence of noise and large dateset. But average linkage perform well on cleanly separated globular clusters.



Here is the dendrogram for the average linkage, which is characteristic by the distribution of the leaves and the clades.

ii) list of clusters extracted and for each of those clusters, please show the first 5 data points and the total number of data points in the cluster.

Using the above implementation of AgglomerativeClustering provided by sklearn, we find the clusters to be

```
print("Cluster 1: \n", np.where(min_pred == 0))
print("\nCluster 2: \n", np.where(min_pred == 1))
print("\nCluster 3: \n", np.where(min_pred == 2))
print("\nCluster 4: \n", np.where(min_pred == 3))
print("\nCluster 5: \n", np.where(min_pred == 4))
```

Cluster 1:

(array([2561, 16914], dtype=int64),)

Only 2 data points, [2561, 16914]

Cluster 2:

(array([0, 1, 2, ..., 23997, 23998, 23999], dtype=int64),)

23994 data points, first 5 data points [0, 1, 2, 3, 4, 5]

Cluster 3:

(array([865], dtype=int64),)

Only 1 data point, [865]

Cluster 4:

(array([9814], dtype=int64),)

Only 1 data point, [9814]

Cluster 5:

(array([1780, 21829], dtype=int64),)

Only 2 data points, [1780, 21829]

3(b) | Repeat Task 3(a) using the agglomerative hierarchical clustering (MAX).

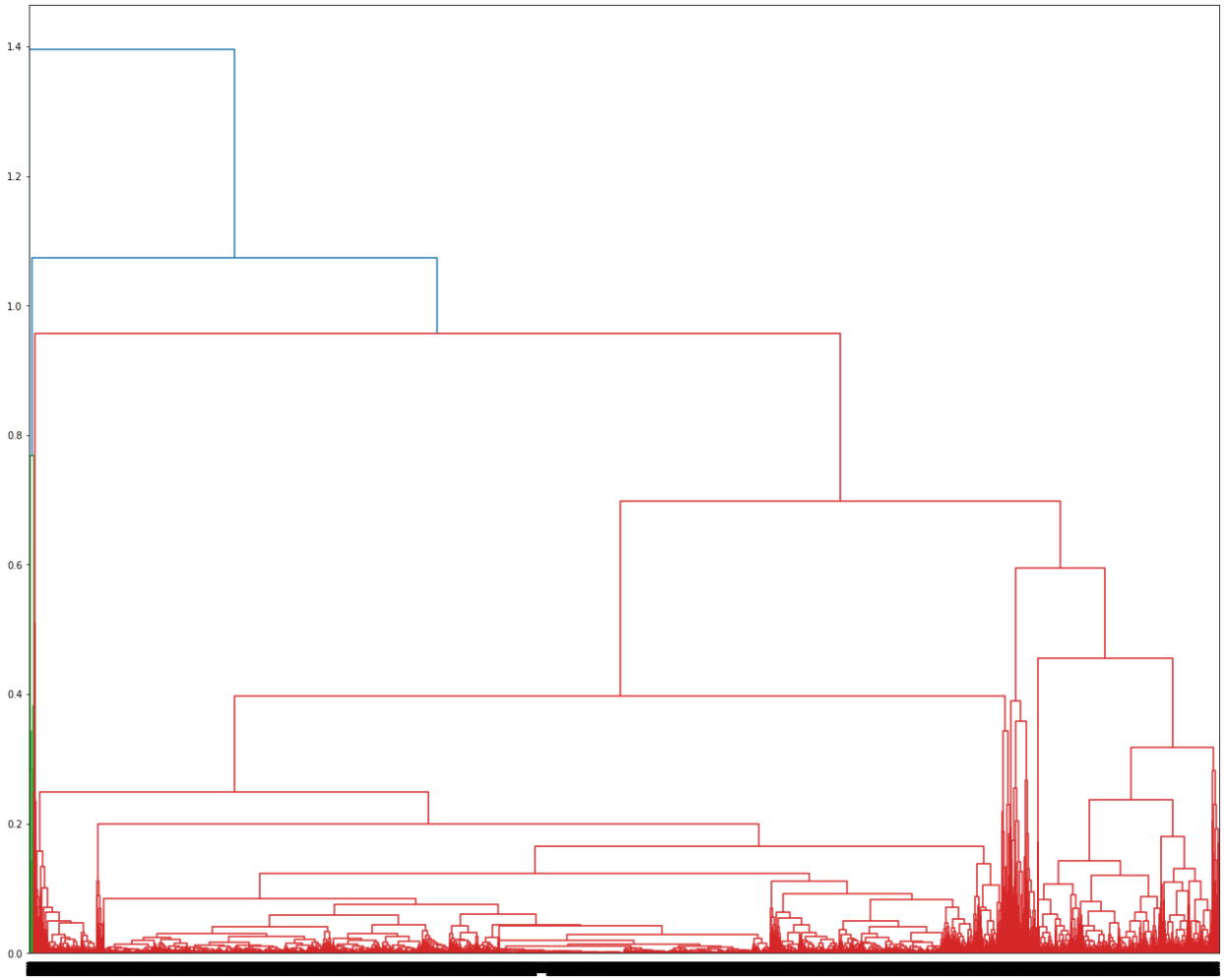
Contrary to the complication faced from single linkage dendrogram, we proceed without issue for Max and Ward linkages.

i) Explanation with illustration and Justification

```
# Hierarchical clustering using complete linkage
max = AgglomerativeClustering(n_clusters=5, linkage='complete')
# Fit & predict
max_pred = max.fit_predict(normalData.values)
```

Using the complete linkage, or rather, max linkage, we are finding the link between two clusters that contains all element pairs, and the distance between clusters equals the distance between those two elements (one in each cluster) that are farthest away from each other, opposite from what we observed from single linkage, or min linkage.

