EXPERIMENT NO. 8

Aim : The aim of data clustering in Python is to group similar data points together into clusters based on certain similarity or dissimilarity criteria. This process helps in discovering inherent patterns, structures, or relationships within the dataset, thereby aiding in data analysis, visualization, and decision-making.

Objective : To partition a dataset into cohesive clusters using Python, enabling insights into inherent patterns and structures for improved data analysis and decision-making..

Software used: Jupyter Notebook.

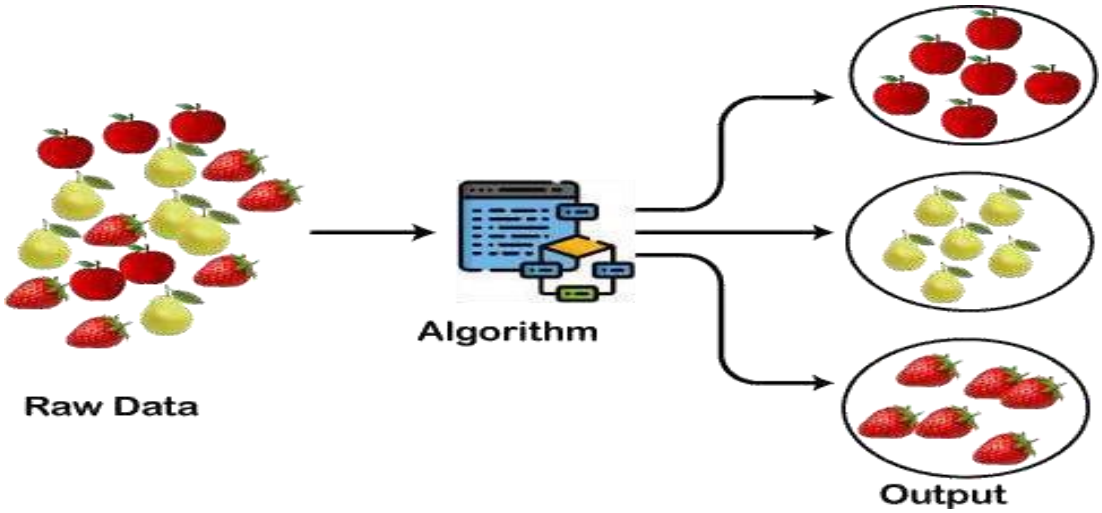
Theory:

Clustering is a set of techniques used to partition data into groups, or clusters. Clusters are loosely defined as groups of data objects that are more similar to other objects in their cluster than they are to data objects in other clusters. In practice, clustering helps identify two qualities of data:

1. Meaningfulness
2. Usefulness

Meaningful clusters expand domain knowledge. For example, in the medical field, researchers applied clustering to gene expression experiments. The clustering results identified groups of patients who respond differently to medical treatments.

Useful clusters, on the other hand, serve as an intermediate step in a data pipeline. For example, businesses use clustering for customer segmentation. The clustering results segment customers into groups with similar purchase histories, which businesses can then use to create targeted advertising campaigns.



There are many other applications of clustering, such as document clustering and social network analysis. These applications are relevant in nearly every industry, making clustering a valuable skill for professionals working with data in any field.

Overview of Clustering Techniques

You can perform clustering using many different approaches—so many, in fact, that there are entire categories of clustering algorithms. Each of these categories has its own unique strengths and weaknesses. This means that certain clustering algorithms will result in more natural cluster assignments depending on the input data.

Selecting an appropriate clustering algorithm for your dataset is often difficult due to the number of choices available. Some important factors that affect this decision include the characteristics of the clusters, the features of the dataset, the number of outliers, and the number of data objects.

You'll explore how these factors help determine which approach is most appropriate by looking at three popular categories of clustering algorithms:

1. Partitional clustering
2. Hierarchical clustering
3. Density-based clustering

It's worth reviewing these categories at a high level before jumping right into *k*-means. You'll learn the strengths and weaknesses of each category to provide context for how *k*-means fits into the landscape of clustering algorithms.

Partitional Clustering

Partitional clustering divides data objects into non overlapping groups. In other words, no object can be a member of more than one cluster, and every cluster must have at least one object.

These techniques require the user to specify the number of clusters, indicated by the variable *k*. Many partitional clustering algorithms work through an iterative process to assign subsets of data points into *k* clusters. Two examples of partitional clustering algorithms are *k*-means and *k*-medoids.

These algorithms are both nondeterministic, meaning they could produce different results from two separate runs even if the runs were based on the same input.

Partitional clustering methods have several strengths:

- They work well when clusters have a spherical shape.
- They're scalable with respect to algorithm complexity.

They also have several weaknesses:

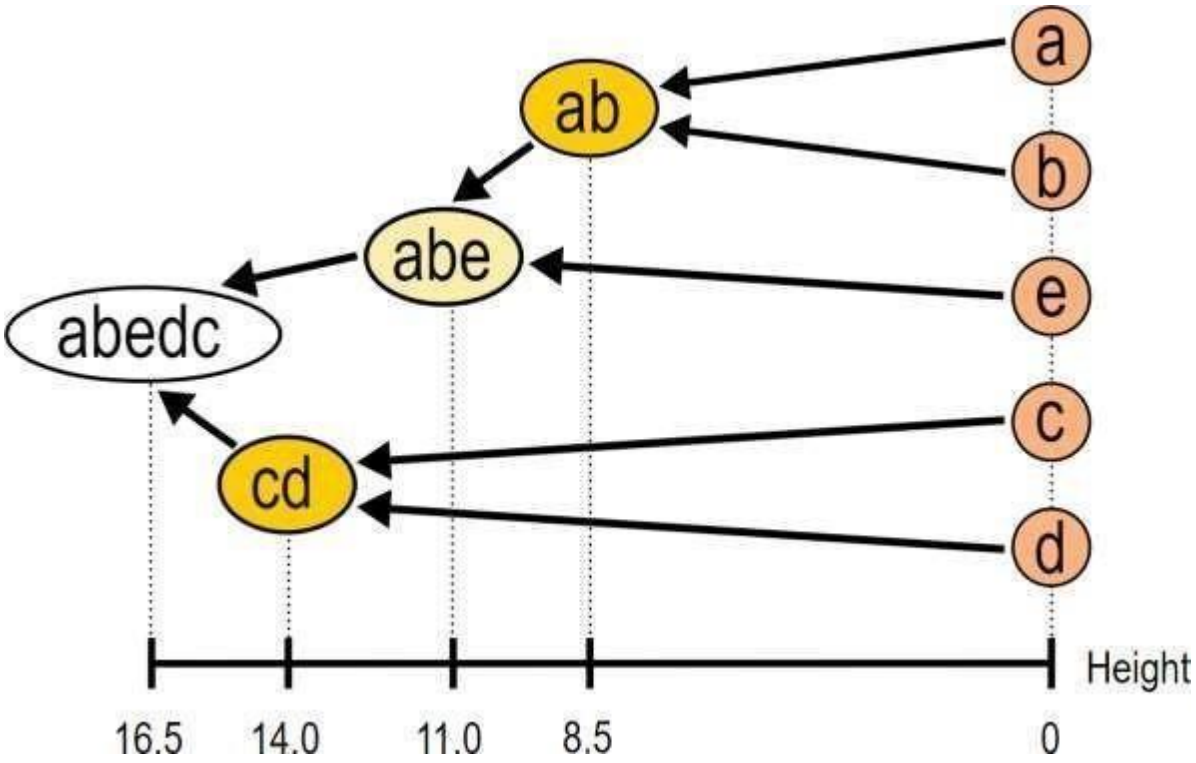
- They're not well suited for clusters with complex shapes and different sizes.
- They break down when used with clusters of different densities.

