# Using Machine Learning Tools: Workshop 8

## Unsupervised Machine Learning

There are four sections to this notebook:

1) Imports, loading and pre-processing a number of different datasets

2) General functions to apply unsupervised methods and display results

3) Example application of the functions to the datasets and exploration of the results

4) Time to play!

Please skim through sections 1 and 2, but spent most time in section 3-4 where you are expected to modify the code to explore different methods and different datasets, to get an understanding of how the methods apply to a range of real (as well as some artificial) datasets. This exploration is key to gaining a true understanding of how the methods can be applied in practice.

### The general flow

As a note, this workshop is a little different from the previous ones. Here there are several data sources being examined, so there is a lot more setup.

## Section 1 [Students, 10mins]: Imports, loading and pre-processing

```
# Common imports
import sklearn
import numpy as np
import os

# To plot pretty figures
%matplotlib inline
import matplotlib as mpl
import matplotlib.pyplot as plt
mpl.rc('axes', labelsize=14)
mpl.rc('xtick', labelsize=12)
mpl.rc('ytick', labelsize=12)

import pandas as pd
import seaborn as sns; sns.set()

# There are a lot of imports below but don't worry about them
#  you can easily figure out when you need them from the docs
#  you wouldn't be expect to know or remember these

# Datasets
```

```python
from sklearn.datasets import load_iris
from sklearn.datasets import load_breast_cancer
from sklearn.datasets import make_blobs

# Clustering methods
from sklearn.linear_model import LinearRegression
from sklearn.decomposition import PCA
from sklearn.preprocessing import StandardScaler
from sklearn import cluster, mixture, manifold, random_projection
from scipy.cluster.hierarchy import dendrogram, linkage

# This is for reading some medical imaging formats
import nibabel as nib

# Convenient variable for specifying colours throughout
global_palette = 'tab10'
```

## Dependencies

It is likely that you should install nibabel. Use either:

!pip install nibabel

OR

conda install -c conda-forge nibabel

```python
# Also, check the version of your threadpoolctl library. It should be
greater than 3.0 to avoid problems
#import threadpoolctl
#threadpoolctl.__version__

# !pip install --upgrade threadpoolctl
```

### Initial loading and description

The following cells load in the data and show a small amount of info on them. A few mundane observations. What a lot of binary input data we have here... it looks like mostly 1's and 0's.

```python
# Dictionary for all datasets - each entry to store (x, labels)
datasets={}

# Dataset on children with physical and motor disabilities
# https://archive.ics.uci.edu/ml/datasets/SCADI
data = pd.read_csv('SCADI.csv')

# We can check the first rows of our dataset
data.head(5)
```

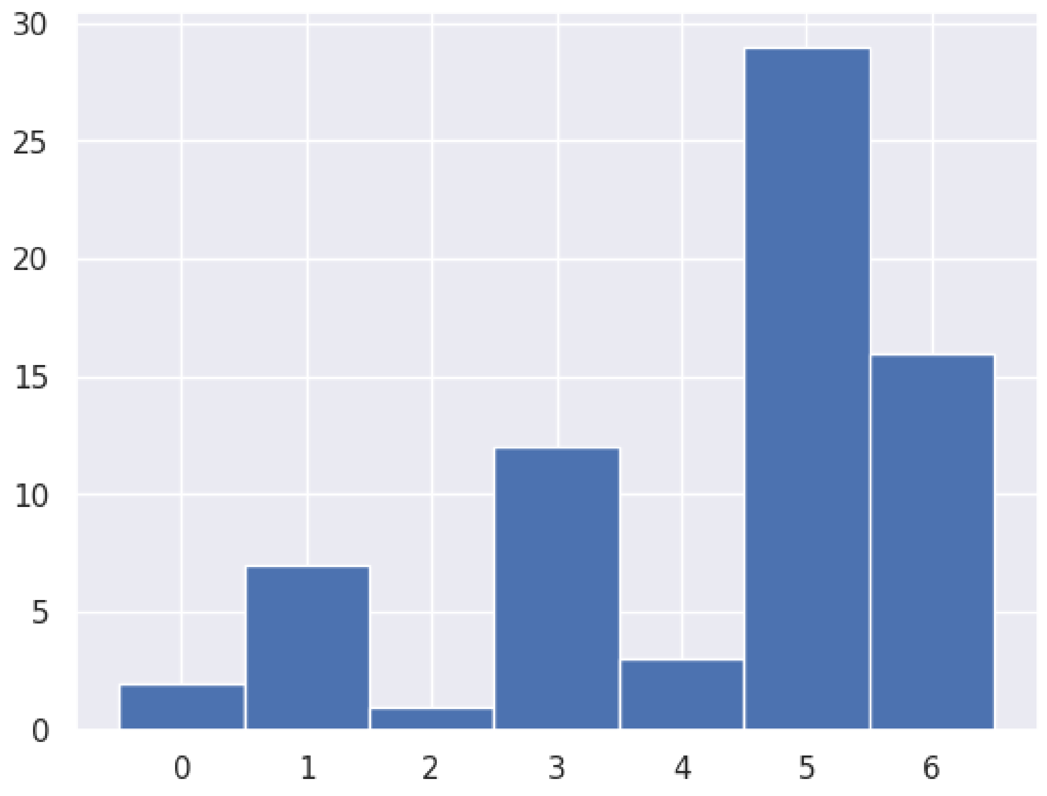
```
{"type":"dataframe","variable_name":"data"}

# We can extract the features
x = data.iloc[:,:-1].to_numpy(dtype=np.float64)
print(x.shape)

# Also, we can extract the labels
labels = data["Classes"].astype('category').cat.codes.to_numpy()

# We can put this dataset in the dictionary datasets, under the name
"scadi"
datasets['scadi'] = (x, labels)
```

```
(70, 205)
```

## Histograms

Sometimes, going to that little bit of extra effort with your histogram makes all the difference. As a trick, whenever you have classes that are numbers 0:N, making your bins work at 0.5, 1.5 etc will ensure they are all labelled sensibly in the final diagram.

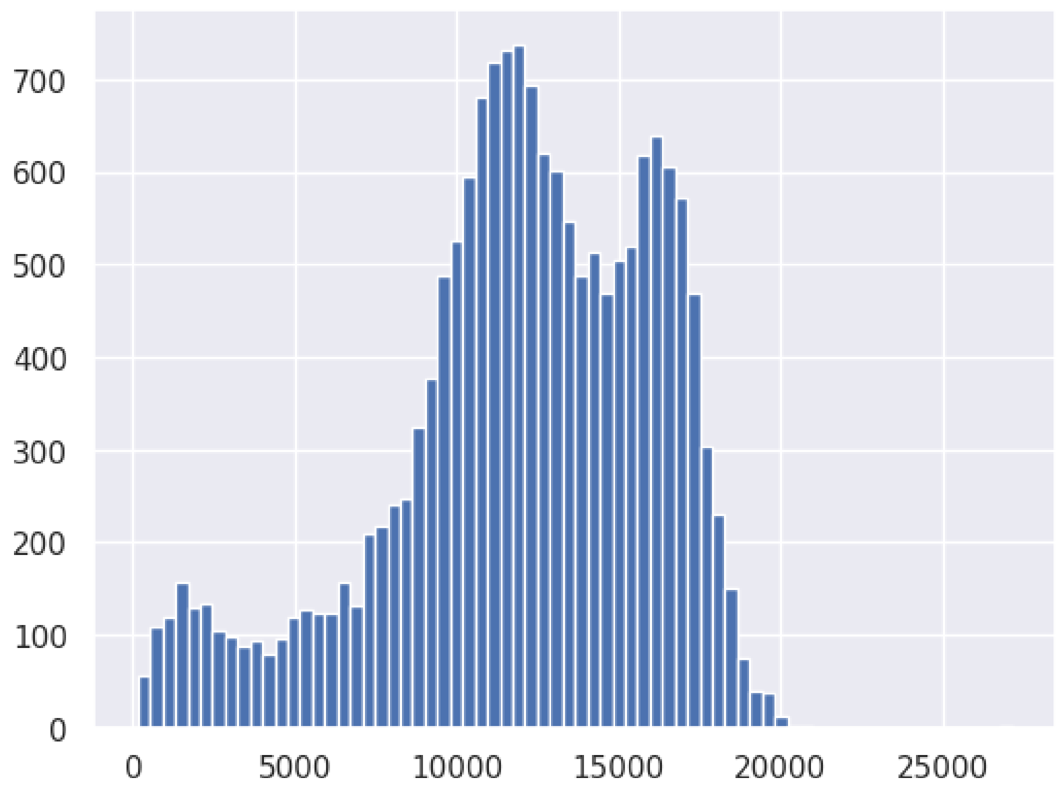
```
# Visualise the labels

unique_labels = np.unique(labels)
# plt.hist(labels)
plt.hist(labels, np.append(unique_labels, len(unique_labels)) - 0.5)
plt.show()
```

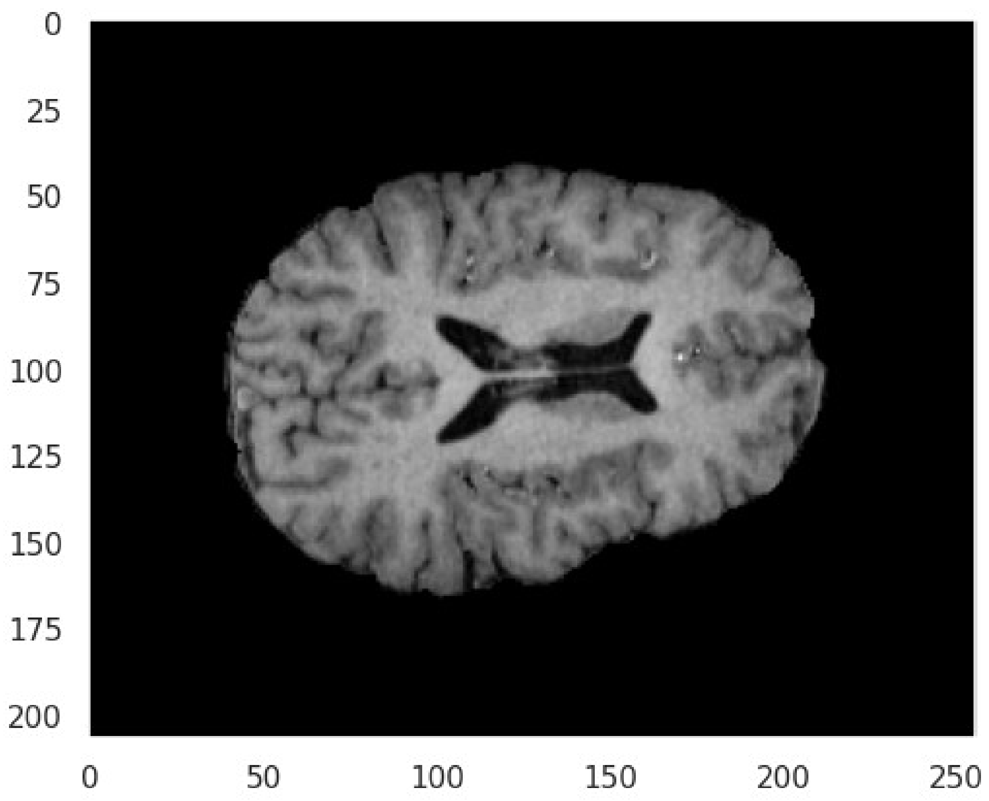


```python
# Brain MRI image slice
im = nib.load('T1_brain.nii.gz').get_fdata()

# We can change this to have a look at different images in the dataset
(default is 95)
image_selection = 95

# We can extract one image
x_im = im[:,:,image_selection]

# This is an interesting step. The reality is, that the images are...
images
# but 2D arrays aren't always the format we need to use.

# Furthermore, why would we do this??
nx, ny = x_im.shape

print("x_im shape:", nx, ny)

# force it to be a 2D array with one column (consistent with other
datasets)
x = x_im.reshape(-1,1)
# Can go back to original 2D image (for display later on) with
reshape((nx,ny))
labels = None
```

```
# We can put this dataset in the dictionary datasets, under the name
"brain"
datasets['brain'] = (x, labels)
print(x.shape)

# Visualise data. Most values are 0, we remove these values to focus
our analysis in the brain mass
plt.hist(x[x>0],70)
plt.show()

# Visualize one image
plt.imshow(x_im, cmap='gray')
plt.grid(None)
plt.show()
```

```
x_im shape: 207 256
(52992, 1)
```





## Intepreting the Brain Tissue Figures

It should be noted, that the main point here is that we are looking at a variety of pixel values, and their intensities. As you can see from the histogram, it would appear that there are different peaks to the data. These correspond roughly to the type of tissue present in the brain. To give a bit of context, different regions of the brain, when run through different scanning protocols will have different densitities. You may have heard of white matter and grey matter? White matter refers to tracts (i.e. nerves) and is the connection between different brain cells. Grey matter reflects neural cell tissue in the cerebral cortex. In different brain diseases the ratio or overall quantity of different types of brain matter is very important.

```
# Breast cancer data
data = load_breast_cancer()
x = data['data']
labels = data['target']

# We can put this dataset in the dictionary datasets, under the name
"breast"
datasets['breast'] = (x, labels)
print(x.shape)
```

```
(569, 30)
```

```
# Classic Iris dataset
iris = load_iris()
```

```
x = iris.data
labels = iris.target

# We can put this dataset in the dictionary datasets, under the name
"iris"
datasets['iris'] = (x, labels)
print(x.shape)
```

```
(150, 4)
```

```
# Interesting 2D data
data = np.load('clusterable_data.npy')
x = data
labels = None

# We can put this dataset in the dictionary datasets, under the name
"interesting"
datasets['interesting'] = (x, labels)
print(x.shape)
```

```
(2309, 2)
```

```
# Artificial dataset
blobs = make_blobs(n_samples=200, centers=4, n_features=3,
random_state=42)
x = blobs[0]
labels = blobs[1]

# We can put this syntethic dataset in the dictionary datasets, under
the name "artificial"
datasets['artificial'] = (x, labels)
print(x.shape)
```

```
(200, 3)
```

## Apply scaling to the data

The reasoning behind this weird dictionary of different datasets becomes more apparent when we can now apply several pre-processing steps to multiple datasets with ease.

```
# Mean centre and variance scale all datasets independently
for key in datasets:
    x, labels = datasets[key]
    x = StandardScaler().fit_transform(x)
    datasets[key] = (x, labels)
```

## Size and Shape

Data is always much easier to interpret when you know the shape of it. Do I have 10 labels, or 200? Or None?

It is a good point to stop and think... why on earth would I have... no labels? (There is a very straightforward answer)

```
# We can see our dataset
for key in datasets:
    if datasets[key][1] is None:
        print(f'{key}: shape = {datasets[key][0].shape} and labels = 
None')
    else:
        print(f'{key}: shape = {datasets[key][0].shape} and labels = 
{datasets[key][1].shape}')
```

```
scadi: shape = (70, 205) and labels = (70,)
brain: shape = (52992, 1) and labels = None
breast: shape = (569, 30) and labels = (569,)
iris: shape = (150, 4) and labels = (150,)
interesting: shape = (2309, 2) and labels = None
artificial: shape = (200, 3) and labels = (200,)
```

## Section 2 [Students, 15mins]: General functions to apply unsupervised methods and display results

### Supporting functions for viewing data and some specific visualisations

It is often useful to automate display functions to suit your purpose. It is very common to need to plot the same 'type' of plot for either multiple different datasets or for different model runs (with different hyperparameters). Things as simple wrapping an existing plot function and adding a few default parameters and maybe titles and axis labels can make your figures far more professional and at the same time, easier to interpret with less coding... win-win.

```
# Define a function to visualize data using scatterplot
def view_data(x, labels, plotdimx=0, plotdimy=1):
    # View data (choose any two features)
    ax = sns.scatterplot(x=x[:,plotdimx], y=x[:,plotdimy], hue=labels, 
palette=global_palette)
    ax.set_title('Original data')
    plt.show()

# Define a function to visualize brain images
def view_brain_image(x, labels, nx, ny):
    im = x.reshape((nx,ny))
    imlab = labels.reshape((nx,ny))
    plt.imshow(im, cmap='gray')
    plt.grid(None)
    plt.show()
    plt.imshow(imlab, cmap='jet')
    plt.grid(None)
    plt.show()

# Define function to create dendogram
def view_dendrogram(x):
    # Hierarchical clustering
    # Calculate and show dendogram
    Z = linkage(x, 'ward')
    fig = plt.figure(figsize=(25, 10))
    dn = dendrogram(Z)
    plt.gca().set_title('Dendrogram')
    plt.show()

# Applying PCA to data and visualize results
def view_pca(x):
    # PCA visualisation
    pca = PCA()  # find all components
    pca.fit(x)

    # show component vectors (weightings per feature)
    ax = plt.plot(pca.components_[0],'r.', markersize=20)
    plt.plot(pca.components_[1],'b.', markersize=20)
    plt.gca().set_title('PCA Component Weights')
    plt.legend(['PCA Arg[0]', 'PCA Arg[1]'])
    plt.show()

    evr = pca.explained_variance_ratio_

    total = evr * 0
    total[0] = evr[0]


    for count_c in range(1, evr.shape[0]):
        total[count_c] = total[count_c - 1] + evr[count_c]

    # show scree plot of explained variance (the "knee", if it exists,
can indicate data dimensionality)
    plt.plot(evr)
    plt.plot(total, 'k-')
    plt.legend(('Contribution', 'Cumulative'))

    plt.gca().set_title('PCA Explained Variance')
    plt.show()
```

## Generic function for data visualisation

This can be used by many methods (most of the code is just for plotting)

```
# Generic function to apply transformation to data and visualize it
def visualisation(method, x, labels, plotdimx=0, plotdimy=1):
    # General Data Visualisation
    y = method.fit_transform(x)  # this line does all the work!
    # do some plots
    ax = sns.scatterplot(x=y[:,plotdimx], y=y[:,plotdimy], hue=labels,
style=labels, palette=global_palette) #seaborn plotting routine
    ax.set_title('Data Visualisation (true labels set both colours and
shapes)')
    plt.show()
    return None
```

## Section 3 [Students, 15mins]: Application of the functions to the datasets and exploration of the results

### Below here is where you start making changes

Separate cells for exploring visualisation (first) and clustering (second)

Also see suggested list of things to do after these cells

### Visualisation and dimensionality reduction

Apply some visualisation methods

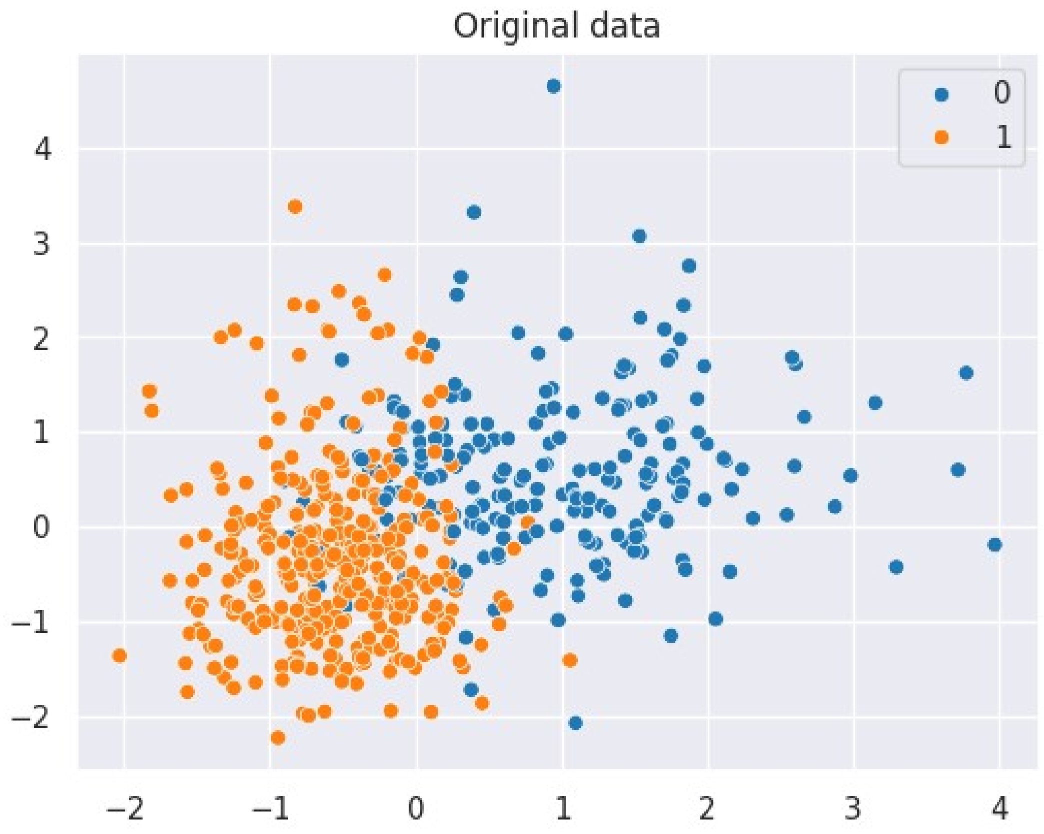
Note that many of these can also be used for dimensionality reduction

Think about the attributes of a dataset that allow the different methods, incl. PCA and dendrogram, to work correctly (or note errors that indicate this)

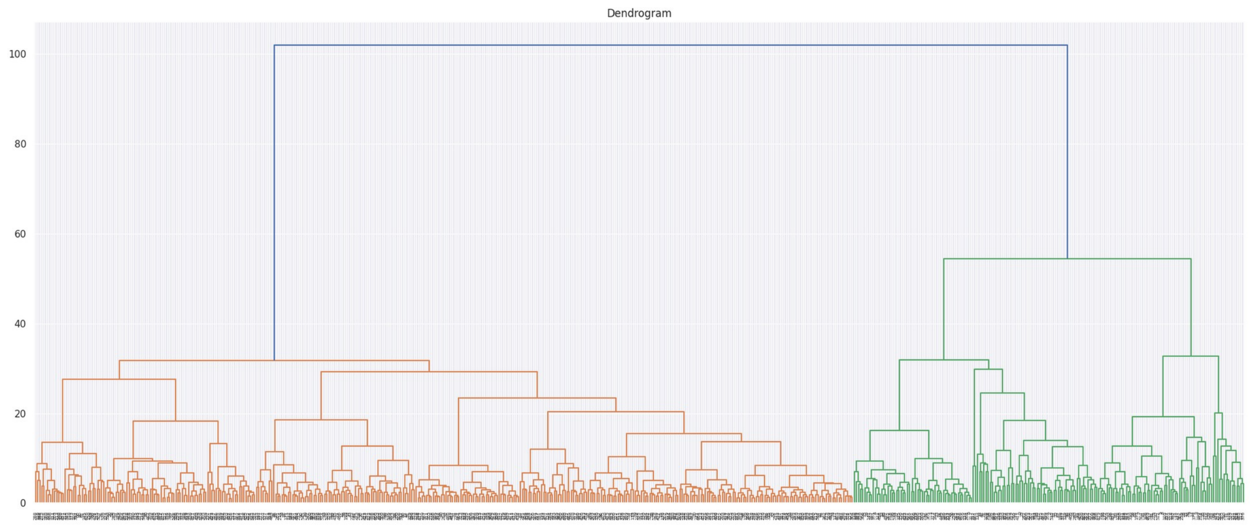
```
n_comp = 2   # YOUR_CHOICE (though for visualisation 2D is common)

tsne = manifold.TSNE(n_components=n_comp)
mds = manifold.MDS(n_components=n_comp,normalized_stress='auto')
lle = manifold.LocallyLinearEmbedding(n_components=n_comp)
isom = manifold.Isomap(n_components=n_comp)
pca = PCA(n_components=n_comp)

# We go to use Iris dataset in first stage
x, labels = datasets['breast']  # e.g. 'iris', 'breast', 'scadi'

# View data allows you to show a scatterplot of your dataset using
only two dimensions of the data
print("Simple visualization using two features")
if labels is not None:
    view_data(x, labels)

# A dendogram. This will not always be appropriate
print("Dendogram visualization")
```

```
view_dendrogram(x)

# Implementation of PCA
print("Getting PCA features")
view_pca(x)

# Implementation of MDS
print("Implementation of PCA")
visualisation(pca, x, labels) # You can implement another model if you
want
```