We can observe the following dendrogram:



ii) list of clusters extracted and for each of those clusters, please show the first 5 data points and the total number of data points in the cluster.

```
print("Cluster 1: \n", np.where(ward_pred == 0))
print("\nCluster 2: \n", np.where(ward_pred == 1))
print("\nCluster 3: \n", np.where(ward_pred == 2))
print("\nCluster 4: \n", np.where(ward_pred == 3))
print("\nCluster 5: \n", np.where(ward_pred == 4))
```

Cluster 1:

(array([0, 1, 2, ..., 23997, 23998, 23999], dtype=int64),)

Cluster 2:

(array([2561, 9814, 16914], dtype=int64),)

Cluster 3:

(array([959, 5070, 8869, 11082, 15532, 17727, 17729, 19622, 21171,
21712, 21852], dtype=int64),)

Cluster 4:

(array([62, 1409, 1780, 2515, 2620, 3303, 5509, 6664, 7361,
7453, 8527, 8775, 9157, 10972, 11131, 11875, 12987, 13291,
13338, 13872, 14670, 16244, 16372, 16982, 17509, 18245, 18620,
20032, 21079, 21312, 21829, 22725, 23917], dtype=int64),)

Cluster 5:

(array([98, 1002, 1087, 1165, 1801, 1805, 2004, 2115, 2793,
2832, 2875, 3504, 3816, 4678, 5311, 5767, 5844, 5941,
6307, 6409, 6920, 7054, 8409, 8602, 8994, 9192, 9407,
9498, 9558, 9701, 10109, 10300, 10512, 10826, 10991, 11352,

11739, 12206, 13438, 13572, 13590, 14015, 14369, 14698, 14783,

14986, 15822, 15968, 16059, 16824, 16939, 17337, 17432, 17543,

17591, 17963, 18359, 18688, 18707, 19172, 19531, 19662, 20485,

20828, 21013, 21291, 21488, 22105, 22143, 22150, 22371, 22802,

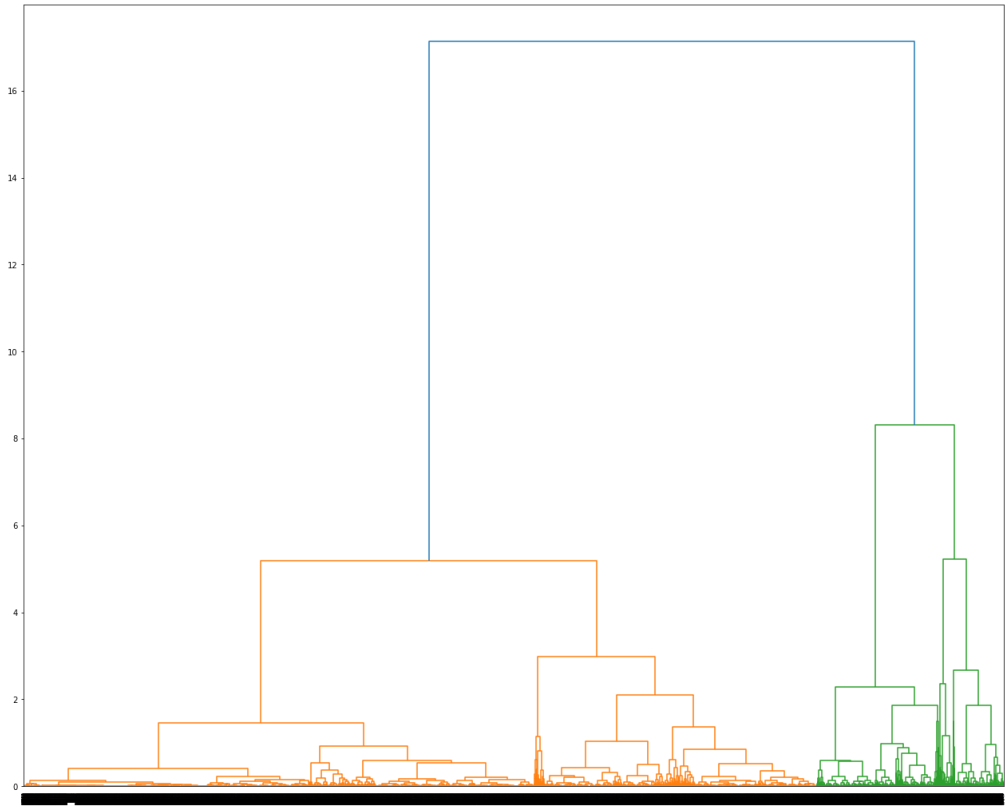
22913, 23023, 23231, 23336, 23472, 23487], dtype=int64),)