Hierarchical Clustering

Hierarchical clustering determines cluster assignments by building a hierarchy. This is implemented by either a bottom-up or a top-down approach:

- Agglomerative clustering is the bottom-up approach. It merges the two points that are the most similar until all points have been merged into a single cluster.
- Divisive clustering is the top-down approach. It starts with all points as one cluster and splits the least similar clusters at each step until only single data points remain.

These methods produce a tree-based hierarchy of points called a dendrogram.



Dendrogram

Similar to partitional clustering, in hierarchical clustering the number of clusters (*k*) is often predetermined by the user. Clusters are assigned by cutting the dendrogram at a specified depth that results in *k* groups of smaller dendrograms.

Unlike many partitional clustering techniques, hierarchical clustering is a deterministic process,

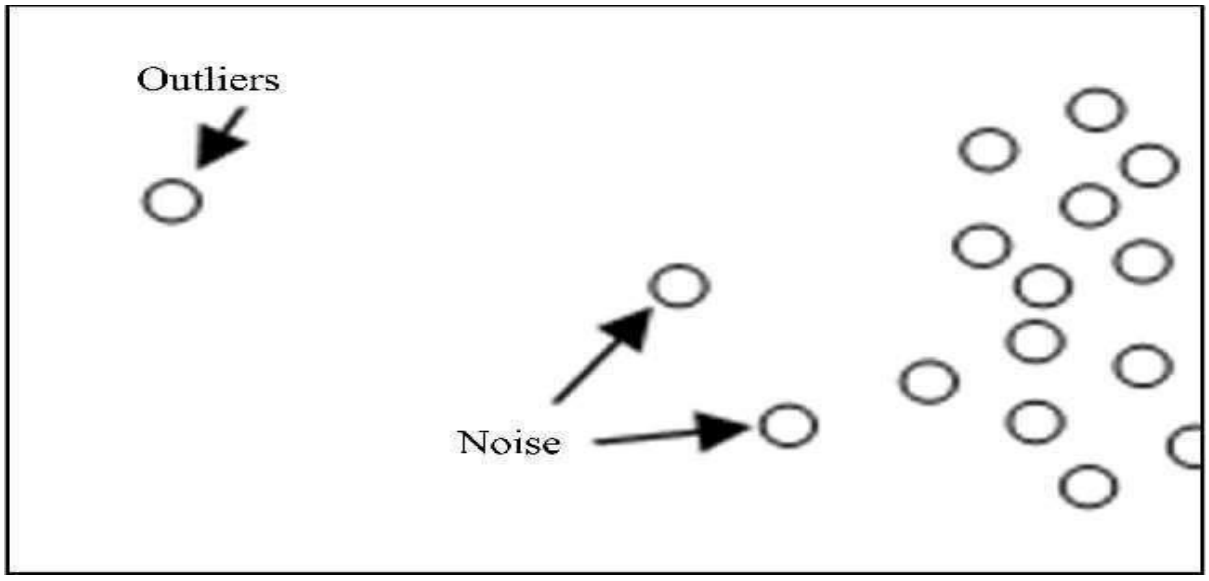
meaning cluster assignments won't change when you run an algorithm twice on the same input data.

The strengths of hierarchical clustering methods include the following:

- They often reveal the finer details about the relationships between data objects.
- They provide an interpretable dendrogram.

The weaknesses of hierarchical clustering methods include the following:

- They're computationally expensive with respect to algorithm complexity.
- They're sensitive to noise and outliers.



Density-Based Clustering

Density-based clustering determines cluster assignments based on the density of data points in a region. Clusters are assigned where there are high densities of data points separated by low-density regions.

Unlike the other clustering categories, this approach doesn't require the user to specify the number of clusters. Instead, there is a distance-based parameter that acts as a tunable threshold. This threshold determines how close points must be to be considered a cluster member.

Examples of density-based clustering algorithms include Density-Based Spatial Clustering of Applications with Noise, or DBSCAN, and Ordering Points To Identify the Clustering Structure,

or OPTICS.

The strengths of density-based clustering methods include the following:

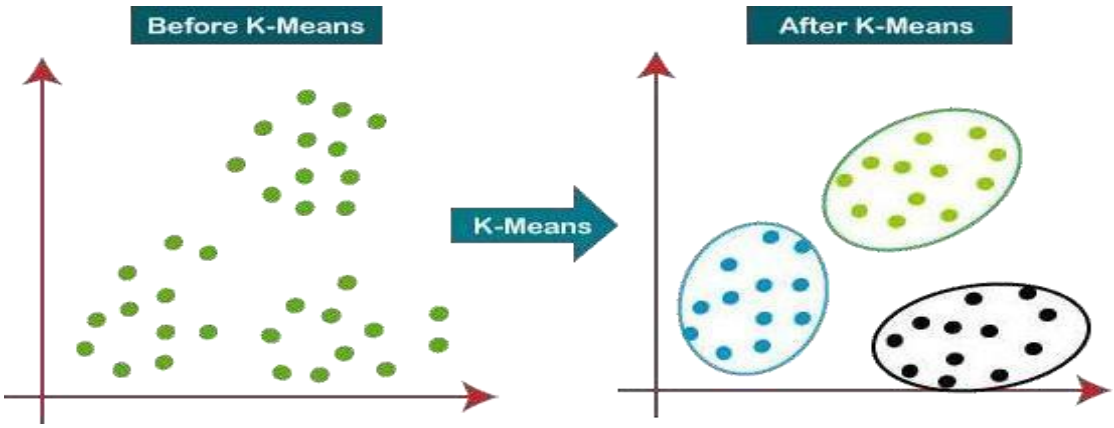
- They excel at identifying clusters of nonspherical shapes.
- They're resistant to outliers.

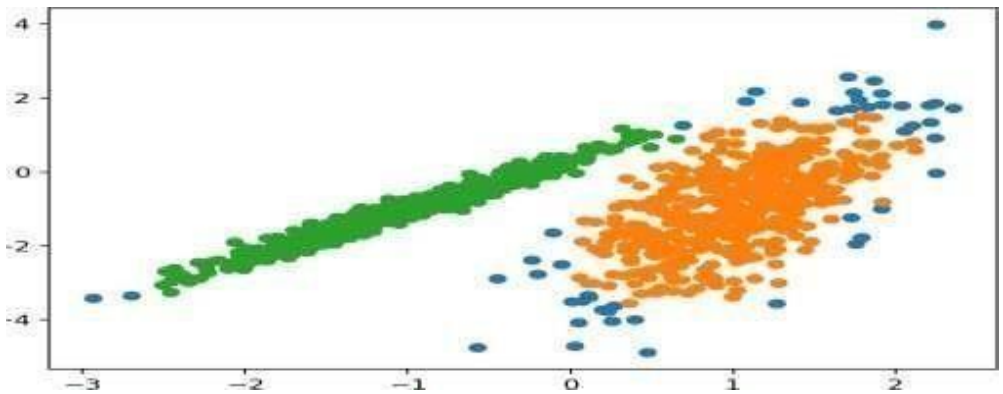
The weaknesses of density-based clustering methods include the following:

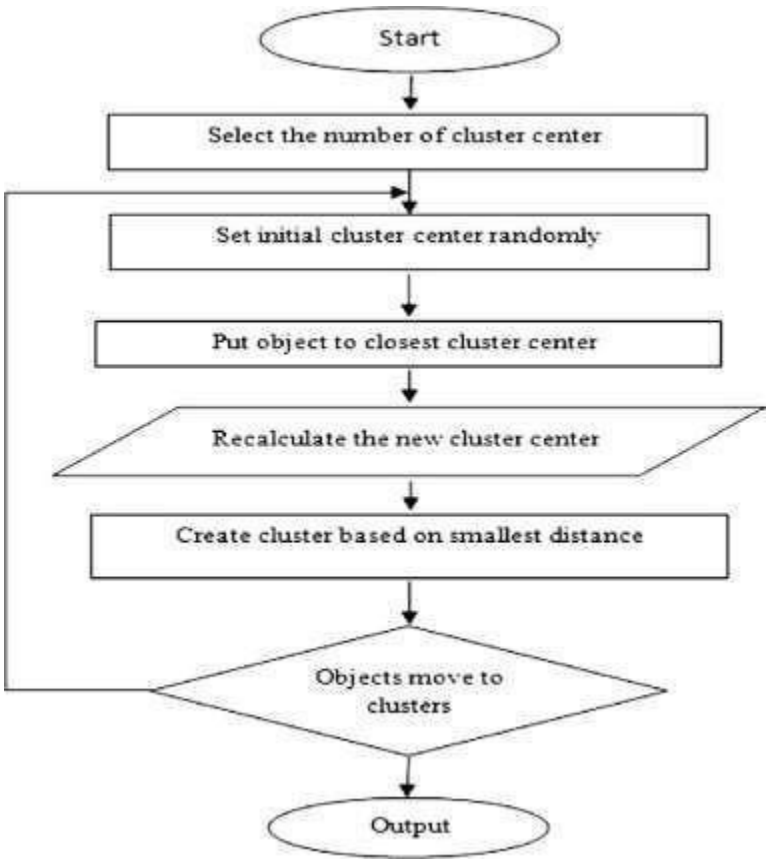
- They aren't well suited for clustering in high-dimensional spaces.
- They have trouble identifying clusters of varying densities.

How to Perform K-Means Clustering in Python

In this section, you'll take a step-by-step tour of the conventional version of the *k*-means algorithm. Understanding the details of the algorithm is a fundamental step in the process of writing your *k*-means clustering pipeline in Python. What you learn in this section will help you decide if *k*-means is the right choice to solve your clustering problem.







Understanding the K-Means Algorithm

Conventional *k*-means requires only a few steps. The first step is to randomly select *k* centroids, where *k* is equal to the number of clusters you choose. Centroids are data points representing the center of a cluster.

The main element of the algorithm works by a two-step process called expectation-maximization. The expectation step assigns each data point to its nearest centroid. Then, the maximization step computes the mean of all the points for each cluster and sets the new centroid. Here's what the conventional version of the *k*-means algorithm looks like:

Algorithm 1 *k*-means algorithm

1: Specify the number k of clusters to assign.
2: Randomly initialize k centroids.
3: **repeat**
4: **expectation:** Assign each point to its closest centroid.
5: **maximization:** Compute the new centroid (mean) of each cluster.
6: **until** The centroid positions do not change.

The quality of the cluster assignments is determined by computing the sum of the squared error (SSE) after the centroids converge, or match the previous iteration's assignment. The SSE is defined as the sum of the squared Euclidean distances of each point to its closest centroid. Since this is a measure of error, the objective of *k*-means is to try to minimize this value.

The random initialization step causes the *k*-means algorithm to be nondeterministic, meaning that cluster assignments will vary if you run the same algorithm twice on the same dataset. Researchers commonly run several initializations of the entire *k*-means algorithm and choose the cluster assignments from the initialization with the lowest SSE.

Output :

1st part

In [ ]:
```
# 1 Categorize
# 2
# Support value and lift
# Rules preparation - top 20 rows
# lift value greater than 1
```

In [9]:
```
import pandas as pd
from  mlxtend.frequent_patterns  import  apriori,association_rules
```

In [10]:
```
titanic  =  pd.read_csv("Titanic.csv")
```

In [11]:
```
titanic.head()
```

Out[11]:

	Class	Gender	Age	Survived
0	3rd	Male	Child	No
1	3rd	Male	Child	No
2	3rd	Male	Child	No
3	3rd	Male	Child	No
4	3rd	Male	Child	No

Pre Processing

In [12]:
```
df  =  pd.get_dummies(titanic)
df.head()
```

Out[12]:

	Class_1st	Class_2nd	Class_3rd	Class_Crew	Gender_Female	Gender_Male	Age_Adult	Age_Child	S
0	False	False	True	False	False	True	False	True	
1	False	False	True	False	False	True	False	True	
2	False	False	True	False	False	True	False	True	
3	False	False	True	False	False	True	False	True	
4	False	False	True	False	False	True	False	True	

Apri Ori Algorithm

In [14]:
```
frequent_itemsets  =  apriori(df,  min_support=0.1,  use_colnames=True)
frequent_itemsets
```

Out[14]:

	support	itemsets
0	0.147660	(Class_1st)
1	0.129487	(Class_2nd)
2	0.320763	(Class_3rd)
3	0.402090	(Class_Crew)
4	0.213539	(Gender_Female)
5	0.786461	(Gender_Male)
6	0.950477	(Age_Adult)
7	0.676965	(Survived_No)
8	0.323035	(Survived_Yes)
9	0.144934	(Age_Adult, Class_1st)
10	0.118582	(Age_Adult, Class_2nd)
11	0.231713	(Gender_Male, Class_3rd)
12	0.284871	(Age_Adult, Class_3rd)
13	0.239891	(Class_3rd, Survived_No)
14	0.391640	(Gender_Male, Class_Crew)
15	0.402090	(Age_Adult, Class_Crew)
16	0.305770	(Class_Crew, Survived_No)
17	0.193094	(Age_Adult, Gender_Female)
18	0.156293	(Survived_Yes, Gender_Female)
19	0.757383	(Age_Adult, Gender_Male)
20	0.619718	(Gender_Male, Survived_No)
21	0.166742	(Survived_Yes, Gender_Male)
22	0.653339	(Age_Adult, Survived_No)
23	0.297138	(Age_Adult, Survived_Yes)
24	0.209905	(Age_Adult, Gender_Male, Class_3rd)
25	0.191731	(Class_3rd, Gender_Male, Survived_No)
26	0.216265	(Class_3rd, Age_Adult, Survived_No)
27	0.391640	(Age_Adult, Gender_Male, Class_Crew)
28	0.304407	(Class_Crew, Gender_Male, Survived_No)
29	0.305770	(Class_Crew, Age_Adult, Survived_No)
30	0.143571	(Age_Adult, Survived_Yes, Gender_Female)
31	0.603816	(Age_Adult, Gender_Male, Survived_No)
32	0.153567	(Age_Adult, Survived_Yes, Gender_Male)
33	0.175829	(Class_3rd, Age_Adult, Gender_Male, Survived_No)

	support	itemsets
34	0.304407	(Class_Crew, Age_Adult, Gender_Male, Survived_No)