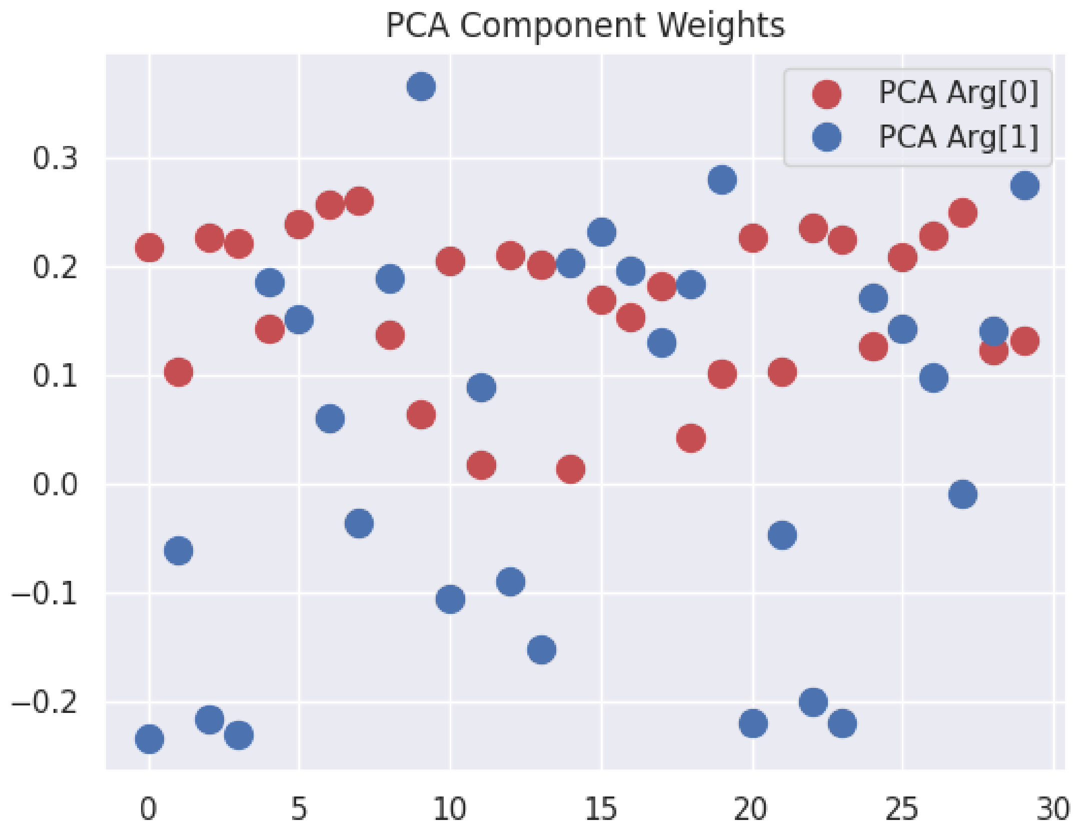
```
Simple visualization using two features
```

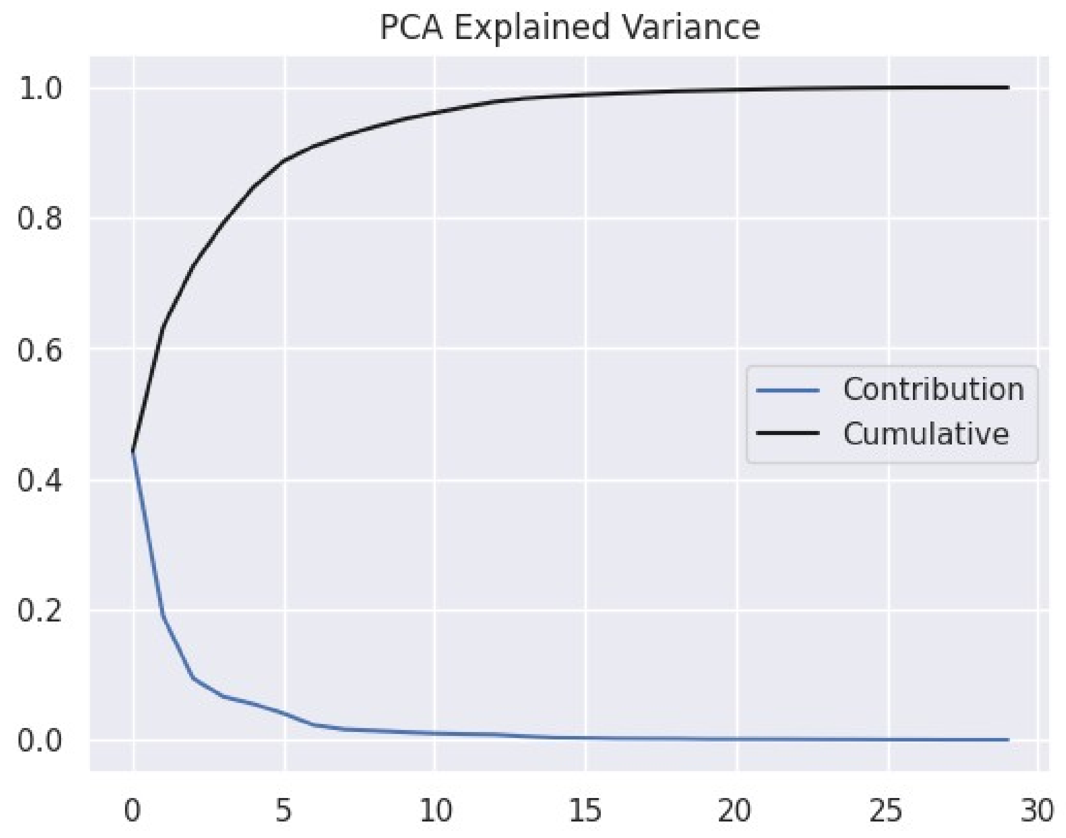


```
Dendogram visualization
```

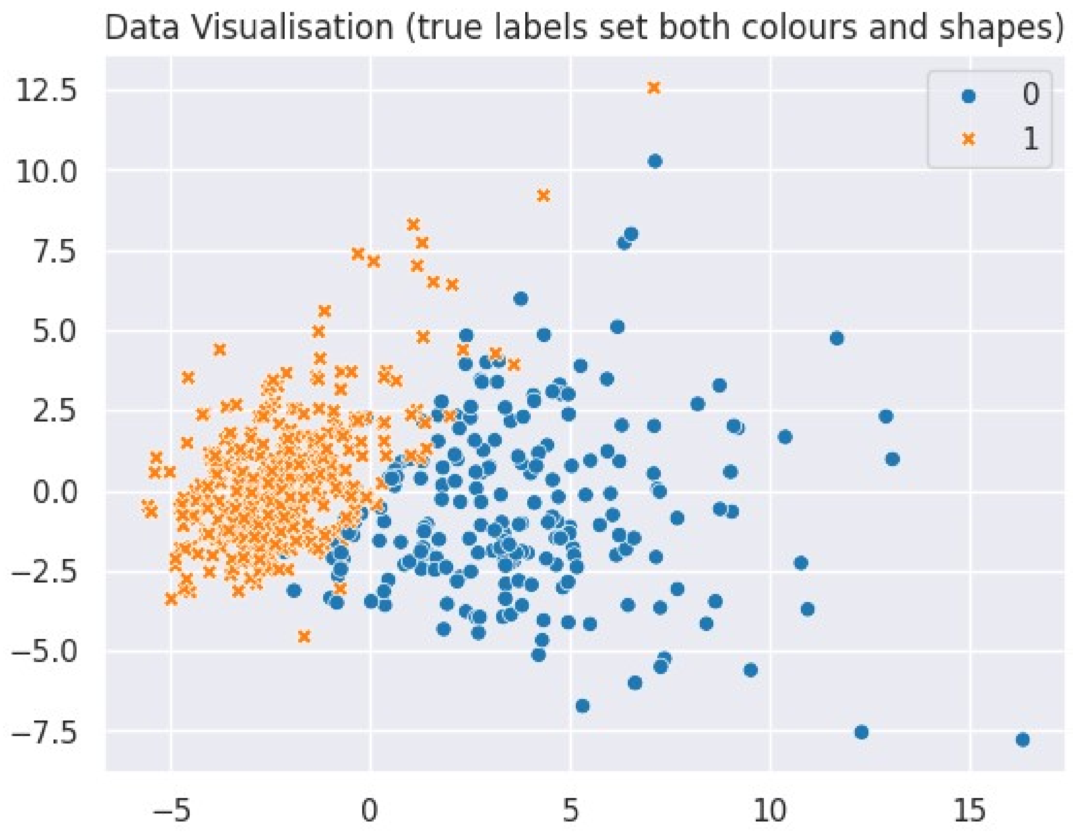


```
Getting PCA features
```





```
Implementation of PCA
```



## Clustering

Generic function for clustering. This can be used by many methods (most of the code is just for plotting)

```
# Define clustering function
def clustering(method, x, labels, plotdimx=0, plotdimy=1):
    # General clustering (note that labels are only used for plotting
purposes)
    plabs = method.fit_predict(x)   # this line does all the work!
    # do some plots
    if x.shape[1]>1:
        if labels is not None:
            ax = sns.scatterplot(x=x[:,plotdimx],y=x[:,plotdimy],
hue=plabs, size=labels, style=labels, palette=global_palette,
legend=None)
            ax.set_title('Colour is based on cluster labels; Size and
style are based on ground truth')
        else: # if there are no labels
            ax = sns.scatterplot(x=x[:,0],y=x[:,1], hue=plabs,
palette=global_palette, legend=None)
            ax.set_title('Colour is based on cluster labels')
    else:  # for 1D data
        # plt.hist(x,70)
        plt.hist(x[x>-0.5],70)
        plt.show()
        temp_x = x[:,0]
        temp_y = np.random.rand(temp_x.shape[0])
        sns.scatterplot(x=temp_x, y=temp_y, hue=plabs, 
palette=global_palette, legend=None)
    plt.show()
    return plabs
```

#**Section 4 [Students, 60mins]: Time to play**

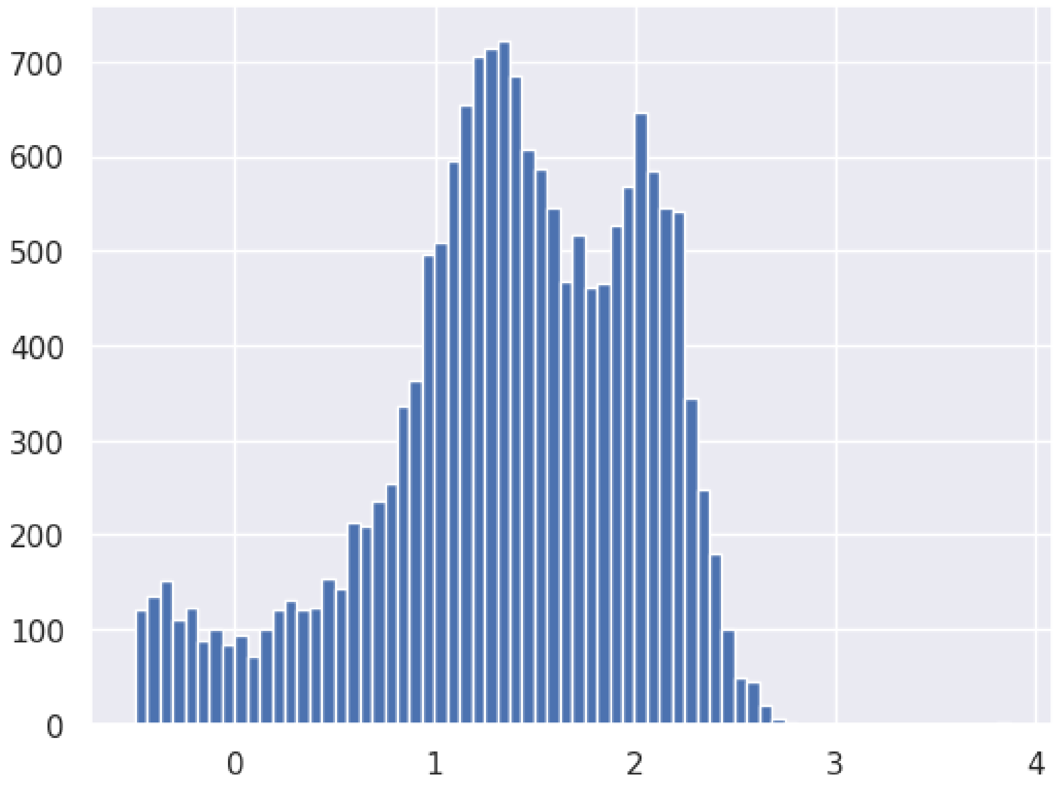
Give the students some time to explore the below sections for an hour or remainder of the class, then discuss solutions/exploration sections at the end.

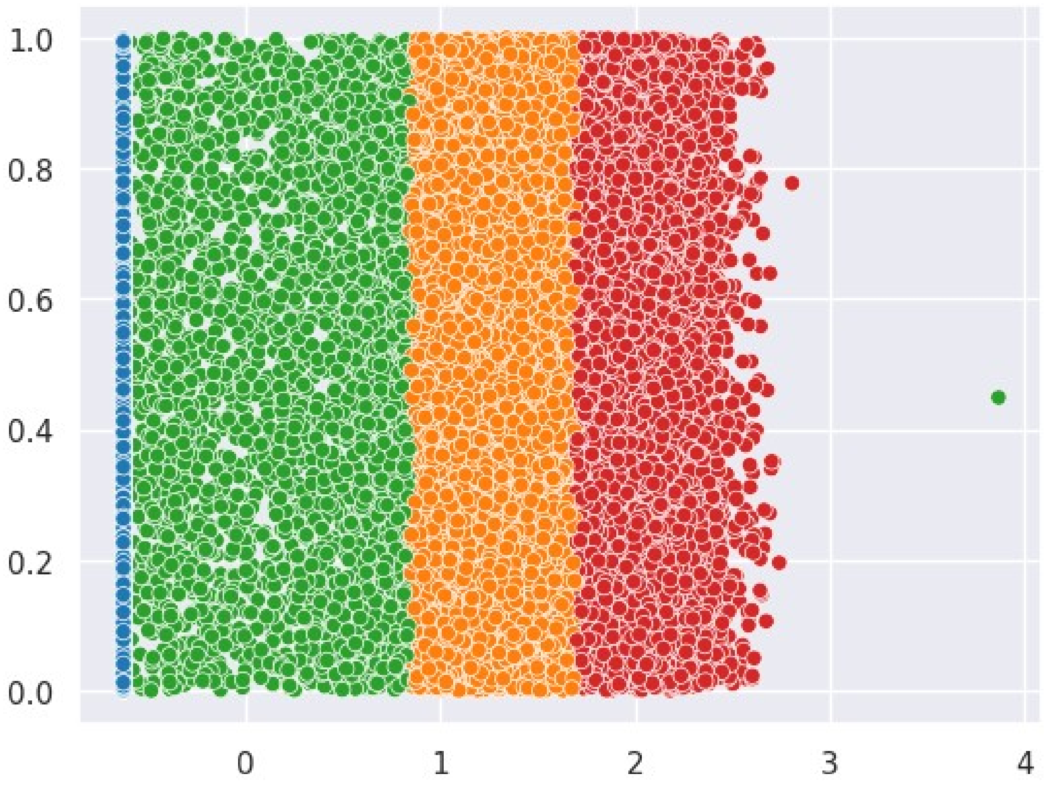
So, what are we doing here? Below are some generic model options ranging from kmeans to DBScan. Each of these models has their pros and cons and each are designed to cluster data. Keep in mind, that although we do have ground-truth data, these methods aren't necessarily using any kind of supervision. We are evaluating the models using the ground truth (evaluate is used loosely here, it more a visual evaulation) but the ground truth isn't influencing algorithms like K-means or DBScan's method for clustering.

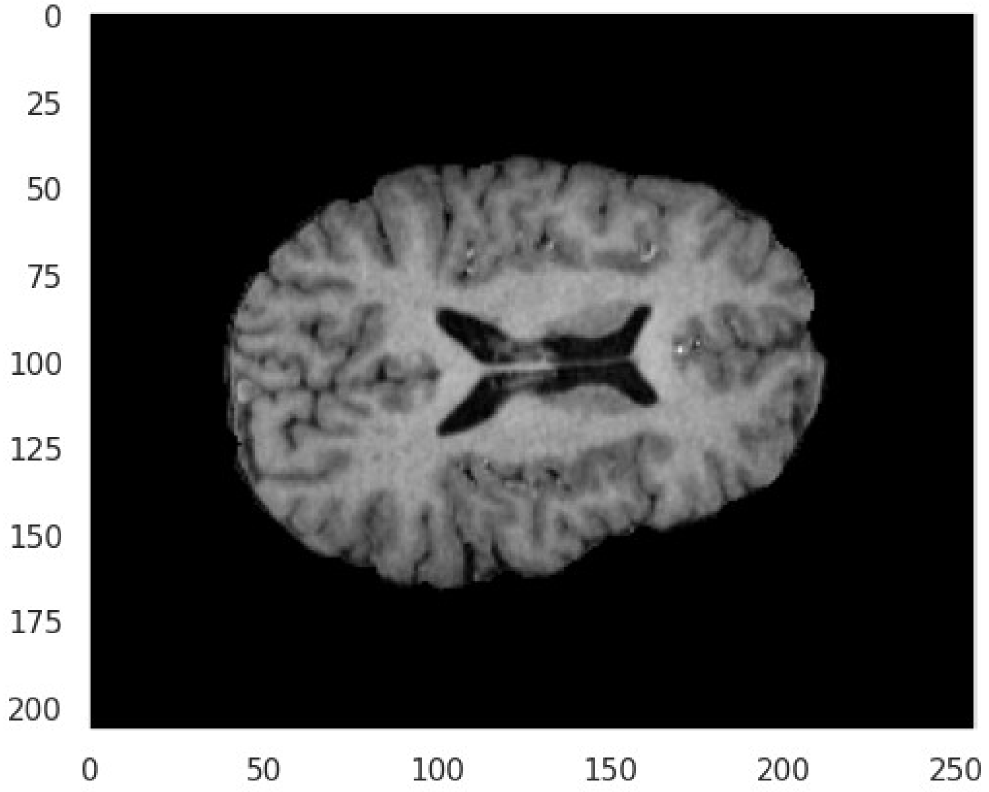
So, by using the pre-written functions below ('clustering') we can test different algorithms with different hyper-parameters on a variety of different datasets. Have fun ^_^

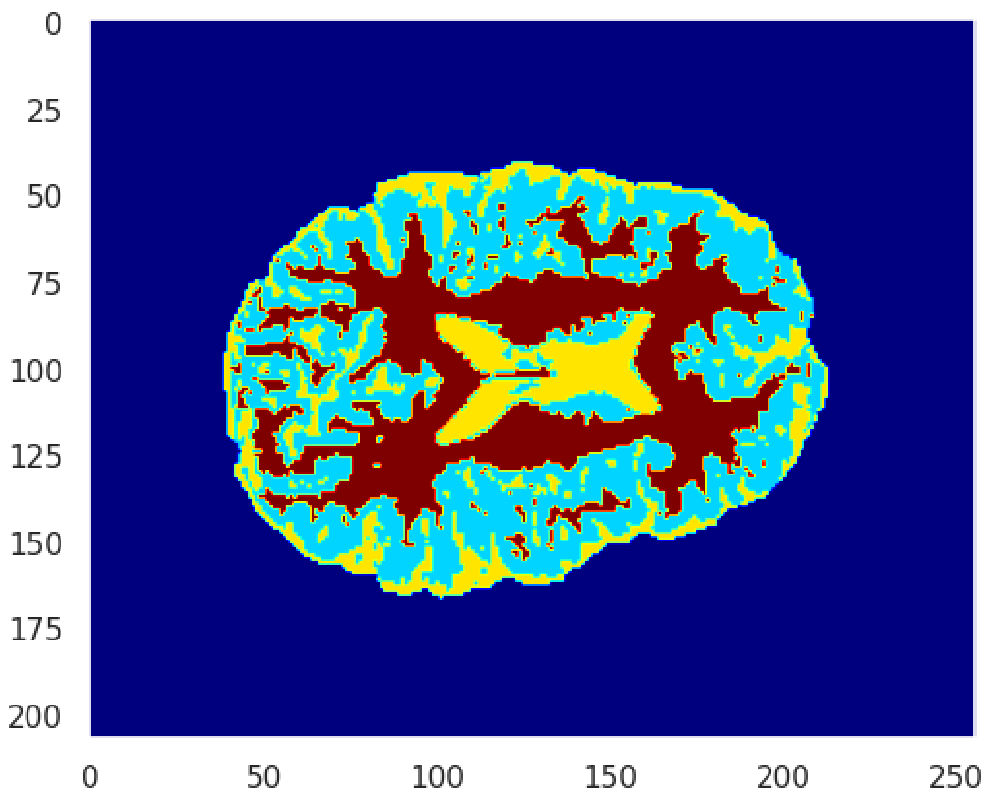
1. List item
2. List item

```
num_clusters = 4  # YOUR_CHOICE (think about the dataset you are 
using)

kmeans = cluster.KMeans(n_clusters=num_clusters,n_init='auto')
gmm = mixture.GaussianMixture(n_components=num_clusters)
ward = cluster.AgglomerativeClustering(n_clusters=num_clusters, 
linkage='ward')
dbsc = cluster.DBSCAN(eps=5, min_samples=10)  # These are the key 
parameters to change
spec = cluster.SpectralClustering(n_clusters=num_clusters)


chosen_dataset = 'brain'

x, labels = datasets[chosen_dataset]   # e.g. 'iris', 'breast', 
'brain' etc
print("Data and Labels")
if labels is not None:
    view_data(x, labels)

print("Clustering")
plabs = clustering(gmm, x, labels)  # e.g. kmeans, gmm
if chosen_dataset == 'brain':
    view_brain_image(x, plabs, nx, ny)    # only use if you select
'brain' above

Data and Labels
Clustering
```




1.0
0.8
0.6
0.4
0.2
0.0
0
1
2
3
4


0
25
50
75
100
125
150
175
200
0
50
100
150
200
250



## Ideas for things to do:

After exploring different visualisation and clustering methods on a range of datasets, consider exploring some of the following in a group or on your own:

1) Results across different datasets using default methods (visualisation and clustering)

2) Effect of parameters on particular methods (for a limited number of datasets)

3) Stability with respect to noise

4) Stability with respect to dataset size (subsampling)

5) Measuring consensus amongst methods

6) Measuring "success" with respect to labels (when available)

7) Try using PCA for dimensionality reduction and feeding the reduced datasets into clustering methods

##**Section 4.2 [TA, 20mins]: Discuss potential exploration/solutions**

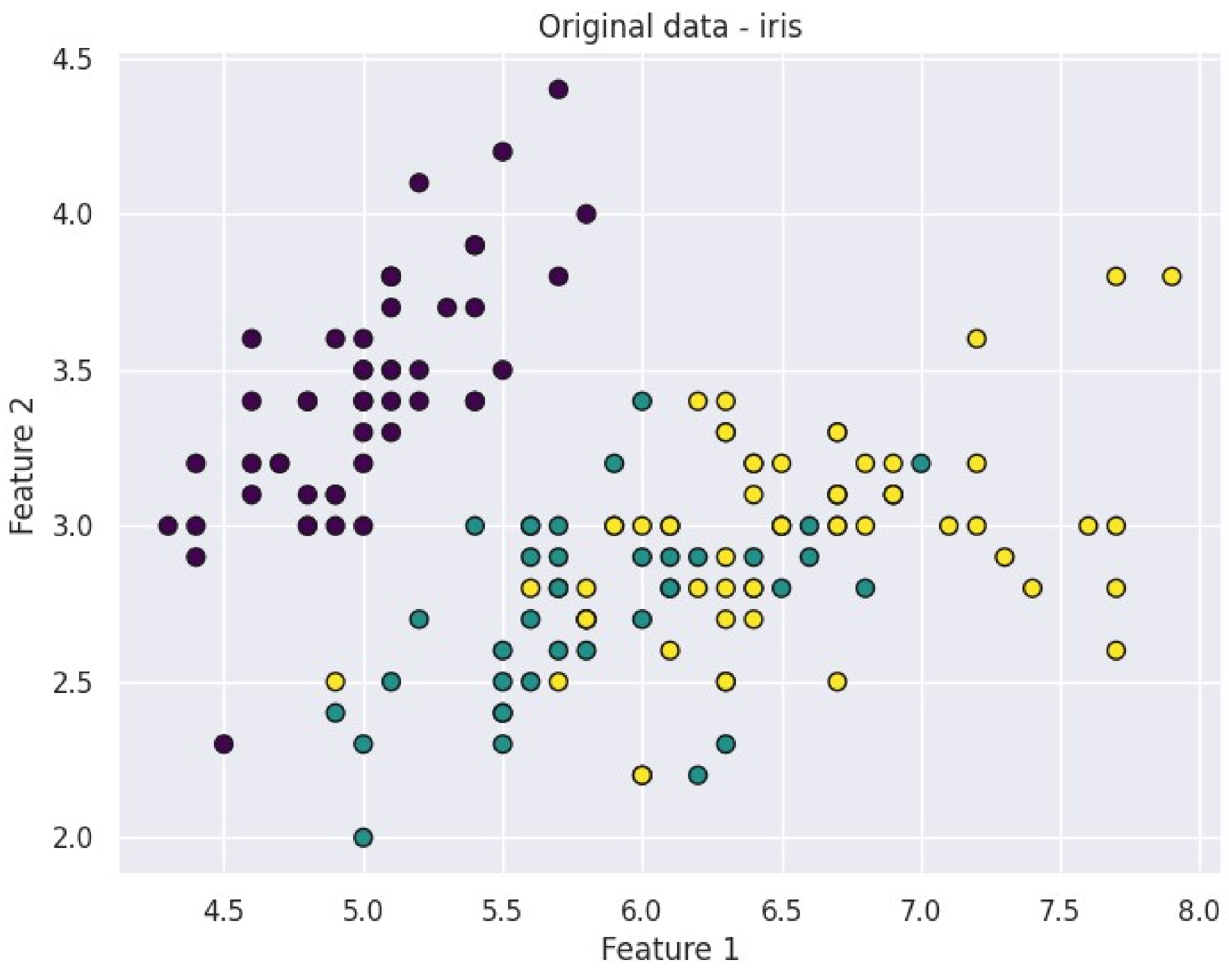
##**1. Results across different datasets using default methods (visualisation and clustering)**

** 1) Results Across Different Datasets Using Default Methods**

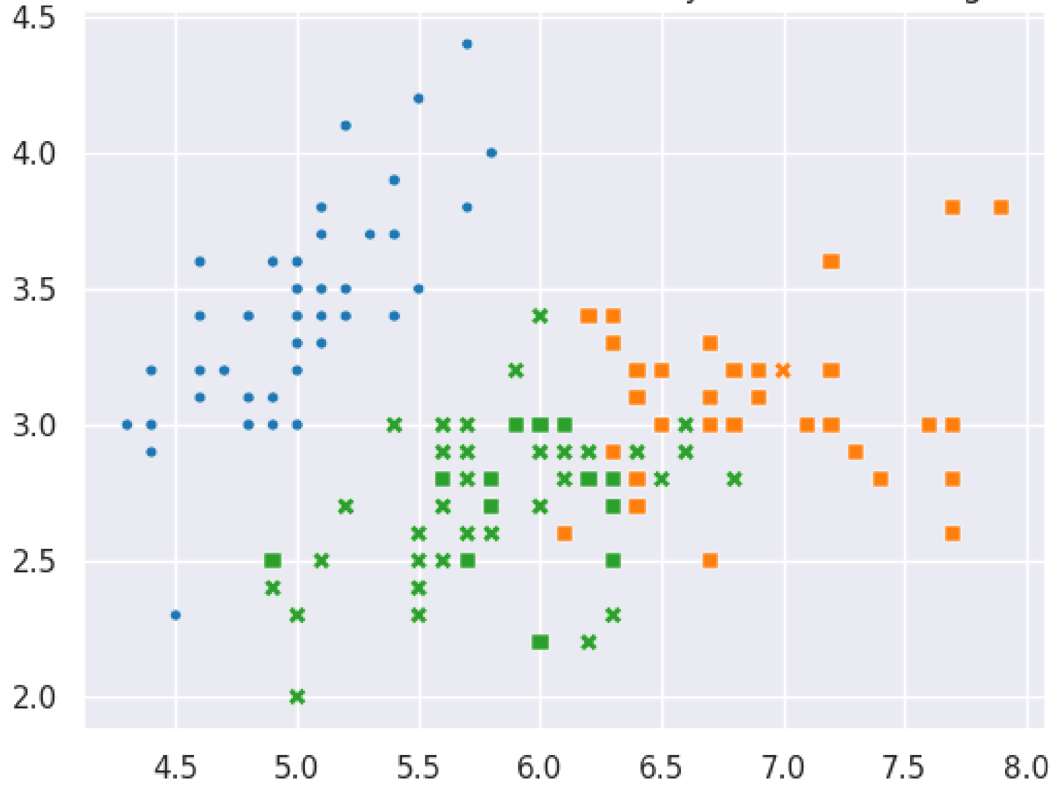
In this section, we explore how the default clustering method (KMeans) performs on different datasets. We will iterate through a list of datasets, apply the KMeans clustering algorithm with a predefined number of clusters, and visualize the results. This helps us understand how well the default method works across various types of data.