3 (c) | Repeat Task 3(a) using the agglomerative hierarchical clustering (WARD).

i) Explanation with illustration and Justification

```
# Import scipy's linkage function to conduct the clustering
from scipy.cluster.hierarchy import linkage
linkage_type = 'ward' # Ward has the highest adjusted_rand_score
linkage_matrix = linkage(normalData, linkage_type)
```

```
from scipy.cluster.hierarchy import dendrogram
import matplotlib.pyplot as plt
plt.figure(figsize=(22,18))
dendrogram(linkage_matrix)
plt.show()
```



ii) list of clusters extracted and for each of those clusters, please show the first 5 data points and the total number of data points in the cluster.

Cluster 1:

(array([2, 5, 7, ..., 23978, 23991, 23992], dtype=int64),)

Cluster 2:

(array([4, 62, 83, ..., 23925, 23956, 23979], dtype=int64),)

Cluster 3:

(array([3, 35, 39, ..., 23996, 23997, 23999], dtype=int64),)

Cluster 4:

(array([86, 98, 202, 410, 466, 523, 553, 576, 609,
649, 661, 853, 913, 957, 983, 1002, 1087, 1111,
1150, 1165, 1171, 1175, 1177, 1337, 1338, 1449, 1466,
1482, 1625, 1720, 1723, 1801, 1805, 2004, 2014, 2030,
2115, 2133, 2169, 2196, 2261, 2396, 2499, 2604, 2750,
2761, 2793, 2798, 2799, 2808, 2816, 2832, 2842, 2875,
2994, 3103, 3119, 3125, 3482, 3504, 3537, 3620, 3715,
3749, 3816, 3866, 3974, 3994, 4041, 4167, 4174, 4481,
4494, 4524, 4564, 4668, 4678, 4688, 4757, 4781, 5145,
5311, 5467, 5495, 5513, 5610, 5681, 5753, 5754, 5767,
5844, 5941, 6008, 6029, 6047, 6052, 6072, 6074, 6077,
6221, 6307, 6409, 6437, 6528, 6623, 6759, 6824, 6920,
7054, 7082, 7115, 7190, 7257, 7316, 7351, 7397, 7531,
7736, 7800, 7846, 8082, 8120, 8279, 8409, 8509, 8587,
8602, 8622, 8690, 8723, 8761, 8785, 8817, 8838, 8898,

8899, 8943, 8949, 8994, 9139, 9192, 9351, 9407, 9498,
9558, 9672, 9701, 9835, 9864, 9914, 9924, 10065, 10071,
10109, 10238, 10264, 10300, 10306, 10512, 10755, 10782, 10808,
10826, 10860, 10944, 10991, 11012, 11015, 11105, 11325, 11352,
11481, 11573, 11655, 11739, 11929, 12023, 12206, 12230, 12261,
12278, 12331, 12468, 12571, 12639, 12720, 12768, 12857, 13137,
13160, 13375, 13391, 13438, 13484, 13500, 13572, 13590, 13834,
13844, 13849, 13851, 14015, 14213, 14233, 14346, 14369, 14646,
14690, 14697, 14698, 14779, 14783, 14868, 14946, 14986, 15053,
15068, 15156, 15250, 15305, 15322, 15513, 15611, 15618, 15715,
15822, 15825, 15968, 16059, 16082, 16199, 16272, 16291, 16417,
16488, 16501, 16585, 16648, 16696, 16744, 16824, 16843, 16868,
16890, 16939, 16965, 17042, 17047, 17075, 17087, 17106, 17192,
17247, 17311, 17337, 17370, 17384, 17432, 17464, 17532, 17543,
17591, 17625, 17630, 17690, 17782, 17963, 18116, 18143, 18188,
18303, 18310, 18359, 18451, 18664, 18688, 18701, 18707, 18749,
18794, 18879, 19014, 19073, 19172, 19184, 19261, 19388, 19531,
19575, 19662, 19732, 20035, 20143, 20302, 20303, 20485, 20554,
20611, 20713, 20738, 20828, 20889, 20966, 21013, 21041, 21076,
21291, 21313, 21336, 21461, 21483, 21488, 21518, 21586, 21668,
21808, 21824, 21907, 22034, 22046, 22098, 22105, 22118, 22143,
22150, 22160, 22176, 22239, 22318, 22329, 22371, 22419, 22549,
22646, 22684, 22802, 22852, 22872, 22913, 22947, 22997, 23023,
23170, 23212, 23226, 23231, 23264, 23336, 23340, 23355, 23463,
23472, 23487, 23536, 23683, 23702], dtype=int64),)

Cluster 5:

(array([0, 1, 6, ..., 23993, 23995, 23998], dtype=int64),)

3 (d) | Compare and report what you find via the process of performing Task 3(a)-(c). Which clustering looks better than others based on what you observed so far? Note that it's all right if your findings at this point is different from your later conclusion.