In [15]:
```
rules = association_rules(frequent_itemsets, metric="lift", min_threshold=0.7)
rules
```

Out[15]:

	antecedents	consequents	antecedent support	consequent support	support	confidence	lift	leverage
0	(Age_Adult)	(Class_1st)	0.950477	0.147660	0.144934	0.152486	1.032680	0.004587
1	(Class_1st)	(Age_Adult)	0.147660	0.950477	0.144934	0.981538	1.032680	0.004587
2	(Age_Adult)	(Class_2nd)	0.950477	0.129487	0.118582	0.124761	0.963505	-0.004492
3	(Class_2nd)	(Age_Adult)	0.129487	0.950477	0.118582	0.915789	0.963505	-0.004492
4	(Gender_Male)	(Class_3rd)	0.786461	0.320763	0.231713	0.294627	0.918520	-0.020555
...	...	...	...	...	...	...	...	...
101	(Gender_Male, Survived_No)	(Age_Adult, Class_Crew)	0.619718	0.402090	0.304407	0.491202	1.221623	0.055225
102	(Class_Crew)	(Age_Adult, Gender_Male, Survived_No)	0.402090	0.603816	0.304407	0.757062	1.253795	0.061619
103	(Age_Adult)	(Survived_No, Gender_Male, Class_Crew)	0.950477	0.304407	0.304407	0.320268	1.052103	0.015075
104	(Gender_Male)	(Age_Adult, Class_Crew, Survived_No)	0.786461	0.305770	0.304407	0.387060	1.265851	0.063931
105	(Survived_No)	(Age_Adult, Gender_Male, Class_Crew)	0.676965	0.391640	0.304407	0.449664	1.148157	0.039280

106 rows × 10 columns

A leverage value of 0 indicates independence. Range will be [-1,1]

A high conviction value means that the consequent is highly dependent on antecedent and range [0 inf]

In [16]:
```
rules.sort_values('lift',ascending = False)[0:20]
```

Out[16]:

	antecedents	consequents	antecedent support	consequent support	support	confidence	lift	levera
65	(Age_Adult, Gender_Female)	(Survived_Yes)	0.193094	0.323035	0.143571	0.743529	2.301699	0.0811
68	(Survived_Yes)	(Age_Adult,	0.323035	0.193094	0.143571	0.444444	2.301699	0.0811
19	(Gender_Female)	(Survived_Yes)	0.213539	0.323035	0.156293	0.731915	2.265745	0.0873
18	(Survived_Yes)	(Gender_Female)	0.323035	0.213539	0.156293	0.483826	2.265745	0.0873
69	(Gender_Female)	(Age_Adult, Survived_Yes)	0.213539	0.297138	0.143571	0.672340	2.262724	0.0801
64	(Age_Adult,	(Gender_Female)	0.297138	0.213539	0.143571	0.483180	2.262724	0.0801
99	(Age_Adult, Gender_Male)	(Survived_No, Class_Crew)	0.757383	0.305770	0.304407	0.401920	1.314450	0.0728
98	(Survived_No,	(Age_Adult,	0.305770	0.757383	0.304407	0.995542	1.314450	0.0728
46	(Age_Adult, Gender_Male)	(Class_Crew)	0.757383	0.402090	0.391640	0.517097	1.286022	0.0871
51	(Class_Crew)	(Age_Adult,	0.402090	0.757383	0.391640	0.974011	1.286022	0.0871
93	(Age_Adult, Class_Crew, Survived_No)	(Gender_Male)	0.305770	0.786461	0.304407	0.995542	1.265851	0.0639
53	(Survived_No,	(Gender_Male)	0.305770	0.786461	0.304407	0.995542	1.265851	0.0639
104	(Gender_Male)	(Age_Adult, Class_Crew, Survived_No)	0.786461	0.305770	0.304407	0.387060	1.265851	0.0639
56	(Gender_Male)	(Survived_No,	0.786461	0.305770	0.304407	0.387060	1.265851	0.0639
102	(Class_Crew)	(Age_Adult, Gender_Male, Survived_No)	0.402090	0.603816	0.304407	0.757062	1.253795	0.0616
95	(Age_Adult, Gender_Male, Survived_No)	(Class_Crew)	0.603816	0.402090	0.304407	0.504138	1.253795	0.0616
50	(Gender_Male)	(Age_Adult, Class_Crew)	0.786461	0.402090	0.391640	0.497978	1.238474	0.0754
47	(Age_Adult,	(Gender_Male)	0.402090	0.786461	0.391640	0.974011	1.238474	0.0754
11	(Class_Crew)	(Gender_Male)	0.402090	0.786461	0.391640	0.974011	1.238474	0.0754
10	(Gender_Male)	(Class_Crew)	0.786461	0.402090	0.391640	0.497978	1.238474	0.0754

In [17]:
```
rules[rules.lift>1]
```

Out[17]:

	antecedents	consequents	antecedent support	consequent support	support	confidence	lift	leverage
0	(Age_Adult)	(Class_1st)	0.950477	0.147660	0.144934	0.152486	1.032680	0.004587
1	(Class_1st)	(Age_Adult)	0.147660	0.950477	0.144934	0.981538	1.032680	0.004587
8	(Class_3rd)	(Survived_No)	0.320763	0.676965	0.239891	0.747875	1.104747	0.022745
9	(Survived_No)	(Class_3rd)	0.676965	0.320763	0.239891	0.354362	1.104747	0.022745
10	(Gender_Male)	(Class_Crew)	0.786461	0.402090	0.391640	0.497978	1.238474	0.075412
...	...	...	...	...	...	...	...	...
101	(Gender_Male, Survived_No)	(Age_Adult, Class_Crew)	0.619718	0.402090	0.304407	0.491202	1.221623	0.055225
102	(Class_Crew)	(Age_Adult, Gender_Male, Survived_No)	0.402090	0.603816	0.304407	0.757062	1.253795	0.061619
103	(Age_Adult)	(Survived_No, Gender_Male, Class_Crew)	0.950477	0.304407	0.304407	0.320268	1.052103	0.015075
104	(Gender_Male)	(Age_Adult, Class_Crew, Survived_No)	0.786461	0.305770	0.304407	0.387060	1.265851	0.063931
105	(Survived_No)	(Age_Adult, Gender_Male, Class_Crew)	0.676965	0.391640	0.304407	0.449664	1.148157	0.039280

74 rows × 10 columns

In [ ]:
```

```

2nd part

In [2]:
```
# import the libraries
from sklearn.cluster import DBSCAN
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
```

In [5]:
```
df = pd.read_csv("Wholesale customers data.csv")
df.head()
```

Out[5]:

	Channel	Region	Fresh	Milk	Grocery	Frozen	Detergents_Paper	Delicassen
0	2	3	12669	9656	7561	214	2674	1338
1	2	3	7057	9810	9568	1762	3293	1776
2	2	3	6353	8808	7684	2405	3516	7844
3	1	3	13265	1196	4221	6404	507	1788
4	2	3	22615	5410	7198	3915	1777	5185