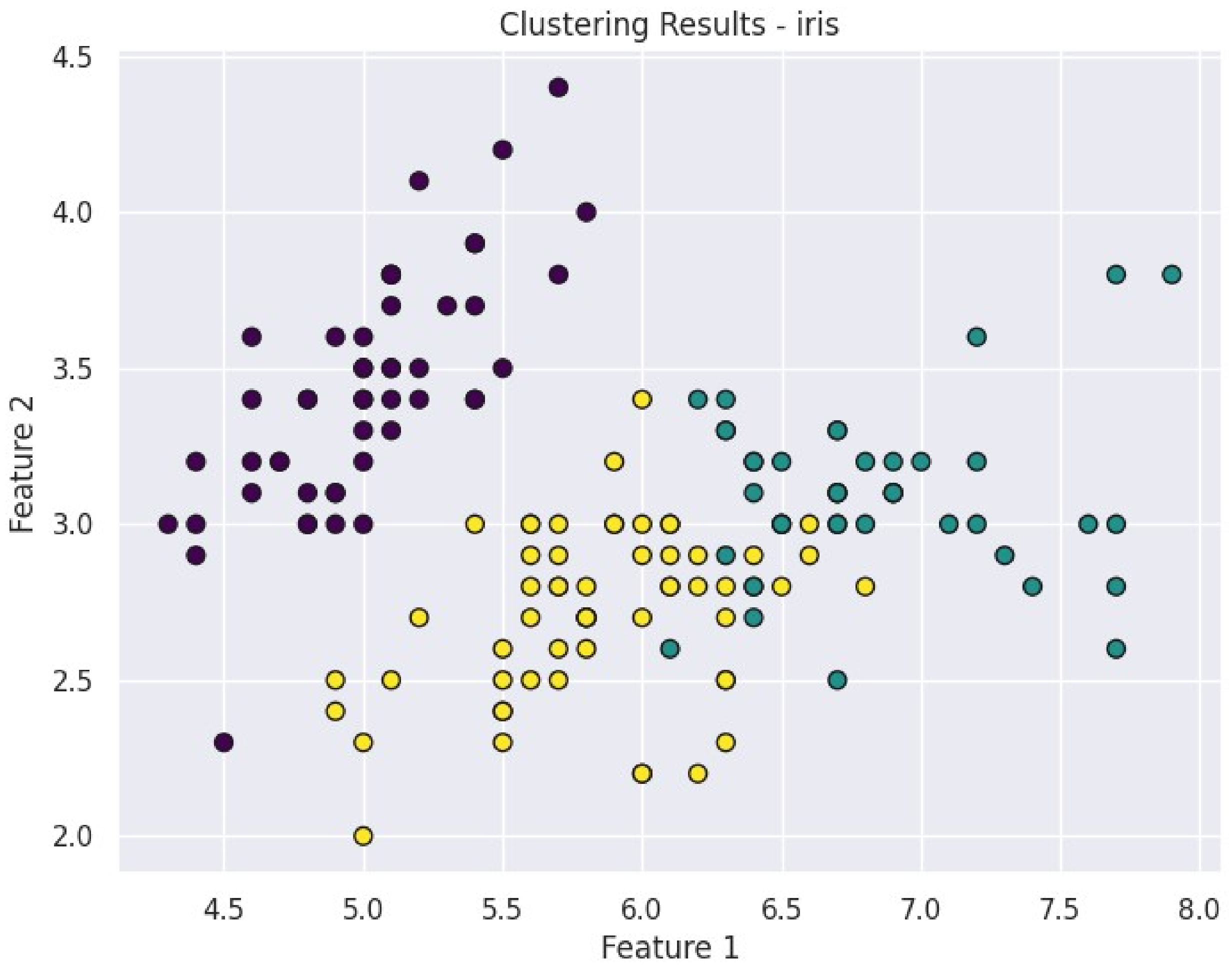
```python
import numpy as np
import matplotlib.pyplot as plt
from sklearn.datasets import make_blobs
from sklearn import cluster, mixture
from sklearn.utils import resample
from sklearn.decomposition import PCA
from sklearn.metrics import adjusted_rand_score, accuracy_score

# Assuming datasets is a dictionary defined earlier with keys 'iris',
'breast_cancer', 'interesting', 'artificial'

# Function to visualize data
def view_data(x, labels=None, title='Data Visualization'):
    plt.figure(figsize=(8, 6))
    if labels is not None:
        plt.scatter(x[:, 0], x[:, 1], c=labels, cmap='viridis',
edgecolor='k', s=50)
    else:
        plt.scatter(x[:, 0], x[:, 1], edgecolor='k', s=50)
    plt.xlabel('Feature 1')
    plt.ylabel('Feature 2')
    plt.title(title)
    plt.show()

# Ensure all plots are generated and labeled
dataset_names = ['iris', 'breast_cancer', 'interesting', 'artificial']

for dataset_name in dataset_names:
    print(f"Dataset: {dataset_name}")
    x, labels = datasets[dataset_name]

    # Visualize the original data
    if labels is not None:
        view_data(x, labels, title=f'Original data - {dataset_name}')
    else:
        view_data(x, title=f'Original data - {dataset_name}')

    # Perform clustering
    num_clusters = 3 if dataset_name != 'artificial' else 4  # Adjust
based on dataset specifics
    kmeans = cluster.KMeans(n_clusters=num_clusters, n_init='auto')
    plabs = clustering(kmeans, x, labels)

    # Display clustering results
    view_data(x, plabs, title=f'Clustering Results - {dataset_name}')
    print("\n")
```

```
Dataset: iris
```

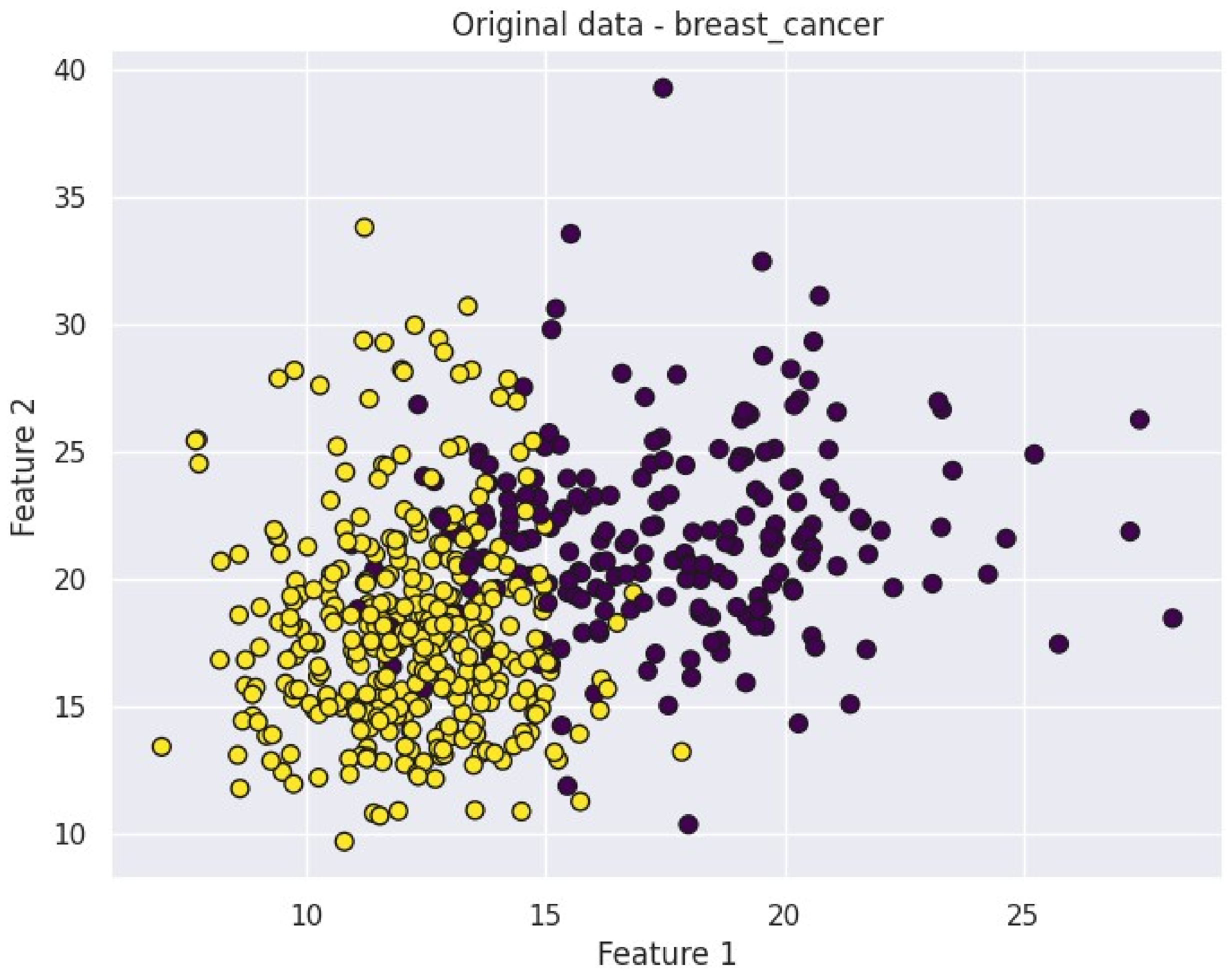


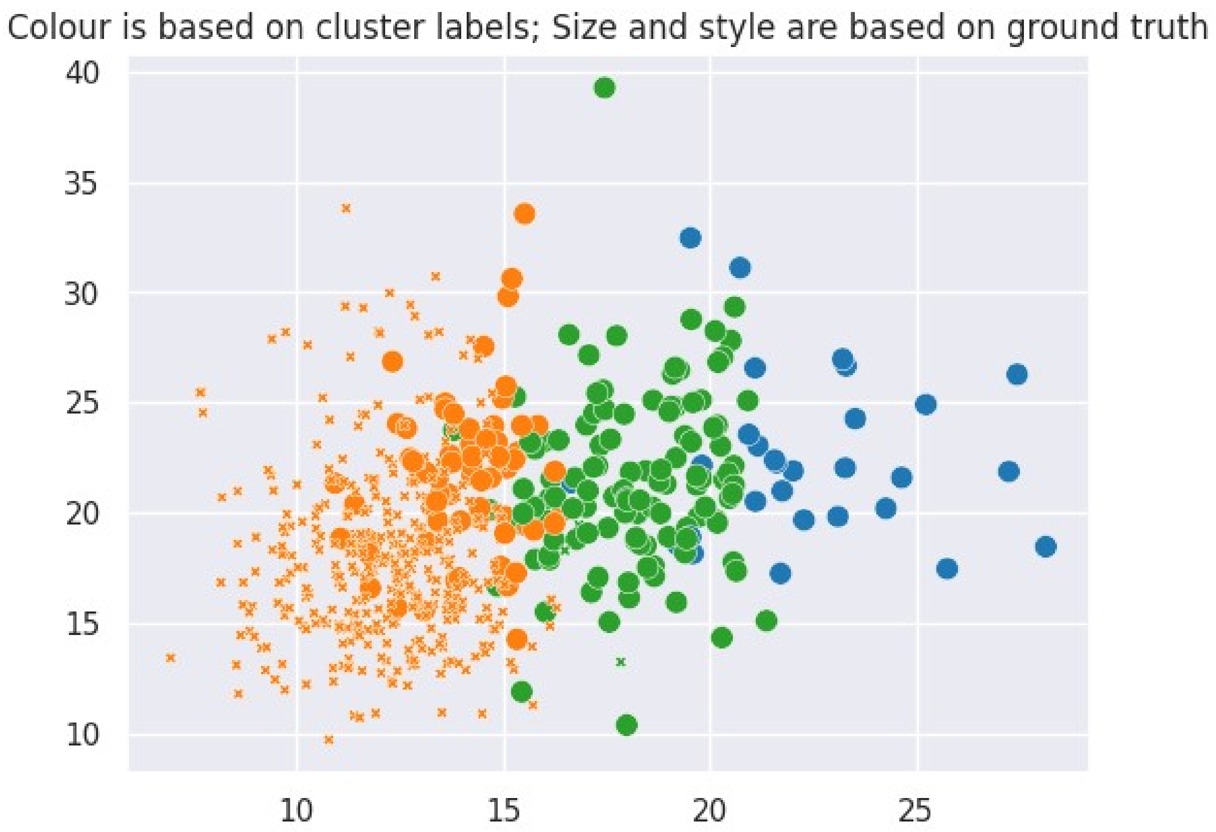
Colour is based on cluster labels; Size and style are based on ground truth

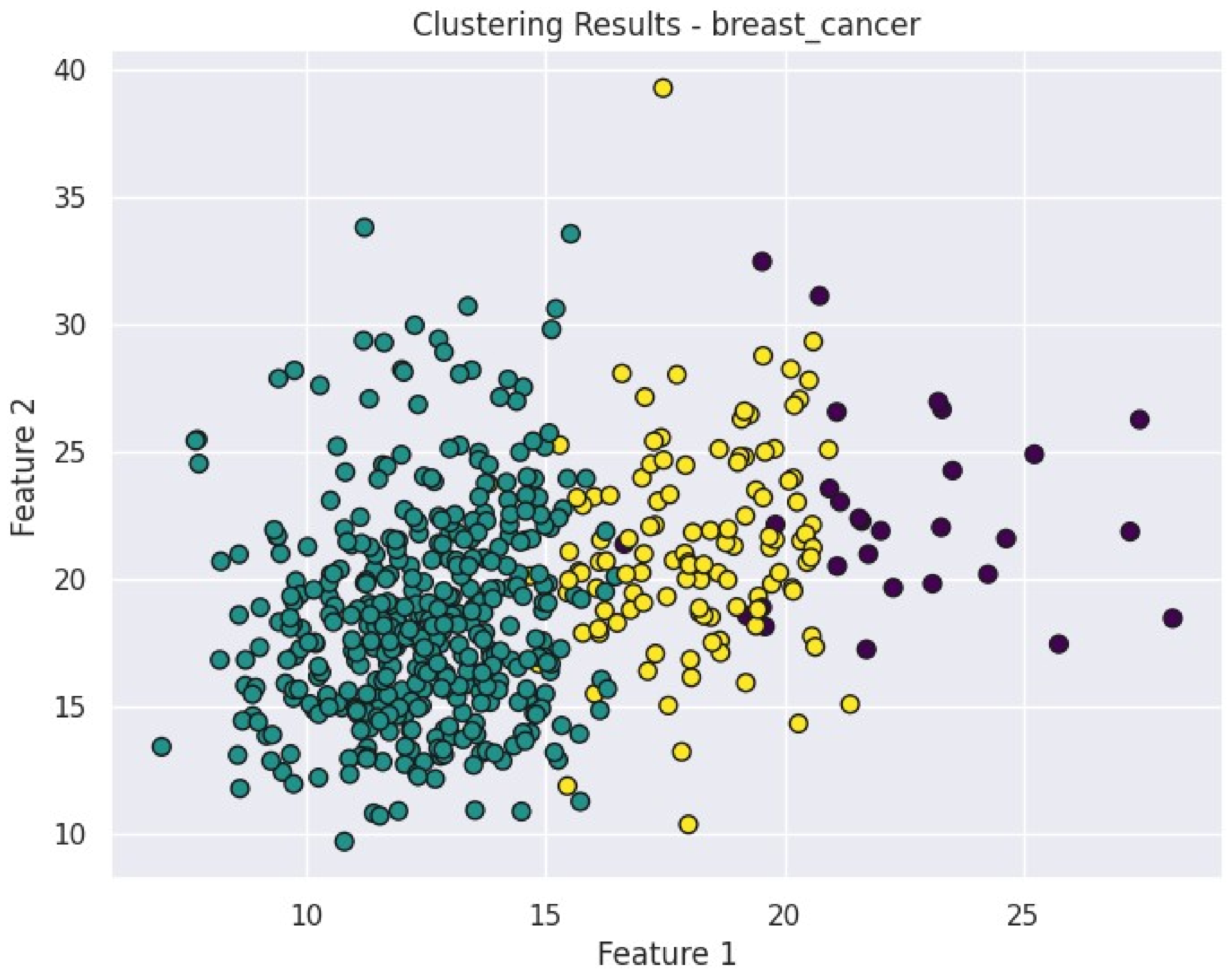




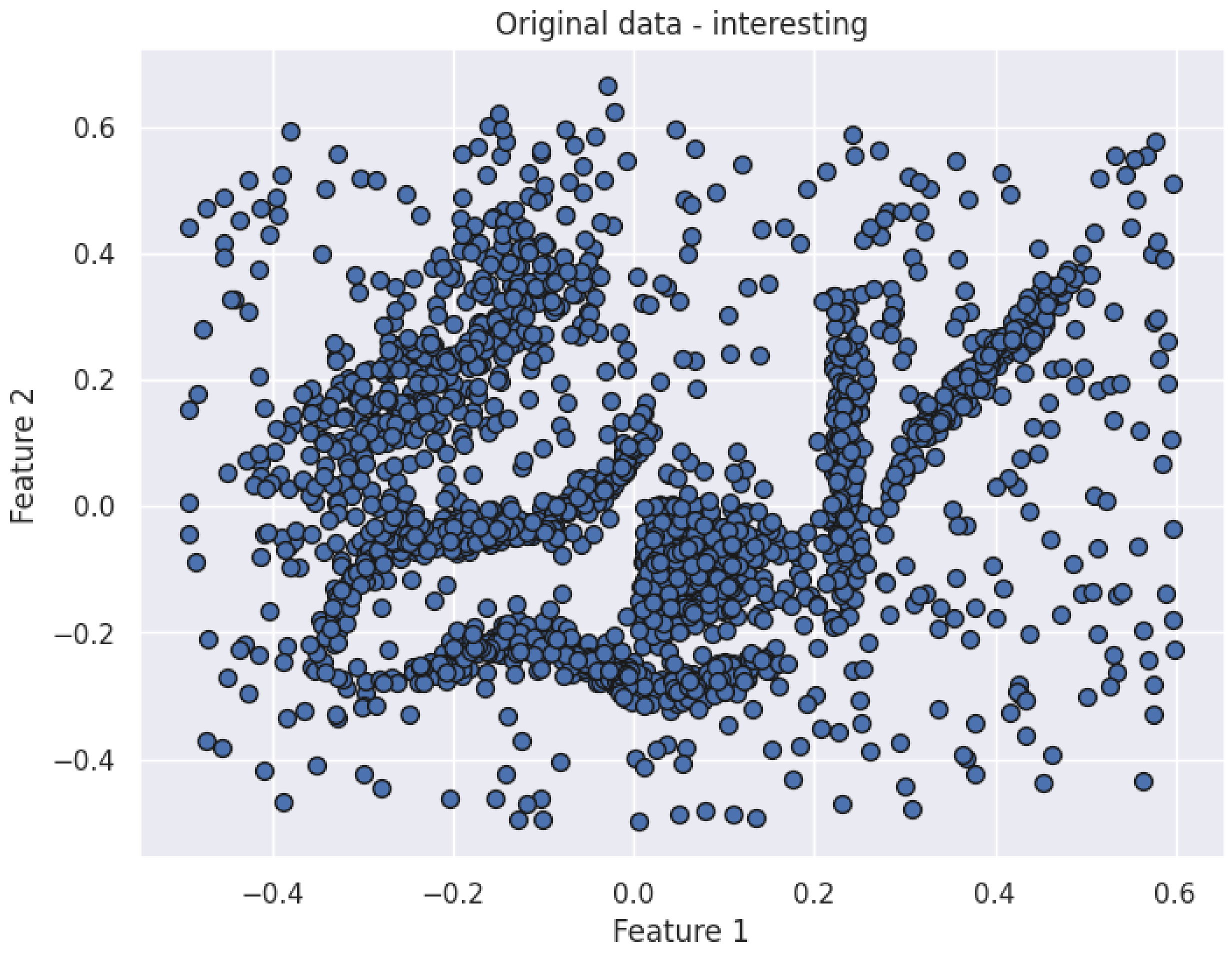
```
Dataset: breast_cancer
```


Original data - breast_cancer
Feature 1
Feature 2
10
15
20
25
30
35
40

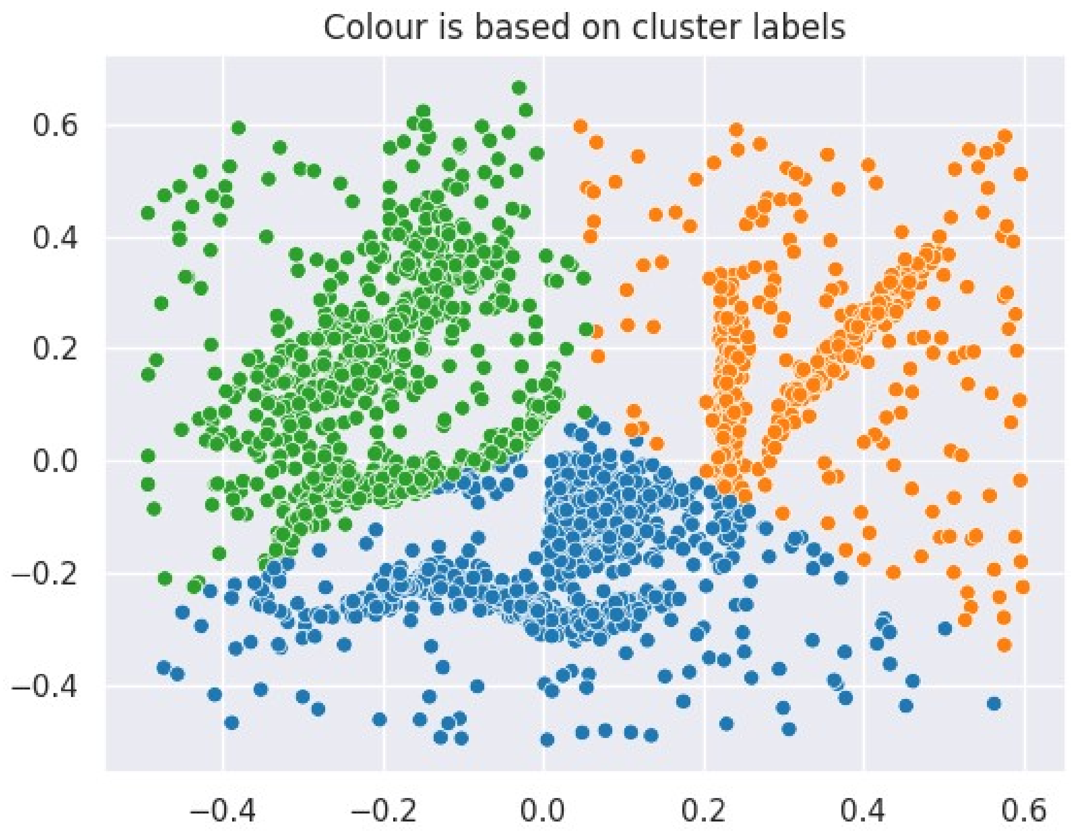

Colour is based on cluster labels; Size and style are based on ground truth
40
35
30
25
20
15
10
10
15
20
25