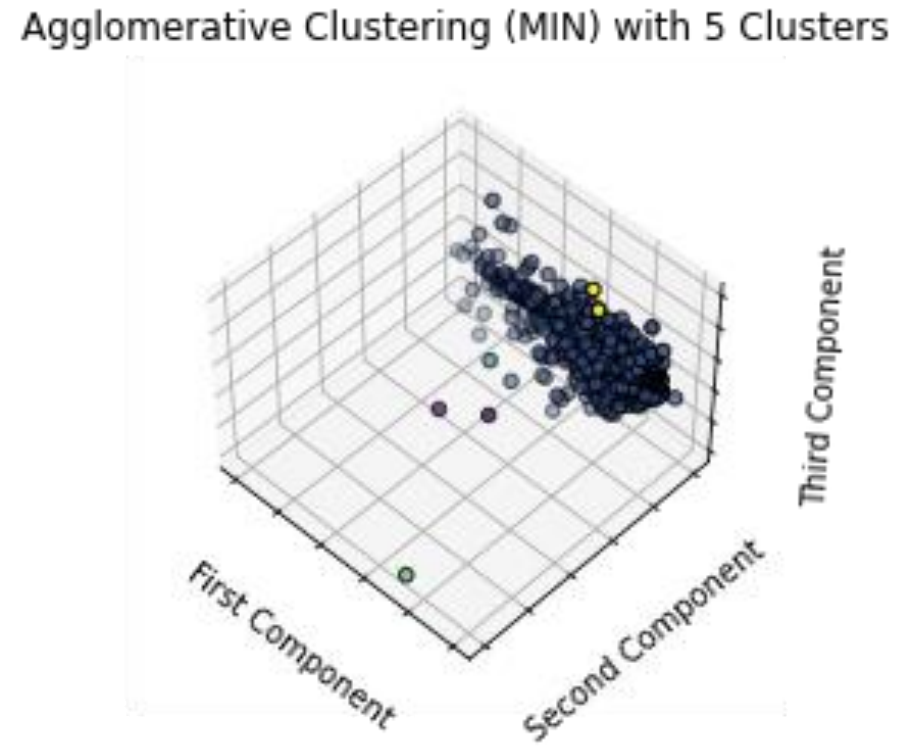
We find that **ward clustering** is the best one out of the three as it generated 5 clusters that each contained more than 5 data points and can be used for the following studies. Unlike the others. Instead of measuring the distance directly, it analyzes the variance of clusters. As learned from the materials, ward's is said to be the most suitable method for quantitative variables, which is advantageous in our use case.

Task 4. Task 4. Cluster Analysis

4(a) | Given the three hierarchical clusterings (i.e., from MIN, MAX and WARD, compute and report their SSE (sum of squared errors). Additionally, show the three clustering results using matplotlib.

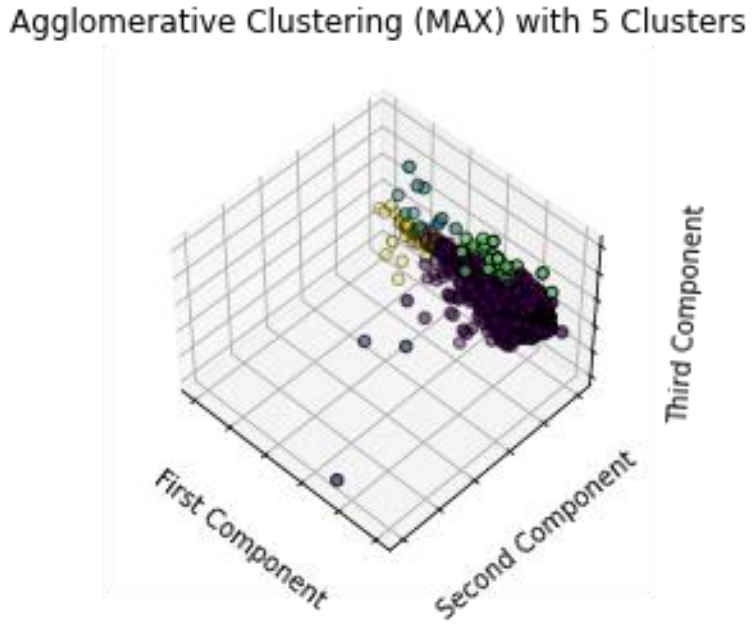
MIN:

Min linkage clustering SSE: [2229.1892284371283]