In [6]:
```
df.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 440 entries, 0 to 439
Data columns (total 8 columns):
 #   Column            Non-Null Count  Dtype
---  ------            --------------  -----
 0   Channel           440 non-null    int64
 1   Region            440 non-null    int64
 2   Fresh             440 non-null    int64
 3   Milk              440 non-null    int64
 4   Grocery           440 non-null    int64
 5   Frozen            440 non-null    int64
 6   Detergents_Paper  440 non-null    int64
 7   Delicassen        440 non-null    int64
dtypes: int64(8)
memory usage: 27.6 KB
```

In [7]:
```
df.drop(['Channel','Region'], axis=1, inplace=True)
```

In [8]:
```
array = df.values
```

In [9]:
```
array
```

Out[9]:
```
array([[12669,  9656,  7561,   214,  2674,  1338],
       [ 7057,  9810,  9568,  1762,  3293,  1776],
       [ 6353,  8808,  7684,  2405,  3516,  7844],
       ...,
       [14531, 15488, 30243,   437, 14841,  1867],
       [10290,  1981,  2232,  1038,   168,  2125],
       [ 2787,  1698,  2510,    65,   477,    52]], dtype=int64)
```

In [13]:
```
from sklearn.preprocessing import StandardScaler
```

In [14]:
```
stscaler = StandardScaler().fit(array)
X = stscaler.transform(array)
```

In [15]:
```
X
```

Out[15]:
```
array([[ 0.05293319,  0.52356777, -0.04111489, -0.58936716, -0.04356873,
        -0.06633906],
       [-0.39130197,  0.54445767,  0.17031835, -0.27013618,  0.08640684,
         0.08915105],
       [-0.44702926,  0.40853771, -0.0281571 , -0.13753572,  0.13323164,
         2.24329255],
       ...,
       [ 0.20032554,  1.31467078,  2.34838631, -0.54337975,  2.51121768,
         0.12145607],
       [-0.13538389, -0.51753572, -0.60251388, -0.41944059, -0.56977032,
         0.21304614],
       [-0.72930698, -0.5559243 , -0.57322717, -0.62009417, -0.50488752,
        -0.52286938]])
```

In [18]:
```
dbscan = DBSCAN(eps=0.8, min_samples=12)
dbscan.fit(X)
```

Out[18]:
```
DBSCAN(eps=0.8, min_samples=12)
```

In [19]:
```
dbscan.labels_
```

Out[19]:
```
array([ 0,  0, -1,  0, -1,  0,  0,  0,  0,  0,  0,  0, -1, -1,  0,  0,  0,
       -1,  0,  0,  0,  0, -1, -1, -1,  0,  0,  0, -1, -1,  0,  0,  0, -1,
        0,  0, -1,  0, -1, -1, -1,  0,  0, -1,  0, -1,  0, -1,  0, -1,  0,
        0, -1,  0,  0,  0, -1,  0,  0,  0,  0, -1,  0,  0,  0, -1,  0,  0,
        0,  0,  0, -1,  0,  0,  0,  0,  0, -1,  0,  0,  0,  0,  0,  0,  0,
       -1, -1, -1,  0,  0,  0,  0, -1, -1,  0,  0,  0,  0,  0,  0, -1,  0,
        0, -1,  0,  0,  0,  0,  0, -1,  0, -1,  0,  0,  0,  0,  0,  0,  0,
        0,  0,  0,  0,  0,  0, -1,  0,  0,  0, -1,  0,  0,  0,  0,  0,  0,
        0,  0,  0,  0,  0, -1, -1,  0,  0, -1,  0,  0,  0,  0,  0,  0,  0,
        0,  0, -1,  0,  0,  0,  0,  0,  0,  0, -1,  0, -1,  0,  0,  0,  0,
        0, -1,  0,  0,  0,  0, -1,  0,  0,  0,  0, -1,  0, -1,  0,  0,  0,
       -1,  0,  0,  0,  0,  0, -1,  0,  0, -1,  0,  0,  0, -1, -1, -1,  0,
        0, -1,  0,  0,  0, -1,  0, -1,  0,  0,  0,  0, -1,  0, -1,  0,  0,
        0,  0,  0,  0,  0,  0,  0, -1,  0,  0,  0,  0,  0,  0,  0,  0,  0,
        0,  0,  0, -1, -1,  0,  0,  0,  0,  0, -1, -1,  0, -1,  0,  0,  0,
        0,  0,  0,  0, -1, -1,  0,  0,  0,  0, -1,  0, -1, -1,  0,  0,  0,
       -1,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0, -1,  0,  0, -1,  0,
        0,  0,  0, -1,  0,  0, -1,  0,  0,  0,  0,  0,  0, -1,  0,  0,  0,
        0,  0, -1,  0,  0,  0,  0,  0, -1,  0, -1,  0,  0,  0,  0, -1,  0,
        0,  0,  0, -1,  0,  0,  0, -1,  0, -1,  0, -1,  0,  0,  0,  0,  0,
       -1, -1,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0, -1,  0, -1,  0,
        0,  0, -1, -1,  0,  0,  0,  0, -1,  0, -1,  0,  0,  0,  0,  0,  0,
        0,  0,  0,  0,  0,  0,  0,  0,  0,  0, -1,  0, -1,  0,  0,  0, -1,
        0, -1,  0, -1,  0, -1,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,
       -1,  0, -1,  0,  0, -1, -1,  0,  0,  0, -1, -1, -1,  0,  0],
      dtype=int64)
```

In [21]:
```
cl = pd.DataFrame(dbscan.labels_,columns=['Cluster'])
```

In [22]:
```
cl
```

Out[22]:

	Cluster
0	0
1	0
2	-1
3	0
4	-1
...	...
435	-1
436	-1
437	-1
438	0
439	0

440 rows × 1 columns

In [23]:
```
pd.concat([df,cl],axis=1)
```

Out[23]:

	Fresh	Milk	Grocery	Frozen	Detergents_Paper	Delicassen	Cluster
0	12669	9656	7561	214	2674	1338	0
1	7057	9810	9568	1762	3293	1776	0
2	6353	8808	7684	2405	3516	7844	-1
3	13265	1196	4221	6404	507	1788	0
4	22615	5410	7198	3915	1777	5185	-1
...	...	...	...	...	...	...	...
435	29703	12051	16027	13135	182	2204	-1
436	39228	1431	764	4510	93	2346	-1
437	14531	15488	30243	437	14841	1867	-1
438	10290	1981	2232	1038	168	2125	0
439	2787	1698	2510	65	477	52	0

440 rows × 7 columns

In [ ]:
```

```

3rd part

In [1]:
```
import pandas as pd
import numpy as np
from  sklearn.decomposition import PCA
import matplotlib.pyplot as plt
from sklearn.preprocessing import scale
```

In [2]:
```
uni  =  pd.read_csv("Universities1.csv")
uni.describe()
uni.head()
```

Out[2]:

	Univ	SAT	Top10	Accept	SFRatio	Expenses	GradRate
0	Brown	1310	89	22	13	22704	94
1	CalTech	1415	100	25	6	63575	81
2	CMU	1260	62	59	9	25026	72
3	Columbia	1310	76	24	12	31510	88
4	Cornell	1280	83	33	13	21864	90

In [4]:
```
# Considering only numerical data
uni.data = uni.iloc[:,1:]
uni.data.head()
# converting ino numpy array
UNI = uni.data.values
UNI
```