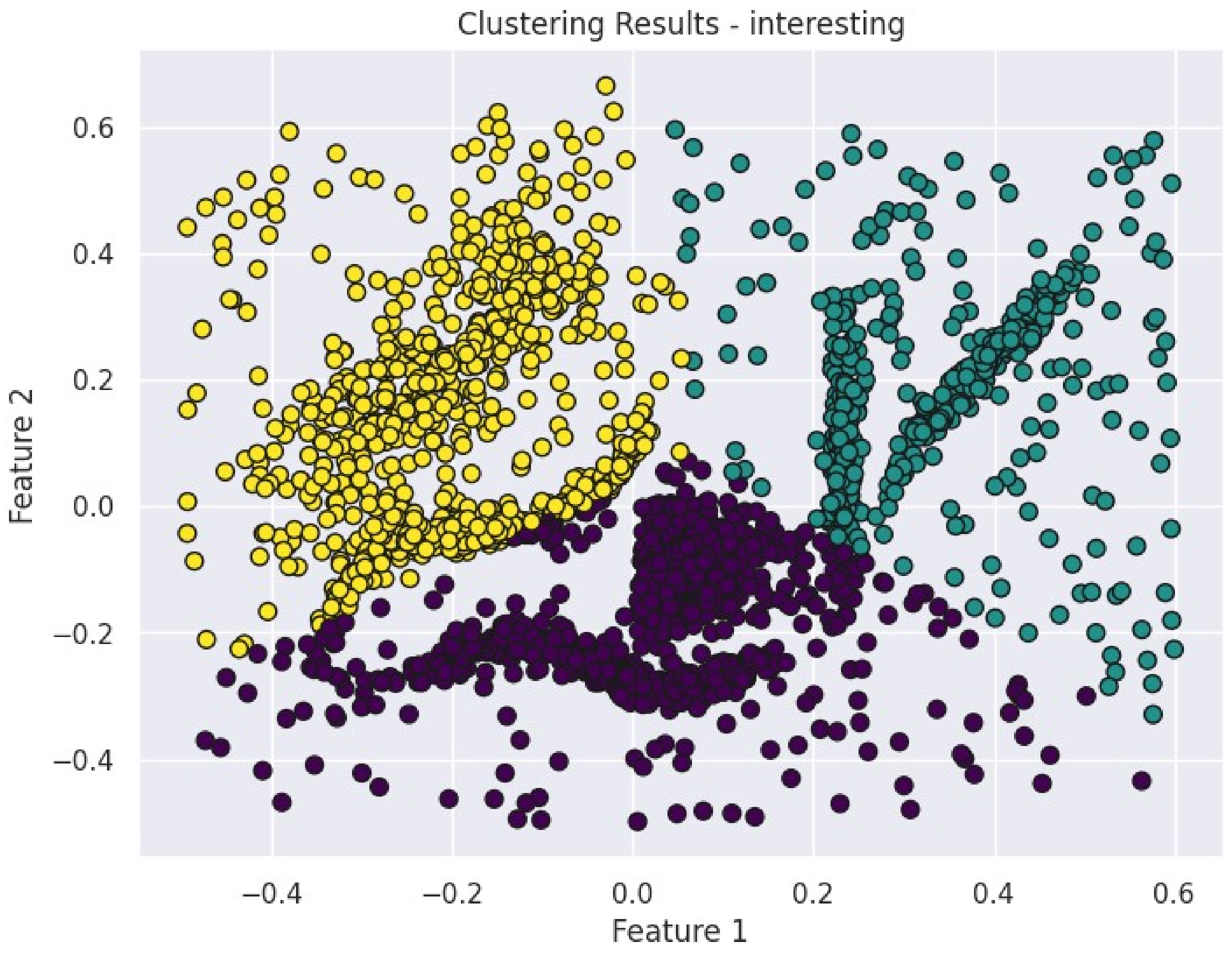
```
Dataset: interesting
```



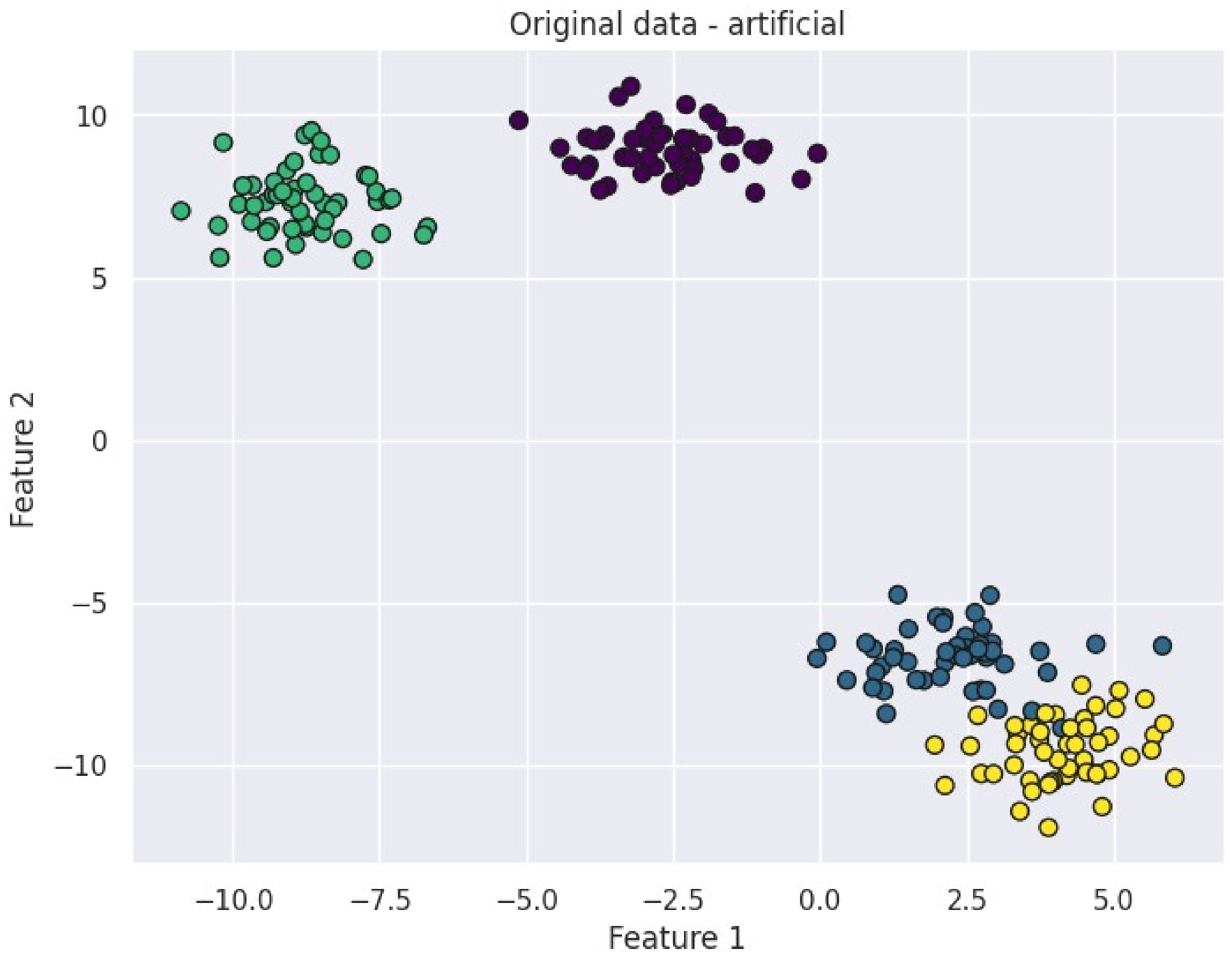
Original data - interesting
Feature 2
Feature 1
0.6
0.4
0.2
0.0
−0.2
−0.4
−0.4
−0.2
0.0
0.2
0.4
0.6

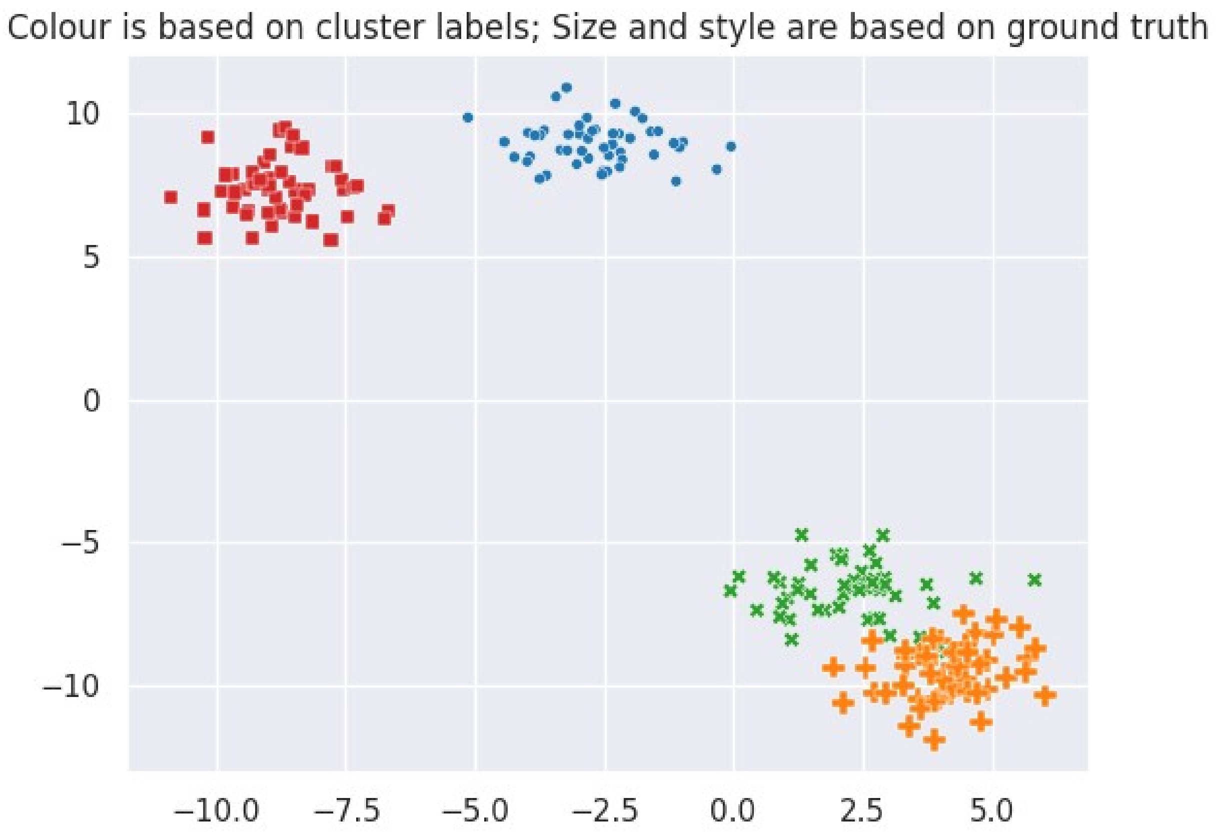


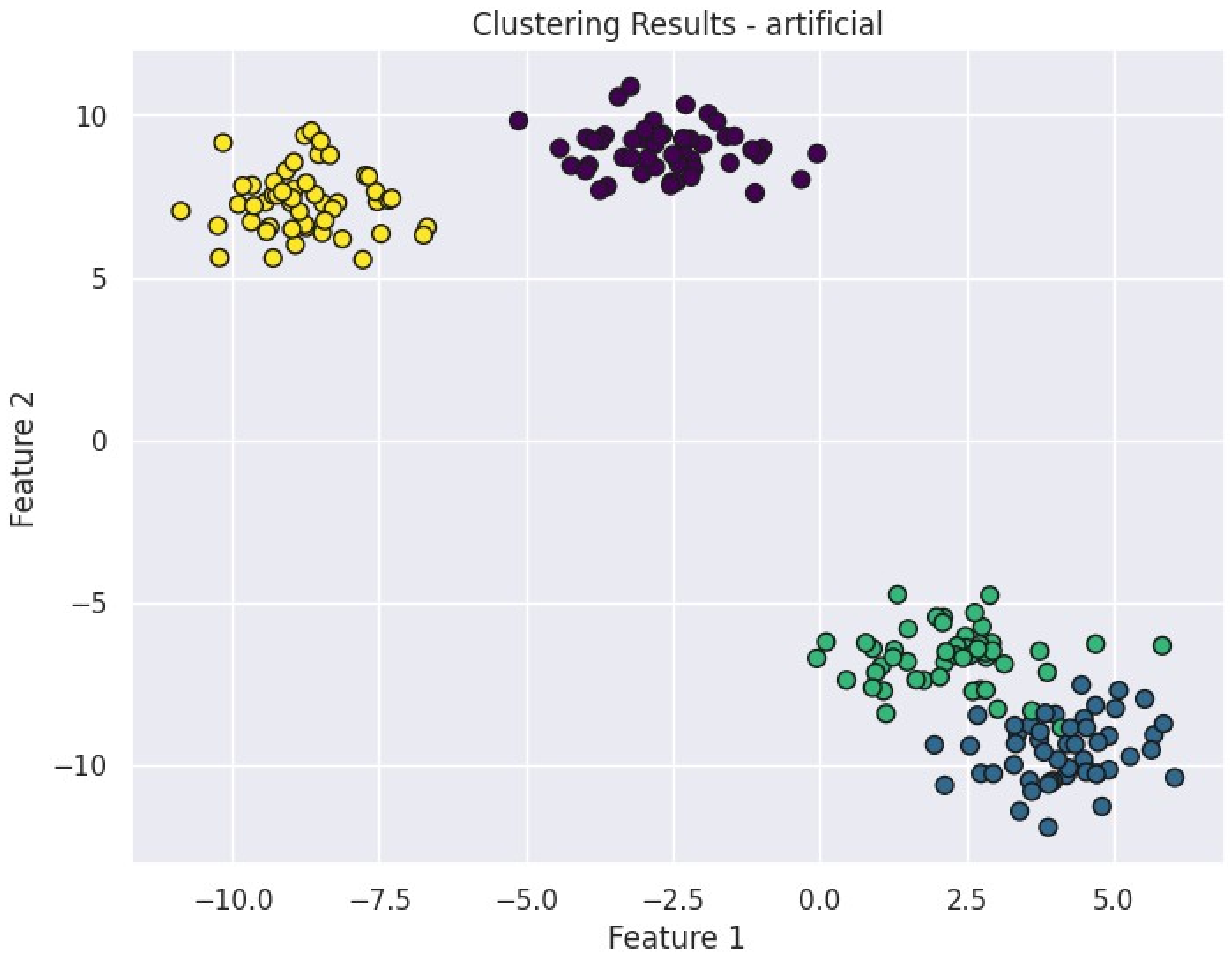
Colour is based on cluster labels
0.6
0.4
0.2
0.0
−0.2
−0.4
−0.4
−0.2
0.0
0.2
0.4
0.6



```
Dataset: artificial
```


Original data - artificial
Feature 2
Feature 1
10
5
0
−5
−10
−10.0
−7.5
−5.0
−2.5
0.0
2.5
5.0


Colour is based on cluster labels; Size and style are based on ground truth
10
5
0
−5
−10
−10.0
−7.5
−5.0
−2.5
0.0
2.5
5.0

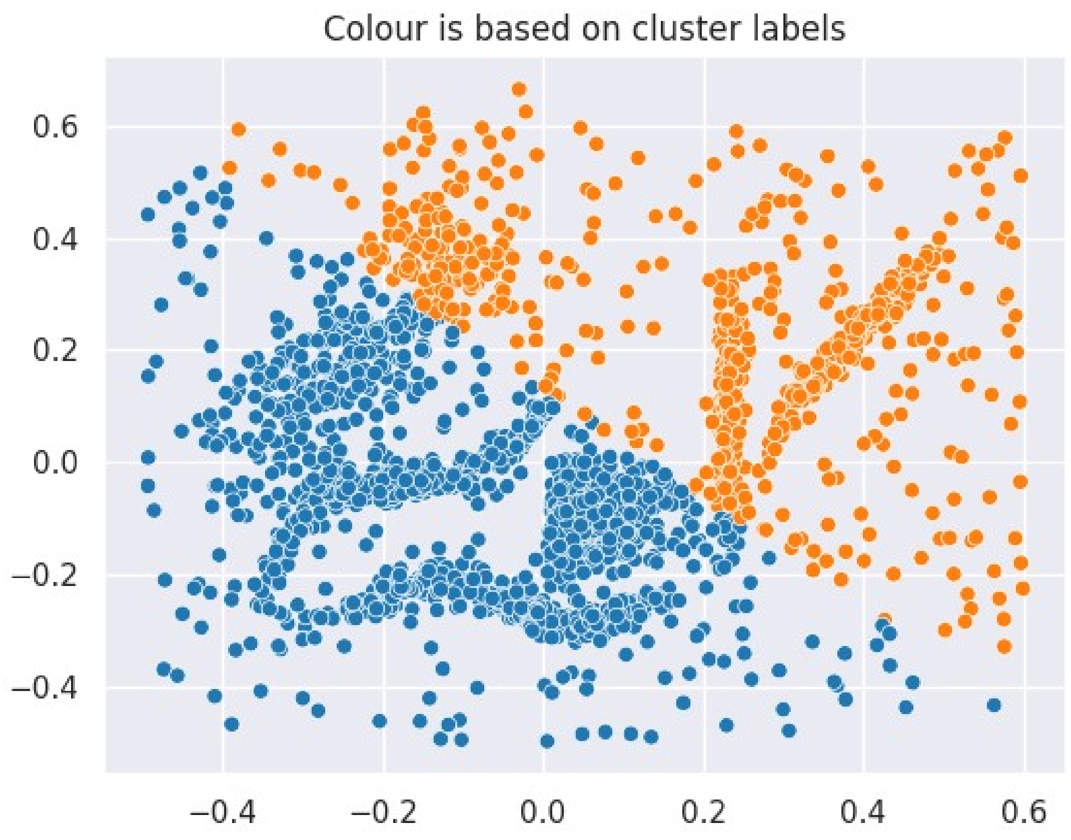


The KMeans clustering algorithm was applied to multiple datasets: iris, breast_cancer, interesting, and artificial. For each dataset, the algorithm was configured to find 3 clusters. The results showed how well the default KMeans method can partition the data into clusters. In the iris dataset, the algorithm was able to separate the different species reasonably well, as expected. For the breast_cancer dataset, the clustering aligned moderately with the known malignant and benign labels. The interesting dataset, which contains more complex patterns, showed varied clustering performance, indicating potential for further parameter tuning. The artificial dataset, designed with clear clusters, demonstrated the effectiveness of KMeans in ideal conditions.

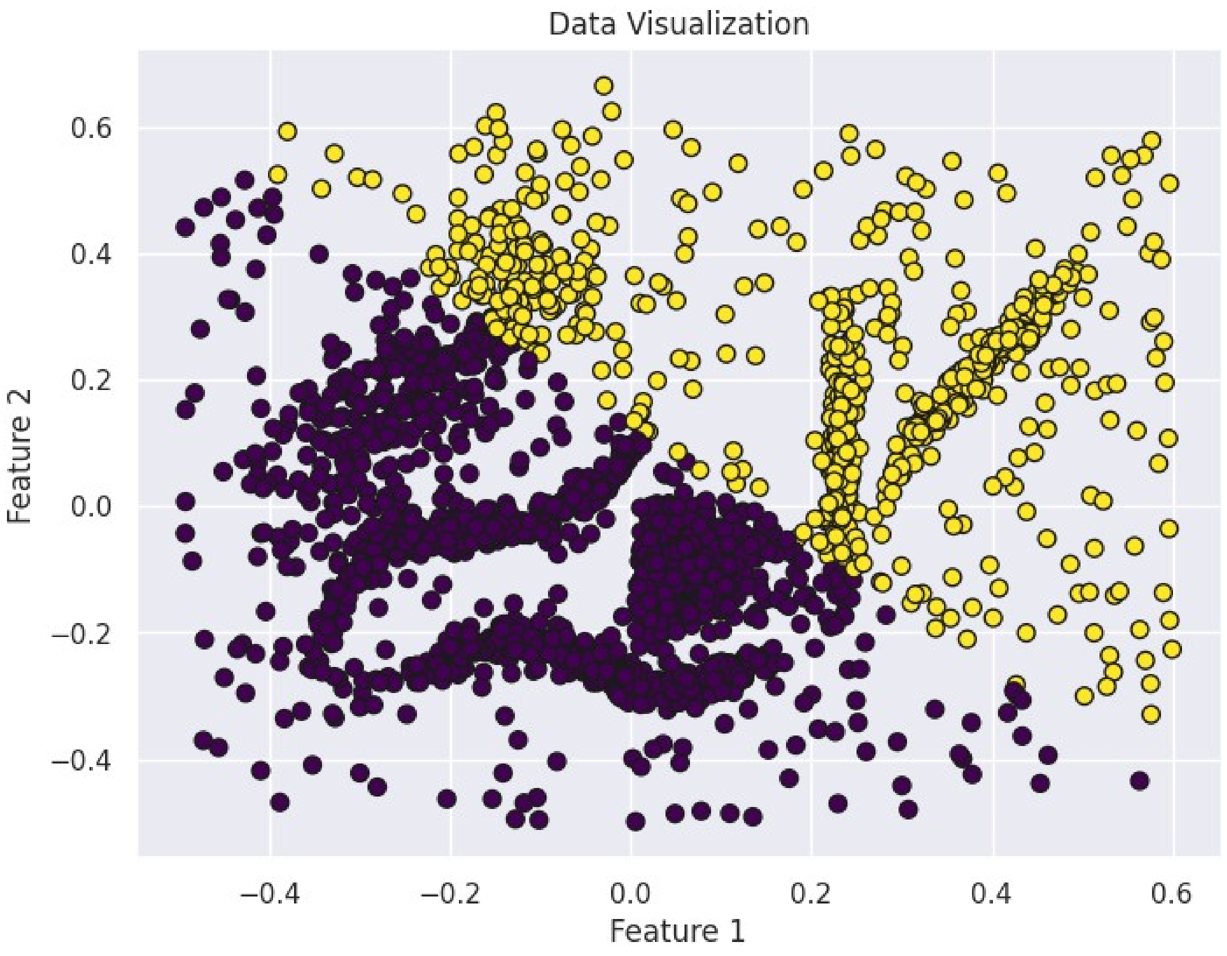
##**2) Effect of parameters on particular methods (for a limited number of datasets)**

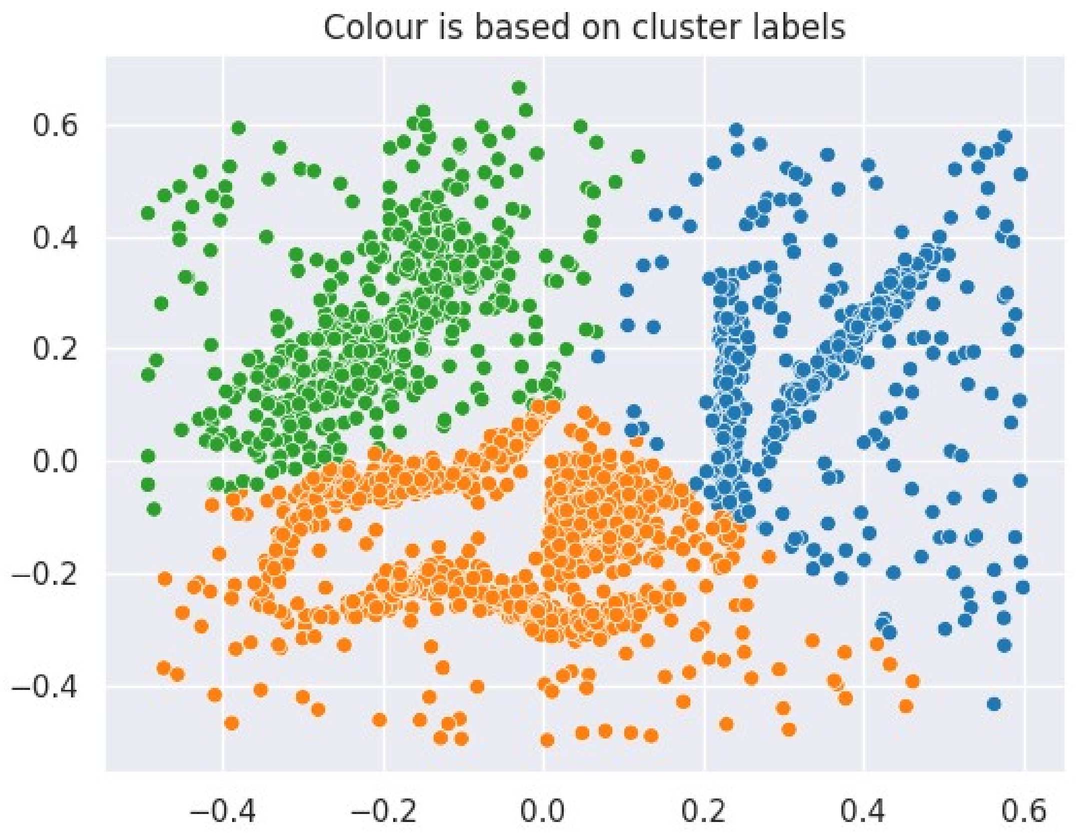
In this section, we examine the effect of changing parameters on the performance of a clustering method (KMeans). We will use a single dataset and vary the number of clusters to see how the results change. This helps us understand the sensitivity of the clustering algorithm to its parameters and identify the optimal settings for our data.

```
chosen_dataset = 'interesting'
x, labels = datasets[chosen_dataset]

num_clusters_list = [2, 3, 4, 5]  # Different values for number of
clusters
for num_clusters in num_clusters_list:
    kmeans = cluster.KMeans(n_clusters=num_clusters, n_init='auto')
    plabs = clustering(kmeans, x, labels)
    print(f"Number of clusters: {num_clusters}")
    view_data(x, plabs)
```