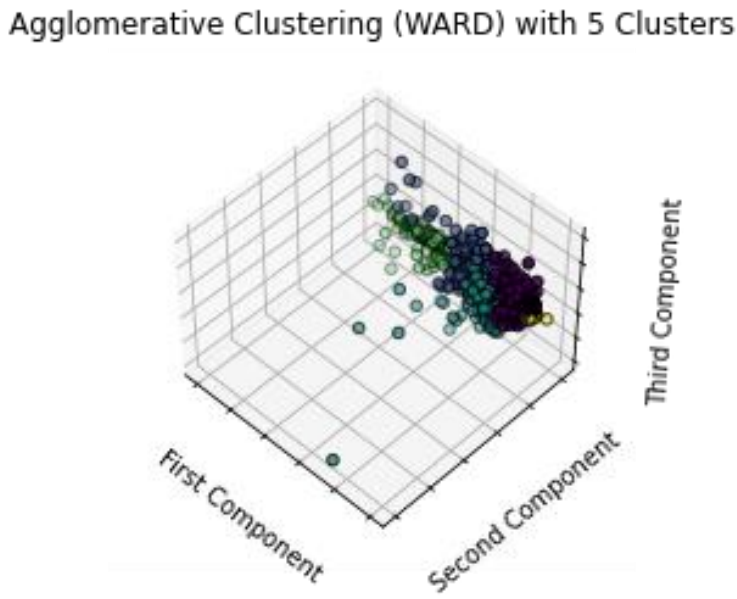
MAX:

Max linkage clustering SSE: [2127.715976670422]



WARD:

Ward linkage clustering SSE: [624.540166663375]



i) We observe that out of the three clustering schemes, ***ward*** generates the least SSE, since for SSE, the lower the better, we draw conclusions that WARD will be the best clustering.

ii) We can observe that the three matplotlib-generated plots indicate similar shapes to the clustering. Single linkage method is prone to "chain" and form clusters of irregulars, often thread-like curved shapes. Since we know, with this method, at any step, two clusters are merged if their closest edges are close enough. No proximity between other parts of the two clusters is taken into consideration. For the three clusters we see they are mostly tied, 5 clusters for each clustering scheme with complete and ward showing the most variations in distribution.

iii) Since we know that SSE is a good measure if we are trying to find spherically shaped clusters, however, given the clusters generated using matplotlib, we observe no spherically shaped clusters. This could be the case if other techniques had been installed. Also, since ward linkage is also known as MISSQ, which is the Minimal Increase of Sum-of-Squares, based on the SSE results, with 624.54, Ward has the least SSE, statement could be made that due to the metric of ward linkage, SSE could be in favor of Ward linkage.

4(b) | Apply external measures, specifically GINI Index, entropy, and percentage of defaults, on the default status to assess the quality of clustering. Which measure is a better tool? Which measure can reveal more interesting information under the context of this application?

1. GINI Index:

```
# First, let's find Gini index of Default status of each cluster
def GINI(data, labels, default, k):
    gini = [1] * k
    clustered = data.copy()
    clustered['LABEL'] = labels
    clustered['DEFAULT'] = default
    for i in range(k):
        cluster = clustered[clustered['LABEL'] == i]
        defaults = len(cluster[cluster['DEFAULT'] == 1])
        if(len(cluster) == 0):
            gini[i] = -1
        else:
            gini[i] -= ((float(defaults) / len(cluster))**2) + ((1 -
float(defaults) / len(cluster))**2)
    return gini
```

```
# Gini index for each of the clusters: MIN
GINI(normalData, singleLabels, data['default payment next month'], 5)
```
Output: [0.0, 0.346696104161352, 0.0, 0.0, 0.0]

```
# Gini index for each of the clusters: MAX
GINI(normalData, completeLabels, data['default payment next month'], 5)
```
Ouptut:

[0.3463505057427154,

0.0,

0.49586776859504145,

0.2975206611570249,

0.41584483892176205]

```
# Gini index for each of the clusters: WARD
GINI(normalData, wardLabels, data['default payment next month'], 5)
```
Output:

[0.34510824635045245,

0.3238656591650413,

0.318949025363262,

0.39900265118040656,

0.35421381483103764]

As discussed in Programming Assignment 1, the range of Entropy lies in between 0 to 1 and the range of Gini Impurity lies in between 0 to 0.5. Hence, we can conclude that Gini Impurity is better as compared to entropy for selecting the best features. We observe that amongst the clusters, ward seems to have the most lowest GINI indices. And since GINI index calculates the amount of probability of a specific feature that is classified incorrectly when selected randomly. If all the elements are linked with a single class then it is called pure. It means an attribute with a lower Gini index should be preferred. Therefore, we can see that Ward is preferred from our GINI results. However, if counting clusters that had outputted 0.0 as gini index, A gini score of 0 is the most pure score possible, which likely points to probable actionable issue, or lack of data points in that cluster as demonstrated in previous task.

2. Entropy:

```
# Secondly, we calculate entropy
from scipy.stats import entropy
from math import log, e
import timeit

def ENTROPY(labels, base=None):
  """ Computes entropy of label distribution. """

  n_labels = len(labels)

  if n_labels <= 1:
    return 0

  value,counts = np.unique(labels, return_counts=True)
  probs = counts / n_labels
  n_classes = np.count_nonzero(probs)

  if n_classes <= 1:
    return 0

  ent = 0.

  # Compute entropy
  base = e if base is None else base
  for i in probs:
    ent -= i * log(i, base)

  return ent
```

```
# Entropy for single/min linkage
ENTROPY(singleLabels)
```

Output: 0.0026558964946346766

```
# Entropy for complete/max linkage
ENTROPY(completeLabels)
```

Output: 0.037521638680803715

```
# Entropy for ward linkage
ENTROPY(wardLabels)
```