Out[4]:
```
array([[ 1310,    89,    22,    13, 22704,    94],
       [ 1415,   100,    25,     6, 63575,    81],
       [ 1260,    62,    59,     9, 25026,    72],
       [ 1310,    76,    24,    12, 31510,    88],
       [ 1280,    83,    33,    13, 21864,    90],
       [ 1340,    89,    23,    10, 32162,    95],
       [ 1315,    90,    30,    12, 31585,    95],
       [ 1255,    74,    24,    12, 20126,    92],
       [ 1400,    91,    14,    11, 39525,    97],
       [ 1305,    75,    44,     7, 58691,    87],
       [ 1380,    94,    30,    10, 34870,    91],
       [ 1260,    85,    39,    11, 28052,    89],
       [ 1255,    81,    42,    13, 15122,    94],
       [ 1081,    38,    54,    18, 10185,    80],
       [ 1375,    91,    14,     8, 30220,    95],
       [ 1005,    28,    90,    19,  9066,    69],
       [ 1360,    90,    20,    12, 36450,    93],
       [ 1075,    49,    67,    25,  8704,    67],
       [ 1240,    95,    40,    17, 15140,    78],
       [ 1290,    75,    50,    13, 38380,    87],
       [ 1180,    65,    68,    16, 15470,    85],
       [ 1285,    80,    36,    11, 27553,    90],
       [ 1225,    77,    44,    14, 13349,    92],
       [ 1085,    40,    69,    15, 11857,    71],
       [ 1375,    95,    19,    11, 43514,    96]], dtype=int64)
```

In [5]:
```
uni_normal = scale(UNI) # normalizing the numerical data
```

In [6]:
```
uni_normal
```

Out[6]:
```
array([[ 0.41028362,  0.6575195 , -0.88986682,  0.07026045, -0.33141256,
         0.82030265],
       [ 1.39925928,  1.23521235, -0.73465749, -1.68625071,  2.56038138,
        -0.64452351],
       [-0.06065717, -0.76045386,  1.02438157, -0.93346022, -0.16712136,
        -1.65863393],
       [ 0.41028362, -0.02520842, -0.78639393, -0.18066972,  0.29164871,
         0.14422904],
       [ 0.12771914,  0.34241431, -0.32076595,  0.07026045, -0.39084607,
         0.36958691],
       [ 0.69284809,  0.6575195 , -0.83813038, -0.68253005,  0.33778044,
         0.93298158],
       [ 0.4573777 ,  0.71003703, -0.47597528, -0.18066972,  0.29695528,
         0.93298158],
       [-0.10775125, -0.13024348, -0.78639393, -0.18066972, -0.51381683,
         0.59494478],
       [ 1.25797704,  0.76255456, -1.30375836, -0.43159988,  0.85874344,
         1.15833946],
       [ 0.36318954, -0.07772595,  0.24833493, -1.43532055,  2.21481798,
         0.0315501 ],
       [ 1.06960072,  0.92010716, -0.47597528, -0.68253005,  0.52938275,
         0.48226584],
       [-0.06065717,  0.44744937, -0.01034729, -0.43159988,  0.04698077,
         0.25690797],
       [-0.10775125,  0.23737924,  0.14486204,  0.07026045, -0.86787073,
         0.82030265],
       [-1.7466252 , -2.02087462,  0.76569936,  1.32491127, -1.21718409,
        -0.75720245],
       [ 1.02250664,  0.76255456, -1.30375836, -1.18439038,  0.20037583,
         0.93298158],
       [-2.46245521, -2.54604994,  2.6282113 ,  1.57584144, -1.29635802,
        -1.99667073],
       [ 0.88122441,  0.71003703, -0.9933397 , -0.18066972,  0.64117435,
         0.70762371],
       [-1.8031381 , -1.44318177,  1.43827311,  3.08142243, -1.32197103,
        -2.22202861],
       [-0.24903349,  0.97262469,  0.04138915,  1.07398111, -0.86659715,
        -0.98256032],
       [ 0.2219073 , -0.07772595,  0.55875358,  0.07026045,  0.77772991,
         0.0315501 ],
       [-0.81416244, -0.60290126,  1.49000956,  0.82305094, -0.84324827,
        -0.19380777],
       [ 0.17481322,  0.18486171, -0.16555662, -0.43159988,  0.01167444,
         0.36958691],
       [-0.39031573,  0.02730912,  0.24833493,  0.32119061, -0.99331788,
         0.59494478],
       [-1.70894994, -1.91583956,  1.541746  ,  0.57212078, -1.09888311,
        -1.77131286],
       [ 1.02250664,  0.97262469, -1.04507615, -0.43159988,  1.14098185,
         1.04566052]])
```

In [7]:
```
pca = PCA()
pca_values  =  pca.fit_transform(uni_normal)
```

In [8]:
```
pca_values
```

Out[8]:
```
array([[-1.00987445e+00, -1.06430962e+00, 8.10663051e-02,
         5.69506350e-02,  -1.28754245e-01,  -3.46496377e-02],
       [-2.82223781e+00,  2.25904458e+00,  8.36828830e-01,
         1.43844644e-01,  -1.25961913e-01,  -1.80703168e-01],
       [ 1.11246577e+00,  1.63120889e+00, -2.66786839e-01,
         1.07507502e+00,  -1.91814148e-01,   3.45679459e-01],
       [-7.41741217e-01,  -4.21874699e-02,   6.05008649e-02,
        -1.57208116e-01,  -5.77611392e-01,  1.09163092e-01],
       [-3.11912064e-01,  -6.35243572e-01,  1.02405189e-02,
         1.71363672e-01,  1.27261287e-02,  -1.69212696e-02],
       [-1.69669089e+00,  -3.44363283e-01,  -2.53407507e-01,
         1.25643278e-02,  -5.26606002e-02,  -2.71661600e-02],
       [-1.24682093e+00,  -4.90983662e-01,  -3.20938196e-02,
        -2.05643780e-01,  2.93505340e-01,  -7.80119838e-02],
       [-3.38749784e-01,  -7.85168589e-01,  -4.93584829e-01,
         3.98563085e-02,  -5.44978619e-01,  -1.55371653e-01],
       [-2.37415013e+00,  -3.86538883e-01,   1.16098392e-01,
        -4.53365617e-01,  -2.30108300e-01,  2.66983932e-01],
       [-1.40327739e+00,  2.11951503e+00,  -4.42827141e-01,
        -6.32543273e-01,   2.30053526e-01,  -2.35615124e-01],
       [-1.72610332e+00,  8.82371161e-02,  1.70403663e-01,
         2.60901913e-01,  2.33318380e-01,  2.38968449e-01],
       [-4.50857480e-01, -1.11329480e-02, -1.75746046e-01,
         2.36165626e-01,   2.63250697e-01,  -3.14843521e-01],
       [ 4.02381405e-02,  -1.00920438e+00,  -4.96517167e-01,
         2.29298758e-01,  4.48031921e-01,  4.93921533e-03],
       [ 3.23373034e+00,  -3.74580487e-01,  -4.95372816e-01,
        -5.21237711e-01,  -6.39294809e-01,  -9.00477852e-02],
       [-2.23626502e+00,  -3.71793294e-01,  -3.98993653e-01,
         4.06966479e-01,  -4.16760680e-01,   5.06186327e-02],
       [ 5.17299212e+00,  7.79915346e-01, -3.85912331e-01,
        -2.32211711e-01,   1.79286976e-01,  -3.09046943e-02],
       [-1.69964377e+00,  -3.05597453e-01,   3.18507851e-01,
        -2.97462682e-01,  -1.63424678e-01,   1.14422592e-01],
       [ 4.57814600e+00, -3.47591363e-01,  1.49964176e+00,
        -4.54251714e-01,  -1.91141971e-01,   1.04149297e-01],
       [ 8.22603117e-01, -6.98906146e-01,  1.42781145e+00,
         7.60778800e-01,   1.84260335e-01,  -2.51103268e-01],
       [-9.77621343e-02,  6.50446454e-01,  1.00508440e-01,
        -5.00097185e-01,  4.87217823e-01,  2.19242132e-01],
       [ 1.96318260e+00,  -2.24767561e-01,  -2.55881433e-01,
        -4.84741049e-02,  8.22745655e-01,  1.52246521e-01],
       [-5.42288939e-01,  -7.95888376e-02,  -3.05393475e-01,
         1.31698758e-01,   5.27399148e-02,  -3.67264440e-02],
       [ 5.32220920e-01,  -1.01716720e+00,  -4.23716362e-01,
         1.69535706e-01,   3.57813210e-01,  -6.60989993e-02],
       [ 3.54869664e+00,  7.78461666e-01, -4.49363319e-01,
         3.23678618e-01,  -3.58332564e-01,  -7.74564151e-02],
       [-2.30590032e+00,  -1.17704318e-01,   2.53988661e-01,
        -5.16183372e-01,   5.58940129e-02,  -1.07932007e-02]])
```