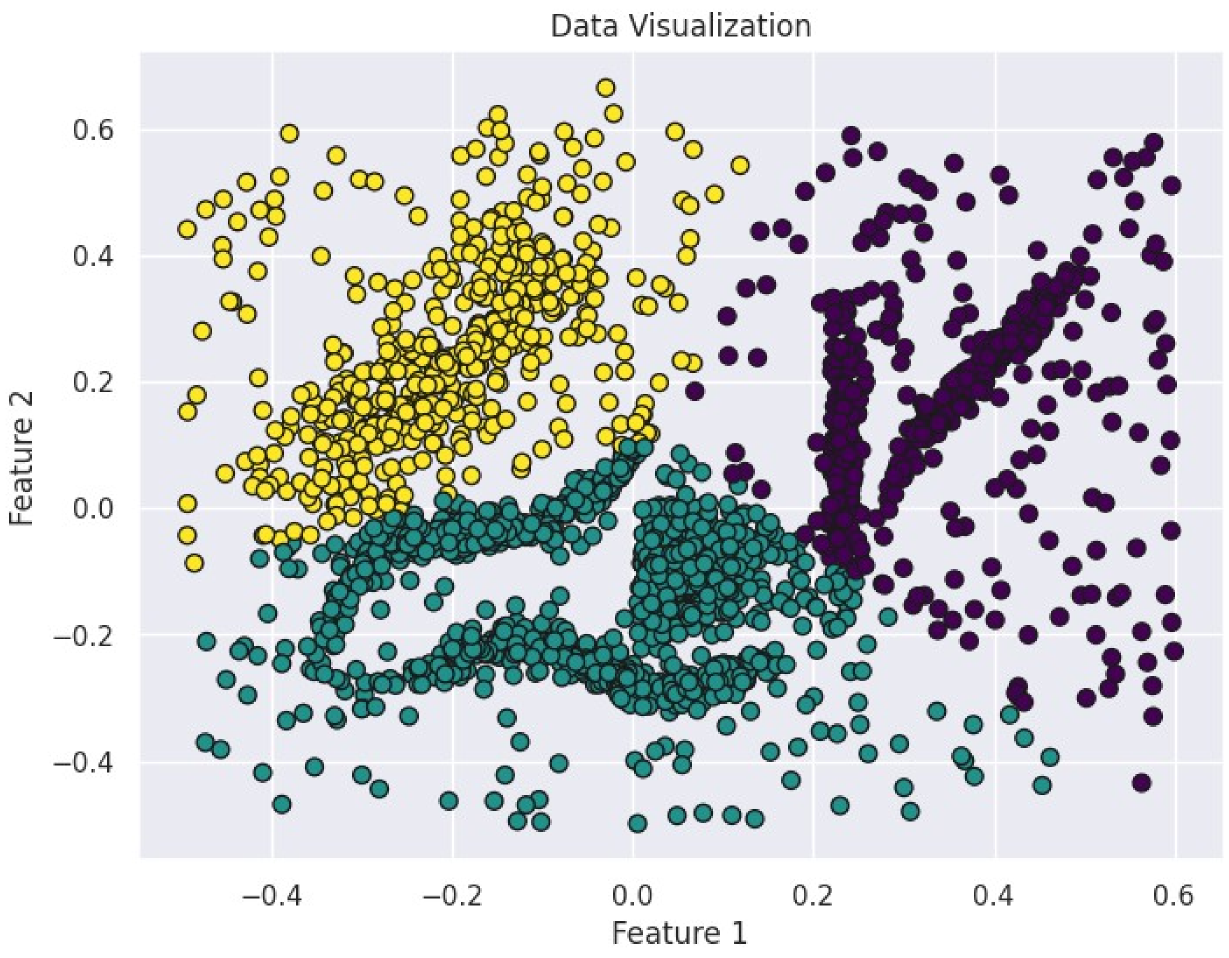


```
Number of clusters: 2
```

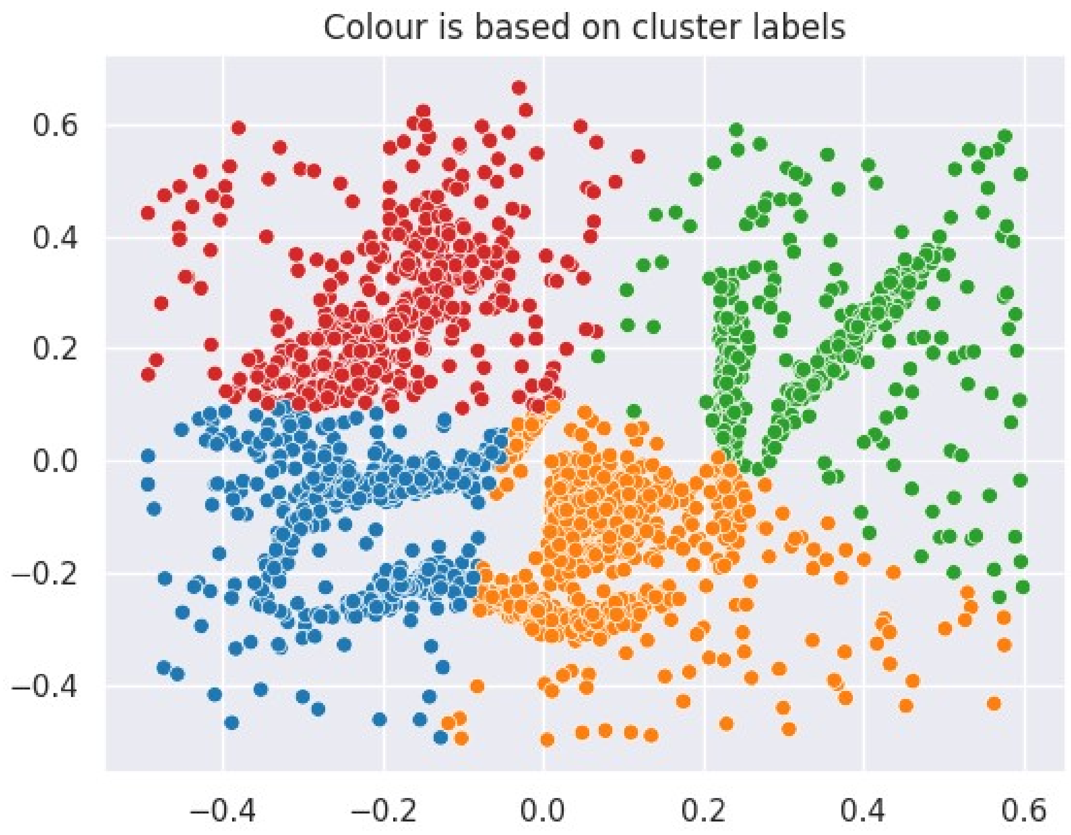

Data Visualization
Feature 2
Feature 1
0.6
0.4
0.2
0.0
−0.2
−0.4
−0.4
−0.2
0.0
0.2
0.4
0.6



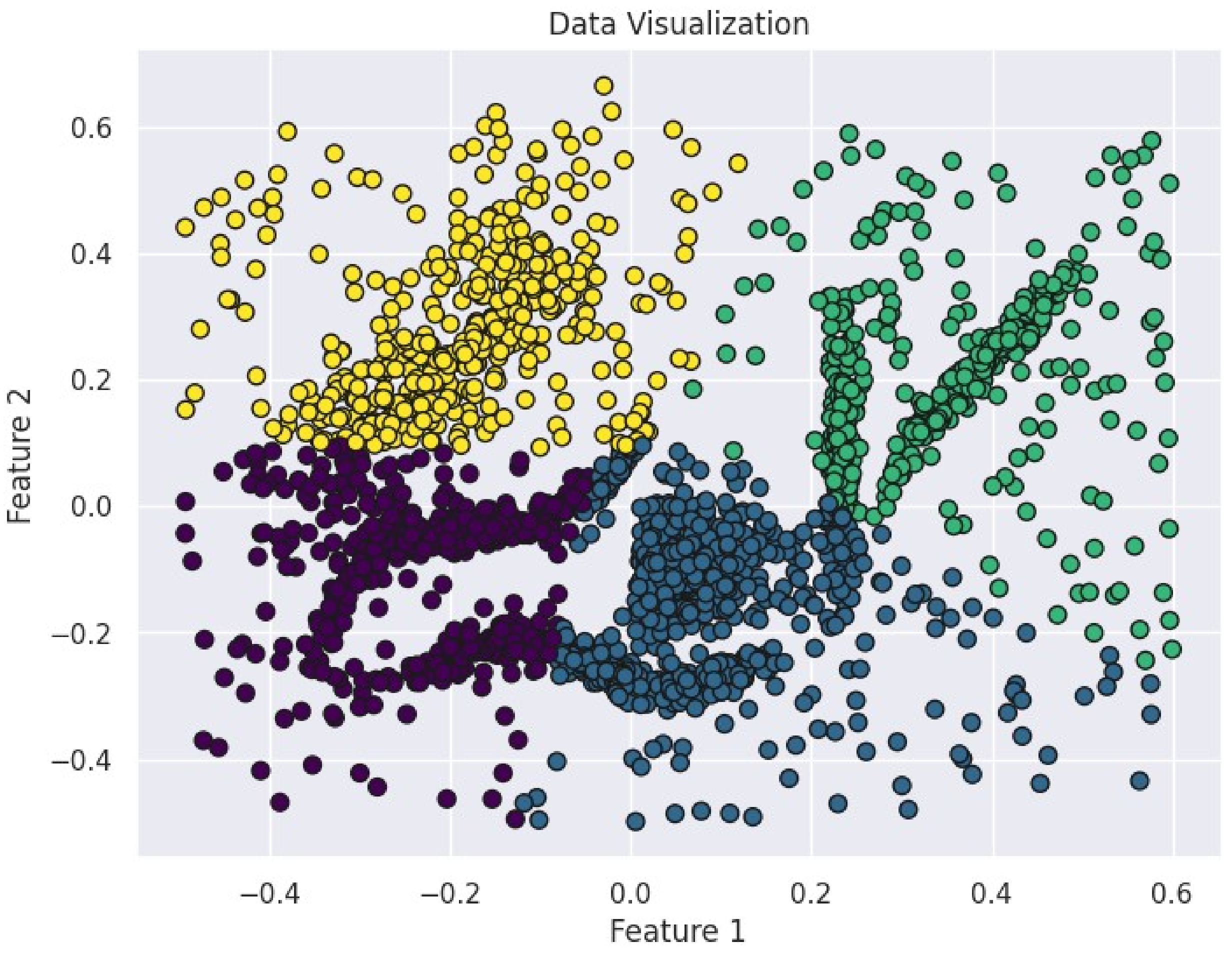
```
Number of clusters: 3
```



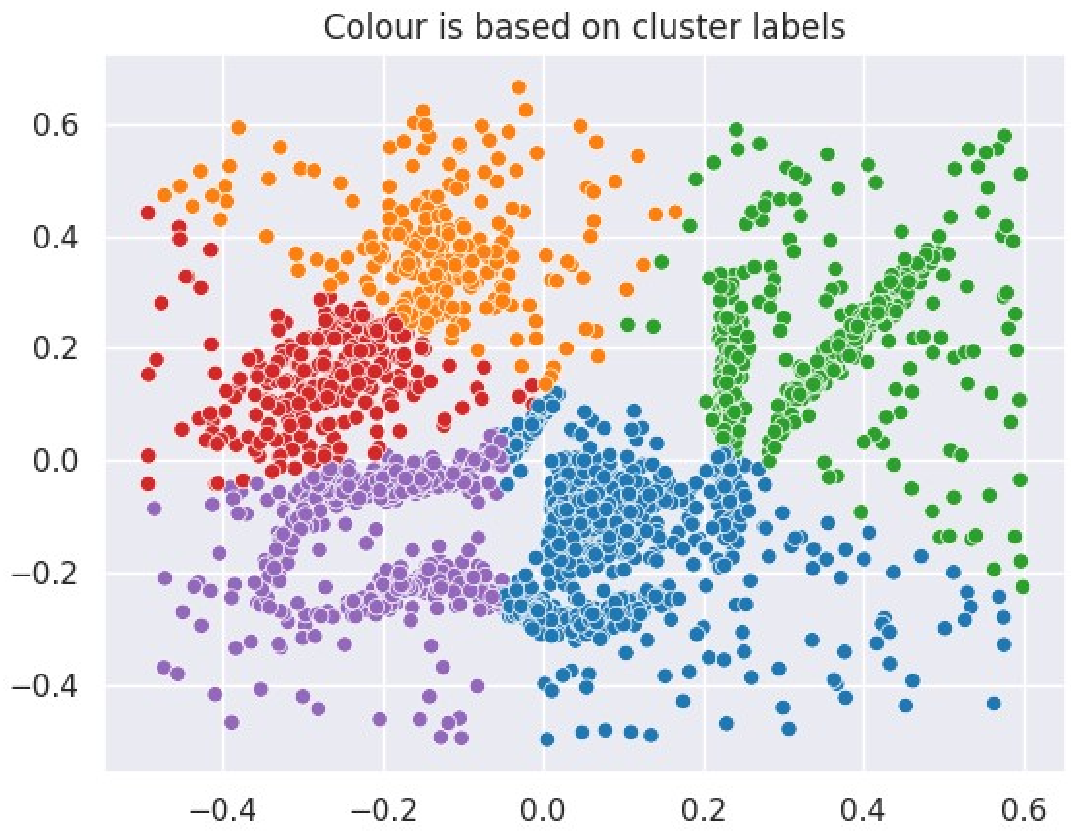
Data Visualization
0.6
0.4
0.2
0.0
−0.2
−0.4
Feature 2
−0.4
−0.2
0.0
0.2
0.4
0.6
Feature 1



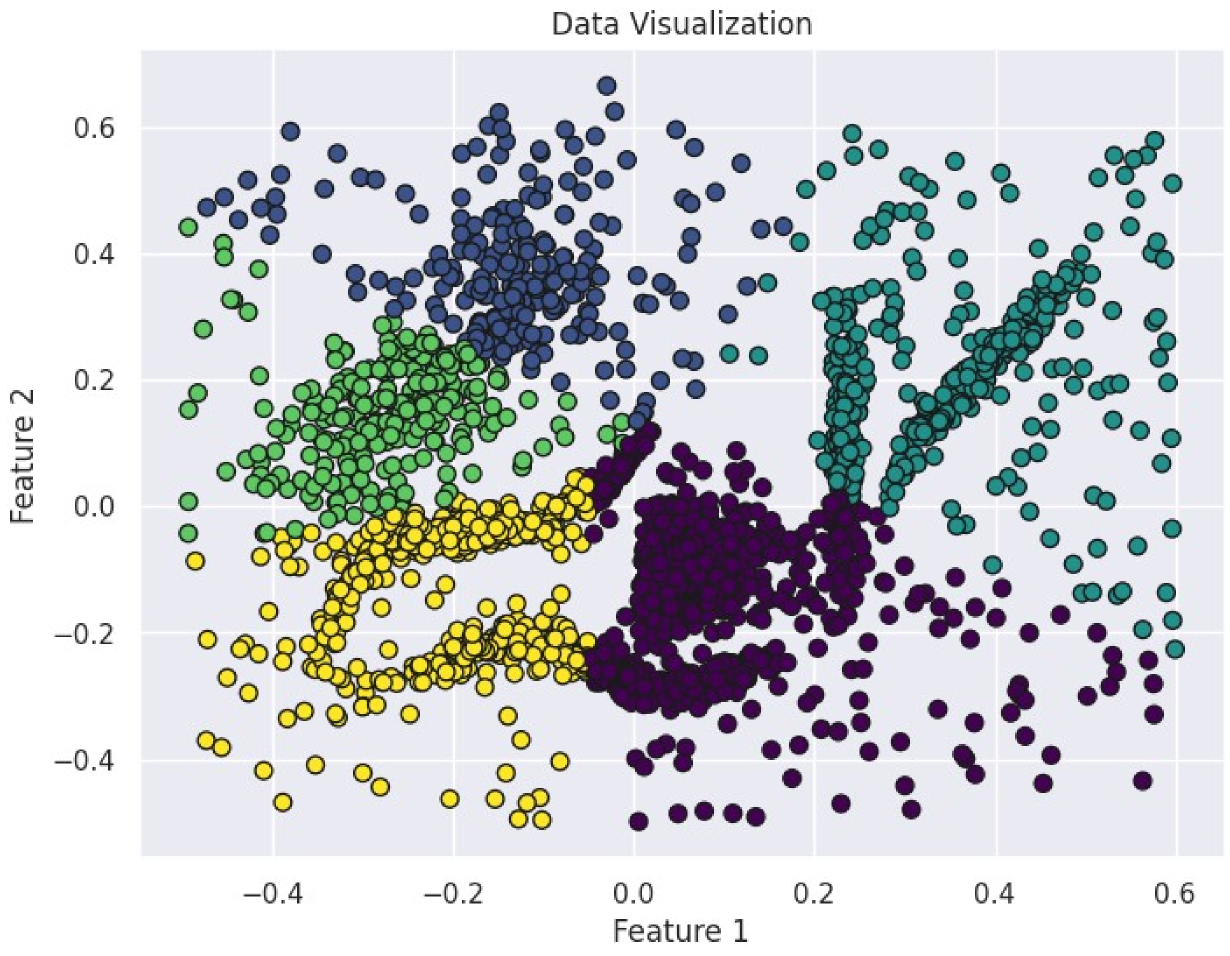
```
Number of clusters: 4
```



Data Visualization
Feature 2
Feature 1
0.6
0.4
0.2
0.0
−0.2
−0.4
−0.4
−0.2
0.0
0.2
0.4
0.6



```
Number of clusters: 5
```



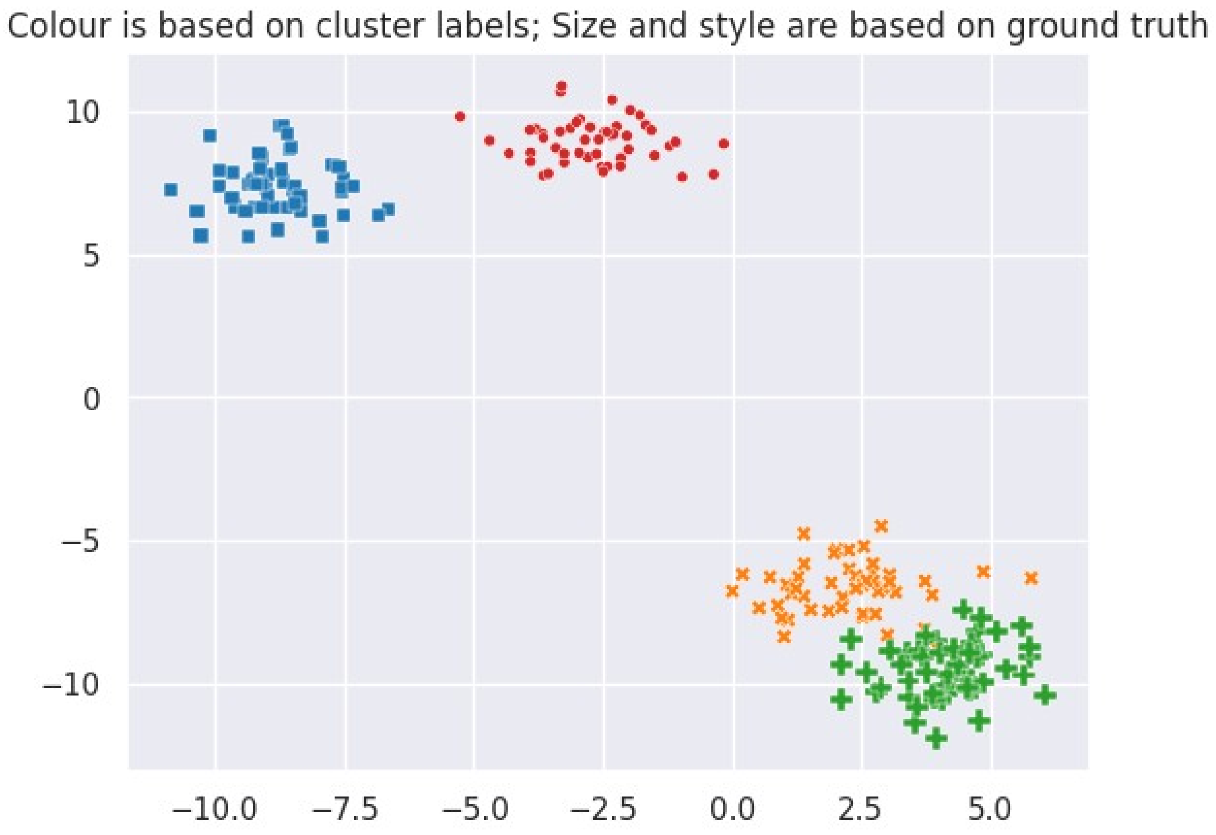
The impact of changing the number of clusters in the KMeans algorithm was examined using the interesting dataset. Different numbers of clusters (2, 3, 4, and 5) were tested. When using 2 clusters, the algorithm provided a rough division of the data, which improved with 3 clusters, likely reflecting the true underlying structure better. Increasing to 4 and 5 clusters began to over-partition the data, indicating that 3 clusters might be the optimal choice for this dataset. This analysis helps in selecting the appropriate number of clusters for accurate data representation.

**##3) Stability with Respect to Noise**

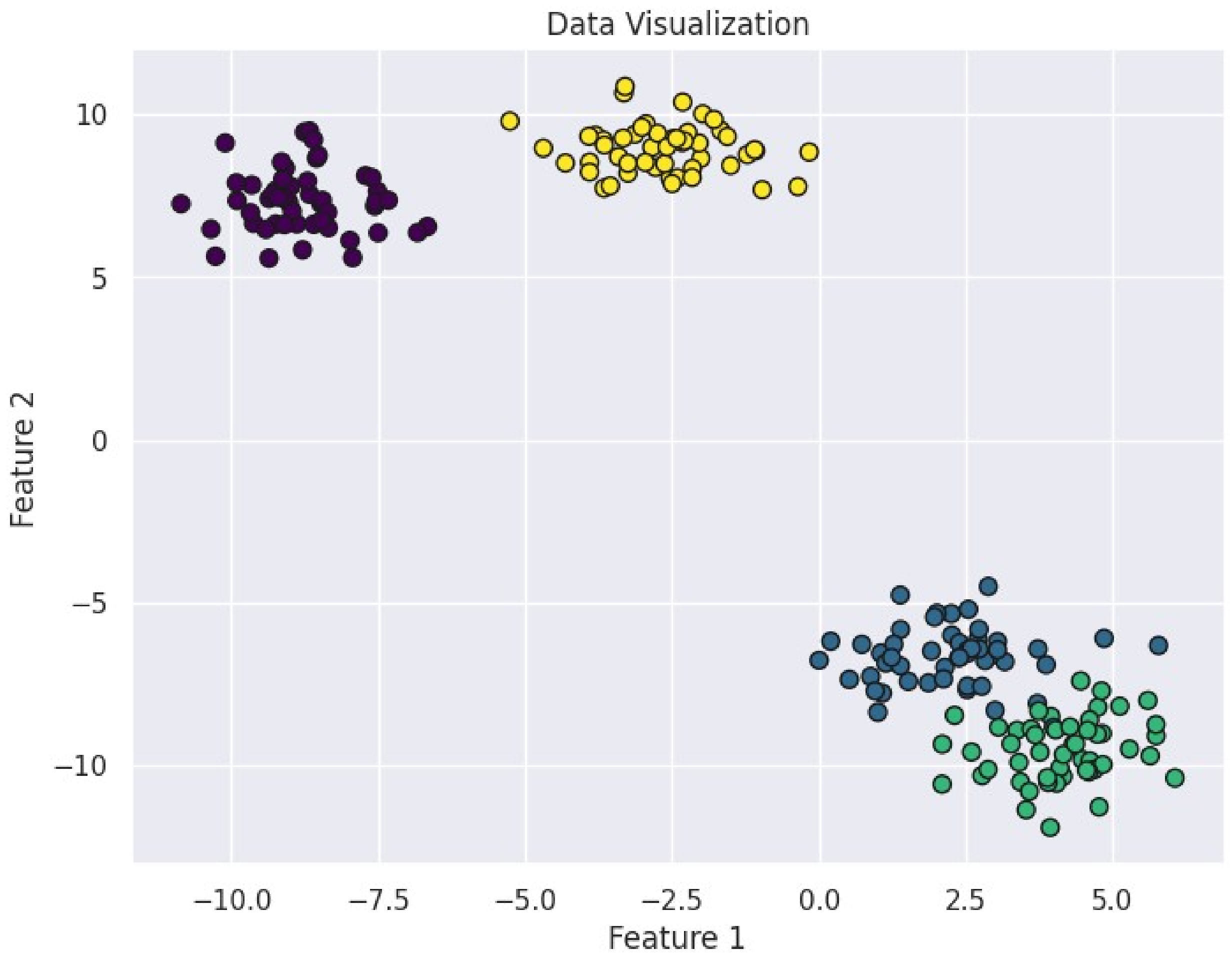
In this section, we investigate the stability of the clustering method (KMeans) in the presence of noise. We will add different levels of Gaussian noise to a dataset and observe how the clustering results change. This helps us understand the robustness of the clustering algorithm to noisy data and identify the noise level at which the algorithm starts to fail.