Output: 1.1721481292552052

Since we know from lecture and past assignments that a high entropy means low information gain, and a low entropy means high information gain. Information gain can be thought of as the purity in a system: the amount of clean knowledge available in a system.

```
# Entropy for single/min linkage
```

0.0026558964946346766

```
# Entropy for complete/max linkage
```

0.037521638680803715

```
# Entropy for ward linkage
```

1.1721481292552052

We observe that single linkage has the lowest entropy, which is preferred over high entropy, the results indicate that single linkage leads to the most accurate the decision/prediction we can make. Min/Single linkage > Max/Complete > Ward

3. Percentage of Defaults

```
# Thirdly, we find the Percentage of defaults in each cluster
def POD(data, labels, default, k):
    pod = [0] * k
    clustered = data.copy()
    clustered['LABEL'] = labels
    clustered['DEFAULT'] = default
    for i in range(k):
        cluster = clustered[clustered['LABEL'] == i]
        defaults = len(cluster[cluster['DEFAULT'] == 1])
        if(len(cluster) == 0):
            pod[i] = -1
        else:
            pod[i] = (float(defaults) / len(cluster))
    return pod
```

```
# Percentage of defaults for single/min linkage
POD(normalData, singleLabels, data['default payment next month'], 5)
```

Output: [0.0, 0.22313911811286155, 0.0, 0.0, 0.0]

```
# Percentage of defaults for complete/max linkage
POD(normalData, completeLabels, data['default payment next month'], 5)
```

Output:

[0.22282722513089004,

0.0,

0.45454545454545453,

0.18181818181818182,

0.2948717948717949]

```
# Percentage of defaults for ward linkage
POD(normalData, wardLabels, data['default payment next month'], 5)
```

Output:

[0.22170900692840648,

0.20323886639676114,

0.1991254624957955,

0.2752808988764045,

0.23001279181323953]

Like the situation encountered from GINI index calculation, we see that ward provides the most comprehensive results, likely due to the abundance of actionable data points produced by Ward linkage as demonstrated in previous task. And like GINI, we can observe that Ward has lowest values amongst the three (if overlooking the 0.0 entries for clustering), besides cluster[3].

From the results of the GINI, entropy and the percentage, we can get a better understanding of the impacts of different techniques used in clustering. It appears that while GINI and Ward reveals the lack of data points from clustering earlier, entropy calculation would show the most variation amongst the clustering techniques.

4(c) | Upon the clustering selected based on the best SSE, perform data analysis of clusters for comparison, e.g., in terms of the characteristics and distribution of data points in those clusters. In the report, show your result and findings with explanation. Illustrations with tables and figures are excellent.

From previous task, we have found that the clustering selected based on the best SSE is the Ward Linkage which uses the "wardLabels". We will proceed to clustering analysis using thee ward labels.

Since the goal of the cluster analysis is to characterize and understand better the clients, we will proceed to analyze through other attributes such as marriage, sex, and education, in addition to the defaults as discussed in previous task.

For better understanding, we retrieve the mean balance over the 6 remaining balance data, for AGE attribute, we convert the age to specific label, i.e., Label1: age⩽30; Label2: 30<age⩽40: Label3: 40<age⩽50; Label4: 50<age⩽60; Label5: 60<age. This way it provides a more comprehensive look to our findings.

Firstly, SEX attribute, we proceed with our cluster analysis with GINI index:

Data type SEX

	cluster0	cluster1	cluster2	cluster3	cluster4
Impurity (Gini Index)	0.48	0.49	0.47	0.49	0.48
label1(local)	40.647%	43.320%	38.446%	43.539%	38.871%
label2(local)	59.353%	56.680%	61.554%	56.461%	61.129%
label1(global)	29.61%	5.63%	12.02%	1.63%	51.12%
label2(global)	28.38%	4.83%	12.63%	1.39%	52.77%

Then, EDUCATION attribute:

Data type EDUCATION

	cluster0	cluster1	cluster2	cluster3	cluster4
Impurity (Gini Index)	0.62	0.62	0.61	0.61	0.63
label1(local)	28.334%	37.976%	32.560%	48.315%	38.823%
label2(local)	51.155%	46.802%	50.925%	38.202%	43.676%
label3(local)	19.096%	12.551%	14.295%	11.236%	16.182%
label4(local)	0.245%	0.567%	0.471%	0.281%	0.512%
label5(local)	1.025%	1.862%	1.480%	1.685%	0.592%
label6(local)	0.144%	0.243%	0.269%	0.281%	0.160%
label1(global)	23.29%	5.56%	11.49%	2.04%	57.62%
label2(global)	31.54%	5.14%	13.48%	1.21%	48.62%
label3(global)	33.35%	3.91%	10.71%	1.01%	51.02%
label4(global)	16.50%	6.80%	13.59%	0.97%	62.14%
label5(global)	32.57%	10.55%	20.18%	2.75%	33.94%
label6(global)	23.81%	7.14%	19.05%	2.38%	47.62%
label0(local)	0	0	0	0	0.056%
label0(global)	0	0	0	0	100.00%

MARRIAGE attribute:

Data type MARRIAGE

	cluster0	cluster1	cluster2	cluster3	cluster4
Impurity (Gini Index)	0.51	0.51	0.51	0.50	0.51
label0(local)	0.144%	0	0.034%	0	0.256%
label1(local)	43.894%	49.312%	47.494%	50.562%	45.659%
label2(local)	54.734%	50.040%	51.900%	49.438%	52.934%
label3(local)	1.227%	0.648%	0.572%	0	1.151%
label0(global)	23.26%	0	2.33%	0	74.42%
label1(global)	27.76%	5.56%	12.89%	1.64%	52.14%
label2(global)	29.74%	4.85%	12.10%	1.38%	51.93%
label3(global)	33.46%	3.15%	6.69%	0	56.69%

Finally, we calculated the mean balance of the clusters:

The mean balance of cluster 0 is 37337.76224499615

The mean balance of cluster 1 is 182500.06666666665

The mean balance of cluster 2 is 98408.85486041036

The mean balance of cluster 3 is 320719.26451310865

The mean balance of cluster 4 is 4892.7793945208405

From this, we can observe that cluster 4 has the least mean balance while the cluster 3 has the most mean balance.

In order to find out how data on previous table affect means balance. I compared cluster 3 and 4 on sex, education, marriage, and age.

For sex and marriage, cluster 3 and 4 show drastic difference in terms of label(Global). While the GINI index shows no significant disparity.

The difference between cluster 3 and cluster 4 also appears on age and education. Cluster 0 has the most percentage of sample with age <= 30 and less percentage of sample with age from 50 to 60 (label4 and label5).

For education, similar pattern ensues, we find label(Global) to to be the highest for cluster 4 in all attributes.

Age attribute:

Data type Age

	cluster0	cluster1	cluster2	cluster3	cluster4
Impurity (Gini Index)	0.70	0.68	0.70	0.69	0.71
label1(local)	35.826%	20.891%	28.591%	15.449%	32.099%
label2(local)	34.714%	45.587%	41.002%	45.787%	36.888%
label3(local)	20.626%	24.049%	21.897%	25.843%	22.194%
label4(local)	7.809%	7.692%	7.366%	10.955%	7.659%
label5(local)	1.025%	1.781%	1.144%	1.966%	1.159%
label1(global)	32.40%	3.37%	11.10%	0.72%	52.42%
label2(global)	26.83%	6.28%	13.60%	1.82%	51.47%
label3(global)	27.24%	5.66%	12.41%	1.75%	52.93%
label4(global)	29.21%	5.13%	11.83%	2.11%	51.73%
label5(global)	25.45%	7.89%	12.19%	2.51%	51.97%

Now let's see how age and education level affect means balance:

```
data type Age
mean balance of label 0 if 0
mean balance of label 1 if 34683.92308529154
mean balance of label 2 if 42560.17354603602
mean balance of label 3 if 40606.82319669527
mean balance of label 4 if 42147.95410367177
mean balance of label 5 if 49387.17682198326
data type Education
mean balance of label 0 if 0
mean balance of label 1 if 34683.92308529154
mean balance of label 2 if 42560.17354603602
mean balance of label 3 if 40606.82319669527
mean balance of label 4 if 42147.95410367177
mean balance of label 5 if 49387.17682198326
```

Now we can find out how age and education level affect means balance, we will proceed to calculate means balance with different age range and education level.

data type Age

	cluster0	cluster1	cluster2	cluster3	cluster4
label0	0.000000	0.000000	0.000000	0.000000	0.000000
label1	38250.613820	178916.199612	97397.286863	304379.839394	6239.567331
label2	36681.770132	182456.874186	99176.116626	315220.080777	3964.013654
label3	36626.578027	183049.277217	97794.118024	327332.398551	4502.601165
label4	37452.424522	190160.382456	98317.827245	343595.068376	5009.604558
label5	41087.246479	185141.409091	108546.191176	362786.500000	3852.695402

data type Education

	cluster0	cluster1	cluster2	cluster3	cluster4
label0	0.000000	0.000000	0.000000	0.000000	1599.071429
label1	34926.290457	181075.269012	97369.079029	327013.688953	2948.035729
label2	38060.839682	185414.020185	98662.034126	313041.948529	6218.803893
label3	38705.905266	177867.602151	99911.747843	317860.983333	6056.446229
label4	47928.294118	177704.000000	102044.833333	277662.500000	1929.221354
label5	39913.309859	165624.326087	97086.943182	353188.861111	5676.858108
label6	37155.416667	223737.222222	97373.916667	244763.666667	4844.958333

We can observe that following the calculation of the mean balance over whole dataset, the mean balance will increase with age and education level.

4(d) | In Task 1(c), the data points are normalized. Is it really better than clustering without preprocessing with normalization? Please use data without normalization to find the best clustering

```
# Now let's run without normalizing
for i in range(1, 7):
    categories.append('BILL_AMT'+str(i))
    categories.append('PAY_AMT'+str(i))
    categories.append('REM_AMT'+str(i))
dataset2 = pd.DataFrame(data, columns=categories)
dataset2.head()
```

We still do PCA to reduce our dimensionality to 3.