In [9]:
```
pca = PCA(n_components=6)
pca_values  =  pca.fit_transform(uni_normal)
```

In [10]:
```
# the amount of variance that each PCA explains is
var  =  pca.explained_variance_ratio_
var
```

Out[10]:
```
array([0.76868084, 0.13113602, 0.04776031, 0.02729668, 0.0207177 ,
       0.00440844])
```

In [12]:
```
var1 = np.cumsum(np.round(var,decimals = 4)* 100)
var1
```

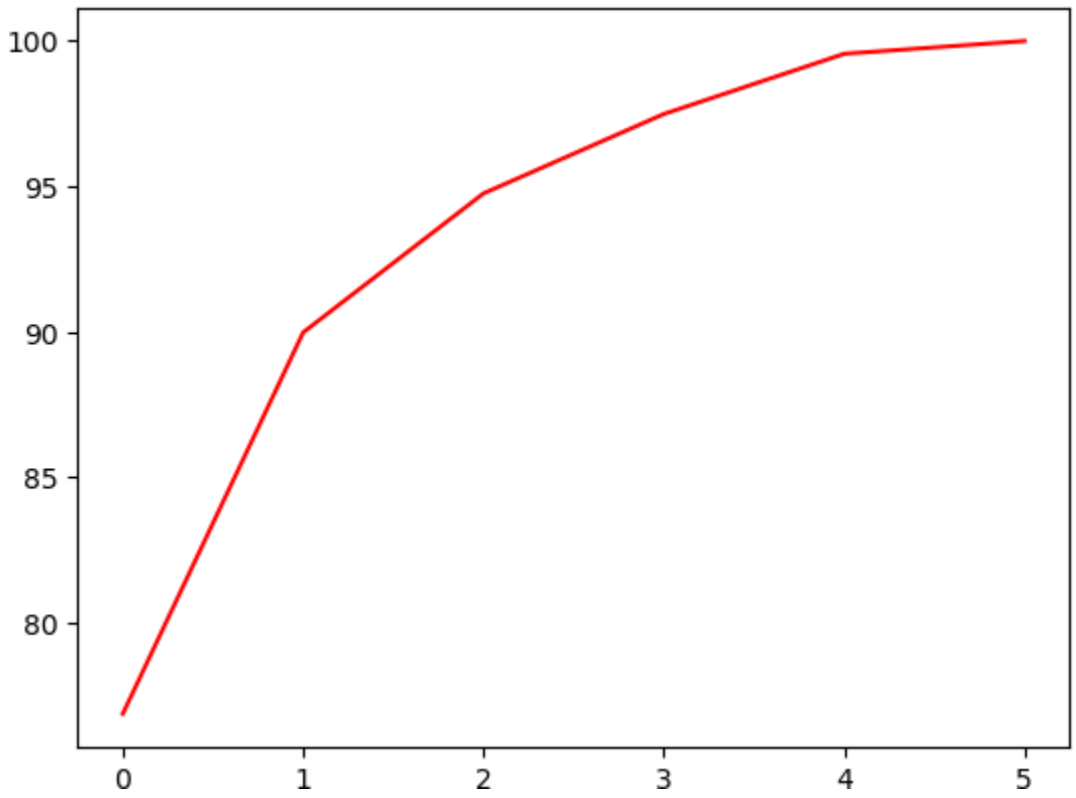
Out[12]:
```
array([ 76.87,  89.98,  94.76,  97.49,  99.56, 100.  ])
```

In [13]:
```
pca.components_
```

Out[13]:
```
array([[-0.45774863, -0.42714437,  0.42430805,  0.39064831, -0.36252316,
        -0.37940403],
       [ 0.03968044, -0.19993153,  0.32089297, -0.43256441,  0.6344864 ,
        -0.51555367],
       [ 0.1870388 ,  0.49780855, -0.15627899,  0.60608085,  0.20474114,
        -0.53247261],
       [ 0.13124033,  0.37489567,  0.0612872 , -0.50739095, -0.62340055,
        -0.43863341],
       [ 0.02064583,  0.4820162 ,  0.8010936 ,  0.07682369,  0.07254775,
         0.33810965],
       [ 0.8580547 , -0.39607492,  0.21693361,  0.1720479 , -0.17376309,
        -0.00353754]])
```

In [14]:
```
# Variance plot for PCA components obtained
plt.plot(var1,color="red")
```