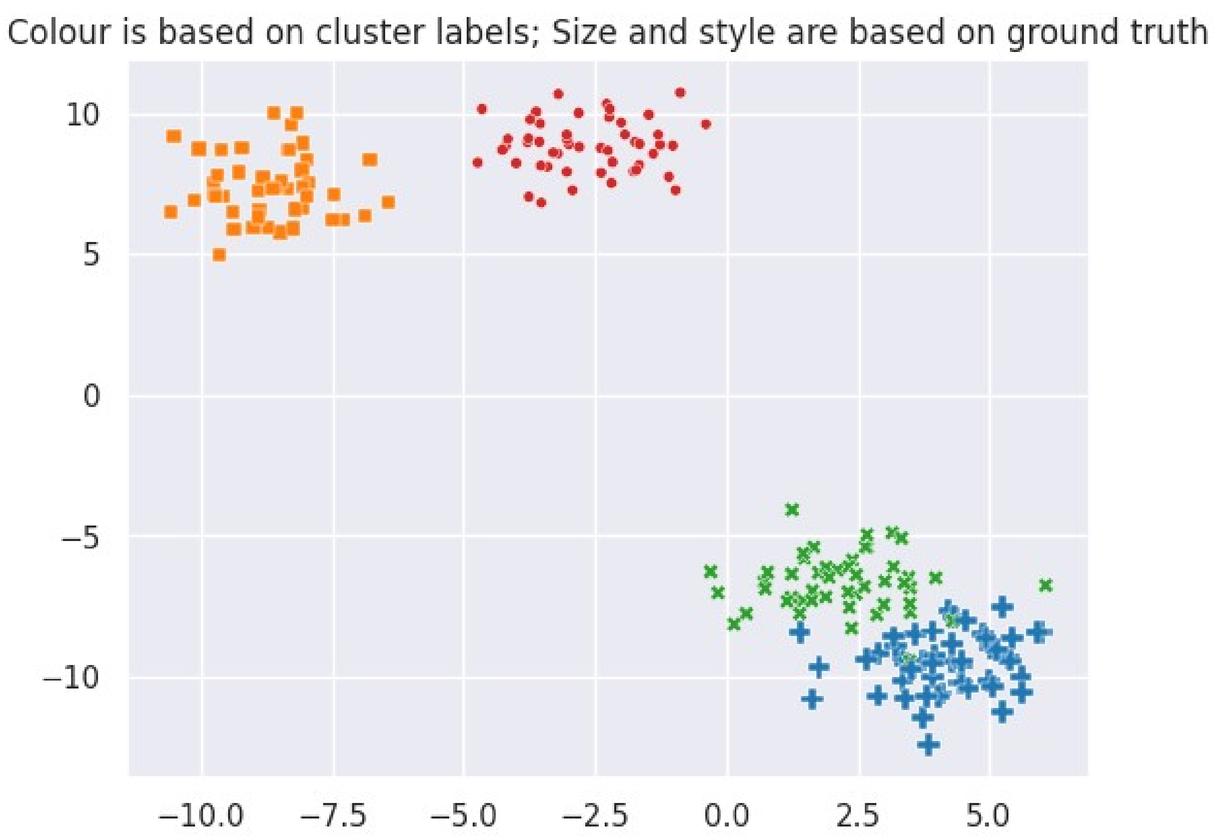
```
import numpy as np

chosen_dataset = 'artificial'
x, labels = datasets[chosen_dataset]

noise_levels = [0.1, 0.5, 0.9]  # Different noise levels
```

```
for noise_level in noise_levels:
    noisy_data = x + noise_level * np.random.normal(size=x.shape)
    kmeans = cluster.KMeans(n_clusters=4, n_init='auto')
    plabs = clustering(kmeans, noisy_data, labels)
    print(f"Noise level: {noise_level}")
    view_data(noisy_data, plabs)
```

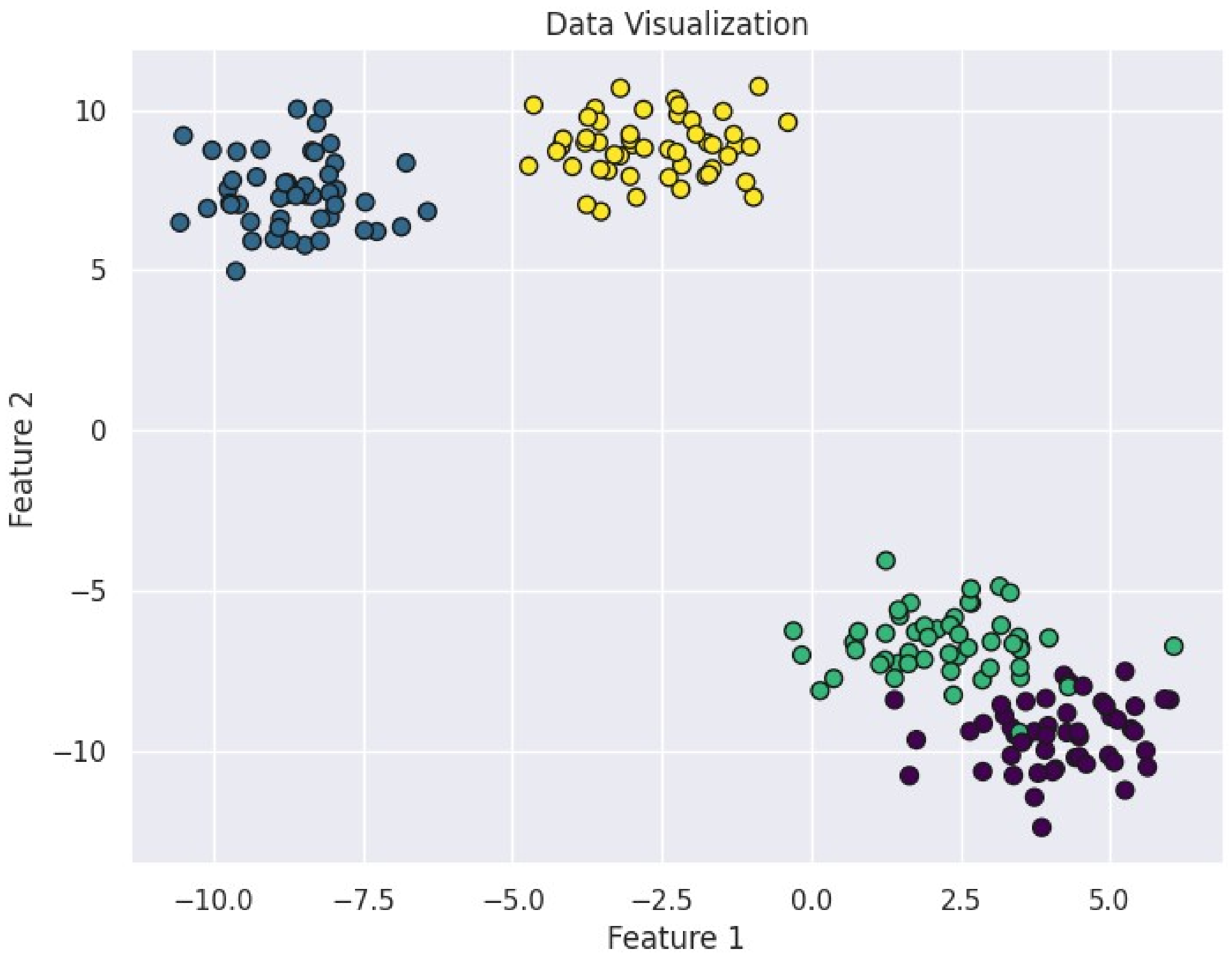


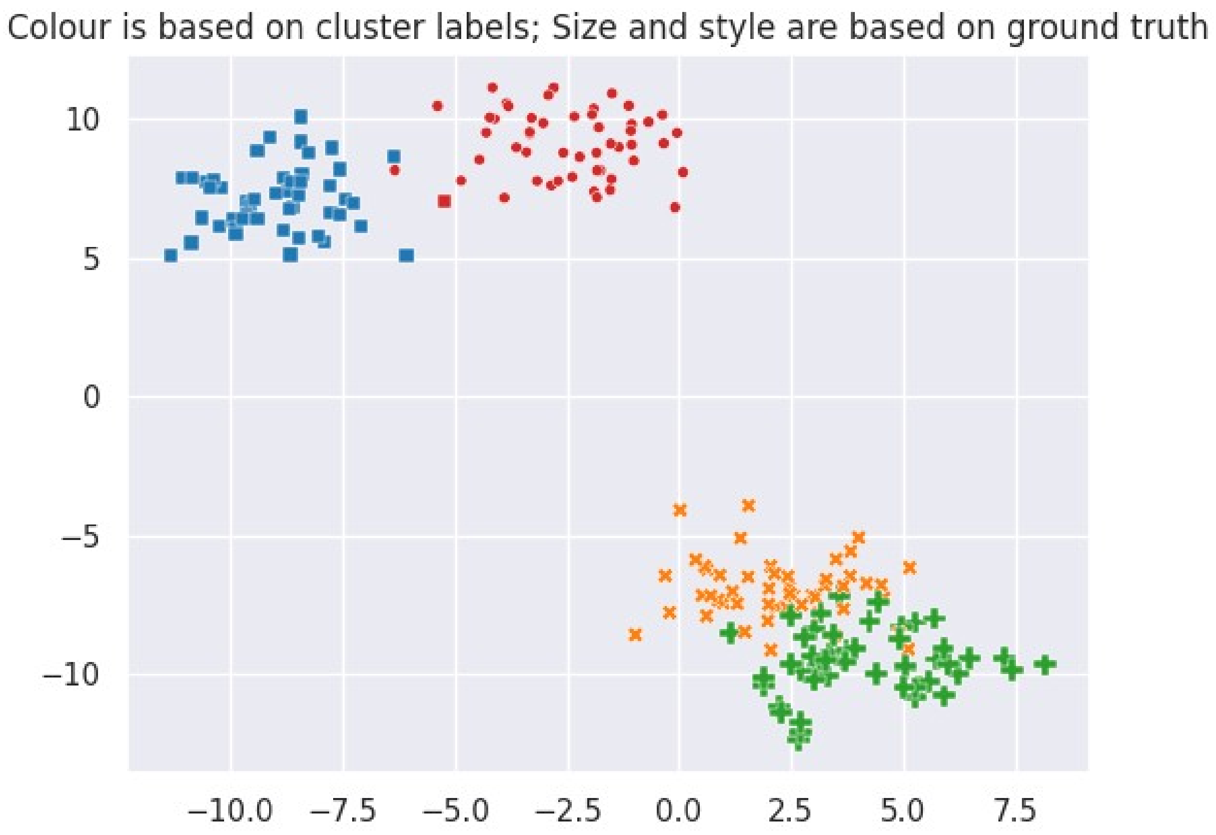
```
Noise level: 0.1
```


Data Visualization
Feature 1
Feature 2
−10.0
−7.5
−5.0
−2.5
0.0
2.5
5.0
10
5
0
−5
−10

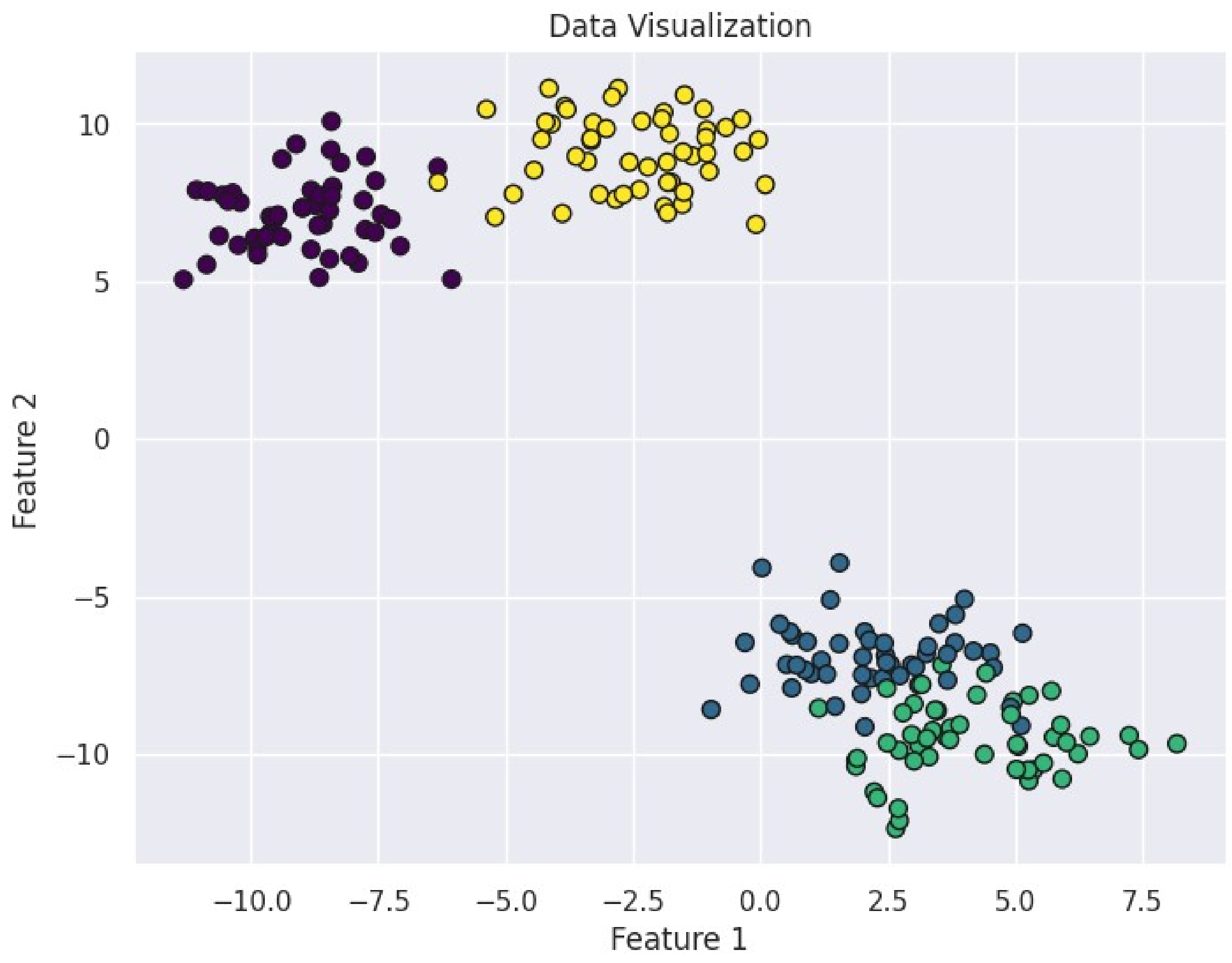


```
Noise level: 0.5
```


Data Visualization
Feature 1
Feature 2
−10.0
−7.5
−5.0
−2.5
0.0
2.5
5.0
10
5
0
−5
−10



```
Noise level: 0.9
```

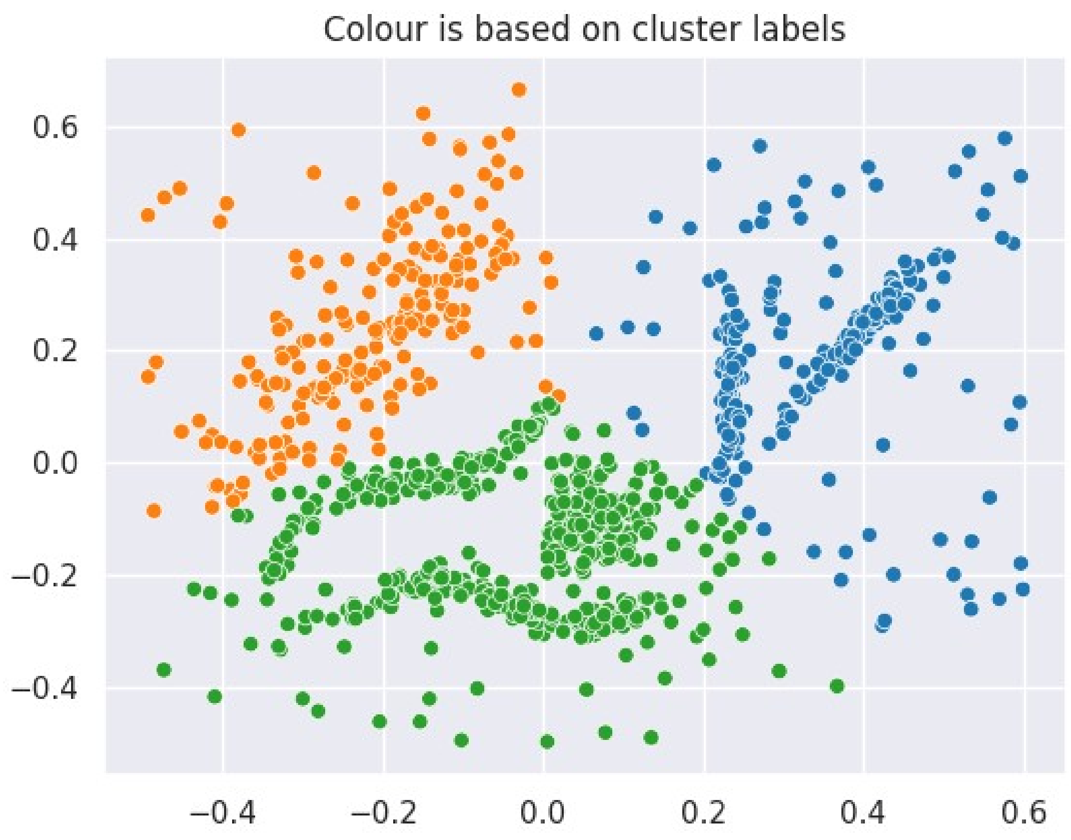


The stability of the KMeans clustering algorithm was tested on the artificial dataset by adding different levels of Gaussian noise (0.1, 0.5, 0.9). For this dataset and noise levels used, the clustering remained relatively robust, maintaining distinct groups. This demonstrates the algorithm's sensitivity to noise and its robustness.

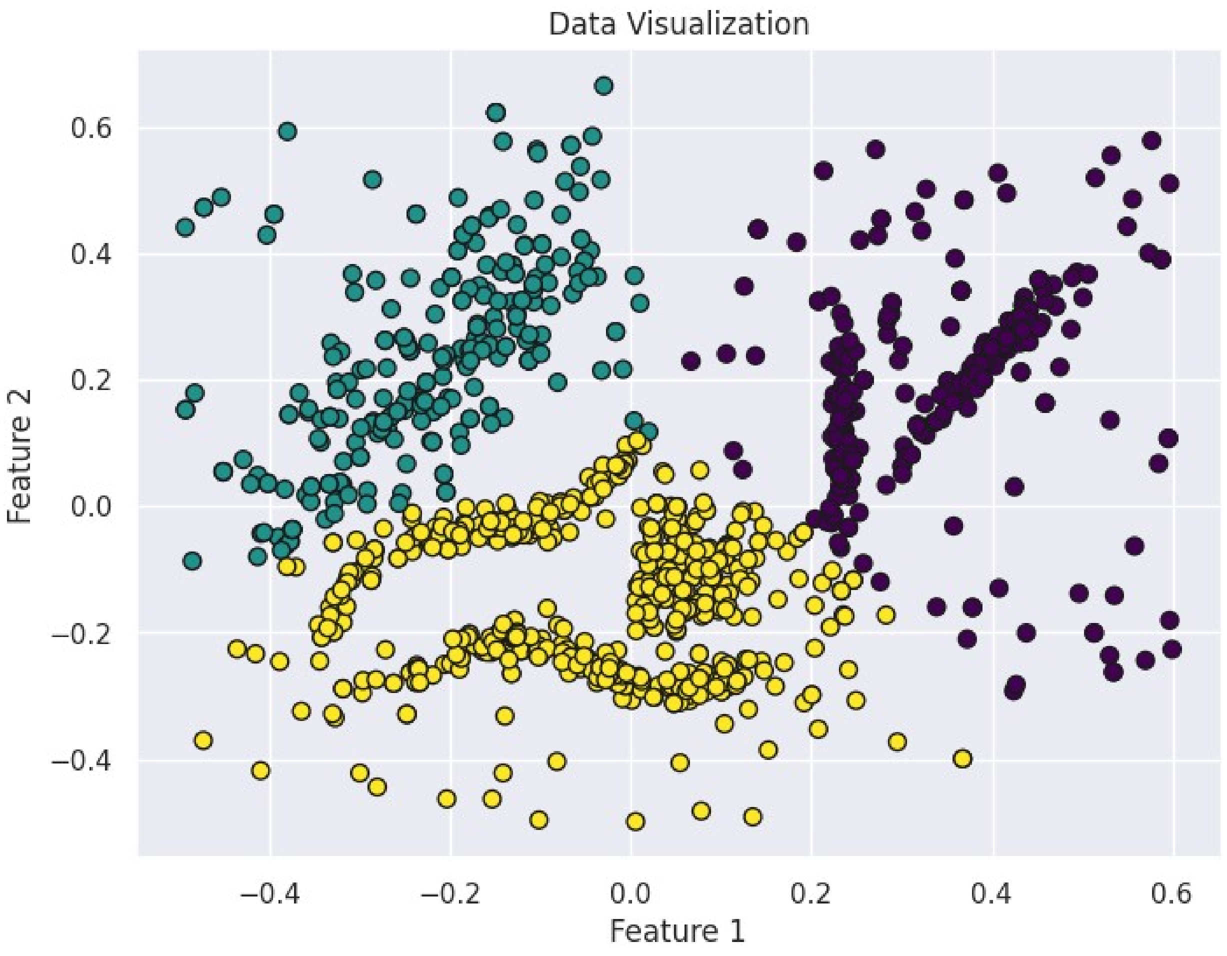
**##4) Stability with Respect to Dataset Size (Subsampling)**

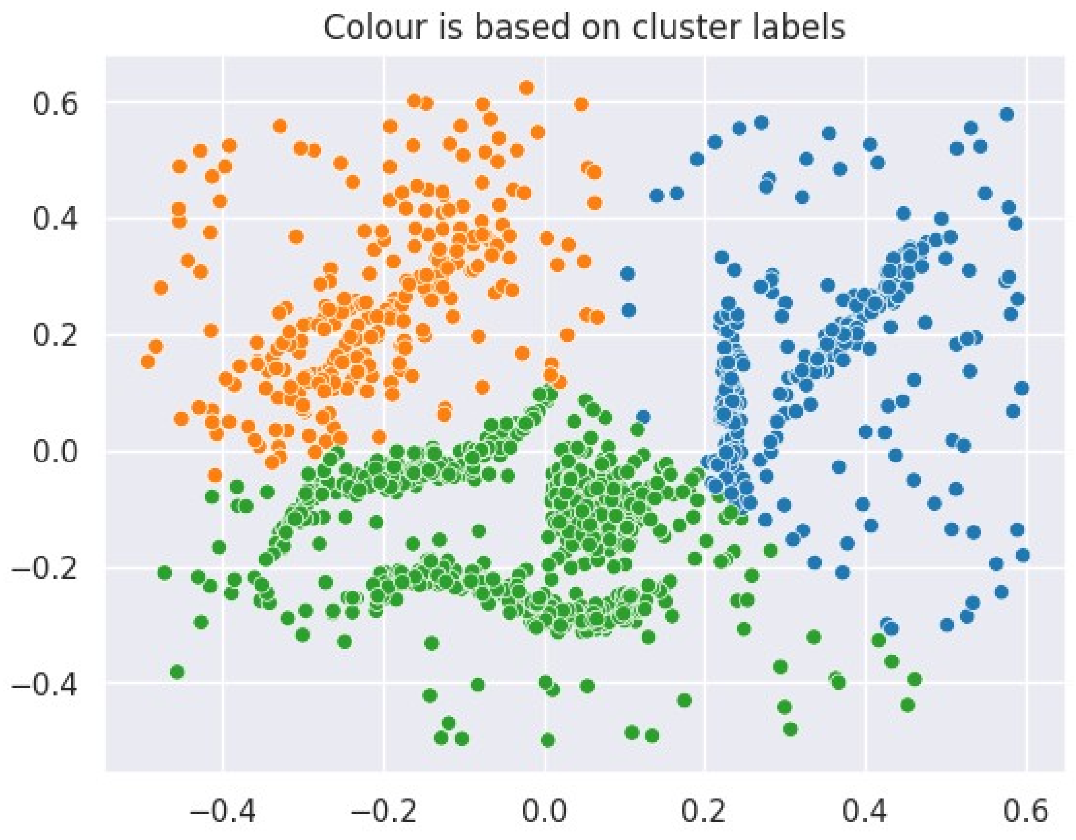
In this section, we explore how the clustering results change when we use different subsets of the dataset. We will subsample the data at various proportions and apply the KMeans clustering algorithm to each subset. This helps us understand how the size of the dataset affects the stability and robustness of the clustering results.

```python
from sklearn.utils import resample
import numpy as np

chosen_dataset = 'interesting'
x, labels = datasets[chosen_dataset]

sample_sizes = [0.5, 0.7, 0.9]  # Proportions of the dataset to use

for i, sample_size in enumerate(sample_sizes):
    if labels is not None:
        x_subsample, labels_subsample = resample(x, labels,
n_samples=int(sample_size * len(x)), random_state=i)
    else:
        x_subsample = resample(x, n_samples=int(sample_size * len(x)),
random_state=i)
        labels_subsample = None

    kmeans = cluster.KMeans(n_clusters=3, n_init='auto')
    plabs = clustering(kmeans, x_subsample, labels_subsample)
    print(f"Sample size: {sample_size}")
    view_data(x_subsample, plabs)
```