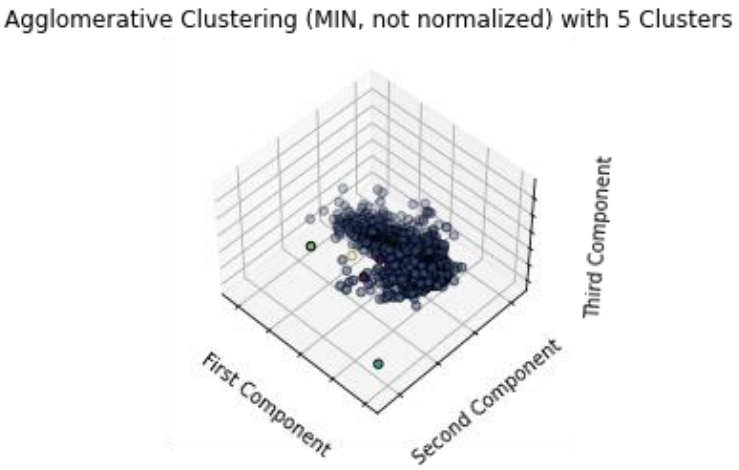
```
# we still do data preprocessing: Peform PCA to reduce dimensionality to 3
pca = PCA(n_components=3)
not_normalData = pd.DataFrame(pca.fit_transform(dataset2))
not_normalData.head()
```

Without normalization to limit the ranges of the data points, we see a greater, more sparse data distribution, which is extra noise we will encounter for the clustering.

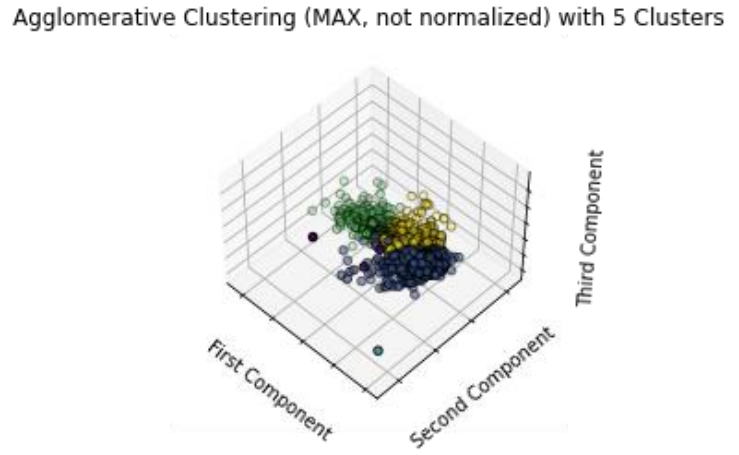
	0	1	2
0	-234900.072834	979.704734	-1030.543185
1	-186011.468525	11715.173306	-7312.021001
2	-17343.912388	8389.925093	-2407.235079
3	265624.057583	60063.369200	-24593.986187
4	957129.792929	-94589.741588	94351.151487

For best comparison of results, we will rerun the SSE calculation and AgglomerativeClustering with MIN/Single, MAX/Complete, and Ward clustering with matlidplot to demonstrate thee results.

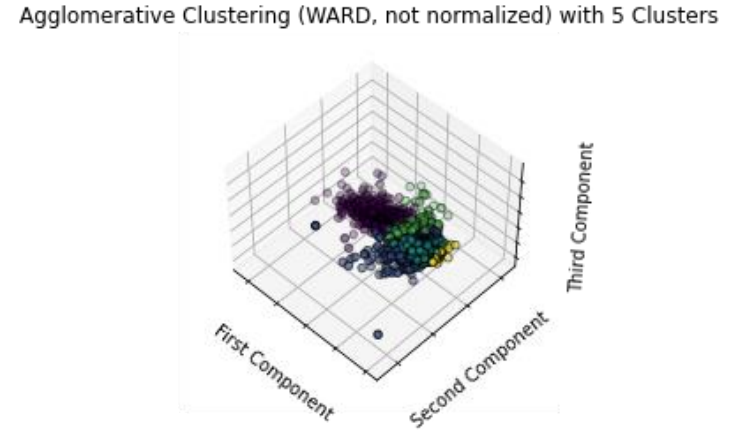
Min linkage clustering SSE: [9845831910362090.0]



Max linkage clustering SSE: [3529148510186874.0]



Ward linkage clustering SSE: [3614092780095645.0]



To contrast the SSE, here we compile a simple table from SSE calculated with normalized data points:

	Min/Single	**Max/Complete**	**Ward**
w/ normalization	2229.2	2127.7	624.5
w/o normalization	9845831910362090.0	3529148510186874.0	3614092780095645.0

We can see the drastic differences in the SSE between data points w/ and w/o normalization, the difference is astounding.

Plots from previous page is the clustering result (MIN, MAX, WARD) without normalization, we can also observe the clusters to be in difference shapes than before (with normalization). The shapes depicted here are sparser and more nonconforming, unlike the more dense versions seen in the previous task. We can see that the data point is sparser which causes our SSE loss to be significantly larger than the dataset with normalization. Which means normalization does indeed improve the performance of clustering.