Out[14]:
```
[<matplotlib.lines.Line2D at 0x1d92120bd10>]
```

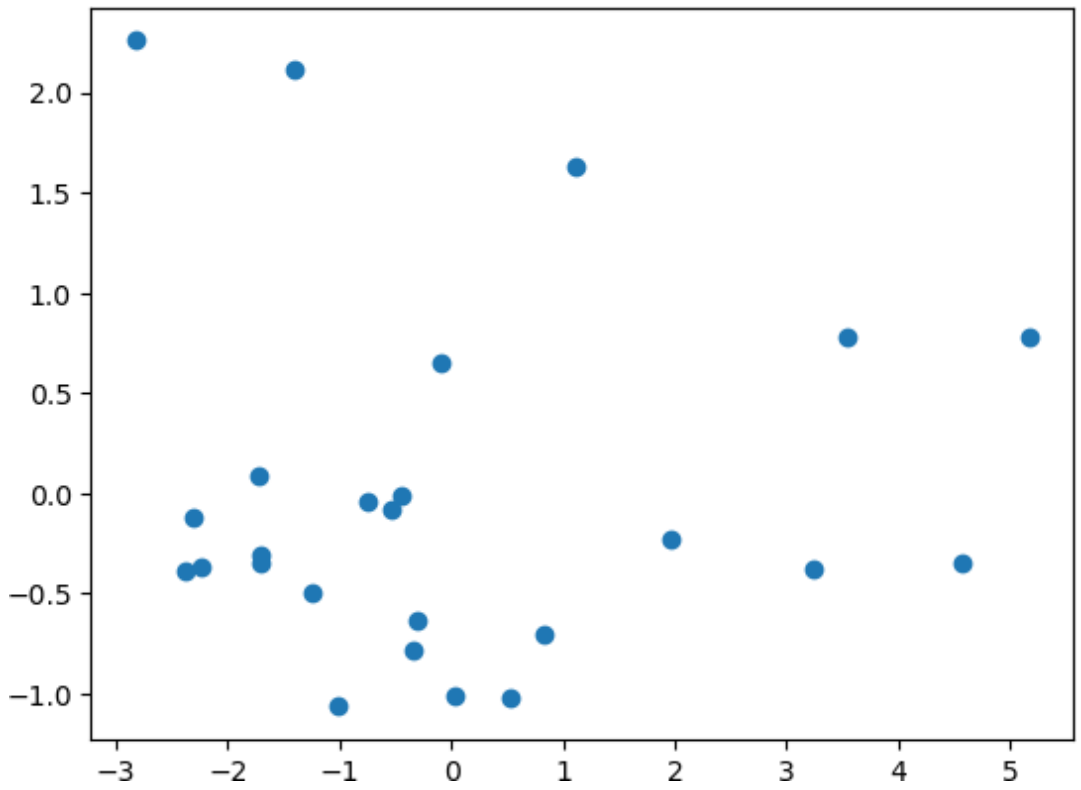


In [15]:
```
pca_values[:,0:1]
```

Out[15]:
```
array([[-1.00987445],
       [-2.82223781],
       [ 1.11246577],
       [-0.74174122],
       [-0.31191206],
       [-1.69669089],
       [-1.24682093],
       [-0.33874978],
       [-2.37415013],
       [-1.40327739],
       [-1.72610332],
       [-0.45085748],
       [ 0.04023814],
       [ 3.23373034],
       [-2.23626502],
       [ 5.17299212],
       [-1.69964377],
       [ 4.578146  ],
       [ 0.82260312],
       [-0.09776213],
       [ 1.9631826 ],
       [-0.54228894],
       [ 0.53222092],
       [ 3.54869664],
       [-2.30590032]])
```

In [16]:
```
#plot betweeen PCA1 and PCA 2
x = pca_values[:,0:1]
y = pca_values[:,1:2]
# z = pca_values
plt.scatter(x,y)
```

Out[16]:
```
<matplotlib.collections.PathCollection  at  0x1d9212e1490>
```

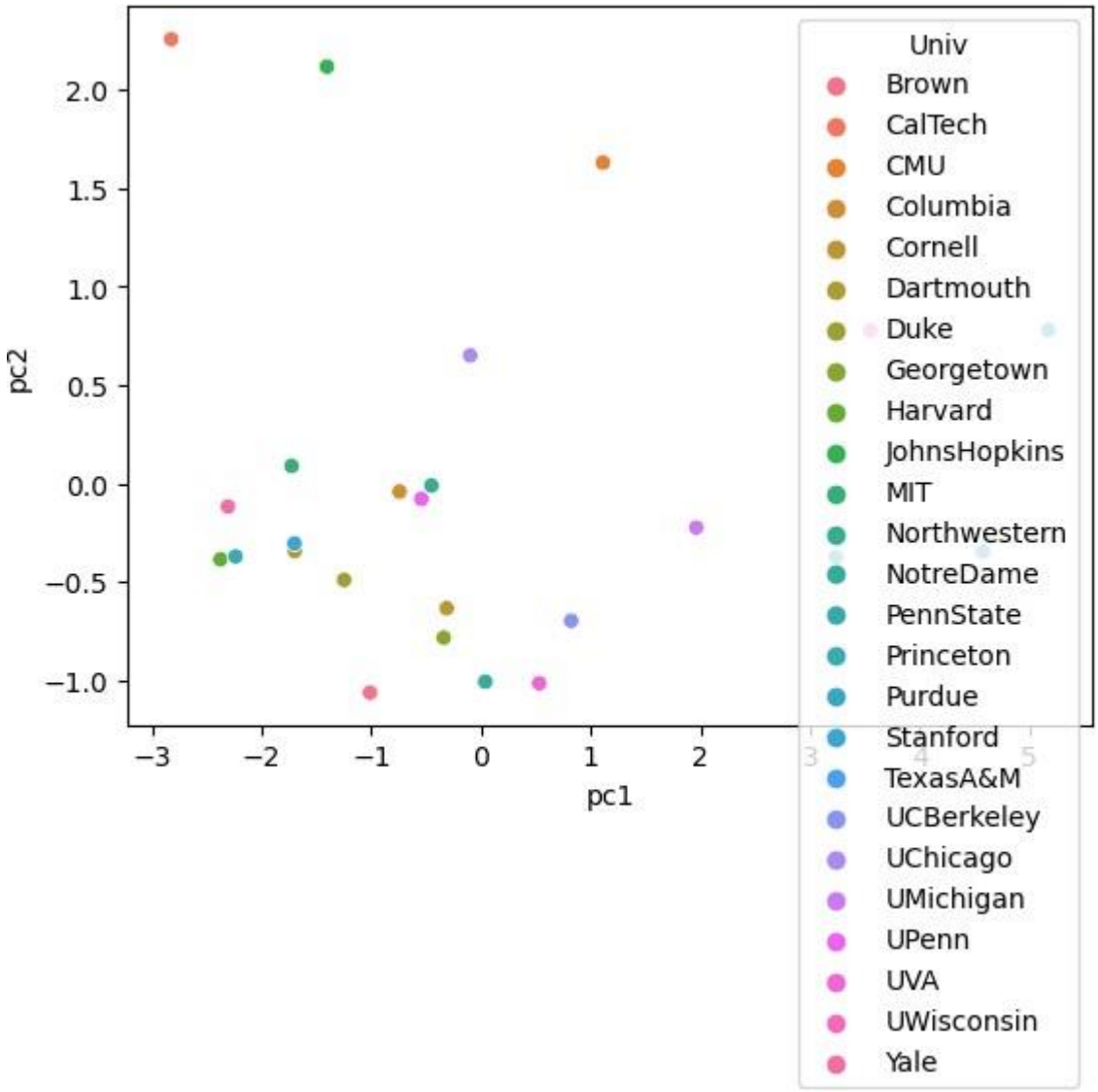


In [17]:
```
finalDF = pd.concat([pd.DataFrame(pca_values[:,0:2],columns=['pc1','pc2']),uni[['Univ'
```

In [18]:
```
import seaborn as sns
sns.scatterplot(data=finalDF,  x='pc1',y='pc2',hue='Univ')
```

Out[18]:

```
<Axes: xlabel='pc1', ylabel='pc2'>
```



Conclusion: Studying and analyzing clustering models provides valuable insights into unsupervised learning techniques that group similar data points together without predefined labels. Through understanding the basics, exploring different algorithms, hands-on implementation, evaluating with appropriate metrics, visualizing results, examining real-world applications, conducting comparative analyses, delving into advanced topics, one can develop a robust understanding of clustering models. By following these steps, individuals can effectively analyze clustering algorithms, interpret results, and apply them to various domains, contributing to data-driven decision-making and problem-solving processes.