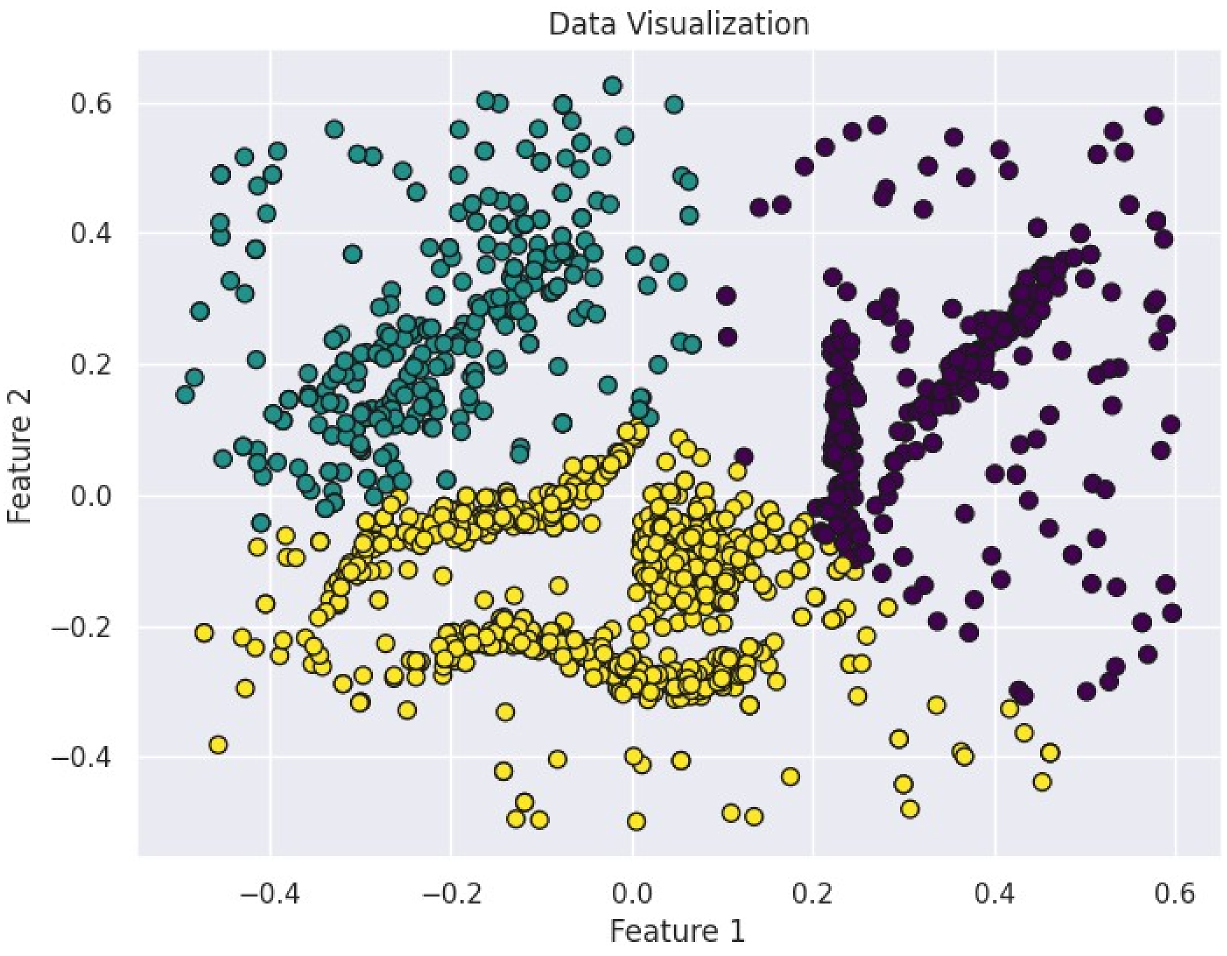


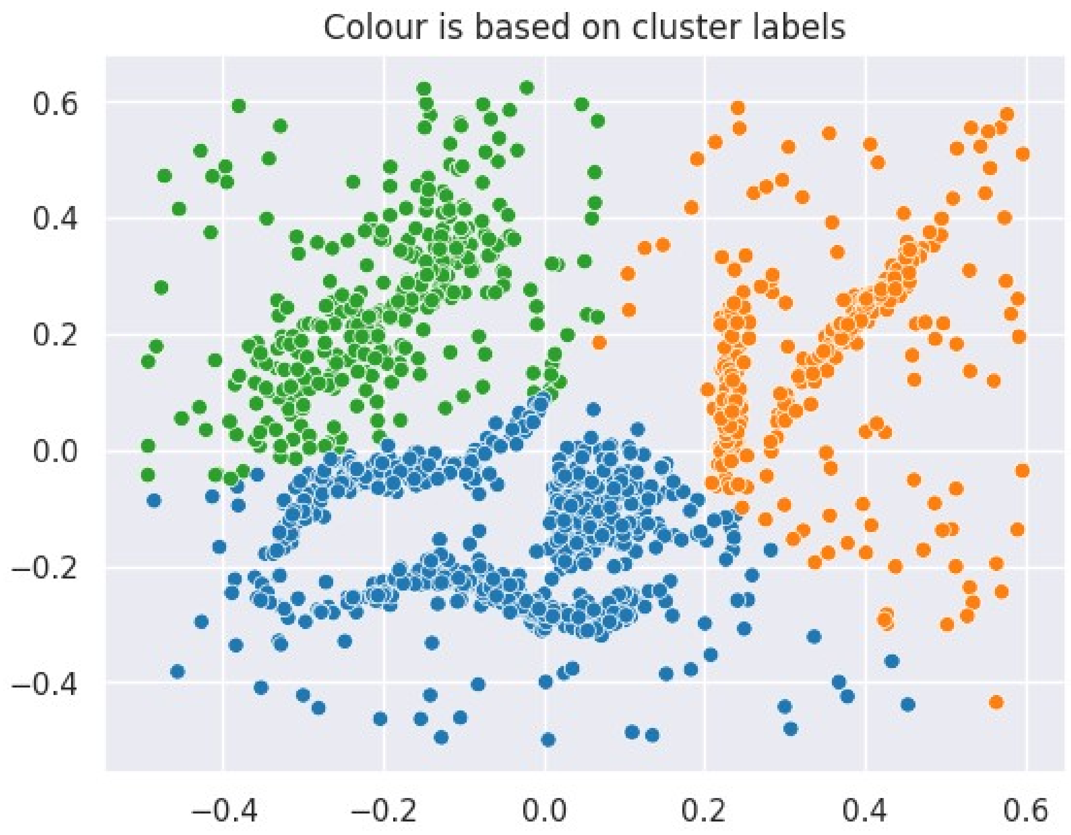
```
Sample size: 0.5
```


Data Visualization
Feature 2
Feature 1
0.6
0.4
0.2
0.0
−0.2
−0.4
−0.4
−0.2
0.0
0.2
0.4
0.6

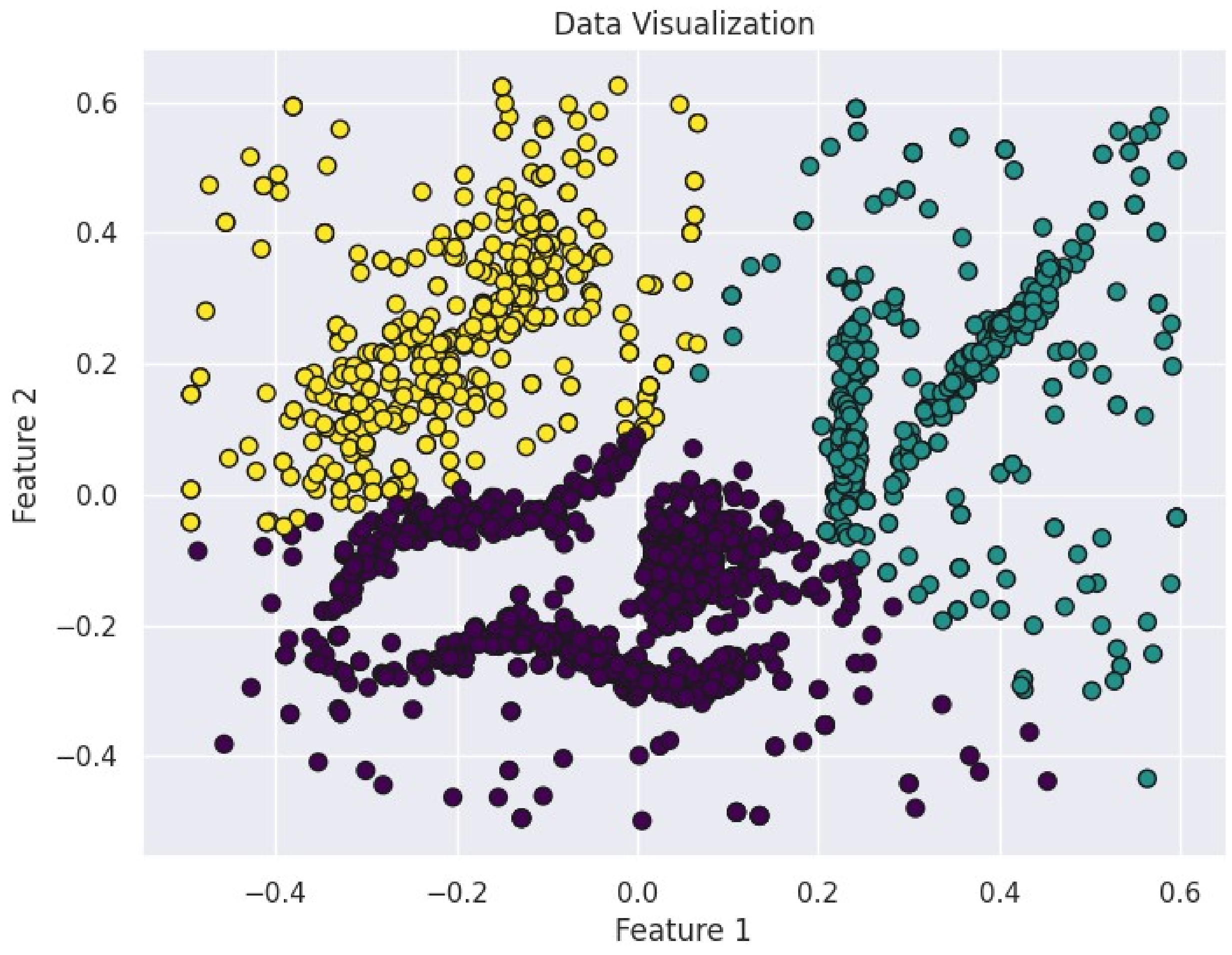


```
Sample size: 0.7
```


Data Visualization
Feature 2
Feature 1
0.6
0.4
0.2
0.0
−0.2
−0.4
−0.4
−0.2
0.0
0.2
0.4
0.6



```
Sample size: 0.9
```

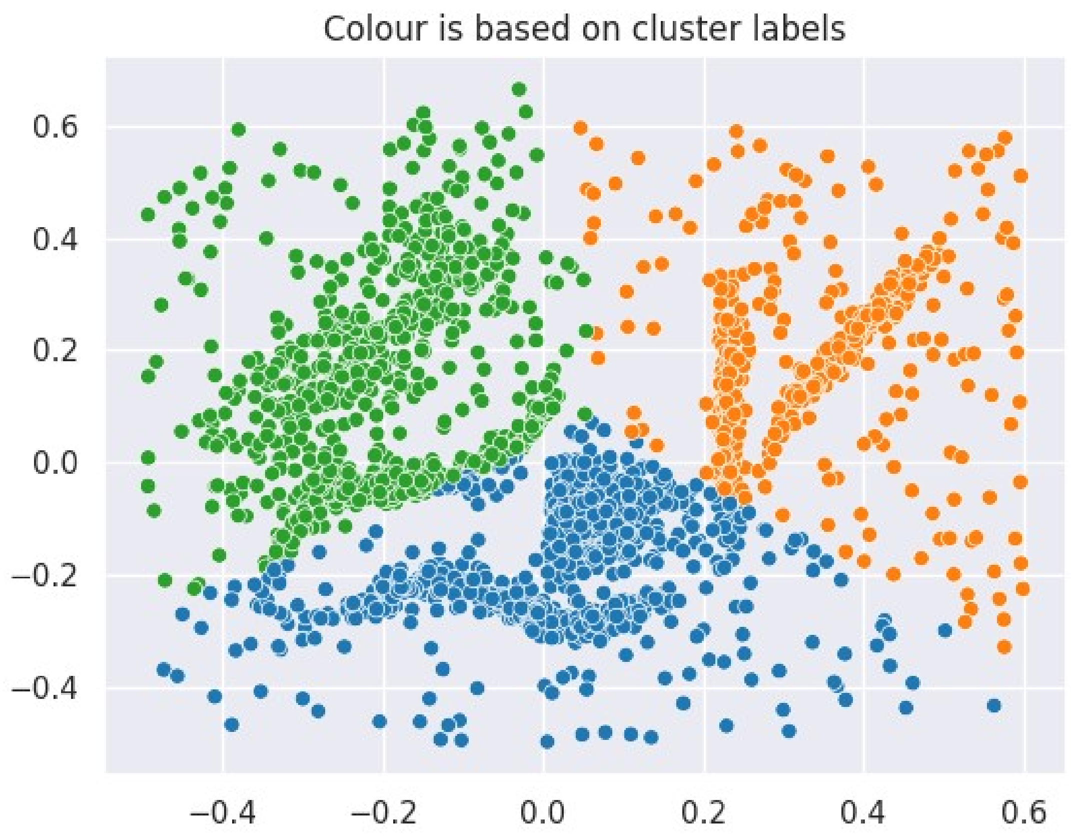


The effect of dataset size on the stability of KMeans clustering was investigated using the interesting dataset. Subsamples of 50%, 70%, and 90% of the original data were taken. For this dataset, the clustering closely matched the results obtained across all subset sizes, indicating stable and reliable clustering performance.

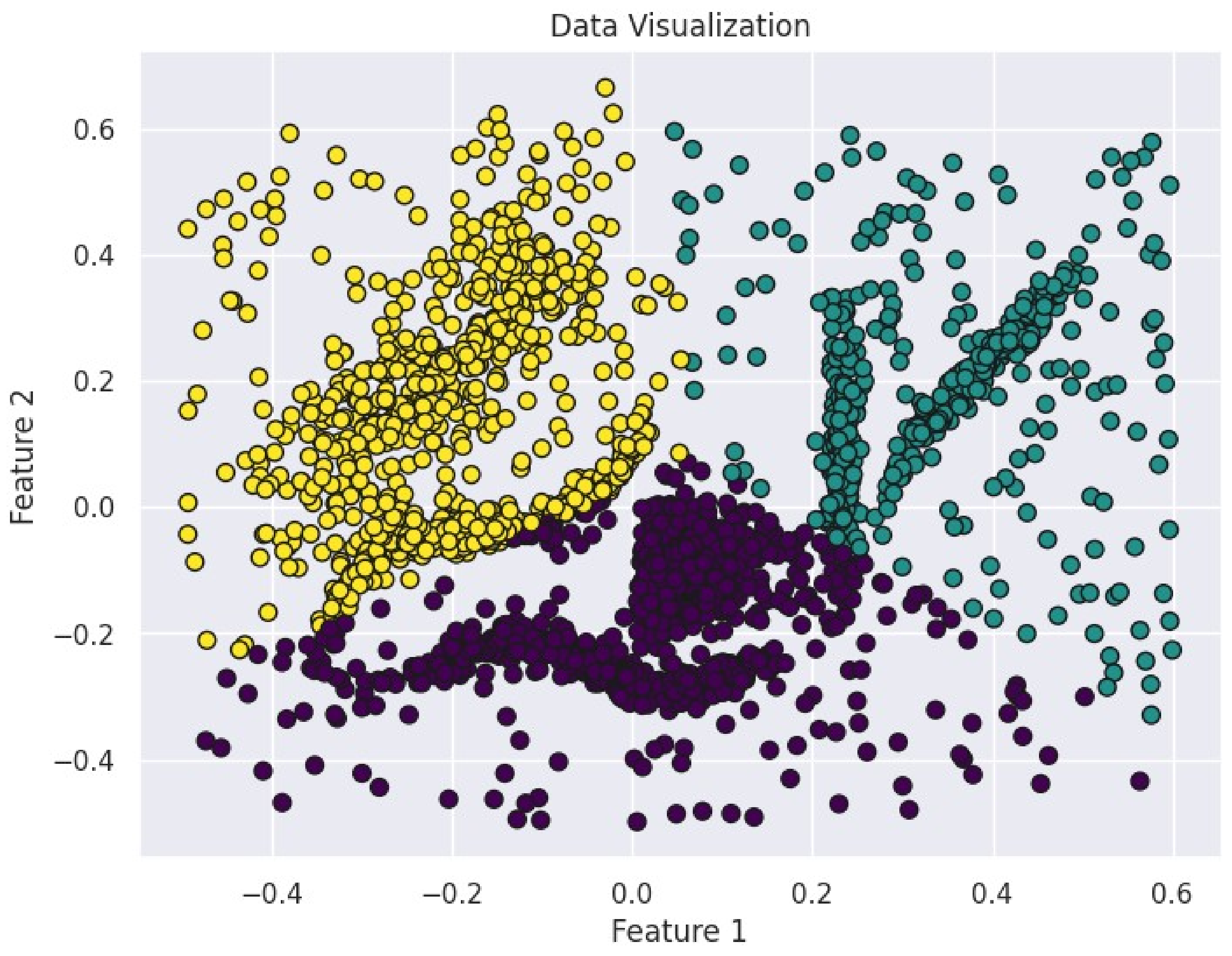
**##5) Measuring Consensus Amongst Methods**

In this section, we compare the results of different clustering methods to see how similar they are. We will use several clustering algorithms on the same dataset and measure the consensus between their results using the Adjusted Rand Index (ARI). This helps us understand the agreement between different clustering approaches.

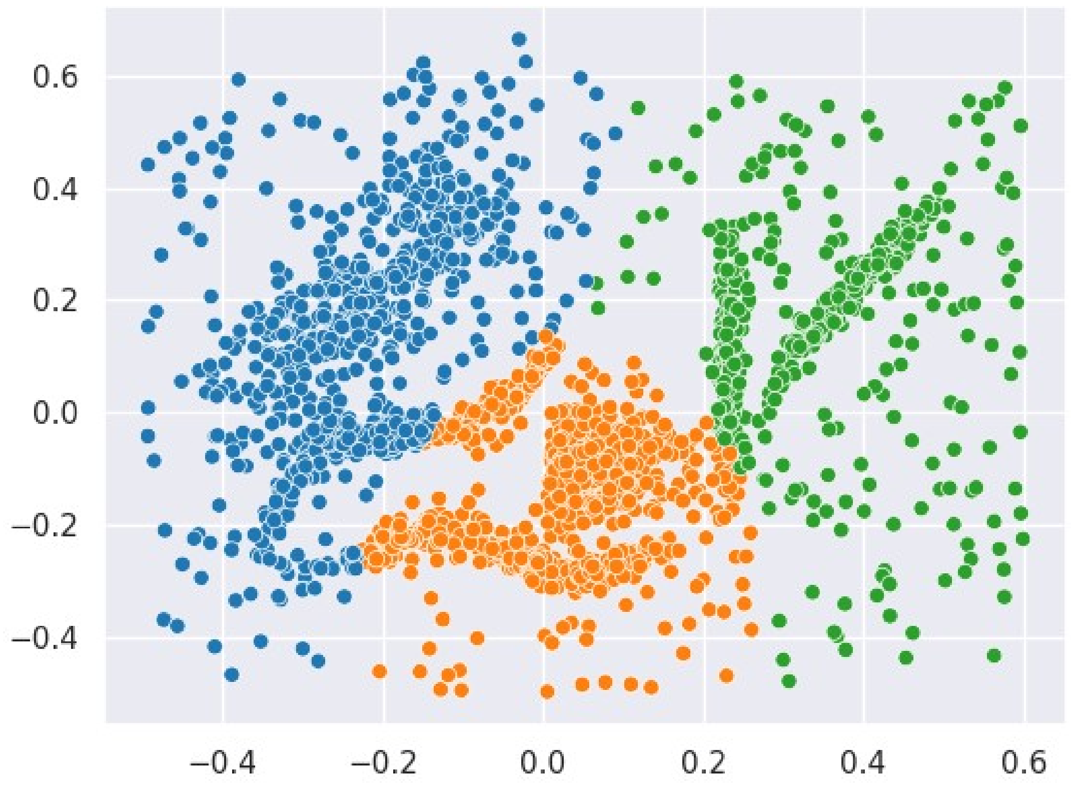
```
# Example of comparing clustering results from different methods

from sklearn.metrics import adjusted_rand_score

chosen_dataset = 'interesting'
x, labels = datasets[chosen_dataset]

num_clusters = 3
methods = {
    'KMeans': cluster.KMeans(n_clusters=num_clusters, n_init='auto'),
    'GMM': mixture.GaussianMixture(n_components=num_clusters),
    'Agglomerative':
cluster.AgglomerativeClustering(n_clusters=num_clusters,
linkage='ward')
}

results = {}
for method_name, method in methods.items():
    plabs = clustering(method, x, labels)
    results[method_name] = plabs
    print(f"Method: {method_name}")
    view_data(x, plabs)

# Measure consensus
for method1 in methods:
    for method2 in methods:
        if method1 != method2:
            consensus = adjusted_rand_score(results[method1],
results[method2])
            print(f"Consensus between {method1} and {method2}:
{consensus}")
```



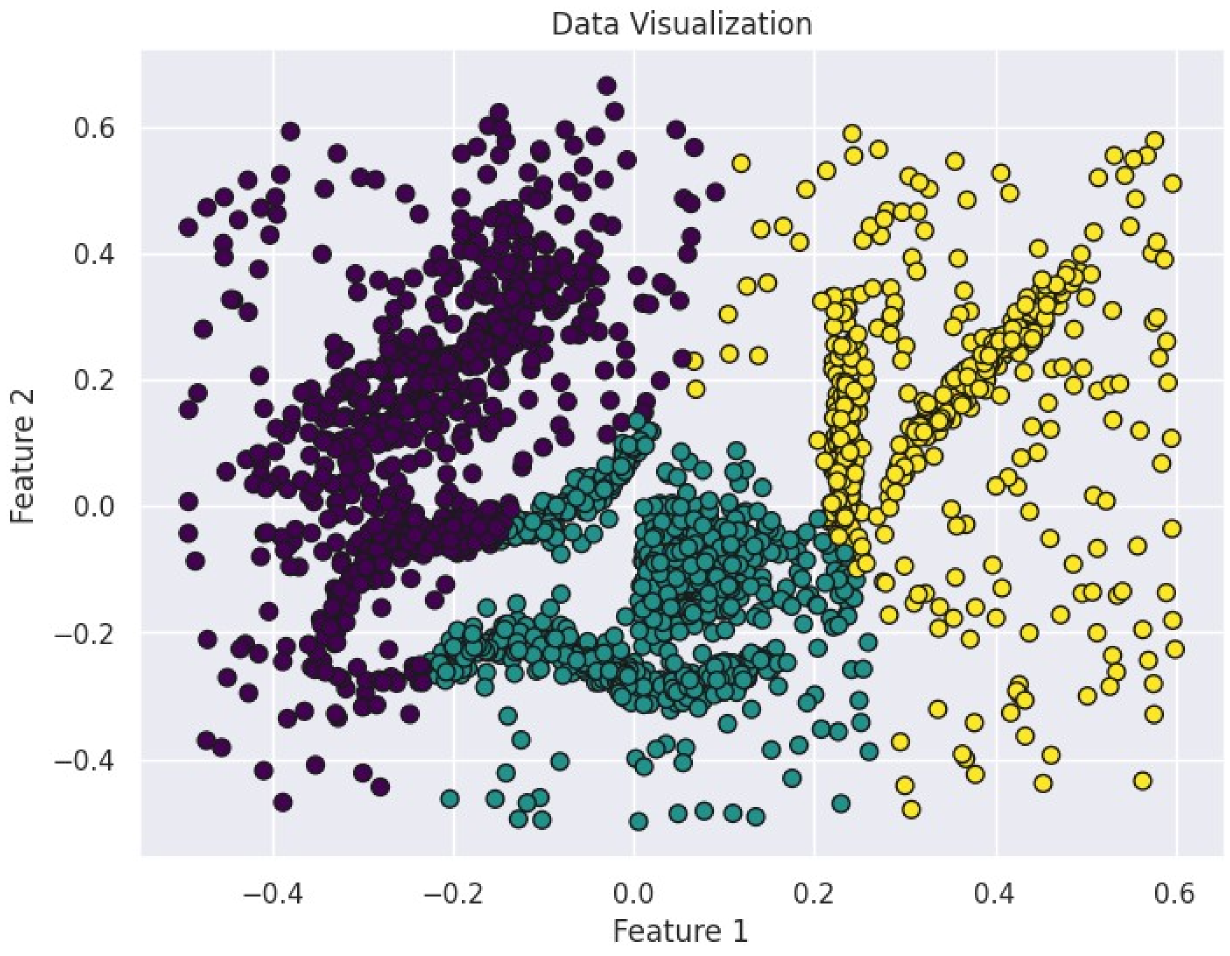
```
Method: KMeans
```

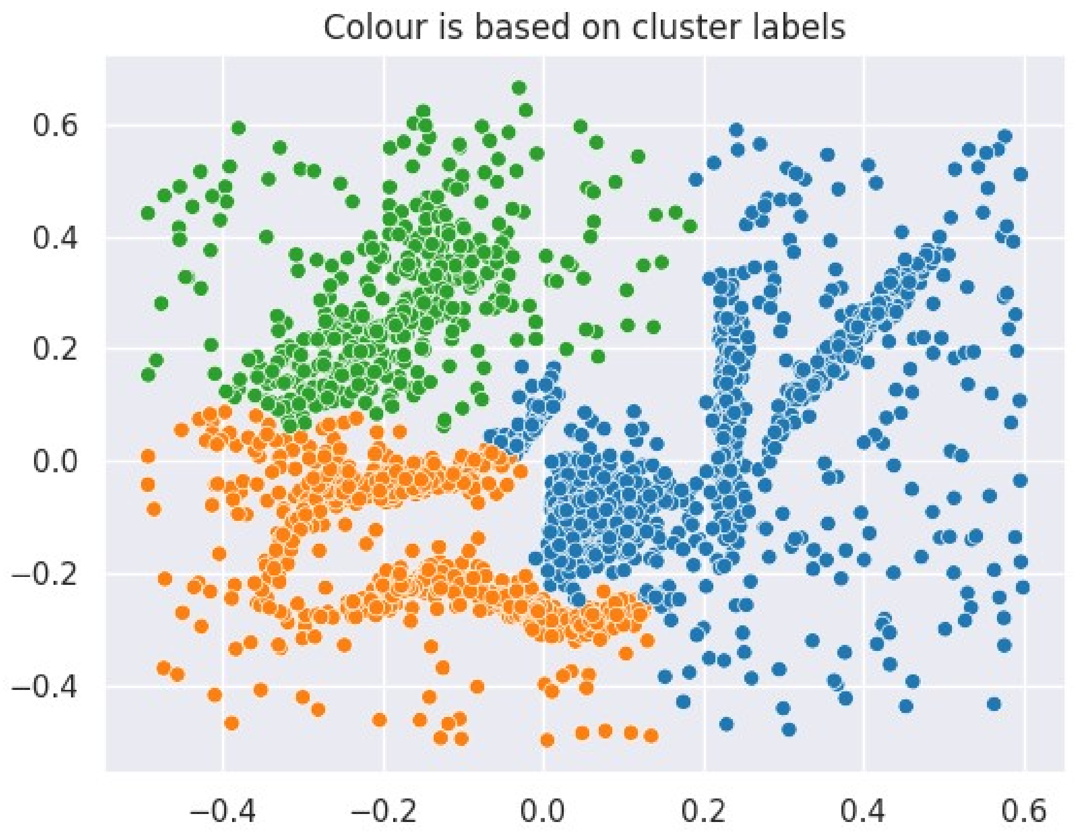


Colour is based on cluster labels

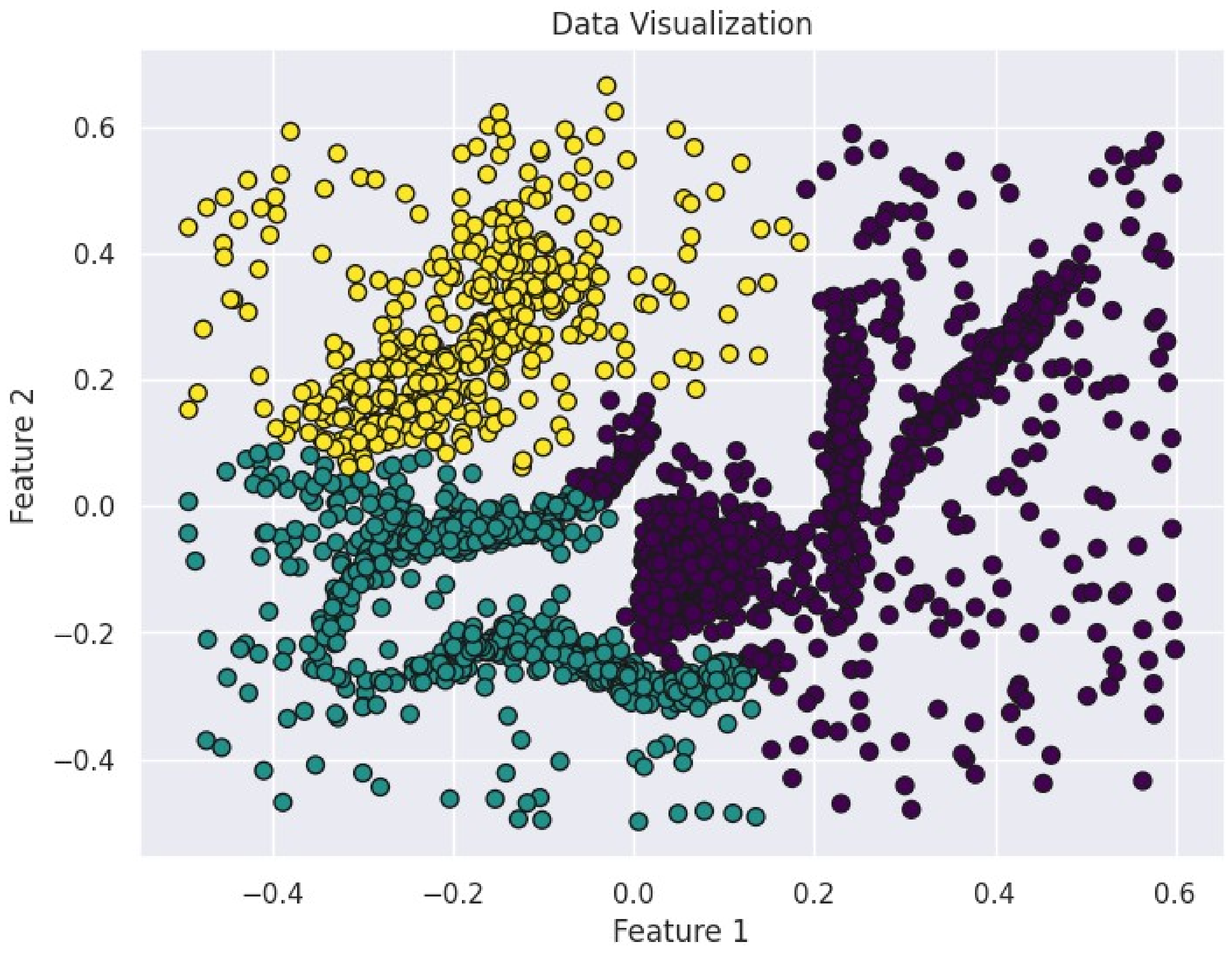


```
Method: GMM
```


Data Visualization
Feature 2
Feature 1
0.6
0.4
0.2
0.0
−0.2
−0.4
−0.4
−0.2
0.0
0.2
0.4
0.6



```
Method: Agglomerative
```



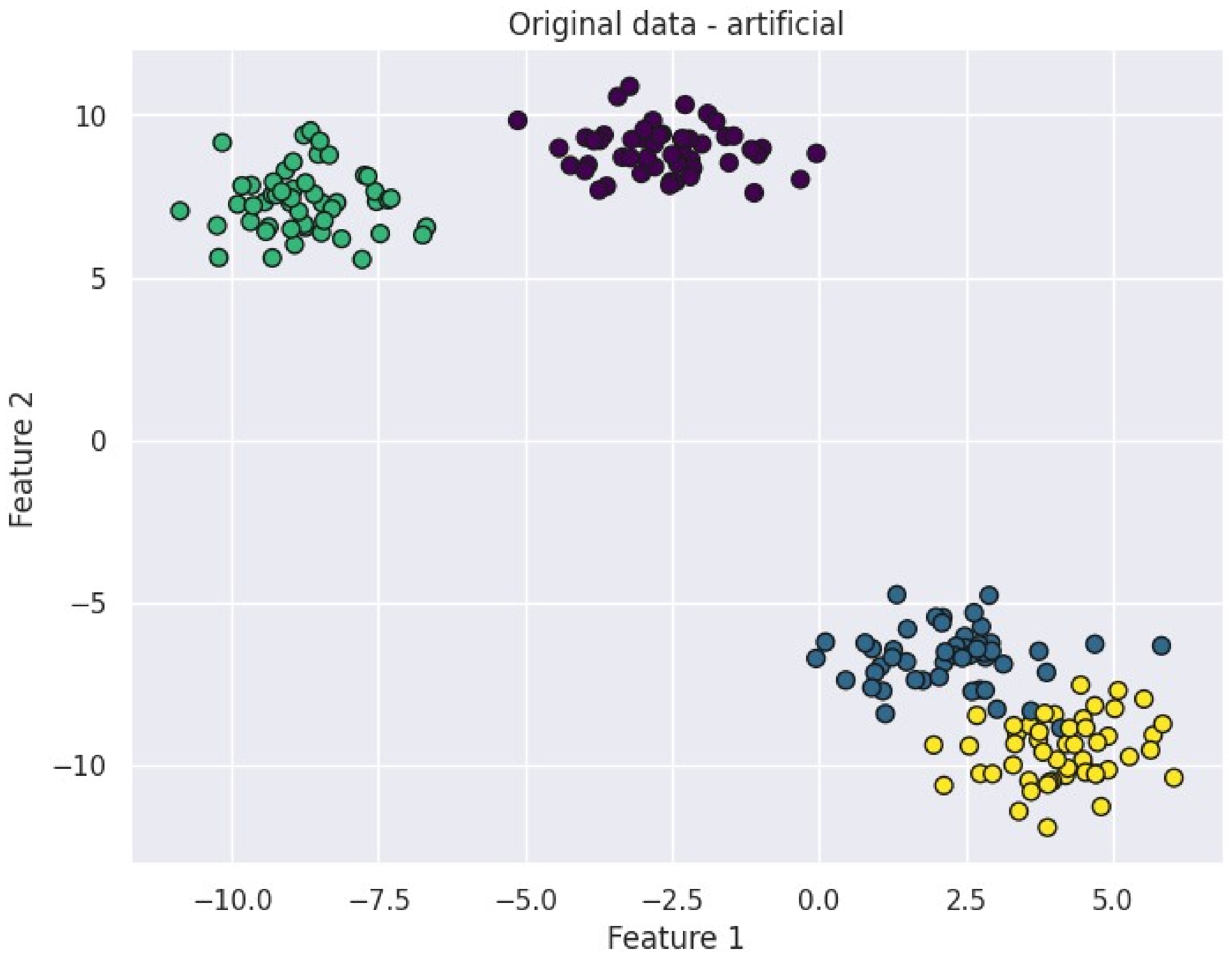
```
Consensus between KMeans and GMM: 0.7389181010668527
Consensus between KMeans and Agglomerative: 0.2731724527079223
Consensus between GMM and KMeans: 0.7389181010668527
Consensus between GMM and Agglomerative: 0.3204781600999838
Consensus between Agglomerative and KMeans: 0.2731724527079223
Consensus between Agglomerative and GMM: 0.3204781600999838
```

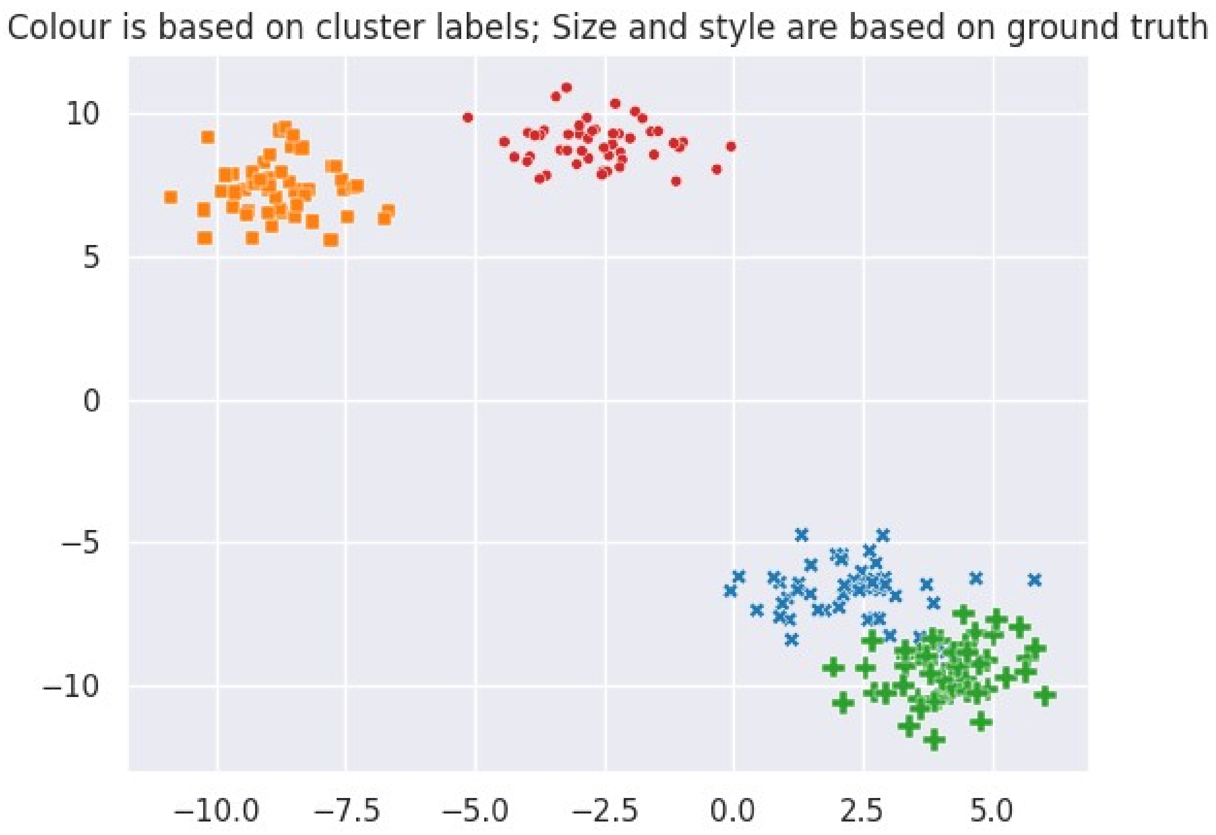
The consensus between different clustering methods (KMeans, Gaussian Mixture Models, and Agglomerative Clustering) was measured using the Adjusted Rand Index (ARI) on the interesting dataset. The results showed varying degrees of agreement between the methods. KMeans and GMM had a high ARI score, indicating similar clustering results. Agglomerative Clustering, while generally consistent, showed some differences, resulting in a moderate ARI when compared to the other two methods. These findings suggest that while different methods can provide similar clusters, some variability is expected based on the algorithm used.

**##6) Measuring "Success" with respect to labels (when available)**

In this section, we measure how well the clustering results match the true labels of the data (if available). We will use accuracy and other metrics to quantify the success of the clustering. This helps us evaluate the clustering performance in a more objective manner by comparing it to known ground truth.

```
# Example of measuring clustering success with respect to ground truth
labels

from sklearn.metrics import accuracy_score
from scipy.optimize import linear_sum_assignment
import numpy as np

chosen_dataset = 'artificial'
x, labels = datasets[chosen_dataset]

# Visualize the original data
if labels is not None:
    view_data(x, labels, title=f'Original data - {chosen_dataset}')
else:
    view_data(x, title=f'Original data - {chosen_dataset}')

def permute_labels(y_true, y_pred):
    # Create a contingency matrix
    contingency_matrix = np.zeros((len(np.unique(y_true)),
len(np.unique(y_pred))), dtype=int)
    for i in range(len(y_true)):
        contingency_matrix[y_true[i], y_pred[i]] += 1

    # Solve the linear assignment problem
    row_ind, col_ind = linear_sum_assignment(-contingency_matrix)

    # Create a new y_pred with the best matching labels
    new_y_pred = np.zeros_like(y_pred)
    for i in range(len(row_ind)):
        new_y_pred[y_pred == col_ind[i]] = row_ind[i]

    return new_y_pred

# Perform clustering
num_clusters = 4
kmeans = cluster.KMeans(n_clusters=num_clusters, n_init='auto')
plabs = clustering(kmeans, x, labels)

# Align cluster labels to match true labels as closely as possible
new_plabs = permute_labels(labels, plabs)

# Calculate the new accuracy
success = accuracy_score(labels, new_plabs)
print(f"Clustering success (accuracy): {success}")
```


Original data - artificial
Feature 2
Feature 1
10
5
0
−5
−10
−10.0
−7.5
−5.0
−2.5
0.0
2.5
5.0



```
Clustering success (accuracy): 1.0
```

The success of the KMeans clustering algorithm was evaluated by comparing the predicted clusters to the true labels of the artificial dataset. The accuracy of the clustering was measured, and the results indicated a high degree of alignment with the true labels, achieving an accuracy = 1.0. This demonstrates that the KMeans algorithm effectively identified the underlying clusters in the dataset, providing a reliable measure of success when ground truth labels are available.

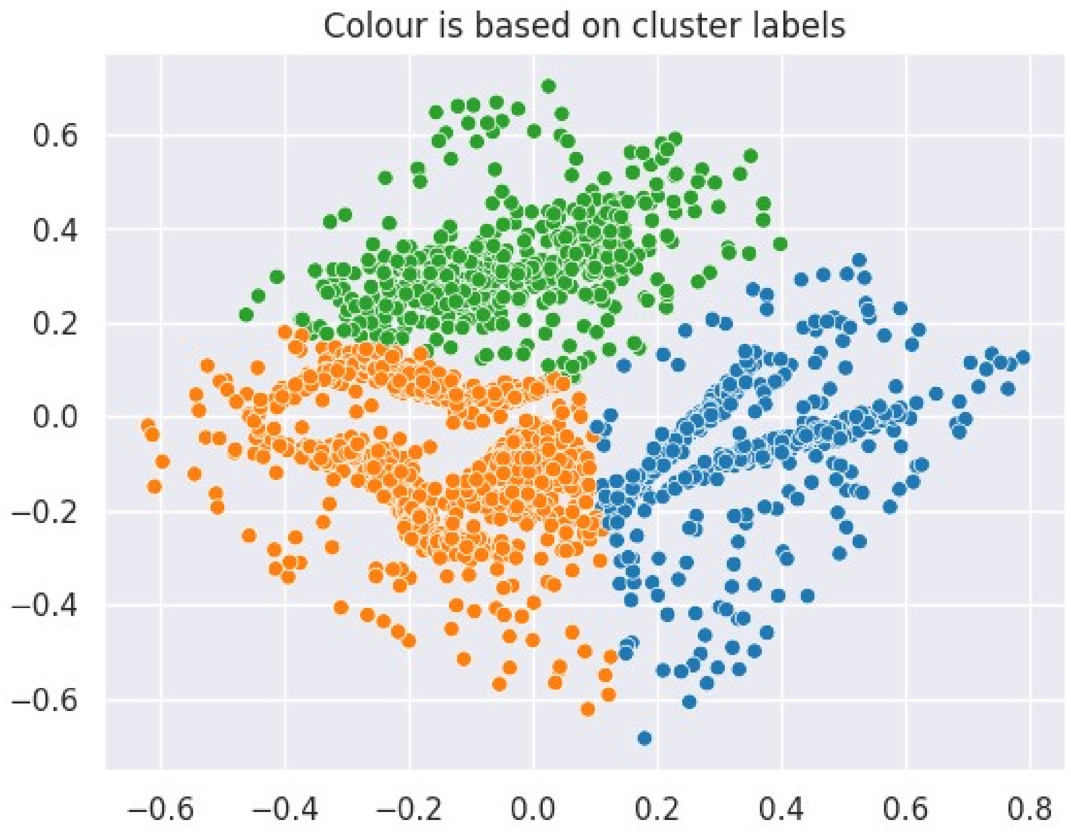
Note: this code also aims to fix cluster label misalignment, as the labels assigned by clustering algorithms are arbitrary and might not correspond directly to the ground truth labels. This can cause an apparent mismatch in the accuracy score. The permute_labels function creates a contingency matrix of the true and predicted labels and uses the Hungarian algorithm (linear sum assignment) to find the best label matching.

**##7) Using PCA for Dimensionality Reduction and Feeding the Reduced Datasets into Clustering Methods**

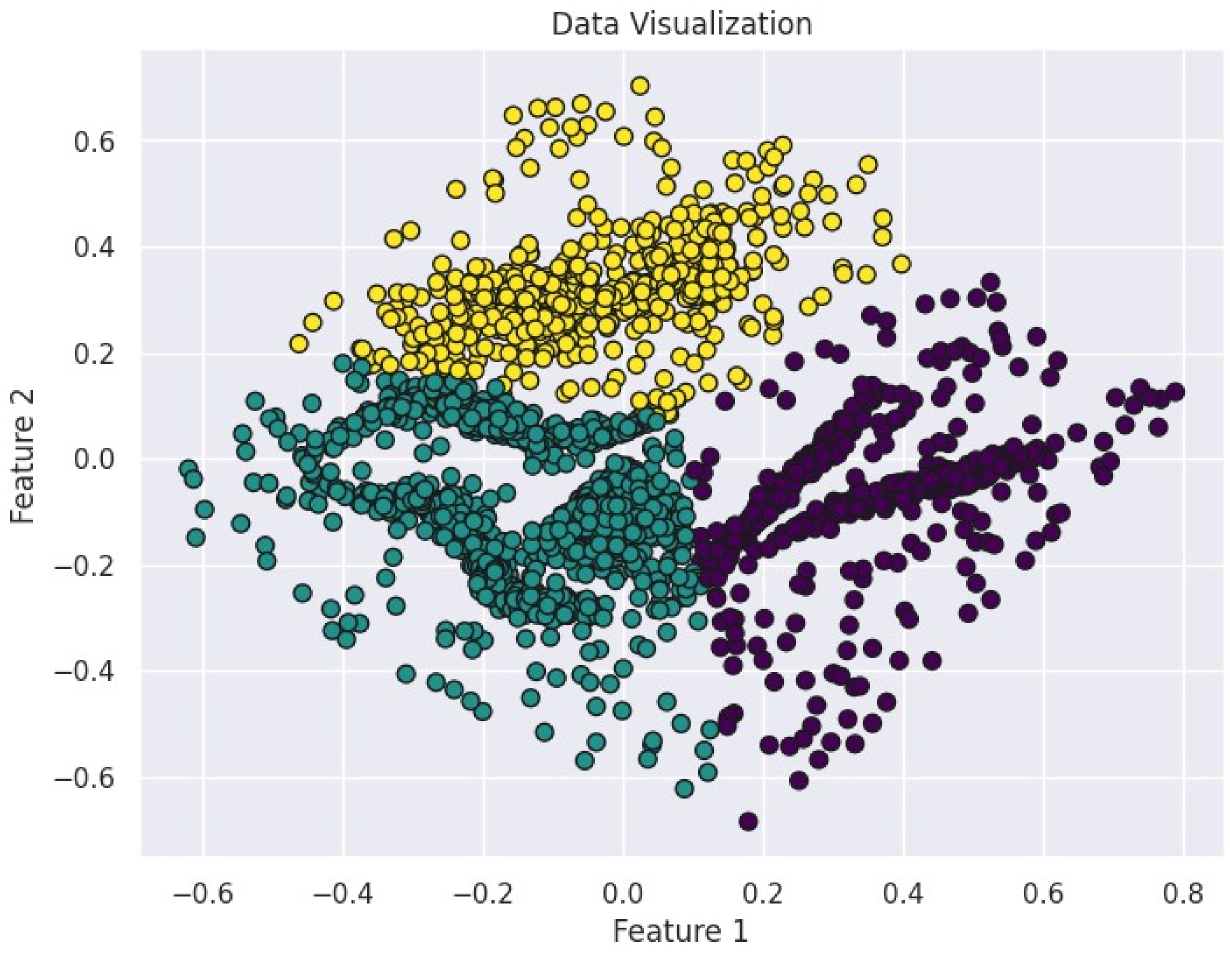
In this section, we use Principal Component Analysis (PCA) to reduce the dimensionality of the data before applying clustering. We will perform PCA to reduce the data to 2 dimensions and then apply the KMeans clustering algorithm on the reduced dataset. This helps us understand the effect of dimensionality reduction on clustering performance.

```
# Example of using PCA for dimensionality reduction before clustering
```

```
from sklearn.decomposition import PCA

chosen_dataset = 'interesting'
x, labels = datasets[chosen_dataset]

# Apply PCA
pca = PCA(n_components=2)
x_reduced = pca.fit_transform(x)

# Perform clustering on reduced data
kmeans = cluster.KMeans(n_clusters=3, n_init='auto')
plabs = clustering(kmeans, x_reduced, labels)

print("PCA reduced data clustering")
view_data(x_reduced, plabs)
```



```
PCA reduced data clustering
```



Principal Component Analysis (PCA) was used to reduce the dimensionality of the interesting dataset to 2 principal components. The reduced dataset was then clustered using KMeans. The visualization of the clustering results on the 2D PCA-reduced data showed that the primary structure of the clusters was preserved with with minimal to no loss in detail compared to the original high-dimensional data. In even higher dimensional data, the PCA transformation may further facilitate easier visualisation and interpretation of the clustering results.